**KTH Computer Science
and Communication**

Agile Development in a Solo Environment

MATHIAS LINDBLOM

Contents

Part I

Background

The following two chapters forms the base for the theoretical and practical part of this reports study. The first chapter explains agile development as a concept and its core ideas. The second chapter explains some of the most common agile methodologies, by mainly going through their practices.

Chapter 1

Agile Development

Most developers have experienced the nightmare of working on a project and slowly witness the increasing number of bugs, errors and time when adding new functionality. In order to combat this disheartening scenario, developers tend to create constraints and routines around their activity, forming a development process based on prior mistakes. However big development projects can be complex and even though experienced based constraints helps, the problems remains and builds upon each other. One solution for this is to keep adding more constraints and routines. So the development process grows and grows and until the process itself becomes so cumbersome and advanced that it creates the problems it was designed to prevent, which was to avoid technical dept and increase efficiency [1].

This so called runaway process inflation was common among big companies around the millennium shift and at the time it was a growing negative trend [1]. Software teams around the world experienced this, including a few industry experts calling themselves the Agile Alliance who sought to fight it [1]. The ideas that make up agile development were not new at the time but it was the Agile Alliance that formulated the underlying concepts that make up agile development. The report will come back to this at section 1.2 The Agile Manifesto.

1.1   Component Abstraction

It is common that agile development is directly compared to the waterfall model which is strange given how they situate on different abstraction levels. What this means is that the waterfall model is a software development methodology just like Scrum. Agile development on the other hand is not a followable methodology by itself. It is more of an ideology or perhaps a collection of methodologies. So how is it even possible to compare the two of them? The answer is that people are comparing the waterfall model to some popular agile methodology like Scrum or some common practices like sprints and standup meetings, indirectly claiming that these elements represent agile development as a whole. This leads to confusion and unfair comparisons. So before continuing reading about agile development

it is a good idea to give a short overview of the components that make up agile development. Bare in mind that this is a simplified overview meant to give the reader a better sense of how agile development and the methodologies with its practices correlate to each other.

In Figure 1.1, agile development have been dividid into four different abstraction levels. First we have the ideology that consist of the core ideas such as The Agile Manifesto and principles. This is basically what defines agile development. Secondly we have the methodologies and strategies that fulfills some or all of the agile principles and philosophies. The third abstraction are the practices that make up the methodologies, similar to how the principles make up agile development. However, the practices are more practically applicable and at a much lower abstraction level. Lastly, at the forth abstraction level, we have the actual tools that are needed to make the practices a reality. These tools can be very simple such as a physical room for meeting practices or more advanced such as the xUnit framework for test-driven development.

The report will frequently talk about these different components so it is a good idea to keep a mental image of Figure 1.1. Moreover, this report will refer to agile development as an ideology in order to not confuse it with the methodologies.

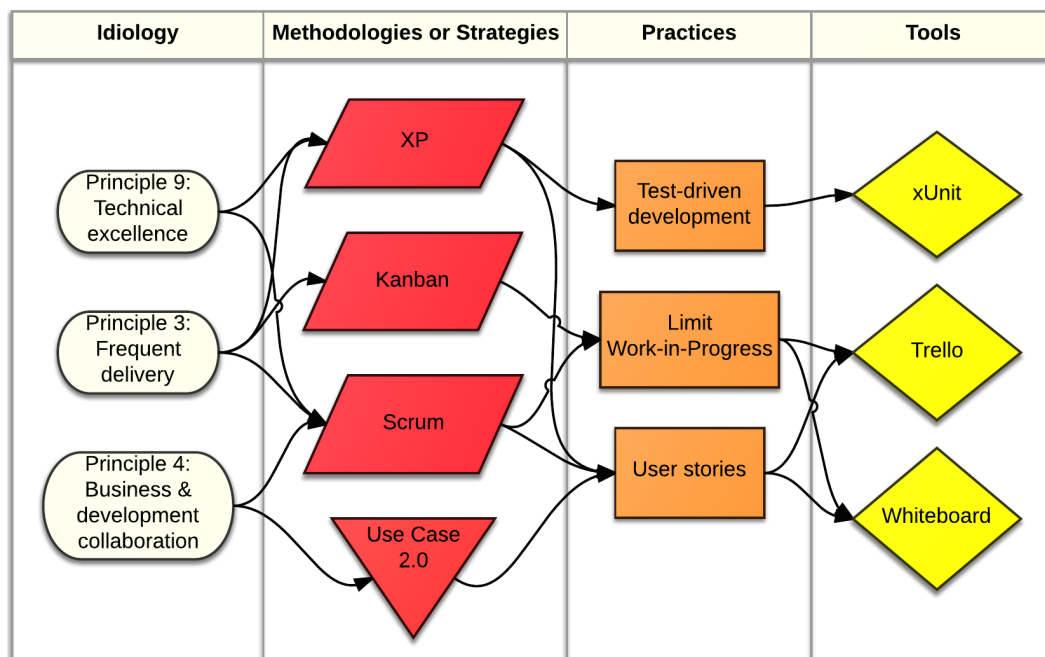
Figure 1.1. Agile development divided into four different abstraction levels starting with the ideology that defines agile development and ending with physical tools. It also shows how the example elements within them correlate to each other.

1.2 The Agile Manifesto

The Agile Alliance is an organization formed by a group of computer industry experts in 2001. They decided to come together to form general development values in order to improve the software development process for companies around the world and to give an alternative to the popular heavyweight methodologies [1] [2]. The result was The Agile Manifesto that states the following:

- **Individuals and interactions** over processes and tools.

- **Working software** over comprehensive documentation.

- **Customer collaboration** over contract negotiation.

- **Responding to change** over following a plan.

The group that wrote The Agile Manifesto consisted of the following people: *Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas* [1] [2]. Several of these people are the authors of the literature used in this report, this is especially true for chapter 2 Agile Methodologies.

The following four subsections will give a short explanation for each of the statements of The Agile Manifesto that stands as the highest abstraction or the core that is agile development. This will give the reader a good overview of the thinking and reasoning regarding agile development.

1.2.1 Individuals and Interactions over Processes and Tools

> Process and technology are a second-order effect on the outcome of a project. The first-order effect is the people.
>
> ALISTAIR COCKBURN [1]

Within football most agree that a football team of average players that communicate well can beat a team of egoistic superstars. The same goes for software development (and probably many more things) where a team of average programmers working together outperforms a team of expert programmers that fail to communicate. At least according to the thinking and reasoning of agile development [1].

Processes and tools are of course important as well but not as important. After all, agile development methodologies are processes that usually requires tools so it would be quite ironic otherwise. Team managers have a habit of setting up the development environment and then assigning a team, assuming that the team will work well since they have the tools they need [1]. However this is focus put at the wrong end of order. Tools and processes can rarely repair a team that doesn't play ball but a functioning team is usually more than capable of configuring their own environment.

1.2.2 Working Software over Comprehensive Documentation

> The program is the specification and documentation.
>
> DOUGLAS CROCKFORD [3]

This might not be that much of a shocker, that functioning software is more important than documentation. The keyword here is *comprehensive* since it is common that teams get hung up on the quest of having close to 100 percent coverage on documentation [1]. The problem with this is that it takes a lot of time and effort to always keep in sync with the code and as soon as you lose the sync, the documentation begins telling lies which is worse than having no documentation. Instead of having every technical part of a program described in a thick book it is usually preferred, both for the reader and the writer, to have a short rational and structured document that explains the software at a high abstraction. When a new member of a team needs more technical information they will get it when working closely and interacting with their team members and the actual software.

A common misconception is that code is for the computer and documentation is for the human to understand the code [4]. Instead of translating the code into human language, why can't we write code so that humans can understand it in the first place? We already do it by naming things like classes and methods with understandable names, but is usually ends there. We make it *good enough.* If we instead focused more on the quality and logic of the code, the value of documentation would be greatly diminished.

1.2.3 Customer Collaboration over Contract Negotiation

Many have tried to create strict contracts with fixed specifications, deadlines and prices for software development. Many have failed [3]. You simply can't treat software as a commodity expecting everything to be known before hand. Trying to do so can lead to all sorts of problems like poor quality or even complete failure. It is not only the complexity of programming that is the issue but also how fast our technical world changes over short time periods.

This is one of the core problems that agile development tries to avoid. By having a tight collaboration with the customer, the specifications, deadlines and prices don't need to be as strict. Instead of having a contract specifying the exact requirements of the entire project it should instead address how the collaboration between customer and developers should be conducted [3]. This leads to a more dynamic development approach where technical problems can be addressed directly. Moreover, needed changes don't risk breaking the contract.

1.2.4 Responding to Change over Following a Plan

The previous section regarding customer collaboration is an indirect example of this prioritization. Customers have a habit of not knowing what they really want or

change their requirements after seeing some of the functionality coming to life [3]. Also like stated in the previous section, the IT world we live in rapidly changes and so can the requirements of a software project. It is nice and tempting to have the full project planned out in some advanced planning software with every functionality written. However as the development team and customer moves forward the plan is bound to change. Some functionality will no longer be deemed necessary and previously unthought of functionality might be added. Some parts of the strict plan might then have been a waste of time and more importantly, the strict plan might hinder these kind of changes that are needed or at least advantageous to the project.

By having a more abstract and lose plan with details only for the near future, for example a couple of weeks, you get a planning more susceptible to change [3]. That detailed plan can be strict but since it is only for a short time period the risks of having unneeded functionality or hindering change is limited. The further away in time we look the weaker the plan should be since we want flexibility to changes and the likelihood of changing requirements grows as the timeline increases.

1.3   The Agile Principles

Using the values stated by The Agile Manifesto, 12 principles were created that act as the characteristics for agile practices [1]. You can interpret these principles as the next small abstraction step from The Agile Manifesto where you get a clearer explanation of what agile development is. These are the principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity–the art of maximizing the amount of work not done–is essential.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
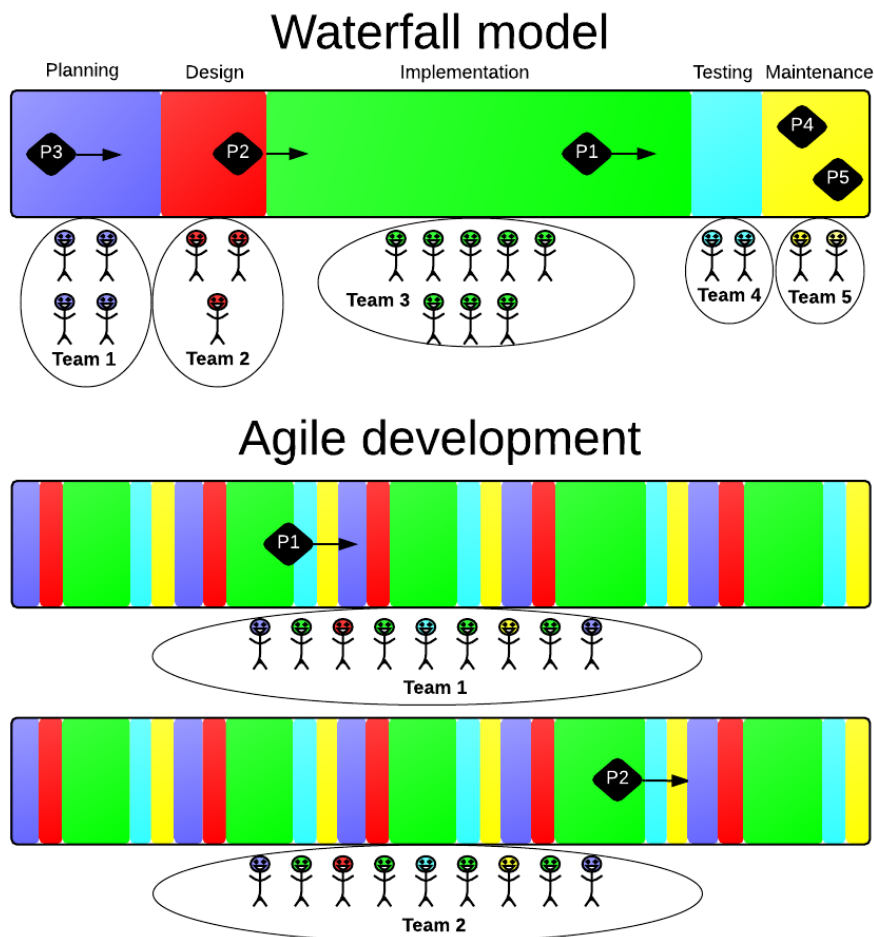
1.4 Agile vs Waterfall

Figure 1.2. Simplified abstract image showing the difference between the waterfall model and agile development. The image portrays how projects move forward in an arbitrary timeline through difference phases. P1 to P5 stands for Project 1 to Project 5.

This section will a short illustrative and abstract view of the difference between agile development and the more established software development methodology called the waterfall model. As explained in section 1.1 Component Abstraction, one should be careful doing detailed comparisons between agile development and other methodologies since agile development is a bundle of methodologies.

As seen in Figure 1.2, the waterfall model takes one logical project step and finishes it before starting the next step, like sequential waterfalls that moves the project from one logical plane to the next. Each team work on different projects at any given time and each team work on one part of the project chain. Which means that the implementation team does not care about design for example. In Figure 1.2, when the design team is finished with project 2 it gives the project to the implementation team and waits for a new project to arrive from the planning team.

In agile development, the project does not normally move from one distinct team to another. Instead, one team holds a project until it is completely finished before starting work on another project. For a team to be able to hold a project from start to finish it is vital that entire team incorporates all necessary skills within a project, which normally is done through so called cross-functional teams that are often mentioned within agile development. If another project starts, another cross-functional team is assigned that project.

1.5 Agile vs Lean

When you read about different software development methodologies they can sometimes be referred to as *lean*, sometimes *agile* and sometimes *lean and agile*. Often without explanation of what specific parts makes the methodology lean and what makes it agile which can be very confusing. To understand the confusion it is best to give a short explanation of what lean development means:

Lean

Lean product development (LPD), or lean manufacturing, dates back to the mid 20th century. The history of LPD begins with Toyota that sought to improve car manufacturing efficiency, quality and flexibility. One way to achieve this, according to LPD, is to have the entire chain from design to full assembly at one physical place [15]. This is comparable to how agile development promotes having short iterations where all parts of the software development chain is incorporated, as seen in Figure 1.2.

Lean software development (LSD) is simply the adaptation of LPD to the software industry that originates from the book *Lean Software Development: An Agile Toolkit* by Mary Poppendieck and Tom Poppendieck [16]. Like with agile development, LSD has a few principles that gives a good overview of what the methodology framework or ideology is about:

1. Eliminate waste

2. Amplify learning

3. Decide as late as possible

4. Deliver as fast as possible

5. Empower the team

6. Build integrity in

7. See the whole

Note that LSD has just over half the amount of principles compared to agile development. Without going into detail for each specific principle we can see that several of the lean principles have similarities to the agile principles. L1 says to remove waste while A1 says to maximize value. L2 is hidden in several of the agile principles. L3 is about delaying decisions as late as possible to make the best informed decision which is a simplified version of A2. L4 and A3 both talks about fast delivery. L5 is basically what A5 states.

So by looking at the principles it is safe to say that the lean and agile ideologies have several similarities. They both talk about adaptive planning, empowering the people, fast delivery, continuous improvement etcetera. The differences lies in focus and scope that becomes more apparent when reading literature about them. LSD focuses on minimizing waste and streamlining workflow while agile development focuses on adaptivity and communication. LSD reaches a higher range than agile software development by going all the way to talk about costumer vision, staffing and maintenance while agile development has a more strict focus on the project development phase.

What is important to note is that agile and lean do not go against each other. This means that methodologies trying to adopt these ideas do not have to be either agile or lean but can in fact be both, with perhaps a focus towards one of them. This chapter will end with a suitable long quote from one of the authors of The Agile Manifesto.

> So as you can see, lean and agile are deeply intertwined in the software world. You can't really talk about them being alternatives, if you are doing agile you are doing lean and vice-versa. Agile was always meant as a very broad concept, a core set of values and principles that was shared by processes that look superficially different. You don't do agile or lean you do agile and lean. The only question is how explicitly you use ideas that draw directly from lean manufacturing.
>
> MARTIN FOWLER [17]

Chapter 2

Agile Methodologies

Up to this point the ideology, that is agile development, has been described. In this chapter the more common methodologies that try to adapt these ideas and mentalities will be brought up. Moreover, the methodologies described here will contain more practically applicable agile practices. What is important to note is that every agile methodology is subject to change. Most authors and advocates of the agile methodologies encourages changes in the methodology itself to suit your development team and/or company. This means that the descriptions of the methodologies can look more or less different depending on the source, especially when they have been written years apart. Another reason is that some parts of a methodology can be vague resulting in different interpretations. However this does not necessarily mean that some descriptions of the practices are wrong. The vagueness can be seen as a way of encouraging changes and alternative implementations of the methodologies which the authors advocates.

The methodologies can be vast in their documentation and this report does not seek to explain each methodology in their deepest details. That would simply require too much work and be overwhelming for the readers. For the interested reader, each methodology will have recommended further reading in its introduction.

One thing that every agile methodology has in common is a set of practices that explain the most important components. Some methodologies refer to them as properties but they all seek to explain the core of the methodology similar to how the agile principles, in section 1.3 The Agile Principles, relate to agile development. It was therefor suitable to use them as the base for this reports overview of the methodologies in order to ensure that the description and comparisons are based on somewhat equal grounds.

A warning to the readers is that going through every practice of all four methodologies in order might be quite heavy reading. An alternative suggestion is to read the practice titles and go back to read the content when needed as the report starts investigating the practices at chapter 3 Theoretical Evaluation.

2.1 Scrum

Scrum is one of the most known and used agile methodologies today. Scrum was originally codeveloped during the early 1990s by two of the soon-to-be authors of The Agile Manifesto, Jeff Sutherland and Ken Scwaber [14]. Several other mentionable people, such as Mike Beedle, have further contributed to the development of Scrum. Since the agile methodology Scrum was conceived just a few years before The Agile Manifesto and by several of the authors, it is perhaps not that shocking that people sometimes wrongfully put an equality mark between agile development and Scrum.

Scrum goes against the traditional predictive way of developing and instead has a more empirical approach. What this means is that Scrum assumes that problems and challenges can't be foreseen in software development. Therefor focus lies maximizing the Development Team's ability to deliver quickly and respond to changing requirements.

The information regarding the practices of Scrum is mainly taken from the book *Agile Project Management with Scrum* by Ken Schwaber, one of the authors of The Agile Manifesto, together with updated information from the official Scrum web page: www.scrumguides.org [14] [10]. There are a total of 12 practices in Scrum divided the categories Roles, Events and Artifacts. Both the book and the web page are recommended for further reading regarding Scrum.
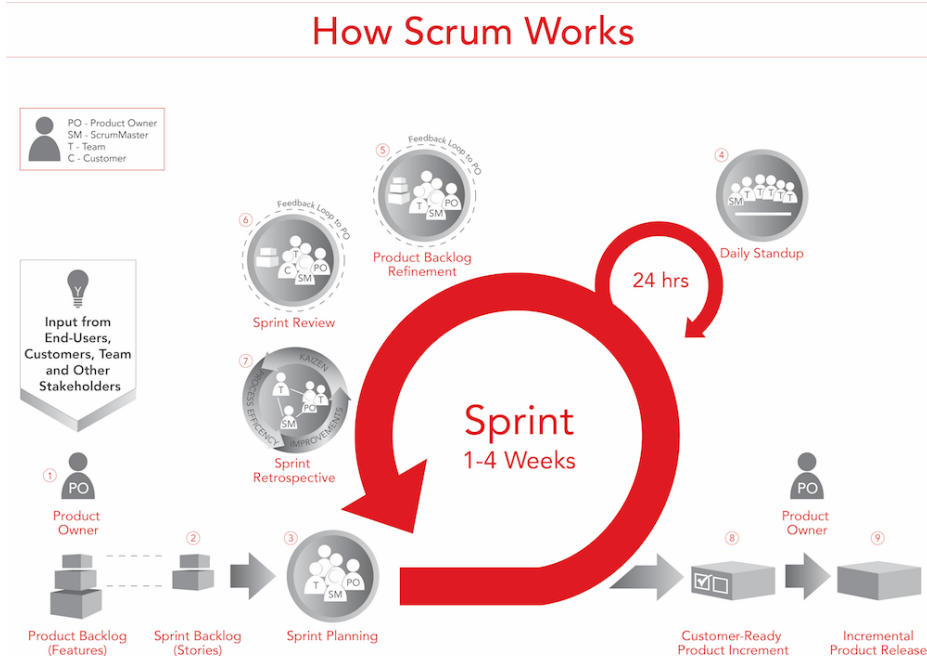

Figure 2.1. Image showing an abstract view of how Scrum works, including different several practices including the three different roles.
Source: Scruminc, http://www.scruminc.com

2.1.1 Roles

There are three distinct roles in Scrum. The Product Owner, Development Team and Scrum Master. The cooperation and relation between these roles is a key component in Scrum. The Scrum team consists of every member within these three roles.

Product Owner

The Product Owner is a single person. He/she is responsible of maximizing the work conducted by the Development Team which in turn means responsibility to maximize the product value. The practical priority is managing the Product Backlog. The managing of the backlog means taking care of the following:

- Ensuring that the Product Backlog items have clear and distinct descriptions.

- The ordering of backlog items to maximize value and reaching goals.

- Optimization the value of the work by the Development Team.

- Making the Product Backlog visible, transparent, clear to all project members and include information about upcoming tasks.

- Having the Development Team on a satisfactory level of understanding of the items in the Product Backlog.

Depending on the size of the project and company. The Product Owner may choose to distribute some of the tasks to other people such as the Development Team. However, the Product Owner is always the accountable person for the above tasks [10].

Development Team

In Scrum the Development Team are in charge of delivering working software, as one might expect. They are in a large sense self governed and chooses how they want to work, with some guidelines of course. The Development Team can be summarized with these characteristics:

- Self-organizing. The Development Team alone chooses how they want to turn the items in the Product Backlog into functionality.

- Cross-functional. The team should encompass all necessary skills to work with everything in the Product Backlog.

- No titles within the group. Everyone is simply a developer.

- No sub-teams. Just like there are no titles for the members, there can be no sub-teams such as a testing team. Not to be confused with multiple teams which is allowed and common in big projects.

- All for one, one for all. Members might have specialized knowledge giving them natural areas of focus but the Development Team should always be accountable as a whole and never rely on individual expertise.

It is recommended that any Development Team consists of 3-9 members [10]. The reason being that too few developers limits the interaction and the team are more likely to have skill constraints. To many members lead to too much complexity that can make coordination between the members difficult.

Scrum Master

Deciding as a company to use a methodology and then explaining how the methodology works to all employees does not mean that the company will be developing according to that methodology. This is where the Scrum Master comes in. The Scrum Master is in charge of keeping the Scrum wheels turning. He/she makes sure that the Product Owner and Development Team are collaborating and generally doing what they are supposed to do. The following is a summarization of the responsibilities of the Scrum Master:

- Remove barriers regarding development so that the Product Owner is free to directly drive development forward.

- Teach the Product Owner how to maximize return-on-investment (ROI) in order to meet his/her milestones using Scrum.

- Improve the quality of life for the Development Team by empowering them and encouraging creativity.

- Assist the Development Team in increasing productivity in any way possible.

- Ensure that each increment of functionality is potentially shippable by improving engineering practices and tools.

- Give transparency to the Development Team's work progress for all parties and ensure it is up-to-date.

The position of the Scrum Scrum Master is normally filled by the project manager. Ken Schwaber indicates in his book, *Agile Project Management with Scrum*, that a Scrum Master should be a leader figure and never act as a regular boss [14].

2.1.2 Events

In Scrum there are five so called events. However the first event called Sprint is the most important one and all the other events are basically a part of the Sprint.

Sprint

As seen in Figure 2.1 the Sprint is very central in Scrum. The Sprint is a short period of time, usually a couple of weeks up to a month [14]. As soon as one Sprint ends, another starts. The Sprint is about dividing the entire project into small iterations. The length of a Sprint should be small enough to avoid complexity but large enough to avoid too much project overhead. The following four practices are all parts of the Sprint and describes more in detail what the Sprint is about.

Sprint Planning

Each newly starting Sprint is initialized with a Sprint Planning. The entire Scrum team should be present during this planning. The Product Owner brings up the Product Backlog and its priorities. These are discussed by all parts and the Development Team goes through what will be achievable for the next Sprint. So to summarize the planning is about deciding what should be done and what can be done for the coming Sprint. The meeting usually has a time limit of 8 hours [14].

Daily Scrum

Also referred to as daily standup as seen in Figure 2.1. This is something that the Development Team does every day and should take around 15 minutes [14]. They do this to synchronize and coordinate their work efforts. There are three important questions to be answered during this meeting which are: What work have each member done since the last Daily Scrum? What is everyone planning on doing before the next Daily Scrum? What hinders exist that stand in the way of the team meeting the current Sprint goals? It is the Scrum Master's job to enforce that the Daily Scrum takes place and that it is only the members of the Development Team that participate [10] [14].

Sprint Review

The Sprint Review is held close to the end of a Sprint. The entire Scrum team should be present and if possible the stakeholders as well. This is supposed to be an informal meeting where the added functionality from the latest Sprint is shown. The Development Team shows what they have done and the Product Owner shows what has happened to the Backlog. The idea is to have a revised Product Backlog at the end that can be used for the next Sprint. The durations of this meeting is usually limited to four hours [14].

Sprint Retrospective

The Sprint Retrospective should take place after the Sprint Review but before the upcoming Sprint Planning. Once again the entire Scrum team is present, but without the stakeholders. This meeting should take about three hours [14]. The focus of this meeting is to improve the work process during a Sprint. The team

reflects about things like tools, relationships, processes and anything else that can improve the workflow of the team and next Sprint.

2.1.3 Artifacts

The three artifacts are the most basic parts in Scrum that are worked on during the different events.

Product Backlog

The Product Backlog is the dynamic requirements list in Scrum. It is an item list comprising of anything that can add any form of value to the product. Typical items in a Product Backlog are new features, bugs, enhancements and functions. Each Product Backlog item also has certain attributes such as a description, priority order, complexity estimate and value estimate.

What separates a Product Backlog from a simple requirements list is that it is a dynamic process and not a static list. The items in the Product Backlog that are to be considered for the "near future", as in having a high priority order, have more details than other items. This is to allow changes to the distant Product Backlog items as the actual product developes. The Product Backlog is in focus for both the Sprint Planning and Sprint Review. The refinement of the Product Backlog items is sometimes counted as a practice by itself as seen in Figure 2.1. However the items can be tuned and refined at any time. All members of the Scrum team have a saying when it comes to the Product Backlog but the Product Owner is the main responsible person.

Sprint Backlog

The Sprint Backlog is the selected items from the Product Backlog for the ongoing Sprint. It is the list of work to be done during the Sprint. All items in this backlog have clear details since the idea is to have them all completed at the end of the Sprint, which is a relatively short time period. The Sprint Backlog changes during a Sprint as the items become implemented and moved to "done". New items can also be added as unforeseen needed functionality is added. It is the Development Team that work with the Sprint Backlog and uses it as their progress report and todo list for the current Sprint. At each Daily Scrum the Sprint Backlog is updated and refined with informations such as remaining work hours.

Product Increment

A Product Increment is the sum of all implemented Sprint Backlog items for a Sprint. It is a sort of "next step" for the product. The requirements for an item to be counted in the Product Increment is that it is fully usable for the product, regardless if the Product Owner chooses to release it or not.

2.2 Extreme Programming (XP)

Kent Beck is the founder of Extreme Programming (XP). He is also one of the members of the Agile Alliance that you can read about at section 1.2 The Agile Manifesto. XP was conceived shortly before the creation of the Agile Alliance. In 2000, just a year before the Agile Alliance surfaced, the first book about XP called *Extreme Programming Explained: Embrace Change* was published by Kent [9]. For this reason, when reading early literature about XP their is no mentioning of the term *agile*, which is not strange since at the time the term did not exist. However more published articles and books have surfaced after the year of 2001 that have incorporated the agile ideology, such as the second edition of Kent's first book released in 2005 [9].

Unlike Scrum, XP almost solely talk about the developers and helping them be as efficient as possible. It does not talk much about the overall business outside the development team. Although if you read further about XP you will find business related information but it is not at all in focus. XP put emphasis on team work, feedback and code quality. The term extreme comes from the idea that it takes a few ordinary development practices and impose them to an extreme. Originally XP was designed for small to medium sized teams but has over time grown to incorporate large teams as well.

There are a total of 12 main practices in XP that will be shortly described in the following sections. There are several corollary practices as well that talk more about the business and goes even further into coding details. To read about this and much more, the second edition of *Extreme Programming Explained: Embrace Change* is recommended [9].

2.2.1 Fine-Scale Feedback

The following practices describes how the members of the team develop their software with feedback in focus. It goes through the basics of how they work in pairs and always use test-driven development (TTD) to guide them. It also describes the relations between business people and developers.

Pair Programming

This is perhaps the most notable practice regarding XP. Pair Programming means that for each developer, there is a another developer observing and assisting without coding. They work together similar as in rally racing where one person is the driver and one is the map reader. The main difference is that in Pair Programming the coders switch places regularly. The purpose is that the observing developer can have their mind set on a higher abstraction level and is not distracted by coding, making it easier to ensure high code quality [1]. Not only are bugs detected more easily but the overall resulting code is simply put better. This way of developing can be quite motivating for the developers since you are never alone and if you grow tired of

typing you just switch places with your pair. If the developers also switch partners every other day, information and knowledge is spread rapidly and smoothly within the entire team. This is especially good when you have specialists with knowledge that could benefit the other developers. Studies conducted by Williams and Nosek has suggested that pairing does not reduce the development efficiency but does show a decrease in defect rate in the software [8].

Planning Game

The Planning Game is played between the business people and the developers. It is about figuring out and deciding witch features to implement and basically form a prioritization of what needs to be done. The business people focuses on the importance of each feature [1]. The developers focuses on the cost of developing the features.

A simple and natural way to play the game is: After each iteration, which for example might be a two week period, the developers updates the cost of future features. They also come up with a budget for how much they will be able to develop for the next iteration. Then it is the business peoples turn. They decide the features, within the budget, that they want the developers to work on for the next iteration. This game plays out after each iteration.

In reality it is much more complicated and a whole chapter is dedicated to the Planning Game in the book *Agile Principles, Patterns, and Practices in C#* by Martin C. Robert and Martin Micah [1]. Things like velocity and user stories are described in detail which are all important components to the Planning Game. What is stressed is transparency since everybody might not be up to play ball just relying on some developers estimation.

Test-Driven Development

The following steps describes TTD:

1. Write an unit test for some functionality.

2. Make sure it fails before implementing it, since it would be strange otherwise.

3. Write the code that implements the functionality.

4. Run the unit test and if it fails, fix it until the test passes.

5. Repeat for every future functionality.

This can seem overwhelming at first but it leads to having a close to 100 percent test coverage which can be very beneficial as the project grows. Another positive aspect is that writing test cases before coding leads to all code being testable by definition. Some may think: *Isn't all code testable?* As soon as you build a system without the slightest thought about testability you will get the answer. Further

more, using TTD you tend to get code that is less coupled and better structured [1]. The reason for this is simply because your mind is set to create code that can be tested independently, indirectly leading to decoupled module like structure.

Whole Team

Saying that XP focuses on the entire team does not simply mean the entire development team. Everyone that is in any way connected to the XP project should be collaborating closely with each other to solve problems. This goes as far as preferably having the customer in the same room or within 100 meters from the developers [1]. The more time the customer can spend with the team and the shorter distance the customer is from the team the better. If the customer is unable to be within close vicinity then it is advisable to have a stand in representant for the customer.

2.2.2 Continuous Process

These practices describes the cycles such as sprints and release plans for the project. Things like frequent integration are brought up and the importance of continuous refactoring.

Continuous Integration

For those that frequently work with multiple people using Git, Continuous Integration will sound familiar. Each programmer should integrate their work several times a day. If you are lucky (or fast), you are first and simply check in your work. If you are not so lucky, you have to merge the changes. In other words, XP teams use nonblocking source control [1]. This means that every developer can work on any part of the system, simultaneously with the other developers. It is important that each system test and acceptance test are run before anything is checked in. If something is broken, the developer fixes it before checking in the changes.

Refactoring

Refactoring is not a backup practice when quality ensuring practices fail. Refactoring is a quality ensuring practice.

Sometimes it can be hard to convince non-programmers the benefits of refactoring. The main reason is that feature wise, nothing is added. At most some optimization might be noticeable. Refactoring is about changing the foundation of the software. It is to make adding new features a simpler task. Code duplication is a sign that refactoring is needed. When developers start to choose bad coding practices simply because they see no other way of implementing something, factoring is needed. When changing code affects unrelated code sections, refactoring is needed. However the idea is not to code poorly, fix it with refactoring and continue. Most developers knows that coming to a situation where refactoring is advisable is pretty much unavoidable, the important thing is not to add fuel to the flames. Pair

Programming for instance seeks to diminish the need of refactoring and make sure quality is ensured [9].

In XP refactoring is not something you do only at the end of an iteration or cycle, not even something you only do at the end of the day. You do it as soon as you see that it is needed. This means that refactoring will come in really small packages that alone do not bring much value. Together however, all the refactoring keep the clone clean, simple and as expressive as possible.

Small Releases

Small Releases means release often and release small. Releasing software can be a pain but that is usually the case when you seldom release, thus having no routine for it. The releases in XP can be split into two types. The iteration plan and the release plan:

The iteration plan
This is a sort of minor release. In XP the usual interval for minor releases of new software is every two weeks [1]. However as short as possible is preferred. The two week period is just a well working standard. At the end of each iteration cycle, decisions regarding user stories and features should be made according to the the Planning Game. It is not obligatory that something gets released at the end of this cycle since the short time period might mean that too few features with value have been implemented.

The release plan
This is basically the bigger version of the iteration plan. This is a more abstract plan over a longer time period that usually spans around three months [1]. A clear difference is that the iteration plan should not change but the release plan is subject to change during its lifetime. A simple explanation for this is that it is expected to foresee implementation issues two weeks ahead while this is not the case for three months. It is also expected that something gets released at the end of a release plan cycle.

2.2.3 Shared Understanding

The following practices are about giving the developers a team feeling and shared understanding of the entire project. The practices talk about code design, abstraction and that sharing knowledge is caring.

Coding Standards

Pair Programming describes how knowledge can be spread swiftly by routinely changing programming partner. For this to work well it is important to have a coding standard so that the programmers don't spend time trying to understand each others semantics and syntax. Without a coding standard, whenever a programmer

starts working on someone else's work, he/she might feel the need to format or even refactor the code.

When setting a coding standard it is important with proper communication so that every developer feels comfortable with the standard and voluntarily wants to uphold it [9]. There is really no reason not to have a coding standard since it will make the software easier to understand which indirectly increases code quality.

Collective Code Ownership

There will always be developers with different specialities. The common way of just working on ones speciality means being more locked to that speciality. XP instead seeks to minimize this speciality difference between the members. Both Pair Programming and Coding Standards helps in evening out the specialities between team members.

Everybody can choose to work on any part of the software such as on the database, the graphical user interface (GUI) or the server. As a XP developer you are not confined to a specialty and assistance should always be given to those who seek to broaden their expertise by coding on unfamiliar parts of the software [1]. This is to increase motivation, reduce risks by spreading knowledge and aid the personal development of the developers.

Simple Design

XP seeks to keep it simple and expressive as stated by the practice Refactoring. The infrastructure is not in place when developing according to XP, the infrastructure gets created when it is currently needed. There are three so called mantras that an XP developer follows regarding simple design:

Consider the simplest thing that could possibly work
When developers are about to implement a new story. They think about the simplest possible scenario that satisfies the story. Then they choose the simplest possible practical way of making that scenario a reality. This means that XP developers do not think about future stories when implementing the current story.

You aren't going to need it
Sometimes you just know that a database will be needed in the future or that a web server is going to be created. Is it not important to make sure the code currently being implemented is easily hooked to these services even if the services don't exist yet? The answer is maybe. The developers only do it if they are absolutely certain that the service will come in the future together with the conclusion that creating hooks for this service before it exists will be more valuable than creating them later.

Once and only once
If duplicate code exists, make sure it doesn't. The best way to remove du-

plication is creating abstraction [1]. For example if a couple of code parts do almost the same thing, use templates. The important thing is that code duplication is removed as soon as it is seen.

System Metaphor

This is the most vague practice in XP. In the book *Agile Principles, Patterns, and Practices in C#*, Martin C. Robert and Martin Micah describes the System Metaphor by describing a jigsaw puzzle [1].

How do one know what pieces go together in a jigsaw puzzle? A blind person could only go by the shapes and try to fit pieces that seem to go together. A more powerful evidence that the pieces go together is the image. If the image seems correct, we don't have to look at the shape, we just know they somehow go together. More importantly, if the pieces do not go together even though the image is correct we know for sure that the puzzle maker has failed with the pieces. A blind person would have a much harder time coming to this conclusion.

In other words, the metaphor correlates to how important it is to know the big pictures in order to make sure that the modules or components are what we really want them to be. No offense meant to any blind person out there. Also note that the description of the System Metaphor using the jigsaw puzzle is a metaphor by itself, mind blown! By using metaphors we can create an abstraction, or the big picture, for complicated software or systems. When using metaphors you often use a system of names for different elements that on a high level describes the overall mechanisms.

The metaphor is the vision for a system. It is the guide for the developers that help them name classes and methods, select appropriate locations for new components and overall helps ensuring that everyone knows where, what and why they are currently implementing something.

2.2.4 Programmer Welfare

This section only has one practice where the focus is on the wellbeing of the developers. The practice Sustainable Pace is about avoiding burning out the developers with stress and overtime.

Sustainable Pace

Like a marathon runner must conserve his/her energy to not burn out before the race is finished, the development team in XP must keep a Sustainable Pace. A software project takes a long time to finish and sprinting from start comes with several drawbacks. It can affect motivation negatively in the long run. Estimations becomes wrongfully optimistic as time passes. Also working extra hard from start, when you have minimum knowledge of the heading of a project, can lead to redundant work.

Therefor in XP you are not allowed to work overtime with the only exception of the final week before a big release [1]. The big releases usually comes once every

third month like stated by the practice Small Releases. That does not mean that the practice is to always work overtime before every release, it simply means that is allowed if a team happen to be extremely close to reaching an important release goal in the final moments before release. Making sure that everyone works in a Sustainable Pace also have the effect of conserving energy so the team is alert for that seldom occurrence when an extra push is needed.

2.3 Crystal Clear

Crystal Clear is an agile development methodology developed by Alistair Cockburn, one of the authors of The Agile Manifesto. Her role in the creation of The Agile Manifesto was to focus on efficiency rather than handling rapidly changing requirements and thus Crystal Clear has focus on efficiency, similar to XP [5]. Just like agile development can be seen as a family of methodologies, Crystal is a family of methodologies. Some of the other variations that exists are Crystal Yellow, Orange, Red, Maroon, Blue and Violet. The darker the color, the bigger project/team it is suppose to handle. Crystal Red is for example for a project consisting of 50-100 people. Crystal Clear, like the name clearly suggests, is designed for the smallest project consisting of 5-8 people. It is for that reason that Crystal Clear was the chosen sub methodology for this report since we seek to find agile methodologies suitable for small development teams.

Crystal Clear has seven properties that stand as the practices of the methodology. The first three properties, Frequent Delivery, Reflective Improvement and Osmotic Communication are the only required properties and the last four are recommended, especially for experienced teams. For a more detailed description of Crystal Clear read the book *Crystal Clear* created by Alistair Cockburn herself [5]. The book is mainly based on expert interviews of development teams and project managers, conducted by Alistair Cockburn himself. The following descriptions of principles are taken from that book if no other reference is mentioned.

2.3.1 Frequent Delivery

It is important to have a strict delivery frequency and not fall into the temptation of extending deadlines since it can be demotivating for the team and jeopardize the future schedule. By ensuring that the team deliver new software on a regular basis the team learns how much it is able to deliver and can improve estimations. This also gives the developers a feeling of early victory. Frequent delivery is not only for the developers, it also aims to satisfy the customer by giving clear progress reports in the form of actual software. The delivery frequency is usually about 2 weeks but depending on the project and development team this can be either extended or shortened.

2.3.2 Reflective Improvement

> Did you get together at least once within the last three months
> for a half hour, hour or half day to compare notes, reflect,
> discuss your group's working habits and discover what speeds
> you up, what slows you down, and what you might be able to
> improve?
>
> ALISTAR COCKBURN [5]

This is similar to what was mentioned at chapter 1 Agile Development in the first paragraph. If you never stop, reflect and evaluate your work you don't know if what you are doing is really working. Agile development is about being open to changes but you have to know the worth of those changes. It does not have to be statistical assessment, it can be something as simple as just sitting down with some colleagues over a cup of coffee and discuss what is working and what is not. The important thing is to make it happen by allowing the team to do this on a regular basis. Crystal Clear is a lot about the social environment and trusts the team to make the right choices when given the right circumstances. Reflective Improvement is important because it allows the team to evaluate their decisions and improve for the future.

2.3.3 Osmotic Communication

This is an important principle for Crystal Clear. In fact, it is so important that it is the only principle that exist in all sub methodologies within the Crystal Family. Osmotic communication is the effect of having a team physically close together and picking up important information in the background chatter. So individuals holding lots of important information will, either consciously or unconsciously, gradually spread it out which is similar to the effect of osmosis. A simple way of ensuring osmotic communication is simply grouping the team in one single room, or the so called "war room", whose name might inspire people resisting leaving their private offices. For large teams this can be an logistic issue but an alternative is to have several rooms next to each other forming sub teams.

2.3.4 Personal Safety

This refers to something that most agree on is important but is generally hard to establish. To be able to give critique at any level of the company or project chain. The most important thing that is needed to allow this is to somehow remove the fear of reprisal. The members must feel safe and have trust in one another in order have the confidence to bring out weaknesses and problems in the project. Especially when weaknesses goes against the word of project leaders or managers. This means that it is important with good leadership to ensure Personal Safety. Simon Sinek, who is a leadership expert, stresses and explains this importance well in a TED Talk [6]. There he makes an example of how the militaries focus on great leadership

influence the soldiers and draws a parallel on the positive effects this would have on companies. In short terms, he brings up the hard to measure positive effects in having leaders truly and fully trusting the team and the team truly and fully trusting the leaders.

A way of ensuring trust, that leads to *Personal Safety*, between leaders and the members is to let the leaders have private discussions with each member. During these discussions the leaders try to make the member expose weaknesses, such as a failure to reach a deadline or any form of work-related information that they don't feel comfortable exposing. An experienced leader can then show a positive attitude and even gratefulness for the information. Furthermore, the leader can cover for the member and offer assistance to demonstrate that when the member revels a weakness or mistake, he/she will actually get assistance. Another tool is to have meetings where difficult problems are discussed where all members of the team are present, including the leader(s). This can lead to heavy argumentation, opening up to critique and hopefully showing that they can solve problems together that would not be easily possible alone.

2.3.5 Focus

> Do all the people know what their top two priority items to work on are? Are they guaranteed at least two days in a row and two uninterrupted hours each day to work on them?
>
> Alistar Cockburn [5]

As a developer, at some point you have come across the situation on working on some functionality, detecting some side problem or challenge, spending lots of time working on it, and finally asked yourself the question: *What am I doing?* As a programmer it is easy to find yourself in this situation of spending valuable time on non-valuable things. It is mainly the executive sponsors job to make sure that this happens as little as possible and the developers stay on the correct course, speaking about value to the project or company.

When working on several projects as a developer, it takes time to mentally switch between one project to another. For the programmer out there, you can draw the parallel to cache misses. Inexperienced managers underestimate this cost and keep assigning new projects to the same developers before the previous projects are finished. This is not only overwhelming to the developers and makes it hard for them to know what to work on, it is also motivationally destructive to regularly report the lack of progress on assigned projects. A simple repair is that the executive sponsor makes a clear prioritization list for each individual. For example prioritizing two projects or items much higher than the rest.

Focus time for the team members is also something important. It can be as simple as guaranteeing two full days work on a project before being allowed to switch or even look at another project. This limits the overhead of switching between projects, guaranteeing some progress to be made before any switch to another

project is made. Another level of practically ensuring focus time is to have a couple of hours each day where interruptions are forbidden, with the exception of certain critical scenarios. This can be very important because just like how project switching leads to overhead time waste, frequent interruptions from colleagues can create the same idle time. Especially when the interruptions are about matters that don't directly involve what the developer is currently working on.

2.3.6 Easy Access to Expert Users

> Does it take less than three days, on the average, from when you come up with a question about system usage to when an expert user answers the question? Can you get the answer in a few hours?
>
> ALISTAR COCKBURN [5]

Expert users can provide qualitative feedback about functionality and design and are generally a great source of information for the developers. Easy access to expert users is a good addition to Frequent Deliveries since it allows rapid and valuable feedback when deploying and testing newly added features. A study conducted in 1995, by Keil and Carmel showed the importance of having direct links to expert users and the positive effects it could have on a project [7]. Their research led to the recommendation to "Reduce Reliance on Indirect Links", as in rely more on access to real users. If a programmer has a design question and it takes several days before a reply comes, from an expert user, it can lead to delays or the programmer making decisions based on his own assumptions. This time delay between questions and answers can have grave impacts on the project. Some practical ways of ensuring Easy Access to Expert Users is:

Weekly or semi weekly user meetings with additional phone calls
Pretty straight forward, allow communication to flow freely between expert users and developers with regular weekly meetings. It can be as little as one or two hors per week. It is advisable to have the meetings more frequent in the early phases of the project. The availability of phone calls is a good addition for those extra important questions that needs a quick reply.

One or more experienced users directly on the development team
This can be hard to achieve and is rarely used since it highly depends on the availability of the expert user. However, if it is possible it is perhaps the optimal way of providing expert user feedback.

Send the developers to become trainee users for a period
This is sort of converting developers to expert users. This leads to the developers seeing the project from a different perspective and can limit the need of having regular feedback from actual expert users.

2.3.7 Technical Environment with Automated Tests, Configuration Management & Frequent Integration

Neither of these technical environment elements are crucial for a successful project but they can make the developers lives easier.

Automated Testing
Many projects do just fine using manual testing. However every programmer, that had switched from manual testing to automated testing, that Alistair Cockburn interviewed swore to *never to work without them again.* The benefits are apparent when several people work on the same code on different days. The next developer can just run the automated tests the day he/she starts and be confident that nothing is broken. It gives the developers a freedom to change without worrying about whether or not something got broken along the way. For those with no prior experience with automated testing, the X-unit framework is advisable to take a look at. There is a version of X-unit for most languages and they share the name ending "unit" while the "X" is replaced with something correlating to the language, like Junit for Java.

Configuration Management
The configuration management system gives a structure to the project that allows asynchronous work, revert changes, select a specific configuration for release and backup to a stored stable configuration when problem arises. You can loosely say that the configuration management system is to an entire project as Git is to software development. According to Alistair Cockburn, configuration management is often cited by development teams to be the most critical non-compiler tool.

Frequent Integration.
Frequent integration leads to easier detection of bugs and also fewer bugs/problems that surface simultaneously. The added code is more fresh in the developers minds and smaller in size, thus leading to fewer hiccups when checking the code to be integrated. It is advisable to integrate as frequent as possible. It is good to be able to integrate several times a days. If that is not realistically possible then once every day is not bad either. When several days goes before integration the problems starts to stack up.

2.4 Kanban

When reading about Kanban it can get quite confusing. This is because Kanban is originally a lean scheduling system for just-in-time (JIT) production developed by Taiichi Ohno at Toyota in 1953 [11]. It is still used today within manufacturing but the ideas have been taken and modified to fit software development. The confusing part is that this software development version of Kanban is called Kanban. If not clearly stated otherwise, future referencing to *Kanban* in the report refers to the

software development version of Kanban. A fun fact is that "kan-ban" is Japanese for "signal card" [11]. Some disagreement regarding Kanban is weather it is more of a lean or more of an agile methodology. The easy answer to this is to simply ignore the question since lean and agile methodologies don't go against each other nor have precise definitions as explained in section 1.5 Agile vs Lean. One thing most can agree on is that Kanban is both lean and agile to some extent.

As David J. Andersson stated in his first book about Kanban, the methodology is lightweight and does not have a clear formal definition in order to promote changes and modifications [11]. This has led to several alternative methodologies, that build upon the original, popping up such as Kanban Ace and LKU Kanban. For these reasons, it is not that strange that the core practices can vary slightly in both numbers and definition. Depending on where you read you might find three or seven practices. For this report the chosen core practices are taken partly from the book *Kanban: Successful Evolutionary Change for Your Technology Business* by David J. Andersson [11] and partly from the overall consensus from various resources available online.
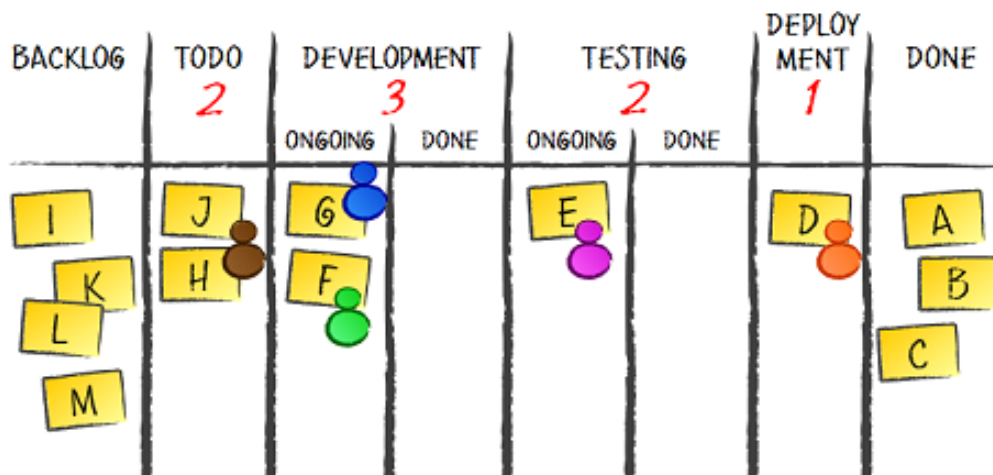

Figure 2.2. Image showing the basics of a Kanban board. The cards move from left to right. The number under each category represent the WIP limit.
Source: Pawel Brodzinski, http://brodzinski.com/

2.4.1 Visualize

This practice is about visualizing workflow. To have transparency between developers, managers and anyone else connected to a project. The standard way of doing this is using a so called Kanban board with cards, see Figure 2.2. This board can either be a physical board with sticky notes, or a virtual board. This board is a visual control system for all the tasks within a project. The board allows for self-organizing by just pulling a new available card after finishing a task instead of waiting for a superior to supply the tasks.

The cards are not only meant to simply hold programming tasks. They are meant to represent some customer value. Therefor the design of the cards is important in order for the developers to make easier decisions on what to work on next. There are no exact rules on what information the cards should hold but a title, description, estimated size and some estimated value number is a good start [11].

2.4.2 Limit Work-in-Progress

Limiting WIP is about having a limit at each step in the product chain, shown in Figure 2.2, that corresponds to the amount of tasks allowed in that step. A simple WIP limit is having it equal to the amount of available working people at each step. When deciding the WIP limit, all members from from the stakeholders to the developers should take part and form an agreement [11].

The WIP limit is important for several reasons. It shows when a pull from one step can be made to the next. It ensures focus on one or a few tasks at a time. It keeps the workload balanced and avoids stock piling of items at some middle step in the product chain. The most important benefit is that it gives the development team a limit to rely on and fall back to when put under too much stress, especially if everyone participated and formed a consensus when choosing the limit for the WIP.

2.4.3 Manage Flow

In Kanban the workflow or task flow is central so it makes sense having a practice focus on how to manage and improve it. In order to improve the workflow one must first be able to measure it somehow. Otherwise when changes are made, how do we know that it was an improvement? Some things are easier to measure and improve such as items stock piling at a certain step. An easy fix would be to add more manpower to that particular step. Some software such as *Kanbanery* offer automatic tools for measuring that can give useful information about how changes affect the workflow [12]. Something important to stress out is that the more precise the estimated complexities and values for the cards/tasks the easier it gets to analyze the workflow.

2.4.4 Make Process Policies Explicit

In the practice Visualize the importance of having a clear design on the cards of the Kanban board was mentioned. This practice is about just that. Without clear rules and a common understanding of how the card system works the discussions becomes subjective and emotional instead of factual [13]. Having the process policies explicit makes them easier to understand and therefor easier to use. They help choices regarding the process become more rational and empirical and the team have an easier time reaching consensus simply because everyone is working from the same policy configuration.

The process policies are rules of the Kanban board. It can state the type and amount of data that each card should contain. It explains the flow of cards traveling across the board. It can contain default implementation details about different classes of cards. It is basically the defined structure of the Kanban board and its cards.

2.4.5 Improve Collaboratively

> If you are not continually improving, but you are doing all of the other parts of the Kanban method, you are missing the point. It's a little like the concept of "doing" agile but not being agile.
>
> MATTHEW POWELL [13]

This practice is more social and less practically oriented than the others. When reading about Kanban you sometimes stumble upon the word *Kaizen* meaning continuous improvement. *Kaizen culture* is also often mentioned that talks about empowering the workforce and encourage them to "do the right thing". Kaizen culture promotes high levels of collaborations and freedom to self-organize work. It is about putting the company and business first and this requires lots of trust which Simon Sinek talk about in his TED Talk about leadership [6].

Kanban promotes using a scientific approach when trying to introduce new changes. That means to evaluate a situation, suggest suitable change, predict the outcome, observe the results and compare with the prediction and previous results. If the team or business have a Kaizan culture together with using the scientific approach then that can lead to spontaneous and regular local improvements to the process [11].

2.5 Other Mentionable Methodologies

The informed reader might wonder why only these four methodologies were chosen for the report. After all there are at least a dozen more agile methodologies worth mentioning. For instance Dynamic Systems Development Method (DSDM) that is a broadly used agile methodology.

The reason for the chosen methodologies are several. First of all we must acknowledge that going through every possible agile methodology is simply not possible. Kanban was chosen since it is both broadly used and very lightweight, making it a perfect choice for testing agile methodologies in small development teams. Crystal Clear was chosen as it clearly tries to address small development teams in its definition and there are not many agile methodologies that does so. Scrum is the most popular agile methodology today and therefor deserves a spot. XP was chosen since it is popular but still quite different from the other methodologies, making it interesting for comparison reasons.

DSDM was skipped simply because its scope is too large, larger than Scrum for example. This makes it impractical for this report since it talks about business cases and sponsorship which might not be the number one priority for small development groups. Feature-Driven Development (FDD) is another popular methodology but it has focus on big projects by discussing things like automated build systems. Some other methodologies are Adaptive Software Development (ASD), Scaled Agile Framework (SAFe), Disciplined Agile Delivery (DAD), Projects in Controlled Environments (PRINCE2) but they were all skipped for similar reasons, or lack of documentation.

Part II

Method

The method is divided in two chapters. The first chapter theoretically evaluates the agile methodologies in how well they would suite small projects and/or development groups. The project specification is also described here. At the end of the chapter a methodology is selected for the project. The second chapter is about the project workflow. It is a diary like evaluation of the selected agile methodology and the work progress.

Chapter 3

Theoretical Evaluation

This chapter aims to compare the four agile methodologies to each other with focus on small development teams. There is no exact definition of what a "small project" or "small team" means. For the sake of clarification we assume that a small project consists of one to five person(s). They can either all be part of the development team or one or two have other responsibilities such as being a costumer, project leader or product owner. We further assume that the small team is a newly founded IT company and not some sort of small project team within a larger company. For example, it could be a handful of friends starting an IT consulting business after getting their master's degree in computer science. So whenever a small company, small project, small team etcetera is mentioned from here on, this is the setup you should have in mind.

3.1   Methodology Walktrough

3.1.1   The Agile Principles for Small Teams

The agile principles are the backbone of agile development. Therefor it is only natural to compare agile methodologies to these principles in order to estimate how agile they are and what their differences are. However the principles aim to satisfy big projects and big teams since after all, the bigger the project the bigger the need for a methodology. Even if it is hard valuing the agile principles, or not even meant to by the founders and followers, it does make sense to reestimate their value for small newly started IT companies. The following evaluation of the principles is just an estimation based on logical reasoning and should be read with a critical mindset. The principles are listed at section 1.3 The Agile Principles.

The first three principles regarding satisfying the customer, welcoming changing requirements and delivering software frequently feel equally important no matter the size of the project team. The fourth principle however, about business people and developers working together daily, is notably less important to satisfy since in newly started small companies they tend to be the same people. The fifth principle lowers greatly in importance since it is clearly focused on bigger companies where you can

33

choose between different teams and people and give them projects to work with. As a newly started small company you kind of have to build the project around the hopefully motivated individuals. The only time face-to-face communication, principle 6, lowers in importance would be when you found a company or work on a project completely alone. Principle 7, 8, 9 and 10 all talk about keeping code quality high over time where the size of the project/company should not matter. Principle 11, which is about self-organizing teams, is pretty much self-fulfilling in a small new company. The last principle, number 12, is about team reflection and should be important in any sized project.

To summarize, the 9 principles 1-3, 6-10 and 12 comes out as more important to satisfy than the rest for small new companies/teams. This quick purge of a few principles will help keep a better focus when evaluating the methodologies for small independent projects.

3.1.2 Scrum

The practices of Scrum satisfy the agile principles well. You can almost match each practice in Scrum to each different agile principle and perhaps this is why Scrum is often used as the example methodology for agile development. The only principle that it does not address is principle 9 that talks about technical excellence and good design. Good code design is only briefly mentioned when talking about getting tasks done before the end of each Sprint and not leaving loose uncompleted ends. The methodology clearly revolves around the Sprints. It talks a lot about having short iterations with many short meetings and ensuring that the Product Backlog is kept updated at all times.

Everything in Scrum goes together pretty neatly as seen in Figure 2.1. Each practice is strongly connected to several of the others. Within the events, all three roles work together tightly and always have the artifacts in focus. Generally this is a good thing since it shows coherence and makes it easier to grasp the meaning of each practice. However to be able to include all practices one would at least need three people to fill the roles, which would mean a Development Team of size one to three for small companies. Another solution would be to have everyone part of the Development Team where two of these have extended roles as a Product Owner and Scrum Master. It is certainly possible to include all practices for a small project but it would also most certainly lead to lots of overhead work when so few people are involved.

3.1.3 Extreme Programming (XP)

XP is quite opposite of Scrum and talks quite a lot about code quality. The practices Pair Programming, TTD, Coding Standards, Simple Design and Refactoring all emphasizes how they have a positive influence on code quality. At first glance one might believe that XP is much more "zoomed in" than Scrum and mostly talks about how the Development Team should code. However, the practice called The

Planning Game is very big and includes components that are similar to several practices in Scrum, such as iterations (Sprints), small stories (Product Backlog) and planning meetings (Sprint Planning). So one could also see XP as a sort of huge framework where Scrum is replaced by The Planning Game. So for companies using both Scrum and XP it's a matter of perspective weather they are using XP on top of Scrum or Scrum on top of XP.

Since The Planning Game together with Small Releases is loosely put another version of Scrum, XP satisfies the same practices that Scrum satisfies. The principle that Scrum did not satisfy, principle 9 regarding code quality, is quite clearly satisfied by many of the practices in XP. Seeing how focused XP is on code quality it is easier to understand the critique it receives regarding being called agile. After all, agile development is usually talked about in respect to cross-functional teams, short planning and regular meetings across the entire company hierarchy.

XP as a whole is quite clearly too vast for small newly founded IT companies. But they do have many stand-alone practices that would be easy to start with and then slowly built upon, practice by practice. For example, all the quality ensuring practices mentioned. However, how agile would it be to have a bunch of practices satisfying only one agile principle? To ensure that the other parts of the agile principles are included one would have to look into the Planning Game. But instead of diving into that vast practice it would make more sense starting with Scrum where the different parts of The Planning Game are more clearly distinguished.

3.1.4 Crystal Clear

Ironically the practices of Crystal Clear is the most vague of all four methodologies. They have chosen a higher abstraction on their practices, somewhere in between the agile principles and the practices of the other methodologies. Instead of describing actual tools to achieve positive outcomes it focuses on describing positive outcomes with tools as examples. Half the practices, Osmotic Communication, Personal Safety and Focus, all talk about social behavior in group environments. The methodology revolves around building safety and trust within the team. In other terms, it has focus on the human mind instead of artifacts and processes.

Since it has this high abstraction level it is hard to say which agile principles are touched. However the practices line of argument means that the more social principles 4, 5, 6 and 8 are being satisfied or are at least are in focus.

Crystal Clear is clearly more lightweight than XP and Scrum which naturally means it would be an easier methodology to follow for small companies. To try to establish a sort of trust oriented culture early in a company, Crystal Clear is certainly a good candidate to follow. However it somewhat lacks practical practices to follow so it might be a good idea to complement it using another methodology or strategy. The methodologies Frequent Delivery and Reflective Improvement is basically the Sprint and Sprint Retrospective so Scrum would be a good candidate to pair it together with.

The last principle, that certainly is more practical than the others, feels a bit

malplaced. The way it is brought forward together with its content compared to the other practices, makes it sound more like general good coding tips than a practice within Crystal Clear.

One thing you can safely say is that Crystal Clear put heavy emphasize on the part of the The Agile Manifesto that states: *Individuals and interactions* over processes and tools.

3.1.5 Kanban

Kanban is the most lightweight of the four methodologies. Like Scrum has a clear focus on the Sprints, Kanban has a clear focus on the Kanban board. It is only one of the five practices that does not revolve around the Kanban board, the practice Improve Collaboratively. It is so lightweight that it is arguable that Kanban is more of an agile/lean strategy than a complete methodology. As food for thought, the practices could be simplified and replaced by one practice called the Kanban Board. So it is not that strange that it is often mentioned when reading literature about Kanban that the methodology is meant to be expanded as time progresses and the project/company grows.

Kanban certainly does not fully satisfy all the agile principles but the practices do nibble on almost all of them. For example principles 1-3 each have its own practice in Scrum that tries to satisfy them directly. In Kanban these are all incorporated though the Kanban board but in a more subtle way. Like with the other methodologies, except XP, Kanban has no real focus on code quality. Also even if the Kanban board is about visualizing workflow for every member of a project it does not talk much about the collaboration between business people and developers. This means that principles 4 and 9 are left untouched. Other than that, Kanban seems to do have a covering but lightweight spread when it comes to satisfying the agile principles.

Kanban seems like a suitable choice to start with for a small new company looking for an agile methodology. It does a decent job covering the agile principles, it is relatively lightweight and most of the practices are easy to practically follow. However as a company/project grows it can perhaps become too lightweight and should be expanded or complemented using another methodology. An example of this is the methodology Scrum-ban which is a combination of Scrum and Kanban.

3.2 Chosen Methodology for the Project

Kanban became the chosen methodology for the project, see subsection 4.0.1 Project Description. There are several reasons for the methodology to be favorable for the project. The methodology covers most of the agile principles but is extremely lightweight compared to Scrum and especially XP. For a small project this is essential since the smaller the project team the larger the overhead for a methodology. And in this particular project there will be just one developer and the owner of the project. The fact that Kanban is meant to be expanded on demand and that

methodologies like Scrum-ban exists also means that it should be easy to let the methodology grow as the project/company grows.

The Kanban board is easy to understand and implement. It will give value even if you are developing completely alone since it provides structure and abstraction. It also offers transparency for the project owner. The freedom of details on the Kanban board and its cards through the process policies means that it easy to start small and slowly buildup the organization of the board.

Regarding Crystal Clear, the focus on social environment makes it a pretty poor choice when working solo as a developer. Even if you are a handful of developers it can be hard to practically follow such abstract ideas. The methodology also does not have the same connection for its practices as the other methodologies.

Chapter 4

Project

This chapter goes through the project and specifically the workflow of the project that was and, at this time, is still being developed using Kanban. The chapter tries to keep a natural timeline but not to a full extent since many logical components of the development process were themselves developed over time. Figure 1.2 describes the problem pretty well. In other words, each component individually, like creating the Process Policies, follow a logical timeline.

Since the project is a private initiative by the product owner, the developer works from his home. Tuesday at 10:00 was initially selected as a weekly meeting time to ensure that the developer and product owner regularly see each other face to face. During the course of the project extra meeting times were added, especially at the start of the project to get the ball rolling.

4.0.1 Project Description

The project, to be developed using agile strategies, is a private initiative from Per-Arne Forsberg who has 25 years of successful international experience and knowledge in managing technically skilled units, products and projects around the world at Ericsson. Having been involved and engaged in change management, using the latest Lean and Agile methodology.

The project is developing a dynamic, scalable and robust interactive tutoring framework prototype for pre-academic students. Assisting students today according to the "old school" is social but also ineffective since it is usually one to one communication or one to a few in some geographical location (you have to meet at pre-determined location at pre-determined times, like classrooms). The long term goal is to assist in communication between students and teachers (tutors willing to help out with their knowledge from their preferred location) in Sweden and in turn combat the negative (knowledge) grade curve that can be seen around the country, according to international studies, PISA.

In short the idea is to create an interactive website that can be used for many to many type of communication between students and teachers. An exact specification of the project does not exist in order to encourage the agile development process.

4.1 Process Policies

that was looked into, after deciding on using Kanban, was the layout of the Kanban cards, in other words the Process Polices. Typically these cards are either simply tasks and/or user stories. Since Kanban itself does not explicitly state the layout of the cards, the strategies described in Use-Case 2.0 were selected to form the basic structure [18]. This will be described further in subsection 4.1.1 Use-Case 2.0.

The Process Policies were not all written at one point in time. As the project grew more policies were added. Using
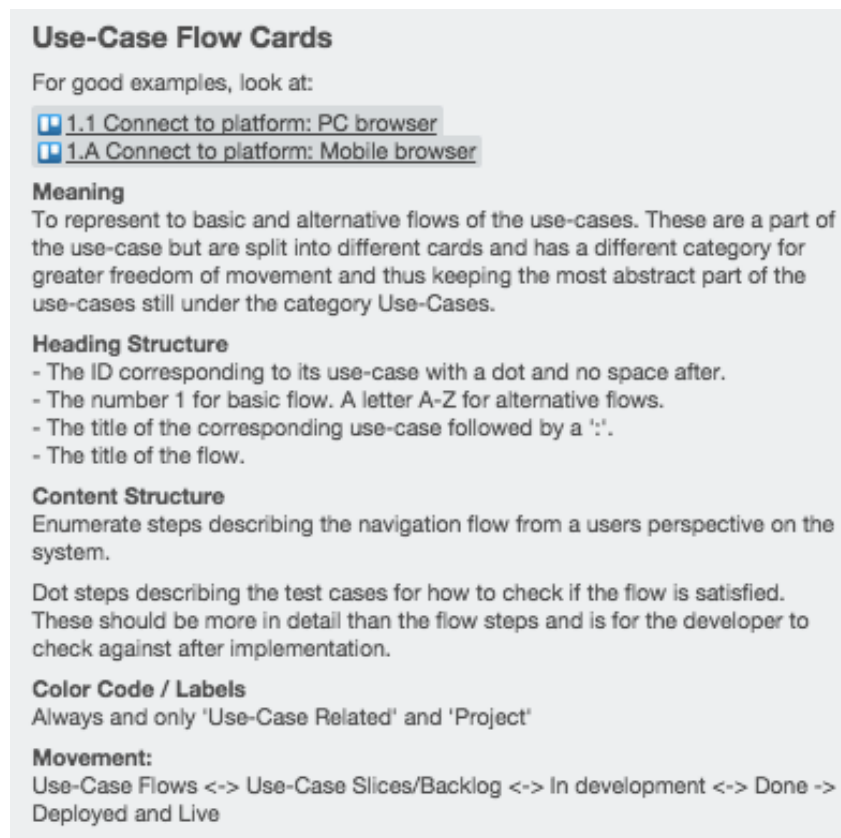

Figure 4.1. A sample of the projects Process Policies. This sample shows information regarding the type of cards called Use-Case Flow Cards. For a full list of the Process Policies, see Appendix A Project Process Policies

The first thing

4.1.1 Use-Case 2.0

User stories is a common approach in agile development to represent project requirements and specifications in a easy to understand, abstract and layered structure. A use-case is a similar alternative approach. The difference is in the scope and

its focus. While user stories tend to be more abstract and use everyday language to keep it open for interpretation, use-cases instead are more user-to-system goal focused and tend to use a more business oriented language. Use cases also tend to have larger scope that includes components such as unified modeling language (UML). It is common that project requirements are built using both user stories and use-cases techniques in some sort of combination. What's Important to know is that the definition for both of these techniques are not strict in order to have an easier time adapting them to any project type and size.

Use-Case 2.0 is a popular technique meant to be lightweight, scalable, versatile and easy to use [18]. If it delivers what it promises it sure fits the description of what you look for in a small project with few members. Also the technique is meant to be used with methodologies such as Kanban and Scrum. Without going into to much detail the technique, when used with Kanban, can be summarized as following:

1. Agree on the goals and scope of the system and identify all actors and ways of using the system. Repeat this every now and then as things can change as the project develops.

2. Each use-case represents a meaningful logical interaction between some actor and the system.

3. Create a narrative for each use-case that explains the actors goal and interaction on an abstract level and in everyday language, similar to how user story are written.

4. Create basic and alternative flows for each use-case that in a step-by-step manor goes through a users perspective in navigating the system to achieve the use case goal.

5. Create basic test cases for how to check if the use-case and use-case flows are satisfied.

6. When development on a use case grows "near", create slices for the use-case. As the name suggests, try to create each slice so they correspond to a suitable and somewhat equally sized work. The slices are created by either splitting a use-case flow or combining multiple flows. The slice can also go deeper in detail if wanted or needed, both in terms of the use-case flow steps and test cases.

7. Work on a certain number of slices at a time (depending on the size of the project team) and when implemented, check them against the test cases.

8. Keep updating the use-cases with information as slices are completed.

Note that this is a simplified summarization of Use-Case 2.0 that is influenced by both Kanban and the project. Read more about how to technique works in in the e-book *Use-Case 2.0, The Guide to Succeeding with Use Cases* by Ivar Jacobson, Ian Spence and Kurt Bittner [18].

4.2 Kanban Board

After deciding to use Kanban together with Use-Case 2.0, about five longer meetings corresponding to almost three full work days were spent on brainstorming the use-cases for the project. To let each case represent a card, post-its notes were used on a blackboard.
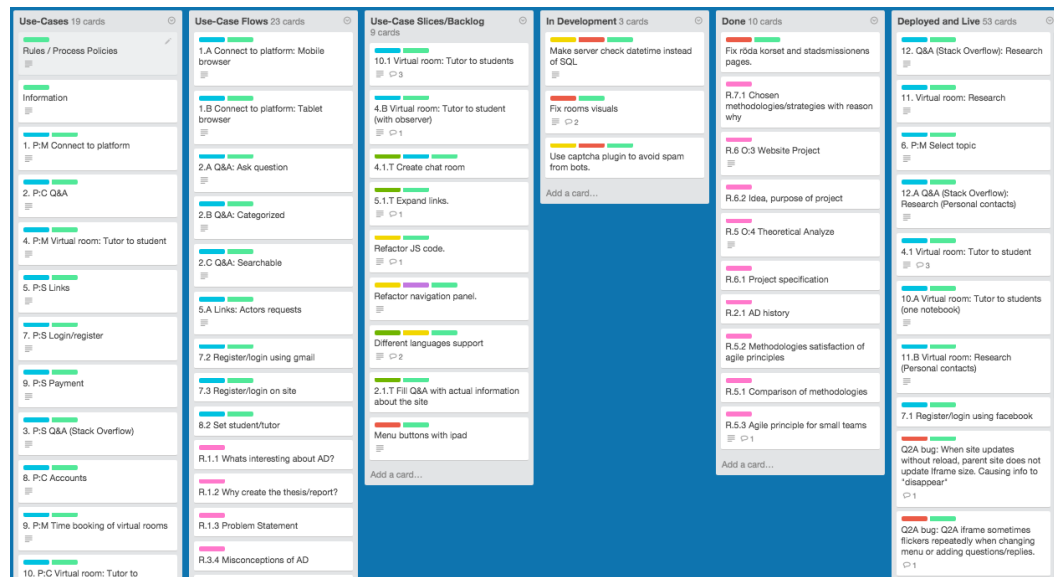

Figure 4.2. Trello Kanban board.

4.2.1 Trello

4.3 Agile Workflow

4.4 Project Result

4.5 Developer & Product Owner Experience

Part III

Discussion

The following chapters will be discussions and final toughs regarding the results, future work, report critique and more.

Chapter 5

Discussion

Chapter 6

Conclusions

Chapter 7

Report Critique

There are several reasons to be critical for the type of research conducted in this report. Some have already been mentioned briefly here and there throughout the report. First of all agile development by itself is quite vague in its definitions and the work regarding agile development and its methodologies are mostly based on expert experience. It is the results of industry experts that have seen problems, come up with ideas to fix them, had companies or themselves try them out finally evaluate the outcome. The problem is that there are so many parameters and influences when evaluating software development methodologies that it is really hard to make sense of the result. Too be clear, the data is very noisy. For instance, you can not make a software using one methodology and then have the development team create the software again using another approach. The results of the new approach will most certainly be influenced by their first attempt. What is really needed is several development teams consisting of developers with somewhat equal experience and background. However, depending on parameters such as if the software type is a website, game or security system the results might vary. Thus most work regarding agile development and its methodologies, including this report, falls into the category of social science or soft science which is important to have in mind. All of this also means that investigating work regarding agile development, such as this report, would benefit in having a more experienced author but this is usually hard to come around when we talk about a university thesis for obvious reasons.

Each agile development methodology has vast amount of information, interpretations and alternative versions going for them. To make a full theoretical analyzes and comparison is out of scope for this report. The first and easy limitation was to not include every possible agile methodology in order to have room for more depth into the chosen methodologies as well as to keep the complexity down and give the readers some focus. However, even with this limitation it would not be viable going through every aspect of each methodology since each methodology would deserve their own full report. So the practices of each methodology was chosen as a solid and common ground to be analyzed. What this means is that a lot of what defines the methodologies have potentially been "cut off" and therefore given an unfair

assessment to some degree. This is also brought up in section 8.1 Future Work.

Chapter 8

Developers Thoughts

8.1   Future Work

Glossary

Early Victory A strategy commonly referenced to in agile development that refers to the sociological benefits of showing early and frequent results, such as increasing morale and trust. 23

Executive Sponsor Also referred to as the project sponsor. Executive sponsor is a role in project management that is responsible to the business/company for the success of a project. 25

Git Git is a popular distributed revision control system mainly used for software development. 19, 27

JIT Just-in-time is a production strategy that seeks to improve the ROI for a business by reducing in-process inventory and associated carrying costs. In simpler terms, reducing the stock up of inventory items in a process chain. 27

Osmosis The movement of a liquid, sometimes gas, through a plasma membrane (which is a sort of filter) which over time results in a balancing from high concentration areas to low for the molecules that are able to pass through the plasma membrane. 24

ROI Return on investment means the benefits, or return, for the investor resulting from an investment of some resource. High ROI. 14

Runaway Process Inflation Runaway inflation is a very rapid inflation (decrease of money value). Runaway process inflation is a term sometimes used within agile development to describe rapidly growing development processes. 3

Stakeholder Has several meanings but this report refers to corporate stakeholders. A corporate stakeholder is a broad term meaning a person or party who has an interest in a certain business or project for a company. It can for example be investors, customers and suppliers. 15, 29

Technical Dept Technical dept (also called design dept or code dept), is a metaphor primarily used in software development referring to the consequences of poor

system design. The dept builds up as you for example introduce bugs and so called spaghetti code that has to be fixed in the future, as in repaying your dept. 3

User Story Within software development and product management, user stories are short descriptions, or stories, that tries to envision the users interaction for some requirement. The layout of user stories varies for different projects but It should capture the "who", "what", and "why" in a short but descriptive way. Usually the stories are highly abstract but before implementation on a story begins they are broken down into more fine graned sub-stories. The term is commonly used within agile development and originates from the methodology XP. 18

Acronyms

GUI graphical user interface. 21

JIT just-in-time. 27, *Glossary:* JIT

LPD lean product development. 9

LSD lean software development. 9, 10

ROI return-on-investment. 14, 49, *Glossary:* ROI

TTD test-driven development. 17–19, 34

WIP work-in-progress. 28, 29

XP Extreme Programming. 17, 19–23, 30, 34–36

Bibliography

[1] Martin, C.R. and Martin, M. 2006. Agile principles, patterns, and practices in C#. Prentice Hall, Massachusetts. 768 pp.

[2] Agile Alliance. 2015. The alliance. [http://www.agilealliance.org/the-alliance/]. Accessed Mars 5, 2015.

[3] Crockford, C. 2007. Quality. [https://www.youtube.com/watch?v=t9YLtDJZtPY&t=175]. Accessed Mars 12, 2015.

[4] Knuth, D.E. 1984. Literate Programming. Comput J. 27: 97-111.

[5] Cockburn, A. 2004. Crystal clear: a human-powered methodology for small teams. Addison-Wesley Professional, New Jersey. 312 pp.

[6] Sinek, S. 2014. Why good leaders make you feel safe. [http://www.ted.com/talks/simon_sinek_why_good_leaders_make_you_feel_safe]. Accessed Mars 13, 2015.

[7] Keil, M. and Carmel, E. 1995. Customer developer links in software development. Commun ACM. 38: 33-44.

[8] Williams, L., Kessler, R.R., Cunningham, W. and Jeffries R. 2000. Strengthening the case for pair programming. IEEE Softw. 17: 19-25.

[9] Beck, K. and Andres, C. 2004. Extreme programming explained: embrace change, 2nd edition. Addison-Wesley Professional, New Jersey. 224 pp.

[10] Scrum Guides. 2013. The scrum guide. [http://www.scrumguides.org/scrum-guide.html]. Accessed Mars 23, 2015.

[11] Anderson, D.J. 2010. Kanban: successful evolutionary change for technology organizations. Blue Hole Press. 278 pp.

[12] Klipp, P. 2014. Getting started with kanban. Kanbanery. 37 pp.

[13] Everyday Kanban. 2015. What is kanban. [http://www.everydaykanban.com/what-is-kanban/]. Accessed Mars 25, 2015.

[14] Schwaber, K. 2004. Agile project management with scrum. Microsoft Press. 192 pp.

[15] Womack, J.P., Jones, D.T. and Roos, D. 2007. The machine that changed the world: the story of lean production. Free Press, New York. 352 pp.

[16] Poppendieck, M. and Poppendieck, T. 2003. Lean software development: an agile toolkit. Addison-Wesley Professional, New Jersey. 240 pp.

[17] Fowler, M. 2008. Agile versus lean. [http://martinfowler.com/bliki/AgileVersusLean.html]. Accessed April 1, 2015.

[18] Jacobson, I., Spence, I. and Bittner K. 2011. Use-case 2.0. Ivar Jacobson International. 55 pp.

AppendixAppendix

Appendix A

Project Process Policies

The Process Policies for the project are written in the online organization tool called Trello. For the time being they can be accessed using the following link: https://trello.com/c/Xtm9s8LF/95-rules

However since the project is ongoing the information will change and the link might become invalid in the future. A sample of how to Process Policies are written in Trello can be seen in Figure 4.1. The Kanban board in Trello is visible at Figure 4.2. Everything below is an extraction of the Process Policies at the time of this reports writing, slightly modified to suit the layout of this report.

A.1 Project Process Policies

Strategies in Use-Case 2.0 are integrated with the Kanban board.

A.1.1 General Kanban Board Policies

- Normally the cards move from left to right on the board. Each card type has more information on this below.

- The categories Use-Cases and Use-Case Flows are only for the Use-Case Cards and Use-Case Flow Cards.

- The Use-Case Slices/Backlog is the "To-Do" list that a developer can grab and move to In Development.

- Everything in In Development should be something being worked on by a developer. If the developer pauses work it should move back to the Backlog and if the work is done it should move to Done before grabbing a new card.

- Cards in Done await inspection before being moved back to the Backlog or being decided to put the code live.

- Cards only move to Deployed and Live from Done. The cards must also of course be deployed and live.

A.1.2 Use-Cases Cards

For a good example, look at:
    https://trello.com/c/JvD4oEyF/47-1-1-connect-to-platform-pc-browser

Meaning
> To represent to most abstract part of the use-cases whose content could change greatly over time.

Heading Structure

1. Use-case ID, a unique number followed by a dot.

2. Priority following MoSCoW written "P:X" where 'X' is either M (Must), S, (Should), C (Could), W (Would).

3. Title name of the use-case.

Content Structure
> Begin with the narrative and use everyday language. What does the actor want to achieve? What is the goal of the use-case? Make sure it is clear who the actor is.
>
> Leave a heading for the basic flow and all alternative flows, if they exist. Link each flow to the card corresponding to that particular flow.
>
> Add any possible notes that are not related to the narrative in the end. Comments can also be used for notes.

Color Code / Labels
> Always and only 'Use-Case Related' and 'Project'

Movement
> Use-Cases <-> Done -> Deployed and Live
>
> The card sit still in Use-Cases and only move to done when all use-case flows corresponding to the card are moved to Done or Deployed and Live. Once there the decision is made weather or not it is truly satisfied and can either move back or to Deployed and Live. The card can also be Archived if decided to be unwanted.

A.1.3 Use-Case Flow Cards

For good examples, look at:
    https://trello.com/c/JvD4oEyF/47-1-1-connect-to-platform-pc-browser
    https://trello.com/c/4krVGIwA/48-1-a-connect-to-platform-mobile-browser

Meaning
> To represent to basic and alternative flows of the use-cases. These are a part of the use-case but are split into different cards and has a different category

for greater freedom of movement and thus keeping the most abstract part of the use-cases still under the category Use-Cases.

Heading Structure

1. The ID corresponding to its use-case with a dot and no space after.

2. The number 1 for basic flow. A letter A-Z for alternative flows.

3. The title of the corresponding use-case followed by a ':'.

4. The title of the flow.

Content Structure

Enumerate steps describing the navigation flow from a users perspective on the system.

Dot steps describing the test cases for how to check if the flow is satisfied. These should be more in detail than the flow steps and is for the developer to check against after implementation.

Color Code / Labels

Always and only 'Use-Case Related' and 'Project'

Movement

Use-Case Flows <-> Use-Case Slices/Backlog <-> In development <-> Done -> Deployed and Live

A.1.4 Use-Case Slices Cards

For a good example, look at:
https://trello.com/c/Dpd63Jy0/114-5-1-t-expand-links

Meaning

These cards are Use-Case Slices or Use-Case Tasks that represents a part of a Use-Case Flow. It can either be a few exact steps of the flow, a task needed to satisfy the flow or a small addition to a flow deserving of its own card.

Heading Structure

1. The entire Use-Case Flow ID followed by no space.

2. The letter 'T'.

3. Title of the slice, or task.

Content Structure

As you wish. The purpose of the card should be clear. However if the card is a strict part of some steps of a flow. Those steps should be written down.

Color Code / Labels

Always 'Project'. Always 'Use-Case Related' if it is a strict subset of a Use-Case Flow. Never 'Report'. The rest are optional.

Movement

Use-Case Slices/Backlog <-> In Development <-> Done <-> Deployed and Live

These cards can be directly put to 'In Development'.

A.1.5 Task Cards

A good example:

https://trello.com/c/5Mrp0crr/121-refactor-navigation-panel

Meaning

A task card does not directly correlate to a use-case and its flow(s). It can be a small feature, bug, design improvement etcetera. Basically anything that can add some form of value but does not fit logically to a use-case. If the complexity of card is large it should perhaps be converted to a Use-Case card.

Heading Structure

Just a fitting title of the task.

Content Structure

As you wish. Be as detailed as possible and needed.

Color Code / Labels

Always 'Project'. Never 'Report' or 'Use-Case Related'. The rest are optional.

Movement

Use-Case Slices/Backlog <-> In Development <-> Done <-> Deployed and Live.

These cards can be directly put to 'In Development'.

A.1.6 Report Cards

Meaning

These cards are for the report writer and only he/she knows its meaning.

Heading Structure

As the report writer pleases.

Content Structure

As the report writer pleases.

Color Code / Labels

Always and only 'Report'.

Movement

As the report writer pleases.