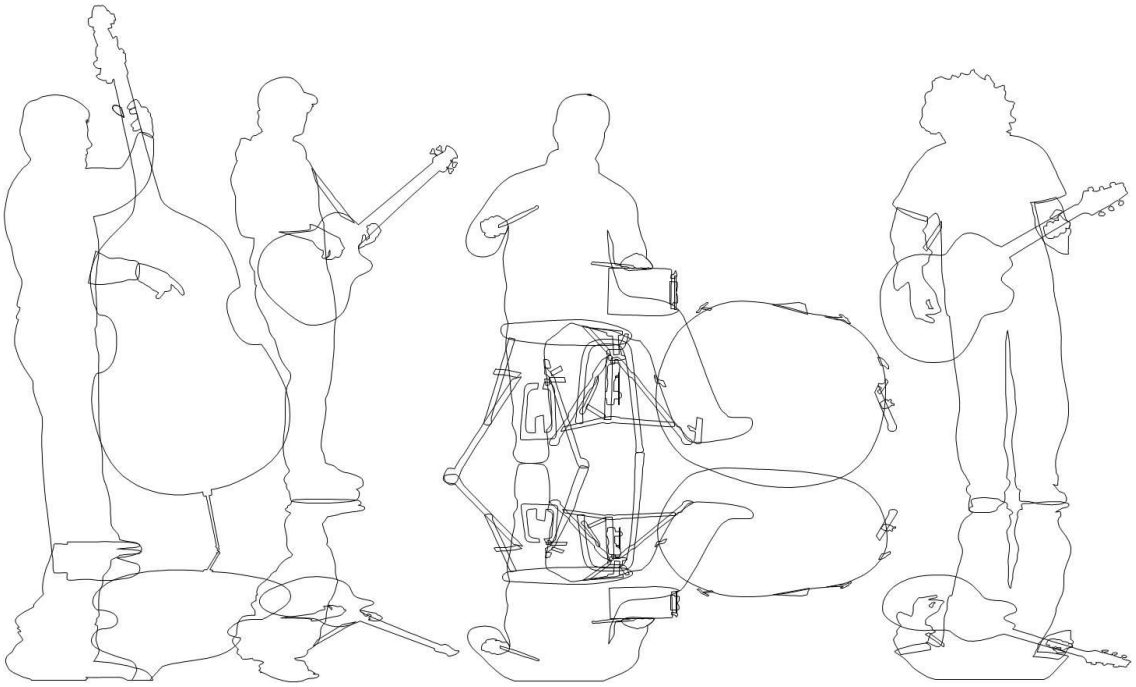
A Brief Introduction to Agile Development

Using Lean and Scrum



Mikael Lundgren

Introduction

Lean and Agile has become buzzwords today (2011) as many organizations are inspired by the success of companies such as Toyota and Scania, and how their product development is both different and efficient. These frameworks for organization and development grew out of collaboration between Japanese and U.S. industries after WWII, (1) among others Ken Schwaber and Jeff Sutherland to formulate the framework *Scrum* for software development in the early 1990's.

There are a number of obvious connections between Toyota's product development system (named *Lean* by MIT) and *agile* principles of work. Where Lean expresses principles and practices for workflow, leadership and organization, agile framework such as *Scrum, XP, and DSDM etc.* depicts various ways to implement several Lean practices in the software industry. Common to all agile frameworks is the concept of working in an **iterative, incremental and time-boxed manner.**

- Iterative – work is conducted in a cyclic manner, where the way to work is often reflected upon and changes are made to work even better
- Incremental – scalable and modular solutions are implemented, enabling teams to regularly release a product that is **ready but not complete.**
- Time-boxed – meaning that deadlines are never moved, but scope and solutions are constantly adjusted to meet the deadlines.

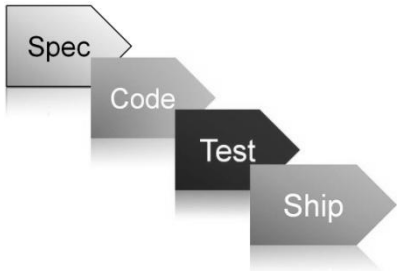
The term *Scrum* comes from the world of rugby, where *a scrum* is a gathering of team members in the field to solve a problem (sometimes earning a bruise or two!)

Scrum is based on cycles of **inspection and adaptation,** a characteristic of *empirical* methods, contrasting to *defined* methods which assume that results can be predicted with high accuracy given enough knowledge of the input conditions and rules of the system. Scrum is inspired by the theory of CAS (complex adaptive systems (2))

One of the problems facing the use of empirical or adaptive methods is that we as software engineers tend to search for defined solutions to each problem – even when it comes to the workflow itself!

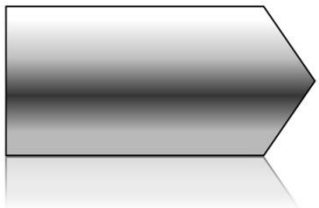
Perhaps we can predict the future if we just spend even more time analyzing the problem before writing the code?

Rewarding early decisions and an overly high belief in the possibilities to use engineering disciplines to predict the future is leading to many development projects being late and costly today.



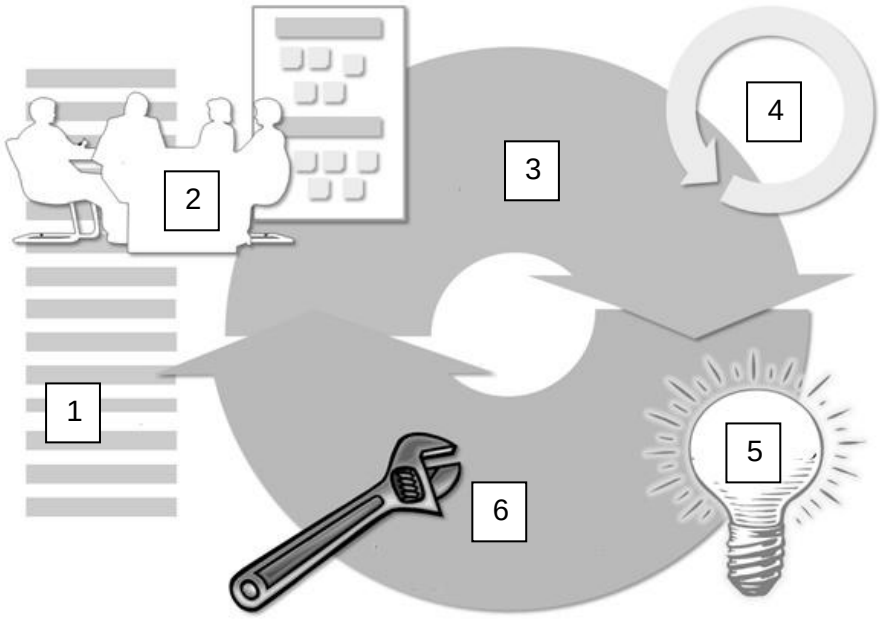
A phase-gate method of working assumes that work in various degree of being finished can be passed on as a baton between people and departments, without losing knowledge or information in the process. That is rarely true, leading to a lot of work needed in every transition.

Using an agile framework, small slices of usable functionality is made, trying to constantly build something that is *ready but not complete.* This allows us to work in short cycles that contain all aspects of a finished bit of functionality.



A closer look at Scrum

This is how I usually depict the framework cycle of Scrum:



This image contains all mandatory parts of the framework:

1. **Prioritization** is expressed through ranking high-level requirements in a list called *a backlog*. The requirements are initially sorted by business value for the user. This is the foundation to planning the development work.
2. **Planning** is performed by the development teams, who are also responsible to follow up on the results compared to the plan. The goals for each sprint is defined when the sprint is started.
3. **Development** is conducted in short, time-boxed iterations called *sprints* which work towards fixed goals and priorities throughout the entire sprint. It is common to start with a sprint of 30 days and adjust the length later.
4. **Synchronization** of the work within and between development teams is performed briefly every day, and the status and prognosis for the results are updated.
5. **Release** is made at the end of each sprint, of a product that is *ready but not complete*. This can be for instance a software system with new and/or improved functions that are fully usable.
6. **Review** of the sprint results and the work routines is made by all involved in the work, and the methods are adapted to increase the possibility for the team(s) to release on time, quality and scope.

The most visible thing when collaborating with an agile team would be the sprints – that the team is working focused towards fixed goals for the sprint. The direction of the development work can be changed – but only at the start of each sprint. Also, status and progress are always available and visible.

It can seem almost as a paradox that development speed tends to increase if short-term goals are established and not changed, instead of having the flexibility to change requirements and priorities every day. But the positive effects are often apparent – increased focus, commitment, less context-switching and less energy spent on worrying over possible changes in requirements. By adding a formal way to deal with change in direction at the onset of each sprint, a slight delay is built into the system, meaning that

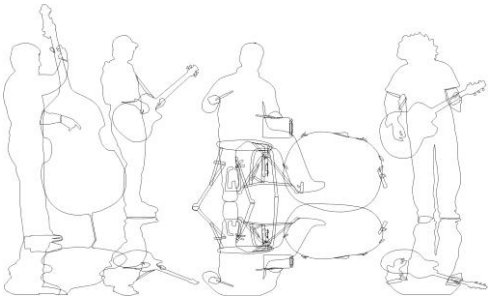
altering the direction is always thought through. In the end, the sprint length will be decided from various factors, including the trade-off between focus and change.

By combining the Scrum cycle with Edwards Deming's PDCA improvement cycle (3) we can see how they match. At first, a shorter work effort (sprint) is **planned** *(Plan)*, which is then immediately **done** *(Do)***.** During implementation, progress and results are **checked** *(Check)* and as the final step changes to the work procedures are **acted** *(Act)* upon to improve the results from the coming sprint. By doing it this way, improvement becomes a continuous part of development work.

Roles of Scrum

Scrum only defines three roles: Developer, Product Owner, and Scrum Master

Developer (or team member)



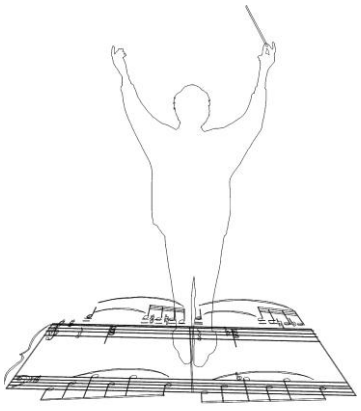
As the member of a Scrum team, you are not only responsible for developing a product, but also to develop your way of working. That means that it is required of a team to improve their work methods over time, and leaders help them to fulfill these obligations when needed. Just as the members of a band, the team play the same song, to the same beat. Even if everyone is a specialist at their instrument, it's the sum of the parts that make up the music.

Product Owner

The Product Owner is responsible for ensuring that the investments in development work are earned back, through prioritizing business value and taking account for cost and complexity of various development solutions. The Product Owners is "singing the song", bringing the vision of the product and functionality to all involved. It is an important part of the Product Owner role to be the *sole decision maker*. This means that the Product Owner has to be available for the development teams to answer questions and make decisions throughout the product development.

Scrum Master

The Scrum Master role is rarely found in organizations today. It is a coaching leader *without formal authority over personnel.* This means that the Scrum Master coaches one or several teams to keep the pace, and to use Scrum to full extent to master the work flow. It is common for the Scrum Master to become the coach to several roles in the organization through his/hers deep knowledge of Scrum and agile development.



Sustainable Development

Everyone developing the product can call themselves *developer*. Not only the programmer, but also test specialists, graphical designers, tech writers, etc. In short, we use a common title for everyone bringing the needed competence and skills to build finished functionality into the product.

This does *not* mean that everyone needs to be able to do everything. In many organizations, experts and specialists are needed to be competitive. But agile development requires them to collaborate in *cross-functional teams* to be able to build finished functionality in short cycles.

As specialists tend to become bottlenecks in development, we try to avoid this scenario by either ensure enough specialists, or slow down the development flow accordingly. To some extent, this is self-regulating as every member of a Scrum team is responsible for their own calendar and availability when planning upcoming sprints. It often becomes apparent when development is slowed by a few specialists, meaning that strategic action is needed.

It is usually necessary to work in cross-functional teams where all competence and skill needed to build finished functionality is represented. The opposite situation, using functional teams and/or individuals handing partly finished components (partially written untested code, etc.) between each other demands an excessive amount of communication and information at each handover. Misunderstandings or loss of information is common, leading to potential problems downstream. As problems are unveiled later in the process, teams upstream are disturbed from their focus to fix requirement errors or bugs. It is not uncommon to end up in a state of being ”80% finished” which is frustrating and hard to predict the outcome from.

Two common (and problematic) handovers are present in many organizations:

- The handover of requirements and expectations from users/requirement specialists to the developers
- The handover from development to test

Cross-functional development teams instead build functionality with tangible business value available often. This means for instance that test specialists no longer work as an “inspection team” near the end of the development chain, bur is a part of requirement work, design and development to ensure that additions to the system is built in a testable manner, producing quality that is built in from the start.

Whenever a final “system test” or verification is called for, it is a pure formality in agile frameworks, and should not unearth strange bugs or design flaws. By keeping development cycles short with focus on quality, time is saved in less post-development work, also making it easier to predict upcoming sprints and development velocity.

Becoming Done

Becoming ”done” or ”finished” with something is so important it deserves a paragraph of its own. The concept of ”done” can seem a bit treacherous when different parts of development (and teams) have a different notion of what it means to be “done”. It quickly becomes hard to keep costs and time if parts of the development starts to think of “done” meaning “code is written but not yet tested”. This calls for a change in the way we view being “done”, and a good first step is to agree on *Done Criteria*.

Done Criteria lists all things that need to be finished before a task can be called ”done”, and before we are allowed to move on to the next task. Here is a short list of Done Criteria that can be used as a starting point:

Implemented
Delivered (checked in, installed, etc.)
Accepted (solution has been accepted by users/product owner)
Documented (as needed by us and the users)
Tested

To some development teams this can be too ambitious to start with, and my suggestion is then to establish the existing Done Criteria (“what do we mean when we say *done* today?”) and list them, together with a mid-to-long term plan to extend the Done Criteria to become more complete (if work with the infrastructure for development is needed, it is usually planned in upcoming sprints)

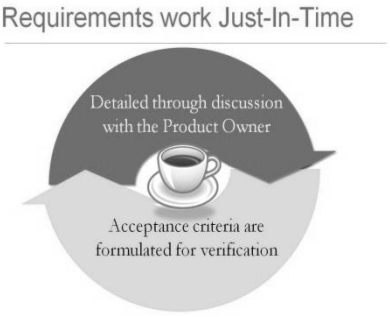
Agile Requirements

Agile requirements work differs from traditional requirements processes in one important point – that requirements work is an integrated part of development. When being faced with a large development effort, some requirements work is usually made upfront to stock the product backlog, but it is expected of development teams to sort out, detail and implement requirements in close collaboration with the Product Owner and other domain experts *as the requirement is in turn to be developed.*

Instead of allocating equal effort to detail all requirements before a project is started, it is assumed that part of the requirement backlog will be changed or re-prioritized as work progresses, and new requirements may be added. Hence, most time and effort is spent in detailing requirements near the top of the backlog (highest in priority) and also closest to being developed. This eventually leads to the backlog assuming the shape of a pyramid – or an *iceberg* to quote Mike Cohn (4) with fine detail near the top and larger requirement chunks near the bottom. As development progresses, the top of the iceberg is implemented (according to the Done Criteria) and lower requirements on the backlog are detailed to a finer degree, also leading to a continuous discussion on priority of requirements, both from a business- and a risk perspective.

From my experience, *User Stories* fits the agile development work quite well, as it helps keeping the user in focus even when detailing requirements on a technical level. User Stories have been described in detail by Mike Cohn (4).

Requirements residing on the product backlog are owned by the Product Owner. As teams commit to backlog items as a part of the sprint planning, requirement ownership is transferred to the team. With requirement ownership comes the responsibility to detail and implement it, but also the right to help from the Product Owner. Often, requirements are detailed through teams presenting several solution suggestions (over a cup of coffee?) to the Product Owner who decides on an alternative, weighing business value and development velocity into the equation.



All priorities are made by the Product Owner.

From Requirements to a Plan

A common misconception about agile frameworks is that work is not planned (at least not longer than the next sprint) and that you "get what you get" at the end of the sprint. That is not necessarily true, as Scrum offers a transparent insight and good control over the development work as the development teams are expected to follow up the current plan and results every day. Scrum focuses on *remaining time* and the development teams are constantly working towards the upcoming goals for the sprint, and the entire project or release by releasing *ready but not complete* product increments in short cycles, updating the plan and prognoses on the upcoming work. For this to work, the organization need to act on deviations from the plan through priority changes, changes in content or solutions, and in external communication, instead of trying to adapt reality to the plan, which is a much more daunting prospect.

Scrum works with adapting the solutions to meet target dates. This means that if a prognosis shows that more time is needed, the Product Owner takes actions together with the development teams to mold the contents to maintain the time-box. Part of the daily work is to estimate the remaining needed time, which is a way to get early warnings on deviations and act accordingly.

Agile development strives for the users to be able to utilize the product as early as possible instead of waiting for a "big bang" shipment towards the end of the project, giving them an opportunity to start earning back the development investment early, if possible.

The Product Owner is typically responsible for the long-term plan for the product's development, and plans project/efforts as well as sprints together with the development teams, who in turn are left to plan each day themselves. This overlap in responsibility is a construction to facilitate a tighter collaboration.

Even bigger planning work, sometimes referred to as *pre-planning* can be performed using Scrum, in sprint format, where the sprint deliverables would be for instance results from research, design prototypes, cost estimates etc. If system development utilize *refactoring* it is possible to even start building functionality early to test concepts and shaping the product.

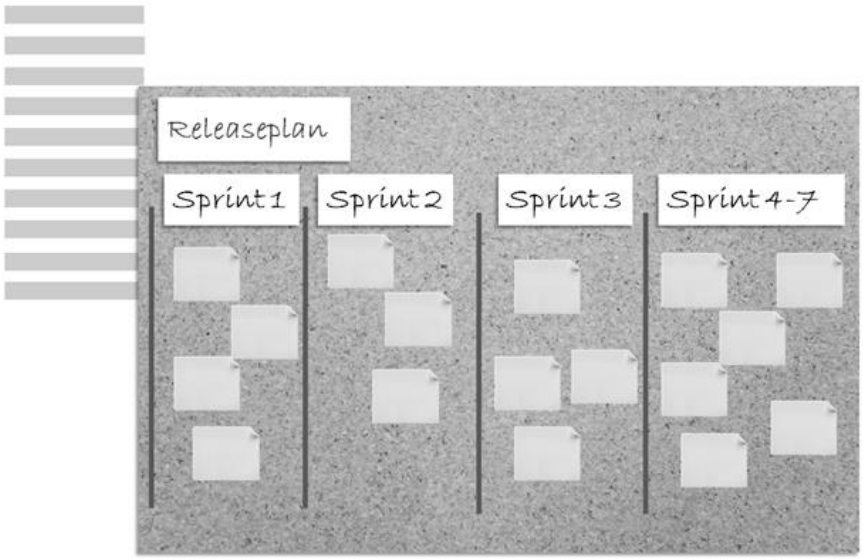
Planning sessions are time-boxed and iterative, where participants spend a proportional amount of time on detailing and researching requirements depending on their priority. Higher priority items will get more time and research, as they are more likely to survive the project unchanged compared to low-priority requirements, which are likely to be changed, swapped out, or removed entirely. Spending too much time with low-priority requirements early is likely to be a poor investment.

This also raises the need to involve at an early stage everyone needed to create functionality throughout the system. The agile world is devoid of concepts such as "GUI-done", "database done" and so on. Functionality is either *done* (offering a value) or not. To have control of the quality produced also becomes important, as a time estimate becomes less credible if no one knows if the proposed solution will end up creating 30, 300 or 30 000 bugs in the code, making it impossible to work with prognoses and continuous improvement. Quality engineers within development teams can help with building quality into the code from the beginning, and measure current quality.

Visual Planning

Visual planning has become popular as agile frameworks have gained a foothold in the industry, and are often inspired from *Kanban boards* from for example Toyota or Scania. They visualize in a simple manner the current flow of production, and allow every worker to control the optimum flow upstream.

For development work, the current trend is to put complicated tools for planning and follow-up to the side, and use visual boards and sticky notes to show and update the current and upcoming plan. By displaying the plan in a simple and accessible manner, everyone is encouraged to participate in keeping it updated and seeing and acting on deviations. Here is an example of a simple roadmap of sprints.



The development team has suggested a roadmap ("sprint map"? ☺) a few sprints ahead, with release contents and goals clearly listed. This can be accompanied by a detailed sprint plan on a separate board, issues and risks, etc. all clearly written on sticky notes and A4 sheets.

A visual plan should initially at least contain the *current ongoing work in progress, a simple formulation of the goal, and some form of progress indicator with look-ahead.* These can in turn be represented for instance by:

- Sticky notes containing to do-activities that are being marked or moved on the board to represent activities as *waiting, ongoing or finished.*
- A sprint goal verbalized in one or a few sentences
- A burn-down chart that indicates the velocity of the team (how fast functionality is being produced according to the *done criteria,* which also is useful to put up, as a reminder)

The most immediate gains with a visual plan are the simplicity and transparency. It is easy for everyone to participate and understand it.

The Anatomy of a Sprint

Sprint Planning

During the start of each sprint, the upcoming work is planned, and the development team commits to as much from the product backlog as they believe they can complete during the sprint, considering their availability and velocity. The Scrum Master coaches the team during this time-boxed session (typically around a day in length) Requirements are detailed as needed into tasks, that are estimated for size and time, and when the team and the Product Owner agree on the scope, a common sprint goal is formulated as the team creates a *sprint backlog* consisting of tasks, often no more than a day in size. If the team is working towards a longer project plan, that is usually updated as well.

Sprint Work

Once planning has been done, the development work in the sprint starts, where the team works with their only focus to reach the sprint goals. By monitoring the velocity and do short re-estimations of remaining work every day, the team tries hard to deal with deviations in the estimates and impediments that occur. In some cases, the sprint goal may need to be re-negotiated with the Product Owner to be able to meet the sprint deadline. The team also synchronizes their work by briefly informing each other what they have achieved, what they plan to work with today, and impediments that may hinder them from working efficiently (sometimes called *Daily Scrum*). The Scrum Master works with removing impediments that the team can not address themselves.

Sprint Review

On the final day of the sprint, the team presents the new achievements in the form of a short demonstration to all stakeholders. After the demonstration, the team and stakeholders discuss how the sprint was, and agree on practices that are important to maintain, but also to change routines or practices that had a negative impact on the sprint. Among typically common improvement suggestions are moving to a common room to work closer as a team, improving or adhering more carefully to the Done Criteria, detailing requirements in a different way, etc.

Improvement suggestions the team cannot implement themselves are listed on a separate *impediments backlog* that is maintained by the Scrum Master, and used by management to improve the organization and workflow.

The Agile Organization

By combining working organization models and meeting techniques from Lean companies with the Scrum framework, larger development efforts as well as the entire organization can be lead in an agile manner.

Larger development efforts and projects

A practical upper limit for group size exists, and it is usually smaller than expected. In Scrum, it is sometimes mentioned that nine members is the largest optimal Scrum team. My rule of thumb is that *each member of the team should be able to grasp what every other member is doing*. When it becomes difficult, it may be time to suggest that the team split into smaller units.

It is important to remember that a Scrum team that has been split into multiple teams really is still a group that was split up for practical reasons. Hence it is often a good practice to keep a common planning and

review for all the split teams together. This technique also works well with teams that are dispersed geographically.

The practice to often build a *ready but not complete* system is more difficult when multiple teams are involved. To ensure that the code does not break the system apart, technical infrastructure and synchronization efforts are needed. The synchronization is done through a meeting called *Integrating Scrum* that is added to the Daily Scrum.

Integrating Scrum

This meeting is held briefly once a week or several days a week, on a similar format as Daily Scrum. The purpose of the meeting is that participants represent their respective teams, stating what the team has achieved since last time, what they plan to do ahead, and *integration issues* that may surface. Usually, the meeting is populated with members of the various collaborating teams (and usually the quality of the meetings suffers if they are populated by Scrum Masters for the teams instead of team members!)

If no risks are apparent, the meeting is over in a few minutes, serving as a brief synchronization. Otherwise, members plan a solution meeting where the integration risk is investigated and avoided. The meeting is responsible to ensure that the code is integrated and architecture integrity maintained.

This meeting needs an *Integrating Scrum Master*. Apart from Scrum Masters (perhaps for one or more of the teams), architects, product owners, and functional managers are good candidates for the role. This person also becomes responsible for maintaining and communicating the common Done Criteria!

Pulse meetings

The practice of short meetings that only focus on synchronization and finding deviations from the plan through simple means can be called *pulse meetings* to borrow a terminology from the world of Lean. It has been used successfully to separate *status meetings* from *solution meetings*. When the two meeting types are confused, they tend to start dragging out over time, ending up in long-winded detailed technical discussions that may not thrill the entire team. The idea behind pulse meetings instead to have such a short time-box for the meeting, that participants have to prepare all status indicators ahead of time, to minimize time waste for the participants. Whenever deviations that impact one or more teams are raised without a solution, stakeholders immediately plan when they will get together to solve matters (often done by staying behind after the pulse meeting)

Some suggestions for pulse meetings that can be inspired by Daily Scrum are:

- Functional line meetings, where functional managers meet to briefly check that all development projects have enough staffing and skills to function well, and help each other out if needed.
- Product portfolio meetings, where Product Owners get together to state what their projects have been doing (earlier sprint results), what they will do (sprint plan) and resolve priority issues between their respective projects using data from the teams.
- Scrum Master Meetings, where Scrum Masters reconcile their impediments backlogs, and share coaching insights and learn other things from each other.

Resources

1. *The New New Product Development Game.* **Nonaka, Takeuchi.** u.o. : Harvard Business Press, 1986.

2. Wikipedia: Complex Adaptive Systems. *Wikipedia.* [Online] [Cited: den 31 Jan 2011.] http://en.wikipedia.org/wiki/Complex_adaptive_system.

3. **Wikipedia.** Wikipedia: PDCA. [Online] http://en.wikipedia.org/wiki/pdca.

4. **Cohn, Mike.** *User Stories Applied.*

Contents

About the author

A former graduate of computer science, I have worked as a developer, project manager, Scrum Master and development manager since 1994 in a wide array of product developing companies, such as online gaming, telecom and biotechnology. In 2004 I was recruited to become a consultant with Citerus, where I work as a mentor and coach for management teams and individuals in agile organizations. I also work as a professional trainer and lecturer.

I have a close collaboration with leading organizations and research in Lean Product Development, to help "our business" – software development – to learn from their practices and experience.

I became one of Sweden's two first Certified Scrum Trainers in 2006 and have trained hundreds of individuals how to become proficient Scrum Masters and Product Owners.

I can be reached at www.citerus.se and mikael@lundgren.com I blog at www.leanspired.com.