JESPER BOEG

REAL LIFE SCRUM

Tools, tips, tricks and tradeoffs when Scrum hits the real world

CHAPTERS AND FOREWORDS BY
LIZ KEOGH AND DIANA LARSEN

InfoQueue

ENTERPRISE SOFTWARE
DEVELOPMENT SERIES

@J_Boeg

Trifork Agile Excellence Mini Book Series

software pilots
TRIFORK.

Biography: Jesper Boeg

Jesper has worked as an Agile and Lean coach since early 2006 and is now VP of the department for ”Agile Excellence” at Trifork. He has a Master’s degree from Aalborg University in the area of In¬formation 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 un¬derstanding “why”. He has a reputation for being honest and straight forward, with a firm belief 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 soft¬ware delivery system to guide success.

Having worked with Scrum, XP, Lean and Kanban for several years Jesper takes a non-religious approach to Agile optimization. He is a CSM, CSPO, CSP as well as an Accredited Kanban Trainer(AKT). He has worked with more than 50 teams across 10 organizations, trained more than 1200 people in different aspects of Agile and authored two books on the subject.

Jesper regularly speaks at Agile and Lean confer¬ences. He is a Member of the Lean-Kanban University(LKU) Management board and has served as track host on numerous GOTO and QCon conferences. Despite having both the perfect reason and opportunity he is not exactly overexposed on the popular social media platforms.

JESPER BOEG

REAL LIFE SCRUM

Tools, tips, tricks and tradeoffs when Scrum hits the real world

CHAPTERS AND FOREWORDS BY
LIZ KEOGH AND DIANA LARSEN

@J_Boeg

Written by Jesper Boeg
Foreword by Diana Larsen and Liz Keogh
Individual Chapters by Diana Larsen and Liz Keogh

Designed by Cecilie Marie Skov - http://cargocollective.com/cecilie
Graphics by Cecilie Marie Skov and Jesper Boeg

First edition December 2012

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 book. The readability and the content have been greatly im¬proved as a result of your comments. A special thanks to Troels Richter, Anne-Sofie Bille, Rachel Davies, Diana Larsen and Liz Keogh for your insightful comments and taking the time to review the book in such detail.

Contents

Foreword by Diana Larsen

Getting Better - it's the whole point of changing how we work - becoming more productive, higher performing, more profitable, more effective; meeting higher quality standards; and forging mutually beneficial relationships with customers and users. Many organizations find Scrum (or a combination of Scrum with other Agile methods) contributes a critical process piece toward their improvement goals for IT and software development. However the path to reaching the full promise and potential of Scrum is rarely smooth. Teams, and team leaders, uncover new questions every day, all with the general theme, "How do we make this work in our real world?" It turns out context matters.

Experience matters too. In this book, Jesper Boeg, an experienced developer, scrummaster, and Agile coach, shares his collection of the questions he hears most often from frustrated Scrum users attempting to make it all work. He offers answers grounded in years of encountering, and resolving, the typical lumps and bumps. Jesper's pragmatic, method agnostic approach has served many teams well.

I first met Jesper Boeg in 2007 while we worked with the same company in Denmark, me as a contract trainer and he as a technical lead. When I led a workshop on leading and facilitating retrospectives, he participated. His engagement made it clear that he "got it" about collaborative teams and continuous improvement. Subsequently, we continued to connect through our mutual involvement in additional trainings and conferences. With each new visit, I saw he had gained expertise in Scrum and in the intricacies of coaching teams for effectiveness. He became a masterful team coach and consultant. He went beyond the "cookbook" Scrum recipes to identifying real needs of real practitioners in real companies and came up with new models and approaches to fit those real needs. I've also been impressed by his commitment to building a strong Agile practitioner community in Denmark by freely sharing what he's learned through his own hard knocks and by hosting user groups and meetups. Sometimes he'd invite me to speak to them.

As you look through the Real-Life Scrum table of contents, read the questions and look for those you're asking, but don't be limited by them.

Think beyond these questions to your own experiences. Have you encountered similar dilemmas? I'll bet you have. Even if your exact question isn't on the list, you'll find benefit in reading about those that come close. If you read down the list and think, "that's not a problem we've ever seen," consider reading the answer anyway. The situation may be right around the corner for you. Also look through the list of "Topic Candidates That Did Not Make It" at the end. Use it as a resource for team research projects or brownbag discussions or a focus for your next retrospective. If you have a burning question that's not on the list, contact Jesper. I bet he'll have an answer for you.

Diana Larsen
Portland, Oregon, USA
partner, FutureWorks Consulting LLC
co-author, Agile Retrospectives: Making Good Teams Great
(with Esther Derby)
co-author, Liftoff: Launching Agile Teams and Projects
(with Ainsley Nies)

Foreword by Liz Keogh

We all know that it's impossible to master any skill in just two days of training, but for some reason we keep trying to do it.

Helping a group of people to work as a team, adopt a process, reflect on it and adapt it, in a fast-changing environment, with high uncertainty as to whether the output will even work, in a profession with no formal criteria for entry, where some of the most polar personalities you'll ever encounter in an office come into play - well, that's not just any skill. The Scrum certifications have helped many to start the long voyage to mastery, but it's an ongoing journey, full of many discoveries.

The development environment in which Scrum plays is innately hostile to its practices, too. For decades, our prevalent tale has been that the best way to deliver software is to break down large requirements into small ones, exhaustively, before writing any code, in the hopes that uncertainty will be driven out by specificity. We've grouped like-minded people with similar skills together, and tried to manage communication between them when multiple skills are required. We've rewarded people as individuals, on depth rather than breadth of skill. These old habits die hard. Of course it would be ideal if we could have business stakeholders co-located with the development team. It would be wonderful if the team had all the skills they needed to deliver. It would be perfect if they acted as a team, without worrying about individual compensation or esteem. But you don't really mean that we should do that, do you? That would be far too disruptive!

As a result, many Scrum adoptions happen in less-than-ideal circumstances. Human beings hate uncertainty, and often react negatively on discovering it. Development team members love to be esteemed as individuals. Business stakeholders can't be in two places at the same time. Everyone has to learn the skills they weren't born with. There is no silver bullet for these constraints, and there never will be. The only thing we can do is to examine where our processes are going wrong, capture the things that are going well, and try out improvements together, always looking out for the next change that might help.

In this mini-book, Jesper Boeg answers questions on common problems encountered with Scrum, and offers multiple suggestions based on real, practical solutions that have worked in many of these difficult environments. By placing each suggestion in context, and helping us reflect on where our problems really come from, he invites us not only to improve on what we're doing already, but also to appreciate how many different ways there are to do it.

If you're also looking out for the next change that might help, you could do worse than start here.

Introduction

Introduction

Though new Agile approaches, like “Lean startup” and Kanban, are emerging and XP practices are being rediscovered, Scrum is still by far the most popular method used for Agile delivery and transition (VersionOne, 2011). This is not by coincidence. Scrum is easy to understand and communicate and can help organizations reap the first benefits of Agile by presenting a simple framework for Agile management.

Having worked with Scrum for 6 years in a variety of settings, as Scrum Master, Product Owner, mentor coach and trainer, in small startup-like settings and big corporate bureaucracies, it is, however, also clear to me that many teams and organizations are not gaining the long-term benefits. Issues can be many and, to a large extent, originate from misunderstanding and misuse of the Agile and Lean values and the practices upon which Scrum is built.

In this mini-book, I have tried to collect the most typical situations we encounter when working with Agile teams and organizations and our experiences whilst dealing with them. Though all contexts are different, it never ceases to surprise me how many similar problems people face across very different organizational, cultural and geographical environments. Most of these problems seem to stem from a lack of understanding of the Lean and Agile principles upon which Scrum is built and an occasionally naive approach to change management. It is too simple to say that “Scrum by the book” is bad or that “Scrum but” is a failure, since neither will get you very far if you do not understand why you are doing it and the actual tradeoffs you are making. Though you are very welcome to read the book front to back, I would suggest that you start by picking those subjects that resonate with your situation from the table of contents. All chapters can be read individually and I have strived to cross-reference as little as possible.

For the reasons stated above and the fact that I truly believe individual solutions must be found to deal with individual contexts, this book is NOT about finding THE right answer, but aims to highlight common problems for teams and organizations in their continuous transition to a more effective use of the principles behind Scrum. It is about solving real problems when working with Scrum and prioritizing value delivery, motivation and customer satisfaction higher than the process itself.

Examples of situations dealt with in the book are

/ The dysfunctional product owner “Role” that seems incapable of providing the right material for the team to work with no matter how many times the team goes “surfing” because the sprint backlog is not ready for sprint planning.

/ Projects that are following almost every Scrum rule except being Agile. Incrementally delivering the projects according to the original spec with little or no feedback on the actual value of the software.

/ Teams are so focused on delivering the maximum amount of story points that they forget the longer-term sustainability of the code base.

It does not have to be like that, however. During the past years, we have helped many organizations kick-start, re-start and improve individual teams and overall Agile change initiatives. In this book, I will share some of the stories, as well as tips and tricks to improve your own situation.

Please note, however, that perception is very closely related to experiences and that your experiences are different from mine. This book is not written to make everybody share my views, but rather to share some of the experiences I have had with Scrum in non-textbook situations and the solutions we have found to work. If you want Scrum religion, there are numerous other places to go.

For the record, I have chosen to make all people in this book male when referring to roles and situations. This is simply due to the fact that ¾ of the people I have worked with are male and it therefore makes it easier for me to describe the situations and characters mentioned and avoid dealing with the he/she/him/her issues.

Throughout this book, you will find references to books, blogs and statements from thought leaders in the Agile community. First and foremost, however, this book is based on my own personal experiences. You should therefore not expect all statements and suggestions to be based on scientific rigorous material and backed up by large amounts of statistically valid data sets. It is a toolbox and my hope is that it will open your eyes to different ways of approaching situations and new possible solutions.
I have chosen a Q&A format to try to bring the content closer to the real-world scenarios.

”Exhausting full day planning session and ambiguous sprint content”

“Exhausting full day planning sessions and ambiguous sprint content”

Question

Dear Mr. Boeg, for a while we have been struggling with our sprint content. We spend full days in sprint planning meetings but still barely manage to get the high-level details in place, and sprint goals are often not very clear. People are starting to hate sprint planning sessions and motivation is suffering because we cannot define when tasks are finished, and during the sprint we seem to be working on everything at once. What are we doing wrong and how can we change our situation?

Problem

Dear Mr., thank you for your excellent question. This remains one of the most typical challenges Scrum teams face. Most often, we find that the problem has to do with the issue of backlog grooming.
Backlog grooming is the art of making sure that the backlog is DEEP: detailed appropriately, emergent, estimated, and prioritized (Pichler, 2010). For sprint planning meetings to be carried out succesfully, the highest priorities on the backlog need to be “Ready” for sprint planning. What this means will vary a great deal depending on the context, but by not paying attention to it, you are almost guaranteed to experience the problems you are stating above. It is, however, not a simple question of “more is better”, and as always, we find ourselves dealing with a tradeoff that needs to be understood before we can make the right decisions.

Simply put, we are faced with a U-curve optimization where both ends of the continuum are equally bad. I have tried to illustrate the situation in Figure 1 – Readyness Is an U-curve Optimization”.(please note that the figure is based on personal experience, not scientific data) and you will find a more in-depth explanation of the tradeoff in the following two sections.

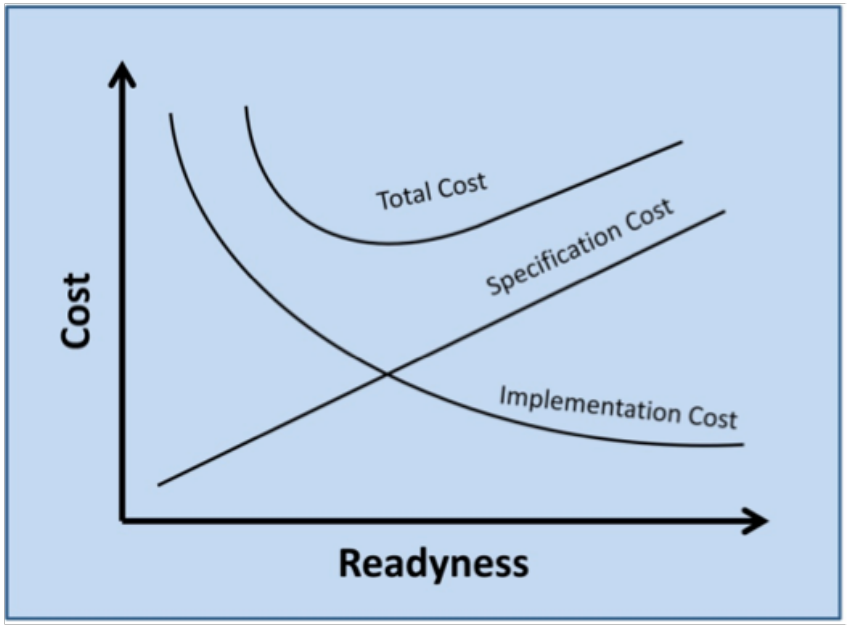

Figure 1 – Readyness Is an U-curve Optimization

Vague Specification

If you specify very little upfront, you should expect very high implementation costs (Cohn, 2010), which is the situation you find yourself in. This is due to the fact that the team will have to spend a lot of time retrieving information about the problem to be solved. They might start a task, work on it for 30 min. and then realize that a large piece of information is needed to proceed. They ask the PO, but it turns out that he needs to talk to a couple of stakeholders to find the answer, who unfortunately are all in meetings for the rest of the day. As a result, a new task is started while waiting for feedback on the first one, but the next day, it unfortunately turned out that this cannot be implemented before an external party has made a webservice available. Having just started the third task, feedback finally arrives on the first and you now have to prioritize that against the feature you are currently working on. Needless to explain, such an amount of task switching is very ineffective.

When requirements are not reasonably clear, defining when the given feature is finished becomes the responsibility of the person doing the job. This can result in a lack of motivation since a clear goal is missing or, even worse, in building more innovation into the feature than is truly needed (Cohn, 2004). You will also often encounter the problem wherein mid-way through implementation, you start questioning whether the feature should be implemented at all or maybe should be done in an entirely different way. One of the reasons we ask the PO to make things "ready" is to achieve flow in implementation. With too little specified upfront, we face a very uneven flow. Another typical consequence is the chaotic and lengthy sprint planning meetings that you mention in your question. This happens because everything needs to be agreed upon in a single meeting, and the discussion often kicks off with the topic of whether or not features should be implemented at all.

Detailed Specification

With all the risks involved in specifying too little, you might think that you are better off hiring 4 extra analysts to help the PO get the specifications exactly right. However, as you will notice in figure 1, the total cost is a U-curve. The reason is that when we approach the other end of the extreme, we start to pay a very high price for information with very little benefit as a result. Should we want to know all details in terms of interfaces to other modules or applications, the optimal position of all UI elements, necessary database table updates, and so on, we would spend more time specifying the details than building the feature (Cohn, 2010).

We have to accept that we are dealing with product development, which, by definition, means that we are working on something new. The only time we are dealing with zero percent uncertainty is if we are building the same thing twice, unfortunately with no added value as a result. Since we cannot eliminate uncertainty without eliminating value (Reinertsen, 2010), we have to accept that there will be times where estimates are off by an order of magnitude or that no users really wanted to use the feature anyway. If we translate the Pareto principle http://en.wikipedia.org/wiki/Pareto_principle to this domain, it means that we get 80 percent of the benefit of "Readiness" with 20 percent of the time/effort invested. If we cannot accept this, the best idea is probably to find another problem domain to work in, since

uncertainty is an ever present and valuable aspect of product development.

One of the biggest anti-patterns I see is development teams insisting that the business has to be sure about everything, rather than collaborating around the uncertainty.

Agile Coach Liz Keogh shares this story on the topic:

"One team I have worked with just went live with a new product. They had arguments around whether Facebook notifications should be turned on or off by default, and eventually went with 'on'. If they had appreciated the uncertainty, they would have created something where they could have turned it to 'off' quickly. But they insisted that the business make the decision, so instead of making it easy to change via a server flag, they put the flag on the client side, so they now have to redeploy to change that feature. And, yes, of course the company got backlash for it, but it is too late. This is a direct result of making the business be 'ready' about changes."

Liz's story clearly shows that it is indeed a tradeoff we are dealing with and that striving for "ready" sprint content without considering the alternatives can quickly lead you down the wrong road. Furthermore, it shows the necessity to address issues from multiple angles since the "business" will often not be aware of the technical possibilities.

Some even suggest that the cost of implementation will start to rise again when we reach the extreme end of the "readiness" continuum and, from personal experience, I would tend to agree. I have seen teams reduced to "code monkeys" by having anything from pixel-by-pixel UI specifications upfront and a list of the necessary DB updates written in the specification. Not only does this hugely impact on motivation, it also means that team members lose interest in the value of the features produced and start focusing solely on how many story points can be finished. Instead of having a team of highly motivated, creative individuals working closely together and challenging both technical and business domain decisions, you are left with a group of individuals that just show up to implement specifications and collect their paycheck.

Figure 2 - Developers Are Reduced to Code Monkeys When We Specify Too Much Up Front

Finding the right balance

As is the case with most U-curves, the bottom in figure 1 is pretty flat and, therefore, you do not have to be perfect to impact on results. An important aspect to consider, however, is that because some teams are used to "waterfall" and are working with long requirement documents, there is a tendency to push too much content and detail onto the backlog. This turns into increased pressure to "groom" and "prioritize" it. You should always strive to start with a small project vision that you can quickly get into production and then focus on delivering whole features within that. When you do that, teams tend to understand much more about what they are doing, thrash less, and collaborate to finish features. "Stop starting, start finishing" applies to the backlog as well.

On the flip side, we find small startups and other organizations that are not used to specifying anything. For them, the challenge is not to shed the old habit of too much detail, but rather to engage in the collaborative process of specifying things at all. From personal experience, I do find this situation easier to handle for two reasons. First of all, this type of organization is often more "change ready" or, in some cases, even "change eager". Secondly, they can almost immediately identify the benefit of doing some upfront specifications since they are often constantly fighting the above-mentioned consequences of having no upfront planning at all.

The key is to acknowledge that both ends of the extreme should be avoided. Too little and too much can both lead to exhausting sprint planning sessions and hurt motivation during the following sprint timebox. I hope you will find that the following toolbox contains some useful advice to achieve the right balance.

Toolbox

1 / Joint sprint backlog validation

One of the most effective ways to handle this problem is a concept we call "Joint sprint backlog validation". The simple rule is: "Nothing new must be presented at the sprint planning meeting." This means that PO and representatives from the development team must all agree that a backlog item is ready for sprint planning PRIOR to the actual sprint planning meeting. If it has not been validated, it cannot go on the sprint backlog candidate list. This way, problems or unclear aspects are identified while there is still time to get them sorted out and not at the actual sprint planning meeting where little can be done. Not only does this improve the quality of specifications, and often reduces full-day planning sessions to 1-2 hours, it also generates a shared understanding and commitment to improve. Since the development team is now also responsible for raising issues that need clarification, the product owner is no longer the single wringable neck. All too often, teams find it easy to blame the PO for problems during the sprint, but with this simple adjustment, it becomes a shared responsibility to make sure backlog items are truly "ready".

We typically suggest using one of the following two options for these backlog grooming sessions

/ Set up a meeting at a fixed time two times a week lasting ½ - 1 hour. This is often the best option if you find yourself having a hard time remembering to get it done or if you are not all in the same location. If you are distributed, it can be done via Skype, but of course F2F is preferred.

/ If you are in a proactive culture and in the same location, you can choose the more flow-based option. The rule is that the PO can ask to get input from the team two times a day. After the morning standup meeting and after lunch. It does not necessarily have to be the entire team, but could be just 2-3 representatives. Note that backlog grooming sessions are not only about the upcoming sprint. The PO might ask the team to estimate new features on the back log, give input to updated milestones or require help in terms of prioritizing non-functional requirements and all other things relevant to the continuous grooming of the backlog.

Many teams choose to adopt a Definition of Ready to make it clear what is expected before a user story is ready for sprint planning, e.g., Validated by team, Acceptance criteria, Estimated in points, Reviewed by end-users. Roman Pichler, author of the book "Agile Product Management with Scrum", offers the advice that the backlog should be clear, feasible and testable to be ready (Pichler, 2010).

2 / Often you don't need detailed specifications for items with a low cost of change

It is important to notice that small things – like items positioned on the screen, fonts, colors, etc. – are easy to change. If things have a low cost of change, they do not really need to be specified exactly, but can be validated through continuous feedback loops during the sprint. I have also experienced teams struggling with day-long planning sessions because they are trying to specify everything upfront. Guest writer Liz Keogh deals with the "real cost of change" in more detail later in the book.

3 / Use user stories to capture demands and requirements

I really, really, really like user stories, but it takes time and practice to become a good user story writer. Over the past 6 years, I have worked with a number of different requirement techniques ranging from traditional waterfall-style requirement specs to use cases and user stories. In my mind, there is no doubt that 90 percent of all projects/products will benefit from using user stories since they offer a combination of user focus, flexibility and validation, which few other requirement techniques can match (Cohn, 2004). There are a couple of different ways to write user stories; for more details, I refer to Mike Cohn's excellent book on the topic (User Stories Applied, 2004). I will only mention some key considerations here.

I like to use the format shown in figure 3.

Story:

As a <role> I want to <acton> so that <benefit>

Notes:

Relevant background details, specific algorithms or formulas, conversation that led to the result etc. References to other material.

Accept Criteria:

1. Given <context/system status> when I <input/action> i expect <result>
2. .....

Figure 3 – User Story Template

When writing user stories, make sure to include "Benefit", which most people tend to forget, but this is arguably the most important aspect (Cohn, 2004). It is vital to focus on user value before diving into implementation details and it often helps discussions stay focused and vastly reduces the time spent discussing unimportant details.

Another important aspect is the "role". Many argue that it does not make sense, since they have many different users and therefore end up writing something like "As a user of the system", which adds little value. Try hard to consider who will be your primary user; sometimes the discussion alone will surface the fact that you do not know and need to investigate that before building anything.
"Notes" are often also forgotten, but chances are that you will find yourself having the same conversation over and over again if you do not write down key elements of the discussion that led you to defining the story and its accept criteria. If you are working in a highly complicated domain, it is also the place to refer to relevant documents describing the context and maybe even formulas and particular algorithms to be used. I often recommend naming a responsible during the grooming or sprint planning meeting to add notes on the individual user stories, but preferably not the PO since he will be very busy participating in the discussion.

Last but not least, you really need to put effort into considering the "right" acceptance criteria. Not only does this make it much easier to define when work is done, the conversation will also highlight areas where expectations might differ a lot. I often get the remark "Is it really necessary to write down accept criteria - we all know what should be done?" when introducing the concept. My typical remark is "Could we try it as an experiment for just one of the small stories?". I have never encountered a single situation where the experiment did not highlight important differences in expectation.

Do not expect all accept criteria to be present when implementation starts and do not expect the ones already there to remain static (Cohn, 2004). Perfect is the enemy of good enough. As you dive more into the details, you will most likely add, remove and change them in close collaboration with the PO. You are optimizing for flow, not efficiency. Moreover, note that accept criteria are not the same as "Definition of Done (DoD)". Accept criteria validate

Definition of Done Team A

All User Stories must be:

- Code Reviewed
- Unit Tested
- Integration Tested
- Deployed to Staging
- Acceptance Tested
- Documented incl. deployment documentation

Figure 4 – Definition of Done Example

the implementation of the individual story, while DoD describes the general process you expect user stories to go through before they are considered done, e.g., an example of definition of done is shown in figure 4.
Some user stories might consist of only a few notes and 2-3 accept criteria, while others might be half a page of notes, including references to other material, and 25 accept criteria to cover the variety of cases, which the system should be able to handle. Making user stories as small as possible is always a good intention, but be careful in breaking things down to a level where there is no value for the end customer or no valid feedback on the implementation can be generated. All too often, I have experienced sprint demos where the PO, customer or end-user did not have a clue what to give feedback on because requirements had been broken down to pieces that were so small, it was impossible for them to evaluate the functionality.

Summary

Having the right level of detail is key to achieving flow and effectiveness during the sprint, but the amount of information needed may differ a lot. Making it a joint effort to validate whether backlog items are ready for sprint planning can increase quality and vastly reduce time needed in sprint planning sessions. Also, you will experience greater flow in your sprint as well as higher motivation since the goal becomes clearer and you avoid the high degree of task switching that you were concerned about in your question. User stories are a perfect way to stay focused on end-users' needs and avoid spending too much time on implementation details that are best left for later.

”We followed every rule in Scrum but our product still failed”

”We followed every rule in Scrum but our product still failed”

Question

Dear Mr. Boeg we are following every rule in Scrum. We have all the roles in place and we have been doing sprint planning meetings, retrospectives and daily standups. Our backlog has been continuously updated to reflect the current prioritization and we have sprint burndowns to track our sprint progress. Our PO has accepted almost every sprint delivery with only minor corrections so we really felt we were on our way. The problem however is that after spending a year getting our first release out the door, manage ment is now shutting us down because the system does not solve the real business problems. I am starting to seriously doubt the value of Scrum.

Problem

Dear Mr., what you are facing is unfortunately a very common situation. Without knowing the details about your situation, from your description, it seems to me you have managed to follow almost every principle in Scrum without being Agile. Possibly you have incrementally delivered the solution but left out the Agile aspect. You might have been more cost effective compared to using a traditional phase-gate model, but in your situation it might just have resulted in you spending 30 percent less building the wrong thing.

Coming from an Agile background, I am heavily biased in terms of how you should approach the problem of software delivery, but having worked with a number of large organizations, I appreciate that things take time and the ideal situation is not always immediately accessible.

That being said, I think it is crucial to at least understand the difference between incremental and Agile development. In the following, I will try to outline the differences and the tradeoff you are making.

To avoid confusion about the use of the words “incremental” and “iterative”, here follows a short explanation about how I understand and use them:
Incremental: Building something in steps. If you were writing a book, you would finish chapter 1, and then finish chapter 2, chapter 3, and so on, until you were finished http://c2.com/cgi/wiki?IterativeVsIncremental.

Iterative: Iterating over the product. Using the book analogy, you would write a draft, reflect on how the different parts fit together, add more detail, change sequence, send a second version out for review, change elements, add more detail, and so on, until you thought the book was ready for publication.

Incremental delivery

Doing incremental delivery means that you are slicing your requirement specification or backlog into pieces of work which you will complete during a timebox. Each timebox will consist of all steps including further analysis, implementation, test and release to a test environment. Compared to a traditional phase-gate model where the entire specification is analyzed, implemented, tested and released in one big batch, this model offers a lot of benefits among others:

/ Better progress tracking: Since real software is released after each timebox, progress is based on “completed” functionality. This is a much more valid measure compared to “80 percent-completed specifications”, which, in reality, tells you very little about the progress of the delivery and your chance to reach the planned scope within budget and deadline.

/ Continuous improvement: Since you get multiple chances to analyze, implement, test and release, you naturally become better at it. Actually implementing the software will also surface issues where the solution cannot be implemented as expected.

/ Efficiency: It has been proven many times over that limiting work in progress increases efficiency. Working with only a small amount of features in a timebox increases focus and decreases task switching.

/ Motivation: Motivation also increases since short-term goals create a sense of urgency and people are no longer caught in marathon

sessions with no sense of completion.

The problem with incremental delivery is, to a large part, that you might find yourself optimizing for efficiency and not effectiveness (Reinertsen, 2010). You may be going faster, delivering "quality" software with highly motivated people on board, but you could be going in the wrong direction. The Standish Report from 2009 stated that 60 percent of features implemented are never (45 percent) or rarely used (15 percent), and you could very well find yourself in the same situation. Compared to a phase-gate model like most implementations of waterfall, it does, however, still present a lot of attractive benefits and could easily increase your performance by 20-30 percent. The reason why many Scrum projects end up doing incremental delivery is that it is often a good fit with the governance model used in the organization. Another reason can be when the PO isolates himself from end-users and stakeholders, becoming the single feedback loop in terms of the value of the software.

If you only get minor corrections on your sprint delivery, it is a good sign that you are talking to the wrong person and the real stakeholder is not present in the delivery meeting. Instead of just showcasing, try putting something small into production and getting people to use it; that will tell you what is really wrong. This will help you go from incremental to Agile, which is the subject we will cover next.

Agile delivery

To use Agile software delivery, you need to add iterative to the incremental concepts (Cohn, 2010). Basically, this means that you should focus as much on getting feedback on the value of what you have implemented, using that feedback to make better decisions going forward. Doing incremental development, you expect to implement the original requirement in a step-by-step fashion, but adding the iterative element means that you plan to harvest feedback early to make better decisions going forward. Those better decisions should result in changing the scope and sometimes even deadlines and budget. As a rule of thumb, scope is often the cheapest thing to adjust. Deadlines are often expensive to move due to coordination and future trust issues, while budget is only a strategic initiative since adding more people to a late project will often just make it even later (Brooks, 1995).

But being Agile is not easy. POs are often so busy that they have little time to ensure real feedback on the delivered software and end up isolating themselves from the business they are supposed to consider their primary stakeholder. Being Agile also means truly working with a Minimum Viable Product (MVP) and striving to release the first version as soon as possible. Don Reinertsen states in his book "Principles of Product Development Flow" (Reinertsen, 2009) that one of the most dangerous things is to build more innovation into a product than is truly necessary. For a PO, this means that much focus should be on finding out what is truly necessary for the first release, since real feedback can only be achieved by releasing the system to a real production environment where the system is accessed by real users. Fortunately, this aspect benefits from the fact that "Time to market" is the number one reason for choosing Agile, according to a 2011 VersionOne study (VersionOne, 2011).

Recently, I facilitated a release planning workshop for a new IPhone/Android app. The customer had prepared a number of requirements and started the meeting by listing a total of 35 features that were considered essential for the app to launch. When I asked them if everything was truly needed, the answer was a very distinct "Yes". Fortunately, they had agreed to work with an Agile prioritization scheme and were willing to do an Agile release planning workshop. By the end of the day, the customer chose only 5 user stories to represent the Minimal Viable Product (MVP), which included only 7 of the original features. This was how little we actually had to build to launch the app and test the primary business hypothesis.

Another problem becomes that real Agility often means challenging existing governance models in the organization. While incremental development can often be adjusted to work within a phase-gate model, Agile is much harder to work in that sense. In many organizations, success criteria are not defined by the ability to deliver value to the customer, but rather the ability to deliver the agreed scope within the chosen budget and time constraints. Working with a method like Agile that actively works to change scope is, therefore, often a difficult fit.

Some describe Agile as common sense, but I tend to disagree. Our entire educational system is built up around the notion of setting a goal and then planning how to reach it. Phase gate is the underlying model for most of the things we are taught, and intuitively, "economics of scale" sounds like the best approach to optimization. Thus, not only do we have to change the me-

chanics of the system, we also have to change the model behind it. I recently conducted an interview with Don Reinertsen and we discussed the aspect of it having taken manufacturing companies +30 years to learn to not optimize for efficiency and utilization and how to address the same problem in product development, especially when our entire educational system is built up around a wrong perception of utilization, phase gates and work in progress. His rather depressing reply was:

"Sad to say, I think it is going to be a more difficult problem to address in product development."

But he hopes that does not mean it will take us 60 years (Reinertsen Interview, 2012). Many POs are yet to make this mental transition and the project is often delivered incrementally without anybody even realizing it. Things are, however, starting to change. According to a VersionOne study, carried out in the later part of 2011, the ability to handle changing requirements is now considerate of the single most important aspect of using Agile, while in 2010, it was thought to be productivity increases (VersionOne, 2011).
A final thing to consider is that not all projects should succeed. Some projects are too ambitious for their budget or markets change so fast that what was initially a good idea is now already outdated or claimed by a competitor. In those circumstances, Agile and Scrum done well should help show the failure early and kill the initiative before more money is lost.

Finding the balance

Should you give up 20-30 percent improvement because your context does not allow you to become truly Agile? No! Should you accept that one year of effort is wasted because there were no real feedback loops present? No! Reality is not black and white, and maybe the toolbox below can help you find inspiration on how to deliver more truly successful deliveries within your constraints.

Toolbox

1 / Give them data!

Management will often resist Agile because they feel they are losing control, and I often tend to sympathize with them. Traditional metrics and status reports are removed and replaced with unclear sprint burndown charts and remarks like "Just trust us, Scrum will make us six times more effective" and "We are not supposed to know where we are going, but fast feedback will ensure we get to the right place". Trust is something you earn, not a request, and Agile change agents should generally pay a lot more attention to what is going on beyond the team level.

If the main bottleneck, in terms of becoming Agile, is to make management feel safe and in control, why not spend a few minutes investigating what that means? Most traditional metrics and reports can still be generated within the framework of Agile development. It may take you some extra time and you may feel they should use different information, but if you earn their trust, it is probably time well invested. On top of those traditional metrics, show them the new Agile ones and explain to them what they mean and how they can use them to make better decisions. Even if they ask for a Gantt chart, just make it. It might look a little different since you are now delivering iteratively with cross-functional teams, but even then, the Gantt chart will give them a perspective on the situation they are used to evaluating.

When they trust you and have started to appreciate the new metrics, ask them politely whether there are any of the old metrics and reports which they feel are not as relevant anymore since you find yourself spending a lot of time making them that could be spent more effectively elsewhere. Most likely, they will point to at least one and you are one step closer. Agile is and should be a very disciplined and transparent process. If you are not able to provide a clear picture on the project's status based on real data, it is probably time to look at the reason "why" instead of pointing fingers at management. Release and Sprint Burndown charts, Velocity, Cumulative Flow Diagrams, Cycle Time and Defect Rates should be very easily obtainable for every Agile project and should all provide excellent input to risk management and other traditional reports.

2 / Communicate your intentions clearly; do not work under the radar.

While working under the radar in stealth mode might generate some interim success, it is rarely a good strategy in the long run. Whether Agile change initiatives should be driven bottom-up or top-down is out of the scope of this book and depends a lot on the situation and the resources available. No matter what, it is important to play with open cards and get permission. Do not expect people to celebrate your success and change the governance model because you delivered an Agile success in stealth mode. Your long-term chance of success is much better if you are honest and transparent. Unfortunately, I have seen many Agile change initiatives fail because the individual project manager or team thought they were free to do what they wanted and once they had proven their success, could just ask for permission later. They did not become a success because they spent most of their time working around the official process and having failed it became much harder for others to get permission to try it out.

3 / Do not think the world is black and white, all or nothing.

Sometimes you just have to accept unhealthy constraints and work within your circle of influence. Sometimes people will approach an Agile transition from a religious rather than a pragmatic perspective. They will spend endless hours complaining that the existing processes are wrong and should be changed and the reason they do not succeed is that people are consistently optimizing the wrong aspects of the delivery process. While they might be right, such a behavior is not always the most sensible. Organizations take time to change, and while you might not be able to go full-blown Agile in the very first step, chances are that you can still do a lot to improve the situation.

In one large organization I have worked with giving up, the phase-gate model was not a possibility initially. There were way too many political issues surrounding it and top management was very reluctant to let go of what they thought was the only way to keep budgets from running loose. After looking closely at what was really needed to comply with the existing model and after having an open and honest conversation with the “enforcers”, we managed to:

- Cut the scope of the first release to roughly ½
- Cut the design and specification phase to only 1/6 of the original intention, keeping requirements users faced on a much higher level
- Use the time the business analysts and end-users normally would have spent during the specification phase to continuously evaluate and give feedback during implementation
- Adjust the change management process so that all small and medium-sized issues could be handled immediately without involving project sponsors, the steering group or setting up a change management board
- Add CFDs to the list of metrics used to evaluate the project progress (eventually it became the primary measure)
- Establish a cross-functional self-organizing team

So yes, we did have to ask permission to pass phase gates. We did have a PM spend time updating rather silly metrics. We did, in some sense, do a big bang release to production (though considerably smaller than the first intention). We did spend a lot of time with bureaucratic processes stating, e.g., that virtual servers should be requested at least 2 months in advance and all the other issues where the organization is simply not set up to handle the pace of an Agile delivery. But that does not mean the project was not a success; it was 10 times more effective than it would have been had we done things as usual.

4 / Train and support your POs

Though the PO, to me, is by far the most difficult and important role to fill in a Scrum project, POs rarely get the support and training needed. 2 days of CSPO and they are off handling 2 teams working on a complex technical platform with various stakeholders. It is not surprising that they quickly revert to former habits and start focusing on the easier problem of just delivering the

initially agreed scope. Being truly Agile means working closely with stakeholders and end-users on a daily basis, ensuring that things are truly validated and new information is reflected in the content and prioritization of the backlog. This is very hard work; without support, training and the necessary time available, most POs will quickly start to focus on incremental rather than Agile development.

5 / Make resources available to your POs

All too often, the PO is left with only half his time to do the job and no one to help. Being a PO for a 7-person team is a FULL-time job; and even then, you need help from others to write user stories and ensure valid feedback. If not, the pattern is almost always the same. Feedback, agility and proactive behavior are the first things to go and the PO becomes so stressed out that hequickly turns to short-term focus and reactive behavior.

6 / Move through phase gates faster

If the phase-gate model is not likely to be changed or modified any time soon, due to organizational constraints, try to go through it faster. If you can reduce the initial scope of the project to half the original size, you will get feedback faster, reduce WIP, decrease risk, improve quality and will, overall, have a much better chance at becoming a success. Though you might still not consider yourself truly effective or “Agile”, you are still much better off.

“Over-commitment – will story points help us? ”

“Over-commitment – will story points help us? ”

Question

Dear Mr. Boeg, we find ourselves over-committing most of the time, which is really hurting our motivation. So far, we have been estimating in hours and stop sprint planning when the level of available hours is reached. A guy on the team has suggested we try story points to get better estimates – will it help us?

Problem

Dear Mr., thank you for your question. What you are experiencing is a very common issue. I suspect your real problem originates from a misunderstanding about the concept of velocity. With the strategy you use, “yesterday’s weather” is not taken into account; essentially, nothing keeps you from being very optimistic every time. While story points could help you to more easily adopt the velocity concept actually, I think the most important thing for you to understand is how velocity is used according to your chosen unit of estimation. Contrary to what many teams believe, “hourly” or time estimates, in general, do not keep you from using velocity effectively.

But do not despair. I remember someone presenting at a conference, stating that only about 20 percent of all Scrum teams know their velocity, which also fits with my own personal experience. Let me try to explain why by describing how velocity works according to the unit used for estimation.

Estimating in Hours

For this unit to make sense as a velocity indicator, we have to decouple the estimates from the actual time available. Most do this by naming them “Ideal Hours” or something similar, thereby stating that they are not just an expression of time passing by. Let us provide a small example where a new team is starting up using ideal hours as a unit:

For the first sprint, everybody is at work and thus have 100 % capacity. At the sprint planning meeting, they start breaking user stories into tasks and they are all estimated in ideal hours. Each time they finish estimating a user story, the SM will ask the team whether they still think it is realistic to do more work in the sprint. Right now, this is purely a gut feeling since they do not yet have a proven track record. When the limit is reached, the SM will tell the team that new teams can usually complete roughly ½ the amount of work they think they can and will ask the team to consider this. On that ground, they accept removing about 45 % and they officially commit to trying to reach the remaining 55 % which, in this case, equals 230 Ideal Hours. In their first sprint, they manage to complete 190 ideal hours, leaving 40 hours uncompleted. This is now their velocity (although it is pretty uncertain since it is only based on a single sprint). In the second sprint, one person from the development team is on vacation, which means they only have 85 % capacity. Since they were able to complete 190 ideal hours in the first sprint, they may now commit to no more than roughly 190 x 0.9 = 171. They end up committing to 177 with a buffer in reserve should they complete more than expected. (Velocity is very uncertain at this point.) Continuing like this the first sprints turn out as shown in figure 5.

Sprint	Completed “Ideal Hours”	Capacity %	Capacity adjusted velocity
1	190	100	190
2	200	90	222
3	160	100	160
4	210	90	233
5	240	100	240

Figure 5 – Tracking Velocity

By sprint number 3, the team is starting to get a better feel for how much they can achieve and they gradually start to reduce the buffer of sprint-ready items that are not part of the commitment. The difficult situation the team is faced with, however, is that 100 percent capacity translates to 450 hours at work for the team. Ideally, this should not be a problem because the team will focus on their proven track record. Reality is often more difficult since both the team and the surrounding stakeholders have difficulty accepting that they, for example, are able to complete only 160 estimated ideal hours in 450. This often means that pressure will rise to push more functionality into the sprint than the team can handle. This typically results in sprint goals that are not reached, quality that suffers as well as time spent

coming up with explanations as to why the last time was a very special situation; and therefore, it still makes sense to commit beyond the team's proven capacity.

Having said this, ideal hours are often still the default estimation unit for many teams since they are easy to understand and relate to and most people feel more comfortable estimating in a time-related unit. The important thing is that you decouple ideal hours from the actual time available. Velocity only works if "yesterday's weather" is taken into account.

Estimating in Story Points

Story Points and relative estimation are becoming an increasingly popular way of doing estimation. The principle is simple:

/ Choose a baseline user story that most people understand and can relate to. Assign an arbitrary amount of points to it, e.g., 8.

/ All existing and future stories are estimated according to this baseline. If it is twice as big, it is 16 points. Half as big is 4 points, and so on.

After a few sprints, the team will have a good idea about the number of points they can complete in a sprint, and since there is no reference to time, it becomes easier to accept your actual velocity. Removing the time reference does, however, make it hard to calculate the overall budget of the project without relating it to time in some way. This is due to the fact that with story points, you really do not know your cost before you have established a stable velocity. Most companies still work with budgeting cycles, making such an exercise necessary to get permission to start at all. To solve the budgeting problem, we sometimes take out a few stories (3-5) and estimate them in hours after having removed the story point estimate. From that, an average factor between points and hours is calculated and then used purely for the purpose of accommodating the budgeting cycle.

Some will also argue that by using story points, you lose the ability to calculate the cost of implementation. This is based on the misunderstanding that though it is dangerous to start relating points to time units, e.g., 1 point equals 2.7 hours, it is perfectly valid to assign a cost. If a sprint will cost you $50,000 dollars and you deliver 250 story points, your cost per point is $200.

Studies have shown that, in general, we are better at estimating in relative size compared to time. While it is hard to track improvement using time estimates (which will always be highly influenced by the team's current ability), relative estimates will show a clearer picture of the team's increasing performance. (They are not totally unbiased, but are more so than time estimates.) If you plan to use planning poker for both user story and task estimation, it makes good sense to choose a baseline story that is between small and medium in size. If it is right between the two, give it an 8, and if it is closer to medium, give it a 13. This way, you can use the cards for detailed task estimation using the same relative measure. A number larger than 20 will tell you that it is often too big and, if possible, should be broken down before becoming part of sprint planning.

Some people will refuse to try story points out at all because, on the surface, they seem too weird. That is just a transition period you have to get through and once the "giggling" stops, it quickly becomes a natural part of the daily work. Since many people have been used to time estimates, there is, however, the danger that those estimating will start to use a factor to translate points to time and lose some of the benefits attached to relative estimation.

Velocity in general

Overall, velocity serves many purposes and whether you are using points or ideal hours, I cannot recommend strongly enough that you start to base sprint commitments and release planning on a real velocity metric. For the team, it is a self-protective mechanism against being super-optimistic (a trait most teams seem to share). Motivation and performance increase as sprint goals are met, predictability is increased as carryovers are limited, and ONLY completely finished features count. In terms of PO's responsibilities, velocity helps level the workload and prepare an adequate amount of work for the upcoming sprint. Release plans become endlessly more useful since they are now based on real data instead of wishful thinking and other people's successes. Finally, velocity increases trust as management obtains data that is valid and easily understood. That being said, velocity can be very easily gamed – if you wish to do so. Therefore, if you find yourself in an

environment that suffers from distrust, local optimization, project poker and gaming, don't count on velocity to show you the true north.

Some choose to combine relative and time estimation and some electronic tools are even defaulted to this behavior. The argument is that since tasks should be no more than 8 hours, it makes sense to use time estimates on those and reserve story points for the user story level. Personally, I am not a big fan of this since, in my opinion, it complicates the overview and you end up having a task velocity in one unit and user story velocity in a different unit. Therefore, I recommend teams to stick to a single unit.

I am aware of the fact that this advice goes against what Mike Cohn (the father of Agile estimation and planning) advocates in his book and blogs on the subject "Agile Estimation and Planning" (Cohn, 2005). He believes that story points are best suited for long-term measures and not for the short term where he advocates doing sprint planning using time estimates, doing what he refers to as commitment-driven planning (maintaining planning until you do not feel you can commit anymore). I used to follow this principle myself, but I found the single-unit approach to work better in most circumstances. As should also be clear from the statements above, I am a big fan of using velocity as a boundary to sprint commitments. In Mike's blog on the subject, he does, however, also state that:

"…the sanity check of having the number of story points committed to (via commitment-driven sprint planning) be within 10% of the team's average is a reasonable rule of thumb." (Mike Cohn's Blog, 2007)

A final thing to consider is that you should be careful in using velocity as a target. Things change and the fact that your velocity is dropping is not necessarily a bad thing. It could just mean that you are now finally committing to delivering quality software and paying back your technical debt. In my opinion, velocity can be used to visualize an upper boundary for the team's current capacity, but that does not mean you should expect to reach or exceed that boundary every time. Therefore, be careful and use it with care to avoid short-term optimization and unnecessary stress.

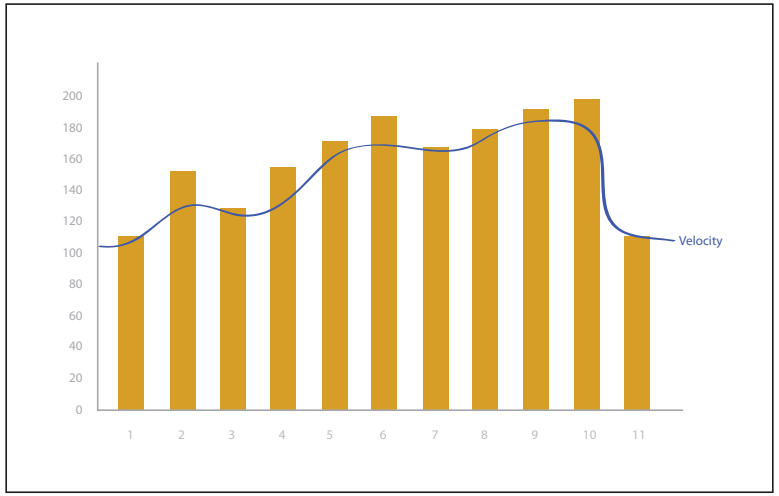

Figure 6 – A Drop in Velocity Does Not Necessarily Indicate a Problem

Conclusion

Though there is nothing keeping you from using time estimates to track velocity, most find it much easier to use story points. As an extra benefit, you get increased precision because most seem much better at evaluating relative size compared to time. In general, however, the most important thing is to actively use velocity as a way to increase motivation, effectiveness and predictability.

While I believe the above statements to be true, you should be aware that new techniques and knowledge are constantly added to this field. A recent study by Vasco Duarte suggests that estimation, in general, is a waste of time. His data shows that normalizing the size of "stories" and simply counting the number of stories completed will give you a better forecast than using story points (Vasco Duarte's Blog, 2012). As for now, there is not enough data to support this as a general statement, but it will definitely be an interesting topic to follow.

More information on how to do Agile release planning and calculate expected deadlines can be found in the chapter"How do we commit to a release plan?"

“We cannot find a PO with sufficient time, skills and knowledge”

“We cannot find a PO with sufficient time, skills and knowledge”

Question

Dear Mr. Boeg, we are facing a difficult situation. Simply put, our Product Owners are not doing a very good job. They show up unprepared for sprint planning meetings and are sometimes invisible for large periods of time doing the sprint. Progress reports are all over the place and they seem to lack very basic skills in writing good requirements. What are we doing wrong and how can we improve this before our Agile initiative fails altogether? Teams are losing drive and motivation, and customers (both internal and external) are losing trust; furthermore, on top of that, one of the POs has had to take sick leave because of stress.

Problem

Dear Mr., what you are facing is a very common problem. To be honest, I find the usual suggestion of “Hire the right PO” to be decoupled from the reality in which most organizations find themselves. Again the situation is not black and white. No doubt, POs with in-depth domain knowledge, technical insight, decision power, full-time allocation, great communication skills, a systematic and structured mindset, networking skills and knowledge about the political processes within the organization are quite effective. They are, however, so rare that they should be on the list of endangered species in most companies (Britton, 2008).

There are good reasons for this. People with domain knowledge and decision power often achieved that position by doing real work. Since they are expected to still be doing that work, they cannot take on a job as a full-time PO. Product Managers are often the most likely candidates, but since they are hired from strategic operational levels, they are often much more interested in the high-level strategic product decisions than the day-to-day work of implementing requirements, and rarely do very well working on a user story level. In addition, since we require one PO per team, the sheer numbers simply do not add up (Britton, 2008). When these people are made to

take on the role any way, it very much feels like trying to make a doctor be a doctor whilst also serving as a secretary.

There is no doubt that the PO is Scrum's Achilles' heel; nothing has proven harder in the last decade and nothing has had such a huge effect on the success of the product/project. Unfortunately, there is no black-and-white "one size fits all" answer.

What you really have to consider is the tradeoff you are making and no matter which solution you choose, you have to be aware of the problems you might expect. Since the perfect PO is not available in most situations, you really have to understand your context to make the best decision. In the following, I will try to outline three of the most popular ways to deal with the problem, including their respective benefits and drawbacks.

Business Owner and Backlog Owner

This is also known as the Proxy PO solution and the empowerment of the backlog owner/proxy might differ a lot. The idea is that the person with the actual decision power and deep domain knowledge does not have the time or skills to work with the backlog. Instead, an experienced proxy PO (or one that has the time to build experience) is put in place to handle the day-to-day backlog grooming, sprint planning, acceptance testing, release plan updates, and so on. The main difference is that actual decision authority remains with the Business Owner (BO). Though the proxy is responsible for having an updated backlog and release plan ready, all decisions above a given threshold are made by the BO. The BO will often state what overall features should be prioritized and it is then the job of the proxy to gather the detailed requirements from the business and validate them with the technical team before the final result is again approved by the BO. To reduce coordination overhead and increase response time, an agreement is usually made that the proxy can reprioritize, change and split requirements up to a certain level.

There are several advantages to this solution. One good thing is that you actually get to build skills as a good proxy PO. Writing good user stories, communicating with the development team and end-users, working with release plans and continuously grooming the backlog are all difficult skills that need to be learned and developed over time. All too often, people spend time and effort working as a PO, but when they finally become good at it, at the end of the project, time is up and they are not going to work with software product development for another 5 years. As a proxy PO, you get the chance to work on numerous projects in different domains and sharpen your skills in being a good proxy PO.

Another good thing is that you actually get a person who has necessary time available. I would prefer a person with the skills and time available anytime compared to a person with the decision power and domain knowledge without time to do the job. You can learn a lot about the domain from talking to the BO and other stakeholders, but even the deepest domain knowledge will not compensate for the fact that you are not available and do not have the time to prepare the content for the upcoming sprints.

Though it might sound tempting, there are, however, also considerable drawbacks to this solution of which you should be aware.

By definition, you introduce coordination overhead when two people are responsible and not just one. This often also results in slow feedback when the BO is not available and he is the only one who can make a decision. Knowing that such delays will frustrate the team, the proxy PO sometimes starts guessing or violates the agreements between himself and the BO. Another, and probably more serious, problem is that feedback on the value delivered is sometimes forgotten. Sometimes the proxy will start to focus too narrowly on delivering the scope of the initial backlog. If the BO is not available, the hypothesis will remain untested and focus will quickly be on delivering the backlog and not value to the end customer. Though this is not desirable, it is easy to see how this situation can evolve. When you do not have formal decision authority and responsibility in terms of the overall scope, you naturally start to focus on the things you can control, which, in this case, becomes delivering the backlog as quickly as possible. Having the BO attend sprint "demos" can help align business value with the actual features being released. It can, however, also create situations of frustration when issues between the BO and PO are escalated in the wrong forum, consequently, trust is lost between the PO and the team. In a proactive culture, these problems will be seen as an improvement opportunity, but unfortunately, that is not always the case.

A fourth problem is often the many areas of “gray”. It is not easy to clearly define where the responsibility of the BO ends and the responsibility of the PO proxy starts. If coordination between the two is not frequent and clear, some issues are not picked up, while others might not be handled in a mutually agreed way.

The problem with hiring a proxy PO is also that you often try to force them to care about the product even though they might not have been involved in the initial discussion about why the product is needed so badly. It is often better for them to work to understand why someone else cares deeply about the result and encourage them to do a good job at that. Though we will often call a proxy PO a “Product Owner”, he or she does not actually own the product.

One to rule them all

One person doing it all is the solution you will find in many books and the one suggested by the Scrum priests or others that have not spent sufficient time in real-life projects and organizations. The problem is not that this cannot work. It can! Some of the most effective and well-run projects have been steered and guided by extremely skilled POs that had the knowledge and decision power available. The problem, however, is that for every one PO that does well, 9 others fail. The reason is clear: being a good PO is very difficult and requires such a variety of skills that few are able to do it successfully. It is therefore simply too easy to argue that since some are successful, this is the right solution for everybody. Most organizations have to choose from a limited number of people to do the job and the goal is to choose the optimal solution with the constraints and capacity available.

But when it works, it works very well. You get very fast feedback and one point of entry. There are no conflicting answers and a prioritization that truly reflects the present risks and business value. Since the same person is responsible for the day-to-day prioritizations of the backlog and the overall strategy, focus is kept clear and there is a direct link between the release goal and the highest priorities. With in-depth domain knowledge and a good network within the business, feedback is better and the chance of the end product being valuable is greater.

Though in theory, there are, however, quite a few drawbacks you should consider regarding a perfect setup.

Since most POs are hired because of their domain knowledge, they lack the skills necessary to perform the job. Creating a release vision, defining the minimal viable change, writing good user stories, working with release plans and communication with various stakeholders are all very difficult tasks to perform and they require time and training to learn. If these basic skills are missing, the PO will often resort to firefighting, project poker (hoping that others will do even worse to hide the mistakes), or a narrow focus on what can be done instead of what should be done (optimizing for efficiency). If you choose the solution, it is therefore crucial that POs are given the necessary training and ongoing support to successfully deal with the challenges. Another problem is the time constraint. Being a proactive PO for a 7-person team is a full-time job in 99 percent of all situations. There is no way around it and as soon as you start to consider that you can do 20 percent of your regular job on the site, you are on a course to failure. First of all, those 20 percent always turn out to be closer to 50, and involve a lot of activities where you will not be accessible, and second of all, you need those 20 percent anyhow.

But even with full-time allocation, you cannot be in two places at the same time. You cannot engage in 2 days of requirement workshops with stakeholders and be available to answer questions for the development team. Often, however, your success as a PO also depends on the people available to help you. Though you are responsible for writing user stories, it does not mean you have to write them yourself. Business analysts or other domain experts can be an immense help in your job as a PO. There is a big workload difference between having to write everything from scratch and working with an initial base of information that “just” has to be split up, rearranged or acceptance criteria added.

One of the main reasons POs fail is that they are so busy, they never get the chance to look at the bigger perspective. They end up optimizing for efficiency, building high-quality solutions very fast that nobody wants. Feedback and long-term perspective are almost always the first things to go when people are pressured.

Therefore, choose this option carefully, remembering that you have to place yourself among the best 10 percent to be a true success.

The Product Owner Team

The product owner team is a construction we have often used. Instead of having a single PO being responsible for everything, responsibility is shared among the team. One might focus on validating user stories for the upcoming sprint, another might visit stakeholders to collect new input and a third might work on planning a release plan workshop. The team works in much the same way as a development team does. A team lead might exist, but not necessarily. Most will choose a visualization technique similar to the Scrum board and many include standups as part of the daily routine. Since the team will have to be available for a lot of ad hoc questions from the development team and other stakeholders, as well as deal with a lot of feedback and ongoing prioritization, most will, however, adopt a more flow-based structure since planning work for the next week often proves impossible.

There are a lot of good things to say about a structure like this. First of all, even with just two people on the PO team, it becomes a much less lonely experience, making it more sustainable in the long run. Also, it very much resembles adding a second or third server, thus making load balancing and dealing with different volumes of work much easier, and you are no longer working with a truck factor of 1. Since you are not relying on a single person, it is also possible to keep existing job functions outside the PO team, though naturally, the same rules regarding non-dedicated resources apply. Accessibility also becomes much higher since the chance that one person from the PO team is available when questions arise is pretty good. Another considerable benefit is the possibility to scale efforts up when needed since more people are already involved. Many organizations realize that, sometimes, one full-time person is not enough to do the job properly and I have even heard very experienced people advocating one PO per 3 developers as the new rule of thumb.

How the PO team is structured will differ a lot depending on the context. One of the organizations I have worked with uses the same structure across almost all their projects. They use a PO team structure where a project manager, an architect, a business analyst and the team lead meet 1-2 times a week to discuss prioritization and divide the PO tasks among them. As with any cross-functional team, there are areas of specialization, but nothing keeps the architect from updating the release plan if that makes sense.

That being said, splitting PO work between more people is not exactly a walk in the park. As with every team, not stepping on each other's toes becomes a problem on its own and even more so with interdependent user stories and the impossible task of keeping everybody up to date on all questions and changes. Another problem is the "single wringable neck" syndrome. Though, to me, it seems a dysfunctional, blame-focused behavior, most organizations like to have one person to point to when things go wrong. With a team of Pos, there is not a single wringable neck to blame, which can present a real problem. Last but not least, there is the cost issue. A team of POs will often be a more costly structure, especially on paper where the added value might not be easily identifiable.

Finding the right solution in your context

Hopefully, I have managed to convince you that there is not just one perfect solution that fits in all possible circumstances and that there are, without a doubt, many other constructions possible than just those mentioned above. Recently, I have even experienced a quite successful company where the PO for one team was part of the development team on another. I have also seen cases where almost all of the PO responsibilities were successfully taken on by the development team. Thus, it is really about finding the best fit in terms of the people and resources available, accepting that no matter what solution you choose, you will almost certainly improve on it over time and that, no matter what, it will still represent a tradeoff. And again, make sure you focus on the "why". Your goal is not to fill a role, but rather to find the structure that will make it possible for you to deliver software effectively, with fast feedback loops to ensure that software development becomes a learning process that maximizes value delivery.

“Only the PO cares about sprint commitment”

“Only the PO cares about sprint commitment”

Question

Dear Mr. Boeg, reading the Scrum literature, it seems that successful sprint commitments can improve the team’s performance, but it is not working for us. Most teams do not care and the sprint’s content is just adjusted as it becomes clear how much can be completed. We often have carry-overs from the past sprint and nobody seems to care a whole lot. Some POs are starting to get frustrated since they thought the team would be able to tell them what they would be able to deliver.

Problem

Dear Mr., thank you for that very interesting question. What you are really dealing with is one of the big topics of discussion in terms of Agile and Scrum. There are nearly as many opinions as there are people discussing the topic. The question we are really trying to answer is:

> *Does effective use of commitment provide increased responsibility, focus, motivation, performance, sense of accomplishment as well as more predictable outcomes?*
>
> OR
>
> *Is commitment essentially batch optimization built on the false premise that if we process a chunk of work in a single go, we will increase efficiency, while, in reality, we are provoking unsustainable pace, inflexibility and sub-optimization by ignoring the natural uncertainty involved in product development?*

In reality, sprint commitments are neither good nor bad and if and how you should use them depends a lot on the context and the people involved and how it is used. From your question, I cannot say whether your teams have

made a conscious and valid decision to skip commitment because it is not a good fit in their context, or if a lack of velocity metrics, teamwork, ready backlogs or resource allocation is making it difficult for them to experience the benefit. Let me, however, share with you some of the benefits and problems associated with using commitment and outline the tradeoff you are dealing with. While commitment may be crucial to the success of one team, it can be very hurtful to others.

Using Sprint Commitment

Big advocates of using sprint commitments will state that without a sense of urgency, work will naturally expand to fill the time available (Parkinson's Law) (Kronfält, 2011). While I do not believe this to be true in every situation, I have definitely experienced teams where a lot of their performance, identity and motivation were built around the concept of commitment. The fact that a team is able to truly commit to the sprint content often also says a lot about other factors than the commitment aspect:

/ It indicates that the team is starting to work as an actual team. They commit as a team and are prepared to help and support each other in reaching the shared goal. This is very different from a group of people with their individual set of tasks. A miserable team will simply not care about commitment.

/ It suggests that the backlog is in a healthy state. Few teams are able to commit to a goal that is vague and where little time has been spent finding the actual acceptance criteria. If the team is able to commit to the sprint content, it indicates that backlog grooming is working and that the team has a profound understanding of the work to be done before they start.

/ They have experienced success and want more. I have not seen a single team successfully using the commitment aspect that did not feel they were doing great and had had success in the past. Thus, the mere fact that people are finding commitment beneficial is a very good indicator that there is drive and motivation to do better and you are breathing a culture of success (which, in itself, can have a huge impact on the actual success).

There are, however, also considerable risks involved in using commitment; unfortunately, we have often witnessed one or more of the following behaviors:

/ Unsustainable pace, resulting in quality problems on a functional (bugs, instability, performance) as well as technical level (test coverage, tightly coupled modules, clean code standards) or too much overtime. In the management classic "7 Habits of Highly Effective People", Steve Convey argues that you must always watch the balance between "Production" and "Production Capability" (PPC balance). If you only care about production, you will ultimately destroy your production capability (Covey, 1989).

/ Too much focus on "Commitment", forgetting the bigger picture, ignoring the business and prioritizing features over value.

/ A lack of knowledge sharing, collective code ownership and communication, since focus remains on the short-term results and optimizing for efficiency within the sprint.

/ Silo optimization where questions and feedback are ignored because they take time and compromise the team's ability to reach the desired scope.

/ Religious behavior where commitment results in a complete lack of flexibility even when the outside world is truly experiencing an abnormal situation and where the lack of flexibility truly hurts the company's finances.

When conducting training or speaking at conferences, I often ask the question "How important is it that developers truly understand their users and the domain they are working with?" Almost everybody agrees that it is essential and that no matter the "readyness" of the sprint backlog, each individual developer will make many decisions impacting on the usability and value of the system. When, afterwards, I ask the question "Have developers at your company spent ½ a day during the past year exploring and observing how users really behave when using the system", typically 5 percent raise their hands. Unfortunately, sprint commitments can help strengthen such a behavior since much pressure is put on the short-term result – sometimes at

the expense of the bigger and more important picture.

However, an important aspect of commitment is using it to drive continuous improvement and learning. That, to me, is the healthy interpretation. When you do not meet your commitment, it is seen as a learning opportunity in terms of capacity, predictability and quality. Sometimes you might want to stretch your limit a bit beyond your current capacity to observe how your system reacts. In some sense, this can be seen as setting the system's "target condition", used at Toyota to systematically drive continuous improvement, though a target condition should ideally be a process characteristic rather than a production output target (Rother, 2009).

In reality, what Scrum asks you to do is to commit to a Sprint Goal, but since such a goal can be hard to articulate in reality (many teams are working on a lot of smaller items), it often becomes a commitment to deliver a certain amount of backlog items.

Working without Sprint Commitment

Some have given up sprints entirely and substituted them with a more flow-based approach, using a Kanban style system where each item is pulled when capacity becomes available. Since we are dealing with Scrum in the context of this mini-book, that is, however, not the case I will be discussing here, but you might want to read my earlier book on the subject of Kanban if you find this interesting. It can be downloaded free of charge from www.infoq.com and has the title "Priming Kanban" (Boeg, 2011).

Some teams choose to bypass the commitment aspect of sprint planning and instead use a forecast approach. There is no commitment ceremony, and the sprint backlog represents the statistical amount they have proven to be able to complete or sometimes just the current best guess. When items are removed or added due to increased or decreased performance, it is a simple matter of making ends meet to have a product increment ready for release at the end of the sprint. Carry-overs to the next sprint are often accepted as a natural part of the process. The goal becomes to have potentially shippable product increment ready, without focusing too much on the exact content.

Advocators of this approach will insist that it is unrealistic, unhealthy and against the very core Agile principles to expect to be able to plan a two- or three-week period down to the level of detail expected in a sprint commitment. Product development is simply influenced by too much uncertainty and having a flexible process that can cope with uncertainty is much better than artificially trying to plan your way to gain predictability (Yuval Yeret's Blog, 2011). Expecting such a high degree of certainty will naturally end up resulting in sacrificing either quality or scope since even on a two-week scale, you cannot lock the project triangle. This is arguably one of the reasons why the latest version of the official Scrum Guide now uses the term "Sprint Forecast" instead of "Sprint Commitment" (Schwaber & Sutherland, 2011). Though this might sound tempting due to the more flexible nature of the setup, there are a number of things you should be aware of:

/ Some teams seem to lose some of the energy, focus and drive from using Scrum when the sprint backlog becomes more of a forecast and less of a goal.

/ Some POs react with disappointment and mistrust when they find out that the team does not consider it a great deal to change the content of the sprint.

/ Dealing with carryovers is not a trivial task since half-finished code is committed to trunk and might break other features. (It is not unsolvable, but does require the team to use techniques like feature branching or feature toggles, of which I prefer the latter.)

/ It becomes too easy to change the prioritization and content mid-sprint, resulting in task switching and firefighting.

/ Individual team members might have a harder time staying focused. For some, the sprint commitment with a clear scope and deadline was the way to fight against uninvited interruptions from other parts of the organization. When the scope becomes a softer definition, they might not be able to fend them off anymore.

/ Scope creeps on individual features and tasks as there is no real boundary.

Finding the right balance

I have seen examples of highly successful teams using commitment and doing without it and teams failing miserably in both situations as well. It very much depends on the people and culture of the company. There is no right or wrong answer, but you will have to make a tradeoff. One thing is for certain, however. Blindly following the processes, hoping that the framework will make you successful by itself, is rarely a good idea. Only by understanding the pros and cons can you make a valid decision and occasionally discussing the tradeoff in an open and honest environment will bring you half way there.

Toolbox

1 / Do it right if you choose to do it!

If you want to use commitment as a way to drive focus, motivation and efficiency, you cannot be sloppy about it! To make it work, it has to be done right:

/ First, commitment should be based on a measured velocity to make sure that there is a realistic chance of achieving the goal (see chapter xx).

/ Secondly, commitment must be done on a sprint backlog that is "ready" enough for it to be clear what needs to be done. You cannot expect a team to commit to something if they have no idea what it entails. This is one of the most common reasons I find commitment not working.

/ Thirdly, commitment must be a real team ceremony. It is not enough that the number of story points match the velocity. The team must clearly state and feel that it is a realistic shared goal.

/ Fourthly, you cannot and should not expect the goal to be reac hed every single time. Actually, it is a warning signal if you do since you will most likely be compromising quality or be working with a buffer that is too large.

/ Fifthly, do not ignore the world around you. If a problem comes of something costing a serious amount of money, do not be religious. Chances are that you will be much more successful with Scrum in the long run if you show some flexibility regarding the business around you. (But make sure it is truly important and worth the sacrifice.)

/ Sixthly, if it is necessary to remove items or de-scope due to lower-than-expected performance, it is not just a formality, but rather a ceremony where the PO is involved and a new commitment is made.
If you have not yet tried it out, do not write it off initially. See how it works for you and then decide whether to proceed.

2 / Have an open and honest discussion about it.

Occasionally, all it takes is having an open and honest discussion. Try writing down the benefits and drawbacks and discuss how they relate to your current situation and way of work. You often know yourself better than you think and if people feel they are in a safe environment with focus on results instead of blame, a team will naturally identify the best fit for their particular situation.

When doing this, keep focus on what you want to achieve as well as your weaknesses and strengths. Do not discuss religion or whether the Scrum rule book advocates one solution over the other. What might be a perfect fit for the team next door might have devastating consequences for you. If you know deep down that you will be sacrificing quality, value focus and sustainability because of the pressure from sprint commitments, do not kid yourself into thinking that will magically change.

3 / Make sure it is the team that makes the commitment

Too often, it is the project manager, PO or just the senior dev. that makes the commitment. For commitment to work, it has to be the entire team that is involved. A good question to ask is: "Do you, as a team, feel comfortable about the idea of getting all this work done in the next couple of weeks, with your current velocity and other obligations taken into account?"

4 / Do not punish the team for not meeting a Sprint Commitment

"You have failed your sprint" is an expression often heard. If the team is working on a lot of new things they have no experience with, it is okay for those things to take more or less time than expected – they are estimates, not promises. If you punish them for things that are outside their control, they will either stop caring or asking for too much information and upfront analysis before they are willing to commit. Again, we are not dealing with manufacturing and we cannot expect the same amount of predictability.

"How should we approach our organization-wide Scrum transition?"

"How should we approach our organization-wide Scrum transition?"

Question

Dear Mr. Boeg, I am writing to you because we have now run the traditional successful pilot project with Scrum; we are now in the process of spreading the initiative to the rest of our IT organization comprising 300+ people. We have managed to replicate the initial success on a single team, but others have failed miserably. This caused me to look at the literature on Scrum/ Agile transitions, but it is hard for me to get a picture of the real tradeoffs we are making. Going beyond the technical aspects of how to actually implement it, what things should we consider before deciding to do a big bang rollout, a team-by-team approach or a third option.

Problem

Dear Mr., thank you for your excellent question. I am happy to report the fact that you are considering this carefully and not just blindly copying other people's successes, this will make you more likely to succeed. What you are dealing with is, however, a very big question and you might even find that you need to use different strategies in different parts of the organization.

A key question you have to ask yourself before deciding on anything is, however, why you want to go "Agile" in the first place. Sometimes there is not a single answer, but having asked the question and found the answer(s) make the whole transition process much more focused and results measurable. Change management is much like a software delivery; the most effective way of managing it is through fast feedback loops, not by planning everything upfront. However, if you want fast feedback on change, you have to know your goal, measure how your actions affect your way towards that goal and adjust your behavior accordingly. The Plan-Do-Check-Act (PDCA) cycle works for managing change as well (Appelo, 2012).

Organizational change is a massive subject with multiple angles and considerations. If you want more specific advice, there are several good books on the subject including part one and two of Mike Cohn's book "Succeeding with Agile Software Development Using Scrum" (Cohn, 2010), Jurgen Appelo's "How to Change the World" (Appelo, 2012) and "Management 3.0" (Appelo, 2011), and books by change agent guru John P. Kotter – "Leading Change" (Kotter, 1996) and "A Sense of Urgency" (Kotter, 2002). One rule to keep in mind is that all organizations are different.

"There is no one-approach-fits-all. And just asking people to change is rarely enough. Diversity is what makes a complex system work, and thus a diversity of methods is crucial when dealing with people." (Jurgen Appelo (Appelo, 2012))

What you are dealing with is not a simple dilemma and you have to consider things like change readiness, need to change, risk, available support, budget as well as technical competencies. Above all, however, you have to remember that Agile does not just mean a change of process, but has much more to do with cultural changes like servant leadership, empowerment, responsibility, courage, uncertainty, as well as respect for people. To create successful change, you have to address all parts of the ADKAR model, created by Jeff Hiatt (Hiatt, 2006): Awareness of the need, Desire to participate, Knowledge of how, Ability to implement, Reinforcement to keep.

Since this is a mini-book, I will limit my answer to primarily discuss the narrow scope of whether to choose a big bang approach, step-by-step change or an evolutionary change. The field of change management and how to change the mindset and culture within a company is a massive field. Therefore, I strongly suggest you dive even deeper into the subject and use help from experienced coaches before continuing. No matter which approach you end up choosing, it is highly relevant for you to consider how social systems react to change. An organization is a complex web of social networks and it is very hard/impossible to predict how such a system will react. Therefore, change cannot actually be "managed" – at least not using the traditional interpretation of the word.

"Culture changes only after you have successfully altered people's actions, after the new behavior produces some group benefit for a period of time." (John P. Kotter (Kotter, 1996))

Big Bang rollout

In a Big Bang rollout, the entire organization changes from a traditional process to Scrum and Agile in a manner of weeks and months. One of the first successful examples of using this strategy was when Salesforce.com made their transition to Scrum in just 3 months (Denning, 2011).

Using this strategy provides a number of benefits:

/ First, most teams rely on other parts of the organization in some sense. Doing a big bang transition, you avoid the problem of having different parts of the organization work at different speeds and the tension that will naturally arise when different processes are used in inter-dependent circumstances.

/ Secondly, successfully completing a big bang transition will reduce the time in "transition" where there might still be doubt and uncertainty. With a big bang transition, top management can show that this is the real deal and full commitment to making it a success. Though you are never done with your Agile transition, there is still a point in time where you would define the worst to be over (Cohn, 2010).

/ Thirdly, since everybody is going through the same thing at the same time, there is a perfect base for knowledge sharing and exchange of successful and failing initiatives.

/ Fourthly, you avoid the feeling of someone being left behind, the feeling of superiority and the resulting "us and them" attitude from those that are not yet part of the initiative and those in transition. In some sense, this can actually help reduce resistance since the always-present skeptics will quickly realize that the initiative receives full support by management in the entire organization.

/ Fifthly, you avoid the frustration and lost opportunity when people come back after a CSM course with lots of energy and drive, only to find the same functional silos are still present and the old metrics and management principles still dominate.

/ Sixthly, since big bang has to be mandated by top management, you avoid the frustrating situations that happen when working "under the radar". That does not mean the world is simple, but trying to be Agile without management support and with the traditional governance model still in place is the direct road to failure.

/ Seventhly, if time is critical and you desperately need change NOW, big bang is the way to go. You should never underestimate the value of a crisis and that alone can help to smooth the transition. A sense of urgency can be a very effective catalyst in any change initiative (Kotter, 2008). As a comparison, Womack and Jones state in their book "Lean Thinking" that they have no knowledge of a successful Lean transition that did not start with a crisis (Womack & Jones, 2003).

We are, however, dealing with a tradeoff, and big bang transitions present a lot of challenges and problems that you will have to deal with.

/ You need to accept that the risk of failure is much higher. Despite the "change readiness", you will encounter resistance, frustration and people blaming Scrum and Agile for all kinds of dysfunctions. If these are not handled and management is not there to fully support the initiative, you might very well find your organization rever ting to traditional processes. If you miss your first shot at Agile, chances are there will not be a second. A lack of understanding of the broader organizational changes required remains one of the main barriers to success (VersionOne, 2011).

/ You need MASSIVE coaching support. And I am not saying this to sell coaching engagements. The Agile transition at YAHOO was not done as big bang and still they suffered from a small coaching budget. Conservative data showed that the positive influence of coaches was worth an average of 1.5 million dollars per year (Benefield, 2008). This was a point further supported in "The Business Value of Agile Software Methods" (Rico et al., 2009), where they state that 79 studies with quantitative data prove an average ROI of 2633% for Agile change initiatives ($26 for every $1 invested). The problem increases manyfold when you are doing big bang. Un-coached teams quickly lose sight of the underlying values and get caught up in pointless discussions about rules,

mechanics and frustrations about the constraints of the surrounding organization.

/ You need a “change ready” organization. Personally, I would not risk this strategy working with an organization where people have seen change initiatives come and go several times and are eager to prove that this one will not see success wither. This has nothing to do with age, but rather the organizational culture and DNA. A recent VersionOne study shows that changing organizational culture is considered the largest barrier to Agile adoption (VersionOne, 2011).

/ There is a risk you will focus too much on the mechanics. With insufficient coaching support and changes going on everywhere, I know from personal experience that people quickly lose sight of the reason and start focusing on the more tangible mechanics. Instead of a cultural change, it becomes a much more superficial process change. Standups might be occurring, but the team reports to the Scrum Master, who then assigns task for the coming 24-hour period. Sprint demos are held, but feedback is avoided and seen as a threat to the budget and release plan.

/ As with any big bang release, feedback is delayed and it is more expensive to change direction. No organization is the same, and you need to tailor both implementation strategy as well as the use of Scrum. Doing big bang, all these feedback mechanisms are occurring at once and on a very large scale. It sometimes feels like steering a supertanker going 100 miles per hour through a narrow straight with potentially fatal rocks everywhere.

/ As previously mentioned, organizations are complex social net works where it is impossible to predict how exactly it will react to change. Therefore, we can only inspire to poke the system and evaluate how it responds (Appelo, 2012). Doing a big bang transition is a massive “poke”; even evaluating the result of a poke this size can prove very difficult.

In general, you should consider big bang high risk and an initially expensive solution, but also with a potential for high payoff and a quick transition.

Step-by-step rollout

Step by step or “starting small”, which Mike Cohn refers to, is essentially a strategy where you start small and gradually expand the initiative of one or a few teams/projects/products at the time (Cohn, 2010). It might be done over a period of months or even years, but preferably as quickly as possible to avoid the many issues attached to making it a marathon transition.

Over the years, this approach has become the preferred way of doing it, to the extent that Salesforce.com initially had real trouble finding qualified coaches that would help them take on the big bang approach mentioned earlier.

/ Chances of success are much better and you get the chance to handpick the projects you think are likely to become a success.

/ You get the chance to be iterative and learn from the implementation of Agile as well as expecting to do it in terms of the product development itself. This means that you can address organizational conflicts early and thereby make the path easier for teams making the transition in the future. You also get the chance to learn what coaching efforts and necessary tailoring prove most effective in your specific context. Small “pokes” make for fast feedback loops.

/ Risk is lower since you do not have to go all in at the beginning. New funding for training and coaching can be granted as initial successes prove to be worth the effort, time and money spent.

/ Most organizations will, at some point, have to address more systemic changes in order to create an environment in which Agile teams can flourish. But changing incentives, organizational structures, governance models and progress metrics does not happen overnight. With a step-by-step approach, you can make the necessary changes to start the implementation and do safe-to-fail experiments, but wait with the rest until you have more data and experience.

By starting small, you avoid the many problems with big bang releases that were mentioned in the previous section:

/ Chances of success are much better and you get the chance to handpick the projects you think are likely to become a success.

/ You get the chance to be iterative and learn from the implementati on of Agile as well as expecting to do it in terms of the product development itself. This means that you can address organizational conflicts early and thereby make the path easier for teams making the transition in the future. You also get the chance to learn what coaching efforts and necessary tailoring prove most effective in your specific context. Small “pokes” make for fast feedback loops.

/ Risk is lower since you do not have to go all in at the beginning. New funding for training and coaching can be granted as initial successes prove to be worth the effort, time and money spent.

/ Most organizations will, at some point, have to address more systemic changes in order to create an environment in which Agile teams can flourish. But changing incentives, organizational structures, governance models and progress metrics does not happen overnight. With a step-by-step approach, you can make the necessary changes to start the implementation and do safe-to-fail experiments, but wait with the rest until you have more data and experience.

On the negative side, you have to deal with:

/ The tension that will naturally arise when different processes are used in inter-dependent circumstances. If part of the organization is still working with shared resources across teams, functional silos and incentives provoking local optimization, it can prove very dificult to integrate that with people trying to reach the next level of agility.

/ Uncertainty since some might doubt whether the change initiative is for real or will soon die and be replaced by something else.

/ The feeling of someone being left behind and the resulting “us and them” attitude from those that are not yet part of the initiative and those in transition.

/ Difficulty integrating new roles and responsibilities with old structures.

/ Lost opportunity when people with lots of energy and drive find that the same functional silos are still present and the old metrics and management principles still dominate in their part of the organization.

/ It might take a long time for you to get there and feel more like a desert walk than a dynamic transition process.

/ A lack of management commitment and attention critical to the change initiative’s success (VersionOne, 2011).

The evolutionary approach

Though not directly related to Scrum, a third approach is worth mentioning since it has generated a lot of traction in the Agile community over the past years. In general, it is known as the evolutionary approach to Agile transition and is advocated by David J. Anderson through the Kanban change method. Instead of defining specific roles, ceremonies and artifacts, the idea is that you should instead focus on the underlying values of Agile by following 5 principles for process improvement:

1 Visualize the Work

2 Limit Work in Progress

3 Make Policies Explicit

4 Manage Flow

5 Improve Collaboratively

This approach involves lower risk since you initially change very little and allow organizations and teams to keep existing roles, responsibilities and job titles, and improve the system one step at a time by identifying and solving bottlenecks. David Anderson describes it this way:

"Kanban is like water. Water flows around the rocks but smooths out rocks over time."

The goal with this approach is not to do Scrum or follow a set of predefined rules, but rather to find out what makes it possible for you to deliver value most effectively in your context by using the basic Agile and Lean principles for optimization. As Karl Scotland stated back in 2009: "Instead of being Agile which may lead to success, Kanban focuses on being successful which may lead to Agile."

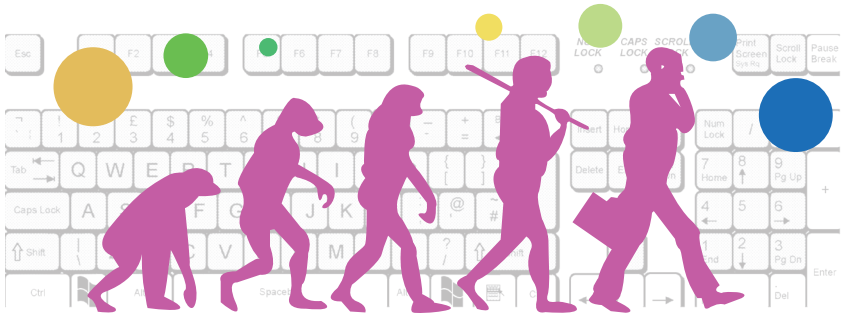

Figure 7 - Kanban Takes an Evolutionary Approach to Process Optimization

Many consider this the ultimate low-risk approach to an Agile transition, but since this book is entitled "Real Life Scrum", I will, however, refer to material like "Kanban" (Anderson, 2010) and "Agile Management" (Anderson, 2012) by David J. Anderson, "Scrumban" by Corey Ladas (Ladas, 2009), or my own mini-book on the subject – "Priming Kanban" – which can be downloaded free of charge from InfoQ.com (Boeg, 2011). In general, I find this approach to be very much aligned with Complexity thinking and Jurgen Appelo's view in the previously mentioned books, and a very "Agile" approach to catalyzing change.

Finding the right balance

Whether to do a step-by-step or big bang rollout really depends on your context. You might even want to go for the evolutionary approach if you find yourself in a situation with massive organizational resistance or simply believe that evolution is better than revolution. In all cases, you have to identify why you want to be Agile and change readiness, need to change, risk, available support, budget as well as technical competencies. It follows on naturally from the statements above that organizations with a high degree of change resistance should be very careful doing a big bang initiative, while the evolutionary and step-by-step rollouts offer a way of doing safe-to-fail experiments.

Though not a subject we will cover in detail, technical competencies are often ignored, but can be a very important aspect to consider. Successful Agile delivery does not only require process and cultural changes, but also requires the team to deliver quality software that can be maintained and changed easily to keep up with process changes. Richard Durnall argues that in most transitions, things break in the following order, which I believe to be true no matter what strategy you are using: People, Tools, Governance process, Customer, Financial Controls and, finally, Organizational structure (Richard Durnall's Blog, 2010). Too often, however, focus is too narrowly on the mindset and process shift people have to make. The reason people break is not only because they need to adopt new processes. It is also the need to acquire new technical skills, which then again quickly starts to challenge the available tools.

And it does not even have to be either-or. After a successful and a not-so-successful pilot project in an organization I worked with, it was decided to do big bang in one half of the organization and to use an evolutionary Kanban approach on the other half. This made perfect sense since there were considerable differences in terms of change readiness and the work being handled.

As mentioned before, this is a huge subject; organizational change is bigger than just adopting Scrum across multiple teams and it has its own set of challenges, skills and practices, of which I am only able to scratch the surface in this mini-book. We have, e.g., not even looked at portfolio management, which also naturally has a big effect on your change initiative since many

organizations keep way too many projects in progress to allow for sustainable delivery. Reading this is therefore no excuse to not study the subject in much more detail.

Still, I hope that you will find some useful suggestions in making the transition below.

Toolbox

1 / Build a team of internal team coaches

To reduce the load on external consultants (and thereby the budget), it makes good sense to build up an internal team of Agile coaches. Doing this, you should, however, be very careful to get the right people on board. First of all, they need to truly want to take on the job, and secondly, they need the personality necessary to help and inspire others. You should not expect the team of coaches to be able to do things on their own from the very beginning. Let them watch and learn from the experienced coaches for a couple of weeks, but do not keep them on the bench for too long. Let them make their own mistakes, explain your observation without judging and let them tell you how they would do things differently next time (Covey, 1999) (Adkins, 2010). Do, however, be careful in selecting the right people. On the one hand, you need what Jurgen Appelo refers to as early adopters to spread the word, but if these early adopters are not respected in the organization, they can work as poison instead, making those who resist even worse (Appelo, 2012).

2 / Build a team of internal management coaches

Since much of the Scrum literature focuses on the team level, management is often an overlooked element of the transition. Nothing has proven more effective than getting top and middle management as well as the Project Management Office (PMO) on board from the very beginning (Cohn, 2010). Not only will that address an area of possible resistance early on, you might actually also be able to direct the energy, which would otherwise be spent resisting the change, towards making it a success. Sometimes I have found that the managers with the biggest influence (not necessarily by title and responsibility, but because of their personality) are the biggest skeptics initially. However, if you can convince them of the value of Agile and the fact that they are essential in making the transition a success, they can become the most valuable members of the transition team. Getting the PMO office on board and establishing a team of internal managers to help spread the word can be a very effective way of smoothing the transition.

3 / Do not neglect the Product Owners

As stated before, the PO is by far the most important and most difficult role to fill, so be very careful in focusing too much on the development team in the initial face. Doing that, you will end up biting your own tail as teams will quickly become frustrated from the lack of quality input. Bad quality in rarely means good quality out. Realizing this fact in the early stage of their transition, one company I worked with decided to turn all focus and coaching support to the PO level since it quickly became clear that all other optimizations at this point would only result in pressuring this bottleneck even more. Therefore, make sure that your newly appointed POs get the training and support necessary to become successful. Chances are that they need even more than the development team.

4 / It is ok to focus on mechanics for a short period of time

Though I might be contradicting myself with this statement, you need to accept that beginners need tools and practices. Thus, do not panic when people in the initial phase become overly occupied with the practices and rules in Scrum. As you will learn from the Dreyfuss Model of skill accusation, you cannot bypass the phase where you are simply testing the mechanics of the process or tool you are trying to master. But do make sure they do not stay there indefinitely. I have seen teams that had been working with Scrum for more than two years spend endless time discussing how to measure velocity the best way, but at a closer look, not a single piece of the software had been delivered and feedback was basically nonexistent.

5 / Make sure that top management truly understand what they are committing to

Agile and Scrum are very different from traditional phase-gate models. While most managers like the thought of better-quality software delivered more effective, not all have fully understood or accepted the "Agile" part of Scrum. Make sure top management is not just expecting you to deliver your traditional requirement spec 50 percent faster than usual by doing it in increments. Make sure they have fully understood that becoming Agile means letting go of command and control-style management and that one person is no longer responsible for making sure that everybody else works to their full capacity on the right type of task. For initial pilots, working under the radar might be ok to get more information, but not including the ones with decision power in the transition is rarely a successful strategy, a view that is also supported by Yuval Yeret in a recent blog post (Yuval Yeret's Blog, 2012) as well as in "Succeeding With Agile Software Development Using Scrum" (Cohn, 2010).

6 / Measure your results according to your success criteria

If you know why you are going Agile, you should find a few carefully selected metrics to demonstrate to yourself and the company that you are moving in the right direction. Do not underestimate the power of demonstrating quantifiable results! Usually, I try to stay away from measuring productivity since it is extremely difficult to do this in any kind of objective way and it only takes very little effort to game the system. Metrics that I find useful and more objective include (though I do not believe any of them cannot be gamed):

Cycle time: The time it takes from the point you start working on something to when it is live in production.

Quality: Overall amount of defects and the number of defects registered per week.

Employee satisfaction: Whether people find it fun and motivating to go to work every day.

Customer satisfaction: Whether customers like the products you are building.

Cumulative Flow Diagrams (CFD): They will answer questions like: Do you have sufficient capacity to handle demand? What is your expected release date with your current velocity? Are you finishing your work before starting something new? How easily you can create and interpret a CFD is explained here: http://leadinganswers.typepad.com/leading_answers/files/creating_and_interpreting_cumulative_flow_diagrams.pdf

7 / Pick the right pilot project

The critical issue is to pick projects that are big and critical enough to be recognized as a real success, but which are also easy to adapt to the Agile process without too many interfaces to sub-suppliers or parts of the organization that are still using existing processes (Cohn, 2010). Also, you could face problems choosing a project that is considered so high profile that it is not likely to really get the freedom to try a new approach. By picking good initial projects, the word of mouth will quickly spread and help virally expand the values of Agile. Storytelling can be an effective vehicle for catalyzing change (Appelo, 2012), but it requires a story that people will recognize and value.

8 / Agree on your target level of Agile fluency from the beginning

In August 2012, Diana Larsen and James Shore published an article entitled "Your Path Through Agile Fluency" (http://martinfowler.com/articles/agile-Fluency.html). In this article, they identify that Agile teams develop through four distinct stages of fluency: Focus on Value, Deliver Value, Optimize Value and Optimize for System. What really stuck with me, however, was not the model itself, but rather their argument that teams that want to strive for reaching the level of "value optimization" or "system optimization" must do that from the very beginning. Essentially, that means that if we want teams to go beyond the team level and strive for "system level optimization", we must consider this from the very beginning.

"Regardless of your target, practice everything needed to achieve that level right from the beginning." (Larsen & Shore, 2012)

But that also essentially means that not everybody should strive for four-star fluency. I bet that is a statement that will give most Agile coaches something to think about – myself included.

"All these levels provide benefits, and every one is the right level for some team. Choose the level that makes sense for your situation. Often, three stars suit small organizations and two stars suit large organizations." (Larsen & Shore, 2012)

"Should we go from 3-week to 4-week sprints?"

"Should we go from 3-week to 4-week sprints?"

Question

Dear Mr. Boeg, I am the Scrum Master on a team and we have been working with Scrum for about a year. We are doing 3-week iterations, but lately, some team members have suggested that we make it 4 weeks instead because they feel we spend too much time in sprint planning meetings, demos and retrospectives. I can see what they mean, but it feels like moving a small step closer to the old waterfall model we used to hate. Am I totally off track?

Problem

My basic answer would be "How about trying a 2-week sprint cycle instead to shake things up". Unfortunately, such an answer would only result in a superficial short-term change of process, so again we have to deal with the underlying principles if we want long-term sustainable process improvement results. What you are asking is not a simple question. It does not only involve finding out what is best to do right now, but also what you believe is the most effective process improvement strategy.

Had you asked a hardcore Lean consultant, you might get an answer like:

"Thank you for asking that very relevant question. Since shorter iterations will limit Work in Progress (WIP) and bring us closer to one-by-one flow; doing so is, however, not a topic that is up for discussion. Instead, I would like you to focus on finding and solving the problems that keep us from working effectively in 2-week cycles."

This is based on the Lean faith that one-by-one flow (or one-piece flow) represents the ultimate target condition for any given process since it will help reduce WIP and increase flow, feedback and quality in the final product. As mentioned previously, process target conditions are the primary vehicle used at Toyota to drive continuous improvement. Working systematically towards, e.g., doing effective 2-week cycles is a process characteristic that would be perfectly aligned with the overall strategy (Rother, 2009).

Had you asked a person inspired by Don Reinertsen's school of thought most popularly expressed in his latest book "The Principles of Product Development Flow" (Reinertsen, 2009), you might have gotten an answer like:

"What you are dealing with is an economic tradeoff. For each iteration, you are paying a transaction cost and a holding cost and what you want to optimize is the balance between those two factors. Batch size is not a matter of faith, but rather a question to which sound economic principles apply. What you should notice, however, is that transaction costs can be changed drastically through process optimization."

After having told you this, he would probably show you a diagram looking something like figure 8 in order to explain the relationship between the mentioned factors

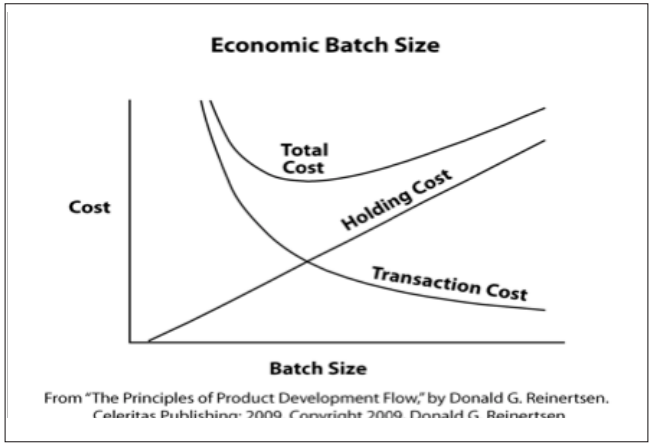

Figure 8 - Economic Batch Size (used with permission)

Asking a Scrum coach, you might get an answer like (intentional sarcasm):

"On the last team I worked with, we used 2-week iterations and that seemed to work very well. Many of my colleagues have had good

experiences going from 3- to 2-week iterations. You should try it as an experiment and see how you feel about it."

In reality, however, it is often a combination of all three that will give you the "best" answer in your situation, so let us look at the tradeoff you are facing in more detail. As you might expect, it is again a matter of understanding the principles behind and not just blindly following advice.

Shorter Sprint Cycles

Shortening your sprint cycle from, e.g., 3 to 2 weeks might be a good idea for several reasons:

/ First, you get faster feedback – not only on the value of the items produced, but also the quality. Since cycles are shorter, retrospectives are held more frequently and process improvements occur more often.

/ Secondly, doing faster cycles, less material has to be ready for sprint planning in one batch. This often levels the load on the PO and since less is prioritized in one go, there is also less risk of moving in the wrong direction for too long. As less material has to be processed in sprint planning meetings, they become shorter and more focused.

/ Thirdly, working on fewer things keeps you focused on finishing things rather than starting. "Stop starting, start finishing" is a good underlying mindset to apply. Also, you avoid marathon sprints where energy and drive are lost mid-sprint.

/ Fourthly, flexibility in prioritization. Shorter sprints mean more flexibility in terms of prioritization and reprioritization. If something new comes up, your average response time is only 1 week and your chance of sticking to your original commitment and avoiding mid-sprint interruptions become much greater.

/ Fifthly, doing shorter cycles brings problems to the surface. As mentioned previously, this is used as the primary vehicle for process improvement at Toyota. While the vision might be one-by-one flow, which in the case of product development means only working on one requirement in all parts of the process, the target condition might be effectively running 2-week sprints. Having set this goal, the team then works iteratively towards solving the problems, which might include: more effective sprint demo, planning and retrospective sessions, better quality-assurance, automating deployment procedures, more in-sprint feedback or whatever bottleneck the team might be facing. If it hurts, do it more often!

A thing to notice is that some teams have managed to drive this principle to the point where they are now releasing software to production many times a day and planning new work on a daily basis. This strategy is called continuous deployment and has gained a lot of attention in the past two years. Such improvements do not happen overnight, but require systematic improvement efforts, advanced technical setups and high discipline. Essentially, they have managed to combine the latest in both process and technical advancement to minimize work in progress to a level where code only spends minutes or hours from being written until reaching production. There are of course other elements to this strategy than just automating test and deployment scripts, and using the Just In Time (JIT) analysis approach that many teams working with Kanban prefer. Unfortunately, this subject is out of scope for this mini-book. I will, however, certainly advise you to explore the subject further since it could very well turn out to define a new era in IT.

Shortening sprint cycles does, however, also involve tradeoffs – implicitly stated in the benefits of Longer Sprints stated below.

Longer Sprint Cycles

Though problematic in a lot of areas, since they represent the opposite of the short cycle benefits mentioned above, there are benefits attached to working with longer sprint cycles:

/ First, it is less stressful when starting up. If you have not been used to iterations and Agile planning, diving straight into 2-week sprints might cause too much stress, frustration and confusion. With 3- or 4-week sprints, you get a chance to settle in and learn the concepts

of Agile without feeling you are going faster than you are able to handle.

/ Secondly, in some situations, it is not easy to get the necessary people from the business and development side together for sprint demo and planning sessions. This might be due to time constraints and geographical placement. In those situations, it might be beneficial to use 4-week sprint cycles and F2F meetings with every body involved, instead of having to rely on videoconferences or disassembled groups. Another way to put it is that transaction costs are simply too high to do shorter cycles.

/ Thirdly, looking at figure 8 above, it follows that transaction costs are lower (per item) when working with longer sprint cycles and thereby larger batch sizes. Put simply: If 2 days are spent in sprint transition meetings, with another half-day getting ready, doing 1-week sprints would result in only 50 percent of time being spent on value-adding work. In many situations, people would regard this overhead as way too high.

/ Fourthly, you get longer periods of focused development time during the sprint. People are different and some need considerable time to get into the "zone" where they are truly focused and effective. While most can become better at this, there is no guarantee that everybody will.

Agile Coach Liz Keogh shares this story about moving toward longer sprints:

"I once lengthened the sprint from 1 week to 2 because looking at cycle time showed me that over 50% of our stories were over 2 weeks long anyway. That was a recognition of where we really were, since the business were not brilliant at slicing up the stories and delivery at that point was really hard. It made sense at the time, but we based the decision on real data."

Though it is perfectly valid to start out doing 3- or 4-week sprints, I personally lean heavily toward shorter sprint cycles, using 2- and even 1-week sprints as my weapon of choice for more mature teams. Though I agree with Don Reinertsen that it is in fact an economic tradeoff and that it is beneficial to understand the principles behind, I also think that shortening sprint cycles is a fantastic tool to highlight problems and improve the process. One of the biggest problems with lengthening sprints is that it takes proportionally longer to do everything else. Essentially, you are just batching, which results in longer meetings and the knowledge from those meetings degrading all the way through the sprint. That does not mean you should do it blindly, but systematically moving toward shorter sprint cycles is often a maturing process.

Toolbox

1 / Do a thought experiment

Do a thought experiment. What would it take for you to do 1-week sprints effectively? Which part of the process would you have to change? How would you go about it? What would be the benefits?

2 / Focus on effectiveness instead of efficiency

The tendency to prefer longer sprint cycles often stems from a focus on efficiency (how much we can complete), where, in reality, effectiveness is the real issue. (What should we complete and is it implemented correctly?)

3 / Do not get too comfortable

Having a stable sprint cadence offers a lot of benefits, including increased predictability and quality. But if you have been using the same sprint length for the past year, now might be the time to challenge yourself. Though a bit faith-based, having experienced the effect of smaller batch sizes, this is one of the few areas I will allow myself to "believe".

4 / Consider adjusting sprint length according to uncertainty and project size

As a rule of thumb, you need short sprints for projects with high uncertainty. This is due to the fact that longer sprints will accumulate more uncertainty

and more time will pass, where, in reality, you could be going in the wrong direction. Project size also matters. Small projects will often benefit from shorter cycles since you otherwise very easily lose touch with both end-users and the business. Before you know it, you are done with the project in one big batch without having had a single real feedback loop in the process.

“Should we fix all bugs immediately or prioritize them on the backlog?”

“Should we fix all bugs immediately or prioritize them on the backlog?”

Question

Dear Mr. Boeg, for a while we have been discussing how to handle defects. Personally, I think all defects should be found and solved immediately to keep a quality focus, but my colleagues are arguing that there is no reason to solve minor defects at all and the rest should be prioritized along with the rest of the backlog for the next sprint since they also have a cost and business value. Who is right?

Problem

Dear Mr., I am hesitant to judge who is right or wrong. In some sense, you are both right and wrong, so let me try to elaborate on that rather confusing statement.

The problem with fixing everything immediately is that it often ends up stressing the software delivery system. By definition, you have to handle all requests as they arise, which often results in task switching and a drop in motivation. On the other hand, paying strict attention to defects and keeping the numbers low enough for them not to grow out of hand helps build a culture of “quality built in” and a fast feedback loop on deliveries.

Prioritizing them on the backlog

While prioritizing defects in the backlog might seem like the sensible solution, it also presents some problems that are often not duly recognized. First, the effect is almost always that minor bugs are placed so far down the priority list that the likelihood of them even being resolved is very low. “But why is this a problem, they are only minor bugs?” you might ask. What most people forget is that your users are unaware that you have chosen to categorize an issue as minor and placed it on a list. The result is that the same minor bug will be reported numerous times. Every time it has to be checked against existing registered bugs to avoid double entries, chances are that when the list grows too large, the overhead of doing that is simply too big and quickly the same thing is registered two, three or even four times in the bug tracking system. The cost of administering, reporting, revisiting and handling multiple user requests on the same list of minor bugs often ends up taking many times longer than fixing the bugs. If you have a group of people in IT that are highly motivated by producing good-quality software and end-users that take a big interest in the system they are working with, you might very quickly jeopardize the motivation on both sides if quality problems cannot be fixed because “they are not important enough”.

But it does not stop there. Even larger issues that do end up high up on the priority list might not even be prioritized for the upcoming sprint. Even delaying a sprint or two, the system might have changed and it becomes hard to replicate the situation that caused the bug to occur. In the worst of circumstances, you end up introducing another bug simply because you misunderstood the bug report.

The dilemma above very closely resembles the alignment trap presented in ”Avoiding the Alignment Trap in Information Technology” (Shpilberg et al., 2007). IT organizations that are highly aligned but not effective are considerably less successful (measured in growth rate) than those with low alignment and high effectiveness. Therefore, be careful in striving for alignment as a virtue.

On the positive side, this approach does, however, make it possible for the team to stay focused on the jobs at hand and avoid task switching. Chances are that blindly fixing any occurring defect immediately or following rigid rules that every defect should be prioritized in the upcoming sprint will generate equally bad results. Both strategies involve considerable tradeoff, so hopefully, I can convince you that there is a better third option.

“Zero Defect” Principle

Personally, I like following the “Zero Defect” principle used in many Lean organizations. Some interpret it to mean the behavior of “fixing everything right now”. That is not how I use it. From my view, “Zero defect” is neither blindly fixing all defects in the system as soon as they are discovered (the

"stop the line" analogy/interpretation) nor is it expecting no defects ever to occur. While "stop the line" might make sense in manufacturing, inherent variability is simply too high in product development for it to be a successful strategy (Reinertsen, 2009).

How I interpret "zero defect" is having a vision about delivering defect-free software. It is a direction and a vision, but is not a goal you expect to reach. This means that whenever a defect occurs, focus is not initially on fixing the defect, but rather on making sure the same kind of defect is not introduced in the future. The essence of a zero defect culture is that you never grow comfortably numb to defects and that you see all defects as a chance to improve the process. In most cases, you will find improvement opportunities, but sometimes you might conclude that this type of defect is simply too expensive to prevent. The important thing is to recognize that software is not a repeatable manufacturing process; therefore, we cannot expect the same results. We can, however, be inspired by the principles used to strive for "Zero Defect" in Lean manufacturing organizations (Liker, 2003):

/ Using mistake-proofing "poka yoke" devices to prevent mistakes from being made.

/ Using Andon buttons (stop the line) to prevent poor quality affecting downstream processes.

/ Creating a continuous improvement culture that values quality.

/ Developing leaders that focus on producing good parts, not just producing parts.

Continuous integration environments, automatically executable tests, definition of done checklists, defect inspection and "no commit on red build" policies are all good examples of translating these principles into the software environment.

Toolbox

1 / Starting from scratch

If you are starting a Greenfield project with almost no legacy software, you are in an easy spot and I would suggest trying out this strategy:

First, you need to differentiate between bugs that are associated with the functionality that has just been released (release bugs) and bugs found on functionality from previous versions (old bugs).

For release bugs, create the policy that they are all to be solved within the current sprint. The underlying message is:

"Thank you for evaluating the software we released; we really appreciate you giving us feedback."

Doing this is a great way of keeping end-users motivated and the feedback loop tight. Do not expect people to be highly motivated in giving you feedback if they are met with the message:

"Thank you for pointing that out; it will be prioritized on our list and we will get back to you when it is fixed/implemented. Expect it to be done within the next six months."

Your velocity should automatically reflect the amount of rework from previous sprints and thereby be self-adjusting. Whenever the cause of the individual bug has been identified, two people from the team will discuss what can be done to avoid creating the same type of bug again. Sometimes they will agree that this was a very special situation and that no new measures should be put in place. That is fine as long as it is a conscious decision.

For bugs registered on old functionality, create a prioritized list with a default FIFO queue principle (First in, First out) to keep things from growing old. If the bug has existed for at least one sprint without being noticed, chances are it does not require immediate attention. However, if a serious bug is found, it is allowed to move to the top of the list. Allow for some fle-

xibility here. The goal is to keep the list under 10, and more than 20 (roughly the amount you can keep in your head) should trigger an alarm where you might consider removing items from the sprint backlog. Again your velocity should reflect this ongoing work and self-adjust. If you keep the list under 10, the variability will be minimal. The same principle applies here: after having found the defect's cause, two people from the team will discuss how a similar bug can be prevented from being introduced in the future. To keep track of the system's defect status and insights into observed variability, I suggest using a very simple diagram, as shown in figure 9:

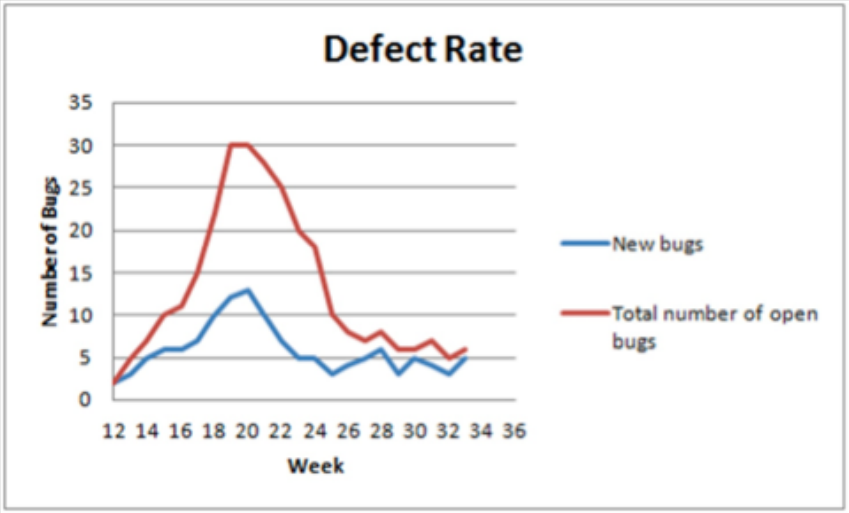

Figure 9 – Tracking Defect Arrival Rate and Total Number of Defects

All you need in terms of data is the creation date and potential fix date for all bugs in the system. Feel free to send me an email if you want an Excel spreadsheet example. All you need to do is export the data if you are using a bug tracking system.

I realize that other factors like time-to-market, risk and feedback affect how you should handle this. Do not be religious but rather use it as a rule of thumb. Sometimes it makes a lot of sense to release a buggy version because basic feedback or time to market is key (Ries, 2011). I do, however, think that you will find it works quite well in most circumstances.

2 / Improving a legacy system

The above strategy makes little sense if you are working on a legacy system with 500 registered bugs. In that case, you might consider one of the following strategies:

/ Delete all bugs older than 1 month! Chances are that with 500 bugs, 2/3 will be outdated anyway and going through them will consume many more resources than recreating those that are still relevant when they are rediscovered. It will cause some frustration, but this can be mended by making the decision public and explaining the reason behind it. Having done that, you can apply a strategy like the one mentioned above. Your velocity might be low at first, but that is just a clear signal that you are paying off on your debt.

/ Do a bug fix sprint. If for political or other reasons, deleting 485 bugs is not a viable solution, try a bug fix sprint. Sometimes you will be surprised how much you can achieve with a focused effort. With so much to clean up, you have to be tough though. If a 1-year-old bug cannot be replicated quickly, you are probably better off deleting it than spending time trying to get hold of the person that reported it to dive deep into the issue that caused it. It feels incredibly good to cut the entire list in half with 2 weeks of focused effort.

/ With high political tension and a deadline approaching, neither of the two options above might be a choice. When this is the case, I often get the remark "we will not invest in any activity with a payoff that lies beyond the deadline". The problem with this is that important deadlines are ALWAYS present. Postponing all decisions to refactor or bugfix will most likely just make sure that you will not reach any deadlines at all. In this situation, the best you can do is "improve a little at a time". Just like you would expect the programmer to leave the code looking a bit nicer than when he arrived, you should also expect the sprint to reduce rather than increase the number of bugs. Track it, visualize it, and make everybody feel like you are now at least moving in the right direction.

“We are releasing too many products”

“We are releasing too many products”

Question

Dear Mr. Boeg, I am writing to you because we find ourselves in a situation where we have managed to dig ourselves into a hole.
We have worked with Scrum for the past 4-5 years and though an initial success, it now seems to be the very root cause of our problems. The problem is that we have managed to launch so many products during the last 2-5 years that we are now facing a huge over-capacity on the business side since most of our development resources are now eaten away by maintaining and enhancing existing products. Due to this mismatch, our pipeline of products/projects seems to be increasing daily and both the business side and the development teams are at the breaking point. We have tried increasing capacity on the development side and hired external consultants, but it seems like we are treating the symptom without addressing the real illness and, if anything, it has proven not to be a sustainable solution. What are we doing wrong?

Problem

Dear Mr., thank you for your excellent question. Though I do not know the details of your situation, we have seen very similar things happen in other organizations, so I will try to answer your question to the best of my ability.

There are many aspects to your problems, which often involve everything from portfolio management, quality and product lifecycles.

Your problem is that focusing only on the execution of the individual project or product will result in unsustainable growth that will end up draining all your resources. Let us take a look at some of the typical issues you might be facing.

Maintenance Cost

What most people do not realize is that around 75 percent of the development efforts for a given system occur after the first release to production. (Actual operating cost in terms of servers, power, etc. is excluded.) This means that you are only 25 percent done (Galorath, 2012) once you manage to get the first version out the door. Using Scrum, this problem actually gets worse for a number of reasons:

- More products will be released due to the elimination of administrative waste in the development process.
- More products will be released since focus is on delivering what the user actually needs and not "everything we can think of".
- Startup costs are much lower since we do not need to know everything upfront.
- Since the business and end-user side is much more involved throughout the project. They too get the chance to learn and become better, thereby increasing their capacity.

While it might have previously taken you 6, 12 or 24 months to get a new product out the door, suddenly you find yourself releasing multiple products per year per team. In some organizations, this presents less of a problem because products are literally "out the door". If you develop software for internal use or own the products, you are responsible for operation and maintenance and this quickly becomes a major issue.

Over-capacity

When development efforts are drained by maintaining an ever-growing product portfolio, the business side is often left with considerable over-capacity. In the worst of situations, their incentives are based on their individual performance as a Product Owner, Product Manager or Business Analyst; they are therefore led to produce a huge amount of business cases, feature suggestions and resource requests that are not likely to ever be handled.

This is not ill-willed, but rather a simple result of the necessary systemic changes that have not yet been made to align with the implementation of Agile. Because of the pressure on the business people to keep busy, external consultants are hired to develop some of the new solutions or features, but since they are only hired for that particular project, they end up making the situation even worse when the responsibilities of operation and maintenance are then handed over to the already overloaded in-house resources.

While it might sound relevant for Product Owners, Product Managers and Business Analysts to keep themselves busy, the real consequences are often devastating. With too many things in the pipeline, information grows old and outdated. Too much information is added before development starts resulting in large amounts of rework or inability to change requirements to fit actual business needs. Worse, this work cannot be done without involving the people already pressed beyond their capacity. Technical aspects of the new business proposal need to be cleared with people from the development teams and the same goes for estimates and even reallocation, as you will see in the next section.

Reallocation

When over-capacity is present on the business side and under-capacity is present on the development side, a strange game that looks like allocation firefighting starts to occur. Being relatively stable in the past, teams are now taken apart to form temporary project-based "tiger teams" in charge of developing the very important projects that need to be done in half the time due to other delays. Not only does this mess with the structure and effectiveness of the teams that are split apart, the new teams are then again taken apart before they have a chance to get to work together effectively. This is either because another and even more important project has come up or because they actually managed to finish the project they were working on. Needless to say, this behavior does not improve on the situation, but rather ends up removing the remaining small amount of value-added time, ultimately resulting in everybody running around chasing each other's tails.

Conclusion

For the reasons mentioned above and probably more, you find yourself in an unsustainable situation. There are no easy fixes, but hopefully, the suggestions in the following toolbox can help improve the situation.

Toolbox

1 / Consider your product lifecycle

Few organizations put serious consideration into product/system lifecycles. When a new system or product is built for in-house or external use, it is vital to continually evaluate whether it is still profitable. The result is that far from all systems or products are worth keeping alive, sometimes the result of such an evaluation is even that some products have never been profitable at all. Usually, profit/value declines over time (as well as the value of software for in-house use). Make sure you kill products that are starting to become a burden. In such cases, it is often a good idea to include top management in the decision. First of all, they are responsible for the strategic direction of the company, but they are also usually not as emotionally attached and it is therefore easier for them to make the right decision with the necessary data available.

If you want new products and do not want to kill existing products, the only way is really to expand the budget.

2 / Focus on quality

Sometimes we find that maintenance costs are not just high because systems and products are being continuously enhanced. Maintenance costs are often high for the sole reason that functional quality and the system's stability are so low that it needs constant attention to keep afloat. It might be important for you to get new things out the door, but if you have to spend too much time firefighting and dealing with bad code quality afterwards, the approach might be too expensive. In those situations, quality might be the actual bottleneck and that is the main reason why you find yourself in an unsustainable situation.

3 / Limit Projects/Products in Progress

When the project pipeline is growing, the pressure to get projects started increases. The logic seems to be that if we can just get things started, they will be out the door sooner. Unfortunately, this could not be further from the truth. Little's Law (http://en.wikipedia.org/wiki/Little's_law) applies to the portfolio level too, and the coordination, task switching and shared resource overhead often makes the situation many times worse. On several occasions, we have identified the organization's inability to "stop starting, start finishing" to be the single most important problem to address for them to produce valuable software to end-users. You will often find that the easiest way to optimize your delivery system is to simply cut down on the number of new products/projects to be able to give the few still running full attention. The Kanban principles of limiting items in progress apply, with even greater success, to the portfolio level.

4 / Focus on effectiveness, not utilization

Unfortunately, one of the biggest challenges in your situation is that most companies focus almost exclusively on utilization (keeping everybody busy all the time). This becomes a major problem in your situation when you are faced with over-capacity. Lean thinking and the Theory of Constraints (TOC) have shown us that optimizing for utilization is one of the most dangerous strategies you can apply. It results in a large amount of work in progress, long queues within the system, and quality problems, and represents an exclusive focus on local optimization. The details of this problem and how to handle it are out of the scope of this book, but I would strongly suggest you read books like "The Goal" (Goldratt, 2005), "Principles of Product Development Flow" (Reinertsen, 2009), "The Toyota Way" (Liker, 2003), and "Kanban" (Anderson, 2010) to gain a deeper understanding of this problem. I will, however, leave you with a few statements to inspire you to look further.

> / From an economical point of view, you would probably be better off having half your staff of business people (or what constitutes your current over-capacity) doing nothing but brow

sing the Web all day. That, however, would be a poor use of your talented people and their motivation for going to work would most likely quickly drop. Instead of having them start more work, have them look into possible improvement opportunities to make the best use of your dev. capacity or make sure that items reaching the dev. team are truly “ready”.

/ If “development” is your bottleneck, your entire system’s capacity is defined by how much they can do. This means that whenever they are working on something irrelevant, things that are not “ready” or work that could be handled by someone else, it equals wasting time for the entire system and everybody involved. For many people, this is very hard to grasp.

/ By not leveling the work of the rest of the system to the capacity of your bottleneck, you are reducing the capacity of the bottle neck even further. This is because your work in progress rises, slowing down feedback loops, increasing quality problems at the bottleneck and the fact that the bottleneck will often have to help plan for the new work to arrive, which results in increased task switching.

“Maintenance and operation tasks are ruining our sprint commitment”

“Maintenance and operation tasks are ruining our sprint commitment”

Question

Dear Mr. Boeg, we often find ourselves with a clear picture of what we want to achieve after sprint planning and the team is committed to reaching that goal. But in the sprint, that focus is disturbed by bugs that have to be solved, users that need to be educated in using the system, or small enhancements to features deployed after the last sprint. All of which need to be addressed to make the system appropriate for day-to-day use. On the one hand, I see the necessity to assist with this, but on the other hand, the team is frustrated by constant task switching and missed sprint commitments.

Problem

Dear Mr., your question has already been somewhat dealt with under “Only the PO cares about sprint commitment” and “Should we fix all bugs immdiately or prioritize them on the backlog”. But since this is a very common situation faced by many teams, I think it deserves an answer on its own.

As you have already identified yourself, there is a real tradeoff involved in dealing with this problem and are even more aspects to consider than those mentioned in your question.

Let us try to look at some of the problems and benefits involved and then I will try to illustrate some of the solutions we have experienced working well.

Screening the team from all outside interference during the sprint

Some people I have met define this as “pure Scrum”. All things to be completed are prioritized in the sprint planning meetings, and all requests for bug fixes, enhancements or support are handled outside the development team and if necessary, prioritized for the upcoming sprints. This setup makes it possible for the team to be very efficient during the sprint and keep 100 percent focus on the job at hand.

But there are also considerable drawbacks to this behavior:

/ On several occasions, we have found that the business and end-users interpret this behavior as the team caring more about the Scrum process than supporting their needs and generating value. If requests for corrections to newly deployed features are refused or they cannot get the necessary support to move on when they are stuck, they get frustrated and come to think of the Scrum team as more of a Scrum cult than a value-adding part of the organization.

/ By insisting on total isolation during the sprint, the team is missing a great chance to get firsthand feedback on the things they have developed. Remember that your goal is to generate value, NOT features. A good friend of mine believes that first-level support is the single most important feedback loop available for developers and an opportunity they should cherish.

/ The team might start to focus on the planned content to the extent that even perfectly valid learning and feedback during the sprint execution is ignored. When issues arise, they will insist that the sprint content needs to be more carefully specified next time to avoid surprises or unforeseen elements during the next sprint. A sprint is a learning process and though it is important to make things “ready” before work is started, you should never expect everything to be fully understood. Multiple feedback loops should be expected during the sprint.

In general, I think it is a big misunderstanding to label this policy “pure Scrum”. As far as I remember when I read my first book about Scrum back in 2005 – “Agile Software Development with Scrum” (Schwaber & Beedle, 2001) – it clearly stated that time should be reserved in each sprint to deal with feedback from the previous sprint.

Solving all issues as they arise

On the opposite end of the continuum, you will find teams where there is a direct line to the individual developer. This offers a very fast feedback loop and there is very close communication between the development team and the business and end-users. Everybody on the team feels the pain when a new feature is not behaving as expected, and appraisal when things work well is unfiltered and direct.

But unfortunately, there are also a number of problems with this approach:

/ When there is a direct line to the individual developer, prioritization will be done by the individual. Potentially, a very important feature in the sprint backlog could be sacrificed to solve a trivial bug. Without any dialog with the product owner, chances are that it could very easily result in bad prioritization.

/ Half the day is often spent task switching between new features and support calls, emails and bugs and enhancements that need to be solved.

/ Sprint commitments are often not met and the team experiences a very unstable velocity.

/ With such a flexible attitude, the PO will start to think less of the sprint commitment as a defined scope for the sprint. New features will start to creep in mid-sprint and existing features on the sprint backlog might be removed or changed drastically, resulting in a lack of focus, more task switching and a drop in motivation.

Conclusion

Needless to say, both ends of this continuum are often not the optimal solution because significant problems are attached to both scenarios. In the following toolbox, I will explain some of the techniques we have used to try to find a better middle ground.

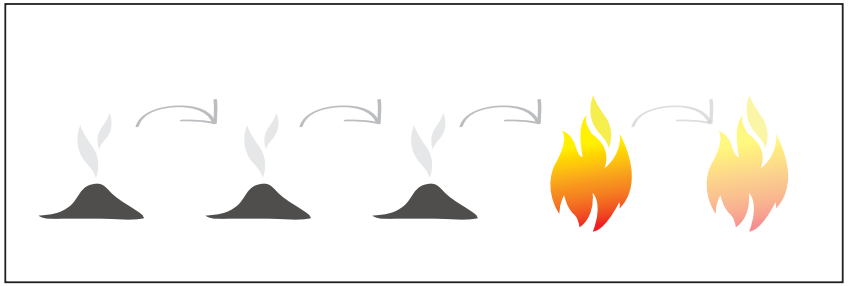

Figure 10 – Long Term Focus is Often Lost When Firefighting

Toolbox

1/ Feedback loops in the sprint

If at all possible, one of the most effective ways of reducing maintenance and operation issues is releasing quality software with very few outstanding issues. This can be done by including feedback loops with the Product Owners and/or end-users as soon as a feature is ready for feedback (could be completed or just barely functional enough to get valid feedback). This way, real acceptance tests are carried out in the sprint instead of leaving it to the sprint demo where days or even weeks might have passed since the feature was completed (Pichler, 2010). This way, feedback loops become much more effective and quality is much higher at the end of the sprint. As an added benefit, the relationship between end-users and the development team becomes much closer.

When I propose this principle, I often get one of the following reactions:

1. That will never work in our context. We do not have access to end-users during the sprint.

2. That will ruin our flow during the sprint. Acceptance tests will take time to set up and result in changes which will slow us down and make it hard for us to reach our goal.

3. That is not our responsibility. We are just responsible for delivering what has been specified.

4. That will ruin the purpose of the sprint demo.

As for number 1, you might be right, but often, not even a small effort has been done to test this hypothesis. I have yet to experience a situation where we have not managed to get hold of at least 2 end-users when we explained the benefits.

As for numbers 2 and 3, they represent a lack of understanding about the goal and long-term perspective. With the help of a good Scrum Master and Product Owner, a very effective setup can be designed that requires a minimum of time from developers. If you agree that early feedback is good, why would you not then want to get it while you can still remember what you were working on? Part of understanding Agile is the realization that no one is an island; only by optimizing the entire software delivery system can we create better results.

In terms of number 4, what really happens is that the sprint demo becomes much more effective. Instead of focusing on a number of small issues, you have time to evaluate the bigger picture and make sure you are really heading in the right direction. Also, you might find that you can do it much faster and reduce the sprint transition overhead.

If your hardware setup allows it, you might even be able to continuously release software to a production-like environment during the sprint and perform the acceptance test on a real environment to include tests for obvious constraints related to hardware aspects.

2 / Use continuous delivery principles

Closely related to the subject above, I have become a huge advocate of using the principles of continuous delivery described in Jez Humble and David Farley's book "Continuous Delivery" (Humble & Farley, 2010). Essentially, it means that software should always be in a releasable state and they provide detailed guidance on how to set up deployment pipelines and other features to make this possible. This can help massively reduce the number of problems found after a release to production and shorten feedback loops by an order of magnitude.

3 / Make it possible for the Scrum Master to help protect the team

I have often heard teams complain that there are too many disturbances during the sprint. When I ask why I often experience dialogue like this:

People are writing/calling me directly with small things from older projects – new features they would like to be implemented our just requests to review some material or participate in a job interview.

Are these tasks more important than the features prioritized for the next sprint?

No, I do not think so.

But you do them anyway?

Yes, if not I feel like a bad colleague and I do not trust the Scrum Master to handle them in a way, so the people asking for my services will feel they have been treated the right way.

or

I have no idea what to do with them if I am not to fix/respond myself.

Basically, you cannot expect the Scrum Master to fulfill his role if he is not given the chance. On the other hand, this is the time where the Scrum Master has to prove that he is not a Scrum fundamentalist and will support the rest of the organization in a meaningful way. The fully allocated team only exists in theory in many organizations, but that is no excuse for constant firefighting and solving things under the radar. Therefore, agree to give the Scrum Master a chance to help protect the team and deal with interruptions. Until proven otherwise, trust him to do this in a way so that the rest of the organization will feel they are treated fairly. Screening does not have to mean ignoring, and a good Scrum Master will find a balance between keeping the team protected and supporting the rest of the organization.

4 / Rotating the maintenance job

To screen the team effectively, but allowing for operation and maintenance jobs to be handled, some teams chose to rotate the job. This can be done in the rather explicit way of simply placing those responsible in another room. Some teams calculate with no available capacity in the sprint for those doing maintenance, but keeping the possibility open of them helping out should time become available. I have also worked on a team where requests were only handled from 8 to 12 in the morning (except priority one issues) and the team would simply rotate the job on a weekly basis.

While it does work quite well in some circumstances, there are issues with this solution of which you should be aware:

/ There is no guarantee that those assigned will have the knowledge to handle all issues. So while it is a good exercise to move outside your comfort zone, in reality you often end up interrupting the development team anyway since all issues cannot be handled.

/ Being away from the development team for short or longer periods of time makes it hard to keep the same team feeling and everybody updated on all the technical and process changes going on. Therefore, you quickly end up missing vital information, which is necessary for you to perform as an effective team member.

/ When your shift is over, you will often face a number of open issues where you will still be considered the primary contact and will either have to spend time handing it over or dealing with it, while in theory, you are 100 percent allocated to development.

5 / Reserve a separate part of the board or color and track.

Probably inspired by the many teams I have worked with that have adopted Kanban principles, I have become a huge fan of using color codes or using a separate swim lane on the board to visualize maintenance and operation tasks. The arrival rate is very steady and predictable and it can therefore be handled as a natural part of the team's velocity. As mentioned in the previous section, on defect handling, you might want to distinguish between different types of request. Visualizing them on the board gives the team a clear indication of the amount and also the sense that this is not something that is going on "under the radar". You might even want to assign certain policies and, e.g., decide that the maintenance and operation queue should always be in a prioritized order and priority 1 issues should be handled as soon as capacity becomes available. Having visualized the tasks, tracking becomes very easy and just counting the number will often give you a good indication about the quality of the system and the time spent handling defect and support issues. (The size of these issues is almost always normally distributed, making it unnecessary to estimate and track time spent.)

6 / Use a maintenance team

Some organizations I have visited opt for the solution of having a separate maintenance team who takes over once the solution is no longer undergoing heavy development. This can be a good strategy if your products are relatively stable and the customer is not in need of constant enhancement requiring deep knowledge about the solution design. It limits the tail of products/projects that individual developers carry around and makes it possible for people that have worked within the company for many years to actually focus on their work. However, you should be aware that, more often than not, you will face one or more of the following problems:

/ It is extremely hard to hire qualified people to be on this team and you probably have to pay them more. Most good developers like to work on new products and quickly lose motivation if they are placed on a maintenance team.

/ Interfaces between development and maintenance will likely become more bureaucratic over time and the maintenance team will require more and more documentation, tests, contracts, etc. before they will take over.

/ You lose direct feedback. As stated above, maintenance can be a very effective way of securing the feedback loop.

/ If one side is awarded for getting things out the door and others for stability, you are likely to face unaligned business decisions.

Developers will most likely, at some point, start to care less about quality, and maintenance will start to care less about business value.

“How do we commit to a release plan?”

"How do we commit to a release plan?"

Question

"Dear Mr. Boeg, I like the notion of sprint commitment, but what I really need is the team to commit to a release plan and not just what needs to be done within the next few weeks. I know that with Agile, you do not know where you will end up, but missed deadlines are still expensive and the trust we worked so hard to build seems to drop every time."

Problem

Dear Mr., what an interesting question. I can guarantee you that you are not the only person faced with this problem. Before we dive into the details, let's agree that if you take away the religious aspects and how things "should work" in theory, Agile is no better or worse off in terms of deadline promises compared to traditional methods. Missing deadlines seem to be the rule rather than the exception in IT no matter what method people use. I would argue that though Agile projects might miss deadlines as often, they generally miss them with fewer financial consequences and by less time, due to the simple fact that less risk and development effort is built into each release. I have no data to support that claim, however.

Some will argue that maybe we do not need deadlines altogether (Olga Kouzina's Blog, 2009) and although this is an interesting topic, it is out of the scope of this book to cover in detail. In most situations, it is not a question of "should there be a deadline?", but rather "what should the deadline be?" or "how do we deal with the deadline placed upon us?" and this is the area we will deal with. Therefore, let us look at some of the benefits and problems with deadlines and, finally, how to work successfully with them in an Agile context.

Benefits of deadlines

Deadlines offer many benefits. Having a deadline means there is an end-point to what you are doing and a goal to work toward. This gives a feeling of progress and being closer to your goal than you were yesterday. Having a deadline also makes you less likely to fall prey to Parkinson's Law. (Tasks will expand to fill or exceed the time available.) Most people that have been part of a successful team will also recognize that deadlines create a sense of unity. Who does not remember the surprisingly fun late hours at work, eating pizza and fighting to get things ready for the upcoming deadline - going to bed exhausted but still with a feeling of achievement? It might later turn out that the deadline was meaningless or the overtime made you produce less with lower quality, but the feeling of being part of a team that fought together persists.

Problems with deadlines

One of the problems with deadlines is that they are too often based on wishful thinking, external pressure or sometimes just made up without considering the reality of the situation. Also, they are often also part of an unsuccessful attempt to constrain all 3 (4 if you include quality) elements of the project triangle, which cannot and should not be done due to the high amount of variability associated with product development (Reinertsen, 2009). In the cases where deadlines continuously push the system beyond its current capacity, a number of bad things start to happen:

- Quality drops
- Stress increases
- Velocity/Productivity drops
- Work in Progress increases, delaying feedback loops and increasing risk
- Efficiency prevails over Effectiveness

And there are probably numerous other consequences.

Next step

So how do we rip the benefits of deadlines without experiencing the negative effects mentioned in the section above? First, it is necessary to realize that since you cannot constrain all elements of the project triangle, you need to make the best financial choice according to your circumstances. In most situations, scope is by far the cheapest thing to keep flexible, for the following reasons:

/ Budget is hard to flex. Increasing the budget often means increasing the number of people. This is often very difficult since adding more people to a late project will make it even more late (Brooks, 1995). Adding more people is a strategic move for the long-term perspective. When an 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 he uses it to show some kind of action. The only case where I have found this not to be true was an understaffed project where we were luckily able to get people on board that had worked on the solution before and could therefore be brought up to speed quickly.

Though deadlines are often the preferred thing to extend when projects are delayed, it is rarely the best choice. While it might seem like an easy short-term fix, I find the long-term consequences problematic:

/ As you stated in your question, missing a deadline affects trust. Whenever you miss a deadline, there is almost always the tendency to ask for more control, more planning, deeper levels of detail and additional legal paragraphs in the contract next time. So while it is tempting as an easy fix, make sure you consider the long-term consequences.

/ Missing a deadline starts a domino effect of coordination. This is especially true on larger projects where communication plans, staff training, software development and hardware deliveries are often closely tied together and one delay causes multiple re-coordination exercises between the different deliveries. The burden can be helped by having close feedback loops and cross-functional teams spanning more than just software. In most larger organizations, it is, however, a struggle to get testers, developers and business analysts to sit together and, including marketing, training and hardware, is not likely to happen in the near future.

/ Missed deadlines decrease the maturity of your Agile initiative. I have seen this many times in the organizations I have worked with. If it is ok to miss deadlines, there seems to be a notion that it is ok not to deliver quality software at first because you can always extend and fix it later. Since deadlines are moved frequently, tracking progress becomes less of an issue, making the likelihood of missing the next deadline even bigger. Because tracking is suspended, there is no clear picture of how much should go in the next release and prioritization, in general, seems to become a less important issue. When the next deadline is missed, the pressure to get more functionality in the subsequent release is increased; thus, the spiral of death begins and suddenly nothing has been released for 6 months.

The problem with the factors above is that the long-term consequences are often hard to quantify and using a simple "Cost of Delay" function will therefore rarely present the right picture. That does not mean it is not important to know your COD and I highly recommend reading the relevant chapters from Reinertsen (2009) if you want to explore this in more detail. Though working with flexible scope requires more discipline (yes – MORE, not less), it is often the cheapest way to handle the variability inherent in product development. When people hear the term "flexible scope", they often interpret it as a laissez-faire approach to software development that really 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 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 it is therefore essential to track your progress to make sure that the project is still feasible and that you are working with the right prioritization.

There are a number of issues related to this, but I trust that you will find the suggestion in the toolbox very helpful. It is inspired by the Story Map technique introduced by Jeff Patton (Jeff Patton's Blog, 2008), the XP planning game and a very simple chart to track progress. I have found this combination to be one of the most effective ways of facilitating any kind of release plan workshop and tracking how you are progressing.

Toolbox

1 / Start with a story map release planning session and track progress.

If you have not experienced story mapping before, I recommend that you try the exercise I blogged about a while ago (Trifork Agile Blog, 2012) and that you read Jeff Patton's blog post mentioned above. The exercise can be done in around 45 minutes and you get a much better feeling about the concept before trying the real thing. With story mapping, you are working with a 2-dimensional backlog which turns out to facilitate not only planning the content of the release, but also working with the constraint of having deadlines. I will not cover the story map process in detail except what is needed to understand how it can be used for effective release planning.
In figure 11, I have tried to illustrate the story map process as well as the final result.

As you will see, the first release consists of both the walking skeleton (red/pink area) and the green area. The walking skeleton consists of the user stories needed to bring a bare minimum of features in production – a "Minimal Viable Product", to use a term from The Lean Startup (Ries, 2011). So why include the second most important features in the green area? A good

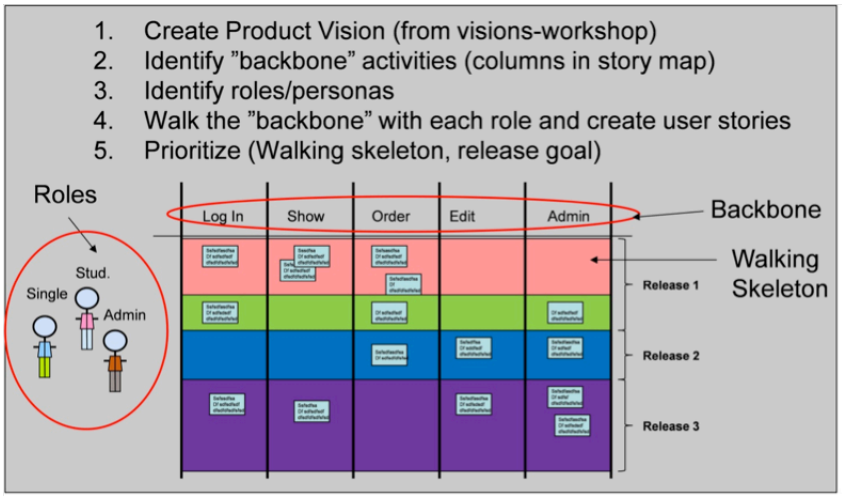

Figure 11 - Story Map Process Illustrated

practice from the Extreme Programming planning game seems to have been forgotten. AFTER you have prioritized the essential features, you should look to fill about 30 percent of your release backlog with content that you consider to be extremely important, but which is not essential. Though important, it can be removed should you find yourself with too little capacity to deliver on your deadline promise. This provides you with some degree of flexibility while still maintaining a focus on time to market and customer value.
But how do you track this? Actually, it is quite easy. Figure 12 shows an example of how to track according to the priority decided on the Story Map.

You should expect velocity to follow an S-curve, not a straight line, since it takes time to get things up and running as well as the unexpected final touches before you are ready to get it out the door. To demonstrate how it works, I have put three possible velocities on the chart. If you are faced with a velocity like number 3, you might want to kill the release early or at least seriously consider whether it is still financially viable. Expecting to cover even the essential features before the deadline seems next to impossible. Faced with number 2, you could decide to wait a while to get more information or already start the discussion about which features to remove from the "green area". With a velocity like number 1, you might want to consider an earlier release date or whether to add more features. (You could downsize

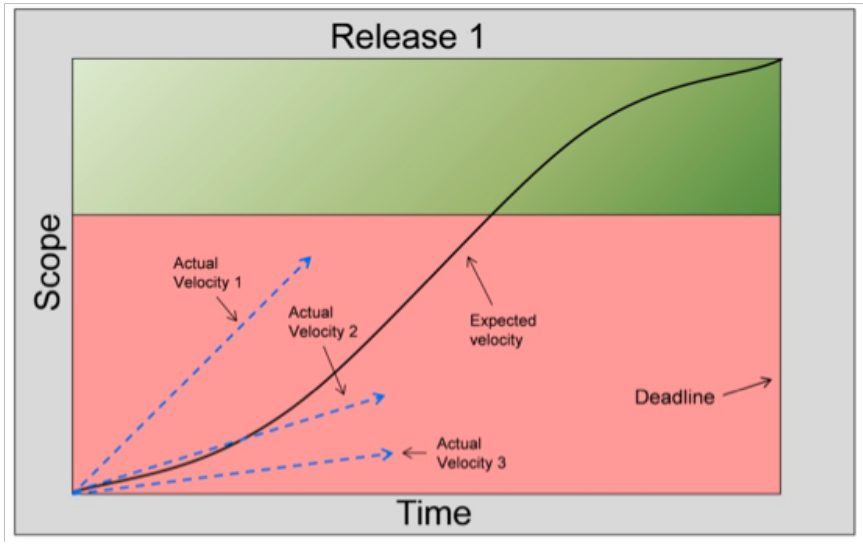

Figure 12 - Release Plan Tracking Example

the team as well, but since I am a big fan of stable teams, that represents a short-term local optimization to me.) For those with experience using Cumulative Flow Diagrams (CFDs), you will notice that there is a close resemblance. If you add "planned velocity" to your CFD, you will get almost the exact same picture as well as other interesting facts like WIP, individual queue sizes, cycle time and the balance between demand and capacity. Figure 13 illustrates using a CFD to track release progress – the S-curve is continuously adjusted to fit the current scope of the release.

I cover this use of CFDs in more detail in chapter 8 of "Priming Kanban". Some keep the actual story map right next to the above diagram to clearly visually demonstrate what is currently in scope for the release. (Remember to mark finished features in a way so that it is easy to distinguish them from those still in progress.)

What happens in almost all cases is that both the size of the red and green areas changes over time. What seemed essential at the beginning might turn out to be lower priorities, and what was considered release 2 functionality turns out to be essential. But if you keep updating the chart shown above, you should have a very clear picture of your current status and the best way

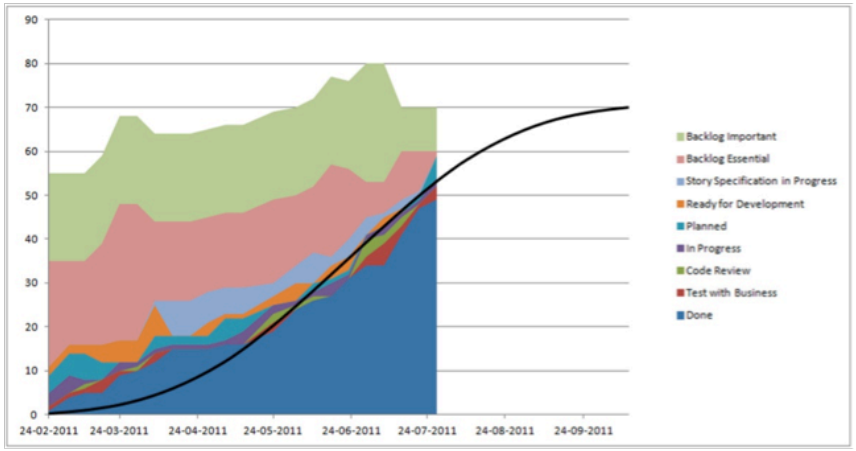

Figure 13 - Release Plan CFD Example

forward. What this means is that you should see the bar between the red and green areas and the top bar moving up and down as you get more data. Schneider Electric uses a similar approach for their release planning activities. Anders Jensen presented his results at the IDA Agile Conference in Aalborg, Denmark on April 30, 2012. He demonstrated that they had gone from missing almost all deadlines to being close to 100 percent on-time deliveries. Not that they knew exactly what would be delivered but products and solutions were launched at the agreed date. It does not have to be complicated to be effective!

Following these simple principles should make it possible for you to create a transparent and useful release plan and track your progress against it. The difficult thing that the above tools will not fix for you is to accept the data and use it to make better decisions going forward. Most people (myself included) have a tendency to come up with weird explanations when things do not go as planned, arguing that this is a special situation and things will improve from this point. Most often, special situations are not that special and you are much better off accepting reality rather continuing down an unrealistic path. As mentioned in a previous chapter, you should always keep in mind that if you wish to do so, velocity can very easily be gamed. If you want to "trick" yourself or others, it is not that difficult to make your velocity appear to be bigger than it is or to artificially reduce the amount of work left.

But again, why would you want to do that? Also, velocity just shows you a trend – it is not the ultimate truth. If you start with the risky stuff first (which is often a good idea), you will also experience a higher degree of velocity fluctuation at the beginning.

"The Real Cost of Change" – by Liz Keogh

"The Real Cost of Change" – by Liz Keogh

Question

Dear Miss Keogh, I work as a project manager and I am writing to you because I continuously find myself having the same discussion with the Scrum team. Some stories often take longer than estimated and it results in us not being able to meet the sprint commitment. This frustrates me since I take pride in meeting commitments and deadlines. I have tried, in vain, to push for the business to make a better analysis before we start coding or at least for us to put some basic change control in place. I would probably be able to convince our business people, but the development team is really pushing back.

Problem

Dear Mr., it is funny how we have a strange desire for control. Let me share a similar story with you to help you understand my position: I was in a planning meeting with my project manager and several of the devs.

"What happened?", the project manager said. "Why did this one story take so long?"

"There was some functionality we needed and did not know about," I replied. "We managed to get it in before the deadline, though." The business had been quite happy with that, and they were notoriously hard to please."

"If they are going to change their mind like this," our PM replied, "we are going to have to introduce some kind of change control."

"Please do not," I begged. "If you do that, the business will spend more time investing in getting things right to start with."

"Exactly."

"But they will still get it wrong. No amount of planning would have spotted that missing piece before we showed it to them! When they get it wrong now, though, they will encounter the change control and they will want to spend even more time getting it right first time. And they will still get it wrong, but now we will have made it more expensive for them to be wrong. We will have a formal process which means it takes even longer for us to find out what is missing, by which time us devs. will have to work to remember the code and the change will take longer. It will slow us down. So they will see that and spend even more time trying to get it right, and before you know it, they will be planning whole projects upfront."

We have a word for that. It is called Waterfall.

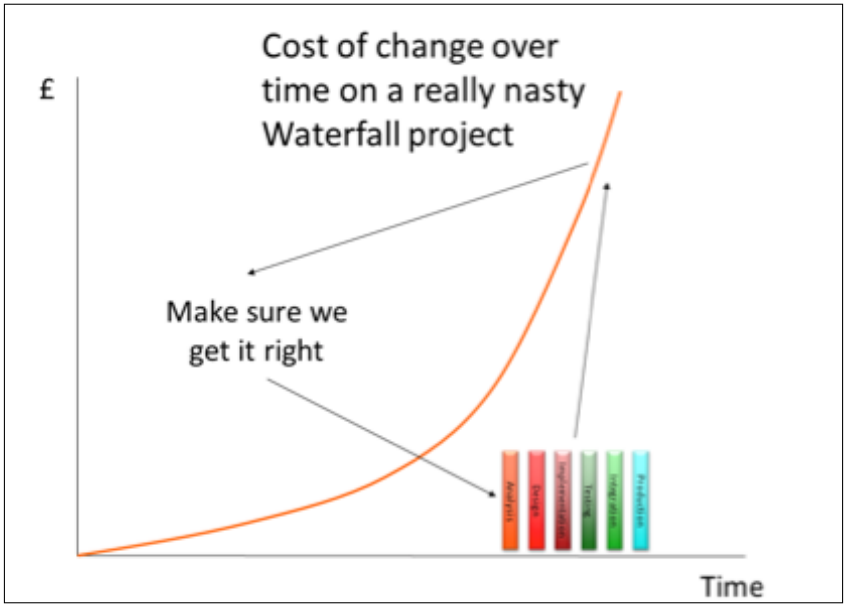

Figure 14 – Cost of Change On a Nasty Waterfall Project

So you see that desire to control change creates a reinforcing feedback loop, in which the cost of change makes us want to invest upfront, which makes the change expensive later on, which then makes us want to invest upfront, and so on. This is shown above in figure 14.

Cost of discovery vs. cost of change

In this case, it would have been pointless, except as a way of shifting blame and risk back to the analysts (and this was an internal project). The cost of change was quite low; we had clean code with a good suite of tests. It was only the cost of discovery, and the implementation that followed, that was really expensive. Do not confuse the cost of discovery with the cost of change.

Discovering something later on only costs more than planning for it if you have made a commitment to something else. In fact, if you do not plan for it, it can cost less. The newly discovered knowledge will be fresh in everyone's minds. Because the ideas have not been known for long, nobody is mentally committed to them either, which makes them easier to question and clarify.

This is even more important when there is a chance that the analysis might be wrong (and there is always that chance; we learnt this from Waterfall). If you plan for something that you later have to revert, you have introduced a cost of change right there. Perhaps the cost is just changing some documents. Perhaps the developers have designed for the plan, and built on top of the thing which needs changing.

Cost of change fluctuates on Agile projects too

But we need to do some planning, right? Otherwise the chances of us being wrong and building on top of the wrong thing are even bigger, and there is even more chance that we will have made commitments in the wrong way. Keeping the cost of change low is even more important than planning because there is always a chance that we will get something wrong. This is what we are aiming for: the ideal, low cost of change on an Agile project, as shown in figure 15.

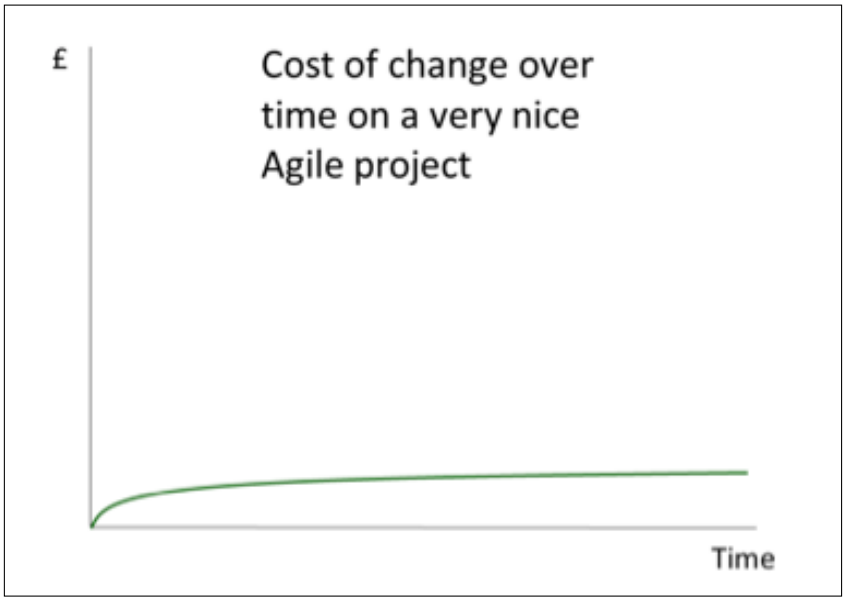

Figure 15 – Cost of Change On a Very Nice Agile Project

The first thing to notice about this is that the cost of change is not zero. That is what drives the team's desire for change control and starts kicking off that Waterfall loop. The second thing to notice is that this bears little resemblance to actual, real Agile projects.

On a real Agile project, it is likely that we have fluctuating levels of stress, concentration, experience and desire for feedback. All of these – or lack of them – will lead us to occasionally write code that is, shall we say, less than ideal. It takes discipline to write code that is easy to change. On a real Agile project, we tend not to do it all the time. Oh, we might say we do, but we do not – not always. And if we do, our teammates do not.

Toolbox

So how do we balance cost of change, discovery and feedback? I have got few guidelines that I would like to share in the following toolbox:

Clean up your mess

The real skill is not in writing clean code – it is in cleaning up the horrible mess we made the week before. And it takes even more discipline to do it afterwards, especially if you are under pressure to just hack the next working thing in too.

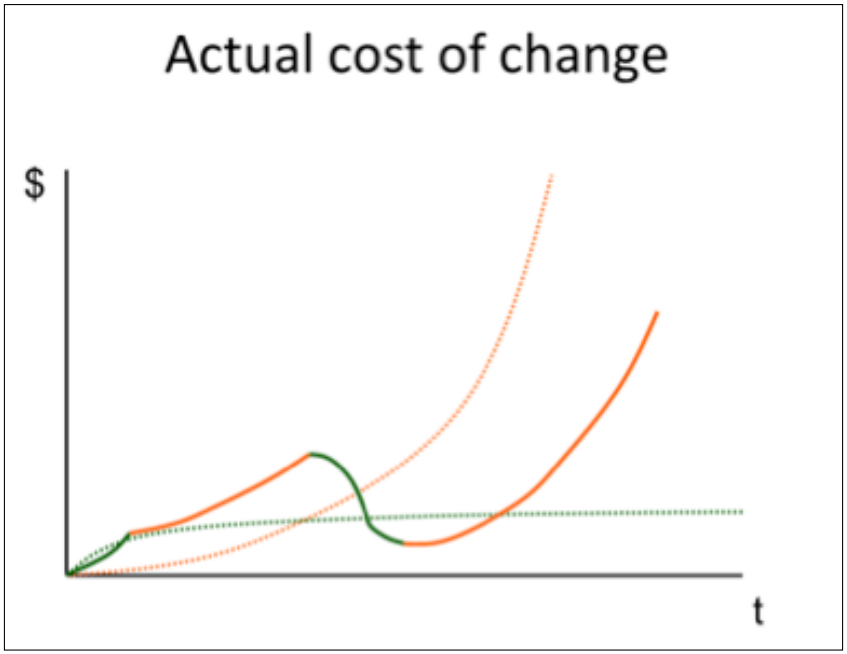

Figure 16 – Actual Cost of Change

If we do not clean up and keep that cost of change low, we are making a commitment to the wrong thing. The longer that commitment stays in place, the higher the cost of change will become. We will find ourselves on that Waterfall cost-of-change curve, and the longer we are on it, the more expensive it is. This aspect is shown above in figure 16.

I have found that the skill to clean up afterwards is even more important in high-learning projects, where a lot of the technology or domain is new, at least to the team. There is no point in writing tests upfront for a class when you do not even know what is possible, or what other frameworks and libraries will do for you, or what the business really wants. In those environments, rather than TDD or BDD, teams tend to Spike and Stabilize. The spikes are not really prototypes; they are small pieces of features, often with hard-coded data behind them, designed to get some feedback about technology or requirements. Dan North, who gave me the term, might write more about this later if we ask nicely, but for now, we can simply bear in mind that the skill to stabilize later – ensuring that the cost of change is lowered – is often more important than the skill to keep the cost of change low upfront, because we get it wrong too.

Assume you got it wrong

Human beings are hard-wired to try and get things right, and to pretend that they did even when they did not. I love this list of cognitive biases on Wikipedia. These are just some of the ways in which we get it wrong and do not even notice.

If we assume that we got it wrong, we then start to look for feedback, and quickly. This is difficult for most human beings. We much prefer to get validation than feedback – to be told that we did it right, rather than finding out what we did wrong. Our brain gives us these little kicks of nice chemicals when we learn that we did something right, and it feels much better than the other kind.

If we can remember, though, that we probably got it wrong, our focus will change. Instead of trying to invest in good requirements and nice code, we will try to find out what we got wrong in the things we have already done. Of course, we need to invest in stabilizing what we have done, or the cost of change goes up, and that will make it more expensive later if we find out we were wrong, which is the assumption we were trying to make in the first place… ah, the paradox!

There is a fine balance to be struck between getting quick feedback – itself often being an expensive proposition, given the business of most domain experts – and getting it right upfront. So where does the balance lie?

Do not sweat the petty stuff

If it is easy to change, do not worry about it. Analysts learn what is easy to change. Typos, colors, fonts, labels, sizes, placement on the screen, tab order, an extra field… these are all usually easy to change and do not normally need to be specified upfront. Even if you have a particular style that you want to see on a page or form, this can usually be abstracted out and changed later – just let the devs. know that you want that consistency at some point.

It is more important for us to know the rough shape of what you want to see, and the flow and process of that information. We do not want to know every field on an order item. We just want to know that it needs to be sent to the warehouse and stored locally because you are going to check the money in the till and count the stock. The fine detail of that is pretty easy to change, so we can get feedback on it later. Getting the fine detail right would definitely be an investment, and we might have got the big picture wrong.

Deliberately discover things you have never done before

Dan North wrote an excellent post on Deliberate Discovery and I have been using it to manage risk on my projects for a while now. It is one of the most important tools in my toolbox, along with Real Options to which it is strongly related, so I want to cover how I use it here.

I really like using Cynefin to help me work out what to target for discovery. We treat a lot of software as if it is complex, and we talk about self-organizing teams and high-learning environments, but in reality, there are huge chunks in most applications which follow well-established rules, have been done a thousand times before and probably have libraries or third-party apps to do the job for you. They are not complex. They are complicated.

They require the application of a bit of expertise and are likely to be done right and never changed again. User registration and logging in are great examples of this. You do not need a big, thick document to describe them. The fine details might change, of course, but we already know not to sweat the petty stuff.
It is okay to plan some aspects of a system as if it is Waterfall, for instance, deciding upfront whether you want to use your own login or let Google authenticate. Even better than requirement documents, and quicker, is to say: “It is user registration. Make it work like Twitter’s, but we also need the user’s loyalty card number. We should offer to send them a card if they do not have one.” Dan North calls this pattern “Ginger Cake” – it is like a chocolate cake, but with ginger. He even cuts and pastes code. And it is OK! Honestly, it is! This code is also absolutely prime for TDDing – if you actually have to write it yourself, that is, since it has been done before, so someone has probably written something to do it for you already. You can also give this code to junior devs., for whom it is new, and guide them in TDDing, making it perfect pair-programming territory. Everything you have ever been told about Agile software development applies, particularly in this place.

Fortunately, most applications have a minimum set of requirements that they share with other, similar applications. David Anderson calls these commodities – table stakes that you have to have just to play the game – so *most* code in an applications will end up going this way.

The places in which we are most likely to get it wrong, and need fast feedback, are places where we are doing something new. They might be technological, particular to a domain, or just things that the team itself has never looked at. My favorite book for understanding risk is “Waltzing with Bears”(DeMarco & Lister, 2003), which starts the first chapter with: “If a project has no risks, do not do it.” It is these new, risky aspects of the project that differentiate it from others and make it valuable in the first place!

For new and risky aspects of a project, the best thing to do is assume you got it wrong and work out how quickly you can get feedback on how wrong you are.

Accept that any new or unknown aspect of a project will need to be changed

I was chatting to one of our analysts. “I can see this feature is in analysis at the moment,” I said. “Does that mean it is the next thing we want the developers to do?”
“Oh, no,” the analyst said. “It is only there because the analysis is quite complex. It is all new stuff, so we have to be careful with it and it is taking a bit of time. Once we get the analysis done, the development should be very easy, so we will do it later.”

“Oh, the development will be easy, I am sure… but would you not like to find out what you did wrong now, rather than later, while it is still fresh in your mind?”

The analyst smiled. The company was very much more used to Waterfall, and the idea that it was OK to get it wrong was something very new.
It is okay to get it wrong, as long as you get feedback quickly, while the cost of change is still low. By working out which parts of a project are unknown or new, and targeting those first, we make small investments while the cost of change is still low. Keep your options open – do the risky stuff first and keep tech debt low.

Think in terms of Real Options

Anyone who has run into me at conferences will know how much I love Real Options, and it is really at the heart of the cost of change. The only reason change costs more is because of the commitment that we already made. Chris Matts describes technical debt as an “unhedged call option”. While Chris came up with the metaphor, it was Steve Freeman who described it. He says: “You give someone the right to buy Chocolate Santas from you at 30 cents each. That is fine, as long as the price of chocolate stays low. As soon as it goes up, you still have to pay to make the Santas, and now you are in trouble and your company is going bust, because you did not give yourself the option to get the chocolate somewhere else.”

Similarly, technical debt is absolutely fine until we are called on to act, and act fast. At that point, we are in trouble. This is the biggest reason for keeping the cost of change low – because it gives us the option for change, later. It is a frequently cited reason for replacing legacy projects – and is, bizarrely, often forgotten when the pressure mounts and the business wants their replacement app.

Do the risky parts first

This is not helped by common practices of estimation and the associated promises which often lead to that pressure building up in the first place. Rather than making these promises upfront, why not try the risky bits first? I often hard-code data so that I can get feedback on a new UI early, or I spike something out using a new library or framework, or connect to that third-party application just to see what the API is really like to use, or have a chat with the team writing that other system we are going to need in June, so I can find out how communicative and receptive to feedback they are. Doing this means that we give ourselves the most time to recover from any unexpected discoveries, and we can worry about the more predictable aspects of a system later.

Once we have got spikes out of the way, adding tests to act as documentation and examples for any legacy code we have created, cleaning it up so it is self-commenting, ensuring that architectural and modular components are properly decoupled, etc., all help us to stabilize the code. At the same time, the effort involved in creating stable code is an investment itself. If there is a good chance that the code might be wrong, it could be worth getting feedback on that – knocking up integration tests, showing it to the business, testing it, and getting it live – before it is made stable.
That way, the commitment made is small and the cost of change is low. Just remember to clean up and keep it that way!

“Retrospective Actions” – by Diana Larsen

“Retrospective Actions” – by Diana Larsen

Question

Dear Ms. Larsen, I am writing to you because we are having difficulties getting the full benefit out of our retrospectives. We have no trouble coming up with problems or discussing possible solutions, but we rarely seem to turn it into actionable decisions and when we do, they are not carried through. It is getting quite frustrating and people are starting to question the value of doing retrospectives altogether. How do we turn the situation around?

Problem

Dear Mr., thank you for that very relevant question. It is quite a common problem and I will try to outline some of the possible reasons as well as solutions to address the problem.

Sometimes teams get stuck at the point of “deciding what to do” in retrospectives. Team members may begin to point fingers and describe things that the ubiquitous “they” must do before the team can move forward or make improvements. This may lead to a team-as-victim, “poor us, we are stuck” syndrome, or blame and finger pointing. “It is their fault we are in this mess!” Blame kills retrospectives and the perception of persecution stalls any hope of forward motion, so the retrospective leader has to shift this conversation, and fast! Team members may also perceive so much room for improvement that they become paralyzed and cannot decide where to start improving their lot.

When there are no improvement actions, if I ask further, I generally discover that the team has:

a) very short retrospectives (20-45 minutes) with not enough time to learn and think together, let alone make joint decisions

b) spent no time setting the stage or gathering shared data, but has jumped right to creating lists of personal (i.e., not shared) opinions about WW/DD (went well, do differently next time)

c) thought that making lists of blockers will magically cause things to change

d) spent no time actually choosing actions collaboratively that everyone buys into

All beg for the Dr. Phil question: “How is that working for you?” The most the team can expect is an opportunity to identify repeating patterns of impediments, if they keep the lists and review them over time.

But as your question reveals that that is not the only problem, often when I ask about a team’s challenges with retrospectives, a recurring theme comes up: Acting on Actions.

I hear, “Our team does not follow through on our plans for action”.
When the team does decide on actions, and then does not implement them, there is a different problem. Odds are the team does not regard their improvement actions as real work. This gets expressed in a couple of ways:

The team maintains an improvement list in addition to their product and iteration backlogs.
Team members assume that someone on the team (or everyone on the team) will remember to do the improvement, in addition to their regular work. It might happen… but usually does not.

Toolbox

If your team’s retrospectives do not result in continuously improving your process, practices, teamwork, or methods, it is a waste of everyone’s time. Try one of these techniques, and let me know how it goes.

Use the Flexible Framework for Agile Retrospectives

In general, I recommend using the Flexible Framework for Agile Retrospectives to design their next retrospective, and see how that works.

Flexible FrameWork for Agile Retrospectives (from Agile Retrospectives)

/ Set the Stage

/ Gather Data

/ Generate Insights

/ Decide What to Do

/ Close the Retrospective

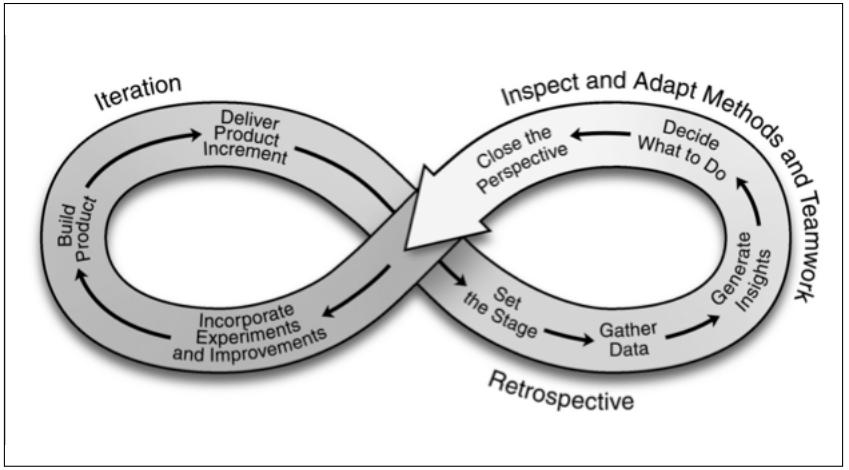

Figure 17 - Flexible FrameWork for Agile Retrospectives Illustrated

A great focus for the next retrospective becomes “getting the most from our retrospectives and improving our improvement actions”.

Use four steps to follow up on actions

To avoid the situation where actions are not implemented, I suggest using four small steps after deciding on the action the team wants:

1. Find out who on the team wants to commit to shepherding the action… and a backup person to work with them.

2. Write the action on a card just like all the other stories (if it is an action that will take multiple steps) or tasks (if it is a single-step action).

3. Estimate the amount of the team's effort it will take to implement the action.

4. Carry the card into the iteration planning meeting following the retrospective and include it in the plan.

Use Circles and Soup

When victim-talk, blaming or overwhelm surfaces, I reach into my retrospective leader's toolbox and pull out a technique to help teams identify the kinds of action that the team can take. I draw three big concentric circles on a whiteboard or flipchart page, making the middle one as big as I can and leaving space wide enough for sticky notes in between each ring.

Label the middle circle "team controls", label the next ring "team influences", and the third "The Soup". I borrowed the useful concept of "The Soup" from David Schmalz, meaning all those elements of organizational climate & culture, policies & procedures, and other systemic factors in organizations that have become so embedded in "the way we do things around here" that the team has little hope of shifting without considerable help and political will on someone's part.

When I have prepared the chart, I explain that everything that affects the team falls into one of these categories, and every action they take also falls into one of three categories:

In the middle, they have control so that they can take direct action.

In the next ring, they have influence, so their action would be a persuasive, influencing or recommending action.

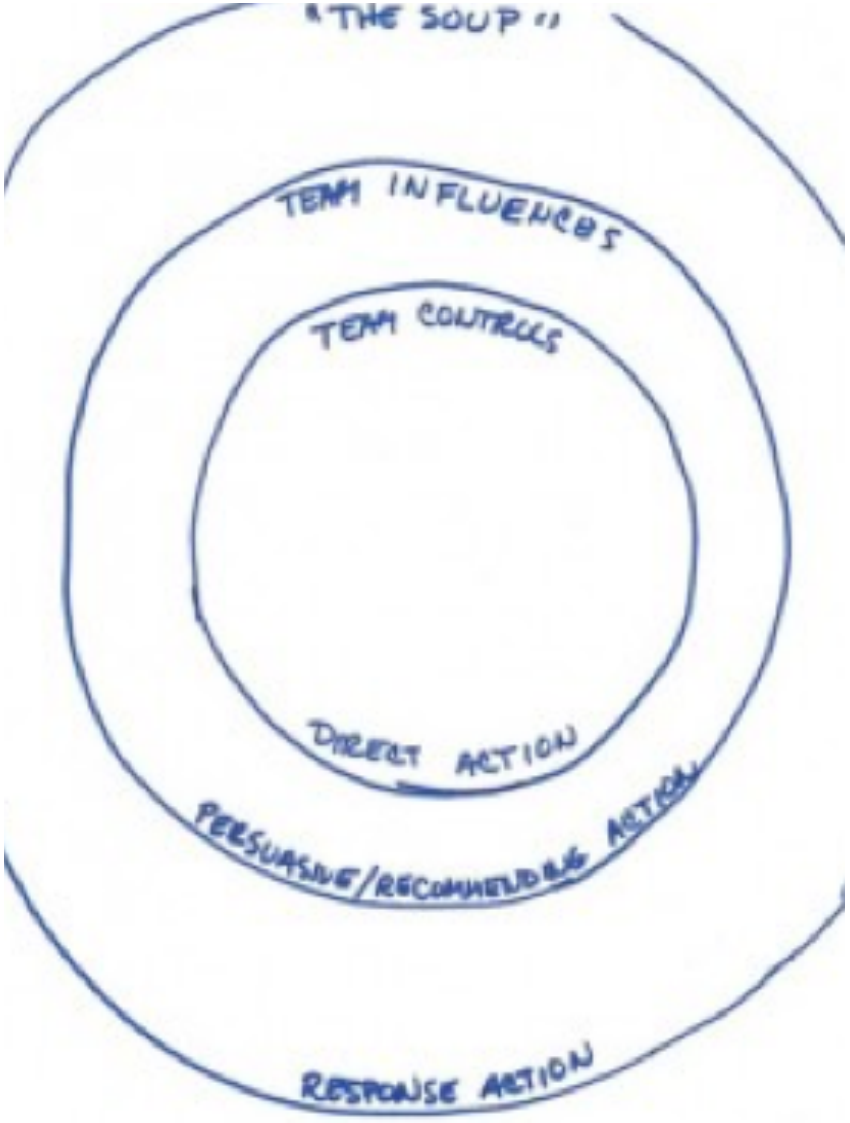

Figure 18 – Circles and Soup

In the last ring, they may have no control or influence, but they can choose actions for how to respond collectively when they find themselves swimming in the soup.

I share a couple of illustrating stories for this one.

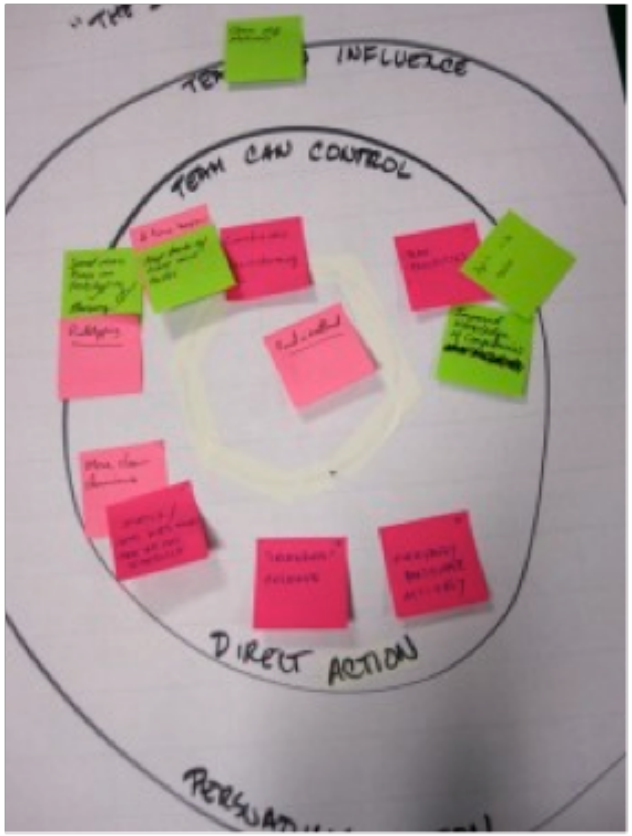

Figure 19 – Circles and Soup Example 1

I invite team members, in pairs, to write the issues and challenges that the team faces on sticky notes – one per note, of course. When they have finished, pairs bring their notes to the chart and stick each note in the appropriate ring. As a next step, the whole team takes a step back to look at the completed chart and identifies what kinds of actions they can take for each note: direct, influencing, response. This activity naturally leads to planning specific actions that will have the greatest beneficial impact. In early retrospectives, it is more effective when the team takes on direct actions to remove impediments or create improvements within its control. When the team has experienced success with direct action, it becomes easier to develop plans for influencing actions for impediment removal or choosing actions in response to systemic constraints.

Figure 19 – Circles and Soup Example 2

The Circles of Control & Influence activity helps the team sidestep the "someone should", "if only they would", "only those guys can", and "we are doomed!" conversations (which generally go nowhere), and shows individual team members that they have more scope for action if they act as a team. This technique was adapted from Stephen Covey's book "Seven Habits of Highly Effective People" (Covey, 1999), also described by Jim Bullock as "Circle of Influence and Concern".

Good luck on your journey

Good luck on your journey

I hope these past chapters have provided useful insights and inspired you to see things from more than just one perspective. If you think your company could benefit from Scrum 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. You can find us here:

Trifork A/S:
Training: training@trifork.com
Website: www.trifork.com
Twitter: @TriforkNews
Blog: http://softwarepilots.trifork.com/

Phone: +45 8732 8787

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

I would love to get your feedback from reading this; all suggestions for a possible second edition are very welcome. You can contact me through the e-mail address above or via twitter: @J_Boeg.

Cheers
Jesper

Bibliography

Bibliography

(VersionOne, 2011) http://www.versionone.com/state_of_agile_development_survey/11/

(Cohn, 2010) Succeeding with Agile Software Development Using Scrum. Mike Cohn. Addison-Wesley, 2010.

(Cohn, 2004) User Stories Applied. Mike Cohn. Addison-Wesley, 2004.
(Reinertsen, 2009) Principles of Product Development Flow. Donald Reinertsen. Celeritas Publishing, 2009.

(Pichler, 2010) Agile Product Management with Scrum. Roman Pichler. Addison-Wesley Professional, 2010.

(Reinertsen Interview, 2012) Interview with Don Reinertsen at the GOTO Copenhagen 2012 Conference. http://www.youtube.com/watch?v=G-NbrI-SyfvM

(Cohn, 2005) Agile Estimation and Planning. Mike Cohn, Addison-Wesley, 2005.

(Mike Cohn's Blog, 2007) http://www.mountaingoatsoftware.com/blog/why-i-dont-use-story-points-for-sprint-planning

(Vasco Duarte's Blog, 2012) http://softwaredevelopmenttoday.blogspot.de/2012/01/story-points-considered-harmful-or-why.html

(Britton, 2008) Derek Britton. http://www.pragmaticmarketing.com/resources/lessons-from-the-eye-of-the-storm-part-2-the-genetics-of-successful-scrumming

(Kronfält, 2011) Overcommitment is Better Than Undercommitment. Richard Kronfält. http://scrumftw.blogspot.dk/2011/02/overcommitment-is-better-than.html

(Covey, 1999) The 7 Habits of Highly Effective People. Steve R. Covey. Simon & Schuster, Revised Edition, 1999.

(Rother, 2009) Toyota Kata. Mike Rother. McGraw-Hill, 2009.

(Boeg, 2011) Priming Kanban. Jesper Boeg. InfoQ, 2011. http://www.infoq.com/minibooks/priming-kanban-jesper-boeg

(Yuval Yeret's Blog, 2011) Scrum Sprint Commitment Rant. Yuval Yeret. http://yuvalyeret.com/2011/10/13/scrum-sprint-commitment-rant/
(Schwaber & Sutherland, 2011) The Scrum Guide. Ken Schwaber and Jeff Sutherland, 2011. http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf

(Appelo, 2012) How to Change the World. Jurgen Appelo. Jojo Ventures BV, 2012.

(Appelo, 2010) Management 3.0. Jurgen Appelo. Addison-Wesley Professional, 2010.

(Kotter, 1996) Leading Change. John P. Kotter. Harvard Business Review Press, 1996.

(Kotter, 2008) A Sense of Urgency. John P. Kotter. Harvard Business Review Press, 2008.

(Hiatt, 2006) ADKAR: A Model for Change in Business, Government and our Community: How to Implement Successful Change in our Personal Lives and Professional Careers. Jeff Hiatt. Prosci Research, 2006.

(Larsen & Shore, 2012) Your Path through Agile Fluency. Diana Larsen & James Shore. http://martinfowler.com/articles/agileFluency.html
(Denning, 2011) Six Common Mistakes That Salesforce.com Didn't Make. Steve Denning. http://www.forbes.com/sites/stevedenning/2011/04/18/six-common-mistakes-that-salesforce-com-didnt-make/

(Womack & Jones, 1996) Lean Thinking. James P. Womack and Daniel T. Jones. Free Press, 2003.

(Benefield, 2008) Rolling out Agile in a Large Enterprise. Gabrielle Benefield. Hawaii International Conference on System Sciences. Vol. 0 (2008), 462, doi:10.1109/HICSS.2008.382

(Rico et al., 2009) The Business Value of Agile Software Methods. David F. Rico, Hasan H. Sayani, Saya Sone. J. Ross Publishing, 2009.

(Anderson, 2010) Kanban. David J. Anderson. Blue Hole Press, 2010.
(Anderson, 2012) Lessons in Agile Management: On the Road to Kanban. David J. Anderson. Blue Hole Press, 2012.

(Ladas, 2009) Scrumban - Essays on Kanban Systems for Lean Software Development. Corey Ladas. Modus Cooperandi Press, 2009.

(Richard Durnall's Blog, 2010) Agile Adoption Patterns. Richard Durnall. http://www.richarddurnall.com/?p=57

(Adkins, 2010) Coaching Agile Teams: A Companion for Scrum Masters, Agile Coaches, and Project Managers in Transition. Lyssa Adkins. Addison-Wesley Professional, 2010.

(Yuval Yeret's Blog, 2012). Who would have thought Personal Kanban would end up being the counter-measure to stalled Kaizen / Continuous Improvement? http://yuvalyeret.com/2012/07/31/who-would-have-thought-personal-kanban-would-end-up-being-the-counter-measure-to-stalled-kaizen-continuous-improvement/

(Shpilberg et al., 2007) "Avoiding the Alignment Trap in Information Technology". Shpilberg, D., S. Berez, R. Puryear & S. Shah. MIT Sloan Management Review, Vol. 49, No. 1, 2007.

(Liker, 2003) The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer. Jeffrey Liker. McGraw-Hill, 2003.
(Galorath, 2012) http://www.galorath.com/index.php/software_maintenance_cost

(Goldratt, 2005) The Goal. Eliyahu M. Goldratt. Productivity & Quality Publishing Pvt Ltd, 2005.

(Schwaber & Beedle, 2001) Agile Software Development with Scrum. Ken Schwaber & Mike Beedle. Prentice Hall, 2001.

(Humble & Farley, 2010) Continuous Delivery. Jez Humble & David Farley. Addison-Wesley Professional, 2010.

(Olga Kouzina's Blog, 2009) Do You Really Need a Deadline? Olga Kouzina. http://www.targetprocess.com/blog/2009/06/do-you-really-need-a-deadline.html

(Brooks, 1995) The Mythical Man-Month: Essays on Software Engineering. Frederick P. Brooks. Addison-Wesley Professional, 1995.

(Jeff Patton's Blog, 2008) The new user story backlog is a map. Jeff Patton. http://www.agileproductdesign.com/blog/the_new_backlog.html.
(Trifork Agile Blog, 2012) Exercise Introducing Story Maps. Jesper Boeg. http://triforkagile.blogspot.dk/2012/02/exercise-introducing-story-maps.html

(Ries, 2011) The Lean Startup. Eric Ries. Crown Business, 2011.

(DeMarco & Lister, 2003) Waltzing With Bears: Managing Risk on Software Projects. Tom DeMarco & Timothy Lister. Dorset House; illustrated edition, 2003.

Topic Candidates That Did Not Make It

Here is a list of the topics that I thought about but decided not to include in the mini-book you have just read. If you have suggestions for other topics or would like to see one or more of the following topics included in "Real Life Scrum 2", let me know.

1. Models for handling fixed-price/fixed-scope contracts
2. Including testers in the Agile team
3. Are we using the right scrum tool?
4. Kick-starting individual teams
5. Unaligned strategic focus with Agile
6. Distributed outsourced Scrum
7. Unrealistically long backlogs
8. Silo optimization
9. Self-organization vs. self-steering
10. I cannot convince management that we should do scrum
11. Scrum only on the team level
12. Multiteam Scrum Master vs. working Scrum Master
13. Silo optimization in Scrum
14. Why break things down into tasks (or why not)
15. Ignoring technical skills and the value of good people
16. Teams finishing everything in the 11th hour of the sprint

Experience matters too. In this book, Jesper Boeg, an experienced developer, scrum-master, and Agile coach, shares his collection of the questions he hears most often from frustrated Scrum users attempting to make it all work. He offers answers grounded in years of encountering, and resolving, the typical lumps and bumps. Jesper's pragmatic, method agnostic approach has served many teams well.

From the foreword by Diana Larsen

In this mini-book, Jesper Boeg answers questions on common problems encountered with Scrum, and offers multiple suggestions based on real, practical solutions that have worked in many of these difficult environments. By placing each suggestion in context, and helping us reflect on where our problems really come from, he invites us not only to improve on what we're doing already, but also to appreciate how many different ways there are to do it.
If you're also looking out for the next change that might help, you could do worse than start here.

From the foreword by Liz Keogh

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 worldwide 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 programmes, covering everything from low-level Agile development practices to large-scale Agile and Lean change initiatives. So, 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: training@trifork.com
Twitter: @TriforkNews
WebSite: www.trifork.com
Blog: http://softwarepilots.trifork.com/
Phone: +45 8732 8787

InfoQueue
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

software pilots TRIFORK.