Product Verification and Validation during Software Development

Andreas Slott Jensenius

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The software development process of an organization can play a vital role for the success of the organization. Choosing a methodology or choosing not to use a methodology, can be a critical choice. An important reason for using a software development methodology in the first place is to be able to deliver products on time and budget. However, some methodologies does not include answers to the two questions "Are we building the product right?" and "Are we building the right product?", which is known as verification and validation respectively. Instead this is left up to developers to obtain these goals with no clear guideline or structure. Other methodologies focus on other areas, which might be verification and validation related, but they are then restrictive in their use and not widely applicable. The nature of software products is that they are used everywhere, and this large variance is also reflected in the large variance when it comes to different projects and teams. Many different methodologies exists - most with either niche uses, or too general to provide meaningful contributions to all three aspects of development - being on time and budget, verification, and validation.

This project aims to develop a methodology that encompasses both getting the product built on time and budget, but also focus on the questions of verification and validation, while being adaptable to a large range of projects and teams.

To accomplish this goal, state of the art methodologies was investigated and taken apart to understand their individual practices, what areas they are challenged in, and where they excel. Scrum was chosen as the basis of the framework, both for it's wide usage in the industry but also for being flexible and light on restrictive, inter-dependent practices.

The understanding from this literature study is then used to develop a framework for software development, that allow it's users to extent the core of the framework using elements in a provided toolbox to suit the needs of each individual team and project. The practices within both the core and toolbox of the framework are all aimed to deliver software products of sufficient quality, built to provide value to the customer, and on time and budget.

The developed framework is tested using a real project provided by IT Minds. The results of this test has been mostly positive, with some suggestions and improvements suggested both by the author, the project owner, and a project lead from IT Minds. The major selling points of the framework are it's information gathering process, which allows developers to really understand their customer, and the adaptability to many different projects- and team types which the toolbox provides.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering under the study line of Information Technology.

The thesis deals with different methodologies for software development, and the development of a new methodology which aims to deliver software products on time and budget, which provides value to the customer, and are built of sufficient quality.

The thesis consists of this report and source code for the developed test cases and code, which was made using the new methodology created in this report.

Lyngby, 03-July-2015

Andreas Slott Jensenius

Acknowledgements

I would like to sincerely thank my supervisor at the Technical University of Denmark, Christian W. Probst, for his help, guidance and advice throughout the project.

I would also like to extend my gratitude to the three persons who have helped me testing the product of my project and providing me with invaluable feedback. Kristian W. Larsen, partner at IT Minds, helped me set up the project and get in contact with the right persons. Magnus Mortensen, owner of Loppeportalen, have been the customer during my testing and provided me with feedback on my work. And a final thank you goes to Mathias Harboe, project lead at IT Minds Copenhagen, who has spent his valuable time evaluating and discussing my work and providing me with suggestions.

Finally, I would also extend a special thank you to Christine Thue Poulsen for her support and patience during the project and the long weekends spent working instead of being with her.

Contents

Chapter 1

Introduction

The software development process of an organization can play a vital role for the success of the organization. Choosing a methodology, or choosing not to use a methodology, can be a critical choice. An important reason for using a software development methodology in the first place is to be able to deliver products on time and budget. However, some methodologies does not include answers to the two questions "Are we building the product right?" and "Are we building the right product?", which is known as verification and validation respectively. Instead this is left up to developers to obtain these goals with no clear guideline or structure. Other methodologies focus on other areas, which might be verification and validation related, but they are then restrictive in their use and not widely applicable. The nature of software products is that they are used everywhere, and this large variance is also reflected in the large variance when it comes to different projects and teams. Many different methodologies exists - most with either niche uses, or too general to provide meaningful contributions to all three aspects of development - being on time and budget, verification, and validation.

This project aims to develop a methodology that encompasses both getting the product built on time and budget, but also focus on the questions of verification and validation, while being adaptable to a large range of projects and teams.

First, an investigation of current methodologies will be conducted as a literature

study. This literature study will uncover in which areas each methodology excels at, and in what areas they are challenged. Building from that, a new methodology will be created which is suited for accomplishing both the aspects of being on time and budget, satisfying verification and validation, and being adaptable to different projects and teams.

Once the new development process is created, it is tested in cooperation with IT Minds, Copenhagen. The test is conducted on a real life project, in which a solution is developed. The process is then evaluated by a project lead from IT Minds and the customer who owns the project.

1.1 Problem statement

To investigate methodologies and practices for software development, their strengths and challenges, and how to combine these in a way that is adaptable to different project and team types. The aim of this new framework for software development is to produce software on time and budget, while focusing on a high amount of verification and validation.

1.2 Reasons for study

IT projects fail more often than they should [10], both when it comes to actual project termination, but also in regards to being on time and budget, and when it comes to delivering a functioning system, which delivers value to it's users. This project aims to identify issues that are a common cause for these problems, especially when it comes to verification and validation of the final product. The nature of IT projects are that they are so different in size, scope and type, that an adaptable approach is needed when it comes to development methodologies and practices, since no size fits all. An adaptable approach should aim to deliver a product which are built correctly (verification) and are build to deliver the requested value to it's users (validation), while staying within time and budget constrains. No such adaptable methodology is widely known, so it is the aim of this project to present such a methodology.

1.3 Structure of the thesis

The overall structure of this thesis is:

Introduction chapter
This chapter contains many of the formal elements, such as problem statement and the project plan.

Analysis chapter
This chapter contains the literature study. First general software engineering is studied, and an important list of common failures for software projects are listed. Afterwards multiple methodologies are studied and reviewed. A subset of the methodologies are selected and analysed for strengths, weaknesses, tools and practices. Project and team properties are presented and how the selected methodologies handles each of these properties are analysed.

Design chapter
In this chapter the framework is designed. First the core of the framework is designed by extending an existing methodology, and improving it based on the common reasons for failure in software project. Then a toolbox is created for the framework, so that the core can be augmented to counter different project or team properties, such as Large Team, og Small Budget. Finally a way to transition to the framework from an existing methodology is presented.

Implementation chapter
This chapter starts out with the case description and an analysis about the properties of the case. Then follows a description of the testing and the results of the testing of the framework. Finally, feedback about the process in action is given and discussed, and the framework is evaluated by an industry professional.

Conclusion chapter
In the last chapter the limitations of the project is stated along with interesting topic for further research related to this project. Lastly we have the conclusion of this MS.c. project.

Each chapter has a small introduction, explaining what the chapter contains, and a conclusion section which summarizes the main points of the chapter.

1.4 Project plan

The project plan and the revised project plan, reflecting the actual work, is found in Appendix A. The project began the 10^{th} of February, which in the project plan became week 1.

The analysis and literature study took a little more time than anticipated, but the design phase was quicker. The main divergence from the plan came when interaction with the customer was needed. Due to the fact that the customer had limited time to spend on this project, the implementation work and chapter deviate from the original plan as well.

At week 12 the meeting with the customer was supposed to happen according to the plan. All work that could be done before that meeting was done and only the introduction chapter was missing. Since the customer did not have time till half way into week 14, the introduction chapter was created and no work was done in week 13 and the start of week 14. The project was handed in one week earlier than planned due to timing constraints.

1.5 literature study

The literature study for this project is found in chapter 2, section 2.4. The sources are research papers published in journals and websites. The websites are chosen for either their reputable author or their publisher. References to the relevant sources are given in the beginning of each section.

A lot of data is used from the annual State of Agile survey [33] throughout this work, as it gives a reasonable representation of the industry right now, such as barriers to agile, methodology adoption rates, and common issues.

Chapter 2

Analysis

In this chapter I will present the findings of my literature study. First I will give a brief introduction to the Software Life Cycle and specify which activities within the life cycle I have chosen to focus on, and why. Then I will take a look at the evolution of Software Engineering since the saying goes *"you can't know where you're going until you know where you've been"*, and list some of the main points of failure in software projects.

Once the basics are established, I will examine some methodologies of varying application area and size, and from them select which to focus on and why. Having selected a few of them, I will discuss their strengths and challenges, compare them against each other, establish under what properties they perform best and worst, and finally present an overview of their tools.

2.1 The software life cycle

The System Development Life Cycle (SDLC) is actual in itself a methodology, as it describes a set of activities to be completed to develop software, but I will here look at it as a description of major activities that most modern methodologies incorporate. It should be noted that some methodologies, such as the Spiral

Model, has a slightly different take on these. However, to facilitate a proper comparison I have fitted many of the following methodologies into the basic steps presented in this section, and made sure that the activities in this section are broad enough to encompass the activities from different methodologies. As such, the step called **Planning & analysis** will encompass both the planning of Scrum, the requirements analysis of Waterfall and analysis, identification and planning phases of the Spiral Model. This results in fewer activities than most other representations of a SDLC. This section is based on the work of Kellner [22] and Boehm [9], and work found on the US Department of Justice [25].

The SDLC then have the following 4 major activities; *Planning & Analysis, Design, Implementation & Testing, and Maintenance.* The activities and their order is outlined in Figure 2.1.

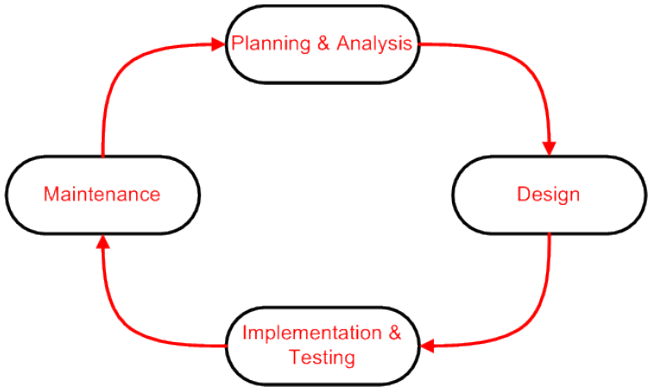

Figure 2.1: The four major activities in the defined System Development Life Cycle

Planning and analysis have been grouped since these activities are both performed before any development is started, and only deals with the system in a very abstract manner. Some methodologies focus heavily on analysis, while other excel at planning, but in both cases the activity is performed for the same reason; to give the developer (and sometime project owner) a better understanding of the task at hand, and the resources needed to successfully complete it.

The *Design* activity contains both design on an architectural level and more low-level design such as class interaction and implementation.

Implementation and testing have been grouped because they are closely linked, and both pertain to the actual development of the product. Some methodologies have these clearly separated, while other have them interleaved. Because of this,

I have chosen to group them, since splitting the activities in methodologies where they are interleaved would distort the function of it, whereas grouping them does nothing to the methodologies where they are separate.

Maintenance includes fixing bugs after release, or making other small improvements or tweaks after final release. In the case of a brand new feature being added post-release, this report considers that as further software development, and thus taking another round in the SDLC.

This report will emphasize the *Planning & Analysis*, *Design* and *Implementation & Testing* activities. The reason for this is that these activities are those performed during the development of software, which is the focus of this report. However, maintenance cannot be disregarded completely since this activity is very important for software which place a high value on reliability, and some methodologies support maintenance very well, which should not be ignored.

2.2 Early evolution of software engineering

As mentioned in the introduction to this chapter, taking a look back to the very beginning of software engineering and it's early evolution can bring some insights into why software engineering is moving in the direction it is, as well as giving some pointers as to what one should be aware of when choosing a methodology. This section is based mainly on the article by Estublier et al [15], and secondarily on the webpage by Ambler [2].

In the beginning when software was starting to be created on an industrial scale, it was natural to look at the established engineering disciplines to find an appropriate workflow. When looking at production lines, pharmaceutical development, and large scale construction, it was clear that first the product had to be analysed and understood, then carefully planned, before finally production started. This lead to the waterfall as the de facto software development methodology. However, as the software industry matured, some issues were raised. The technology was moving at such a rapid speed that practitioners often faced the following, as quoted from Estublier et al [15]:

- *Novelty ("I never did that before")*
- *Uncertainty ("I do not know how it works")*
- *Instability ("it is buggy")*
- *Requirements evolution ("I want this new feature")*

This led software engineering to have some quite different characteristic than traditional engineering. I here quote the characteristics presented in Estublier et al [15], which differentiates the software engineering development process from traditional engineering processes, since the authors presents it quite nicely:

Usual workflow systems assume that:

- *A process is a partial order of steps to be executed in order; completion of the last step means the process goal is reached.*
- *Processes are a fully repeatable sequence of steps.*
- *The product, goal of the process, is considered finished at the end of the process and, therefore, keeping its versions is usually not a concern.*
- *Most often, workflow systems use a unique copy of the data, stored in a global store, and either forbid concurrency or leave concurrence unmanaged.*

In contrast in software engineering:

- *The goal is never reached and actions are undertaken in apparently a non-deterministic way*
- *Concurrent processes are hardly deterministic; each execution is significantly different.*
- *The product, goal of the process, is never finished; keeping intermediate versions of the product is a critical issue.*
- *Many copies of the product under way are continuously and concurrently modified.*

It can be seen that the waterfall process tried to emulate the traditional engineering disciplines, even though the field of software engineering is not quite appropriate for this methodology. therefore it is important to find an appropriate methodology, and much work has already been done in this field. However, the traditional waterfall-like methodologies still has many practitioners since the jump from waterfall-like to more agile methodologies like Extreme Programming (XP) is quite large, so another approach to this might be needed. Furthermore, we see that the activities outlined in the waterfall model often are used in modern methodologies, so the waterfall model is an important stepping stone in understanding these methodologies.

2.3 Main points of failure

Before moving on to the methodologies, I will list the most common causes for failure in software projects. This gives us some important notes to take into consideration when comparing methodologies, and especially when designing a modified methodology in the next chapter. This section is based on the articles by Charette [10] and Frese et al [18] and surveys done for the State of Agile [33].

I list the points of failure grouped in three categories, External issues, Internal & Validation issues, and Verification issues. This grouping is done to easily grasp which of these we can act upon with a proper methodology. The Verification Issues relates to troubles in understanding the customer and his needs, while Internal & Validation issues deal with dynamics within the team and how the develop.

A **External issues**

(1) Outside pressure (Commercial, rivals, first-to-market, organizational)
(2) Lack of support from executives and/or organization

B **Internal & Validation issues**

(1) Insufficient risk management
(2) Poor status reporting
(3) Use of immature or unsupported technologies
(4) Team not able to handle the complexity of the project
(5) Bad development practices
(6) Poor resource management and estimates

C **Verification issues**

(1) Lack of customer involvement
(2) Unspoken, unknown or unrealistic project goals and requirements
(3) Poor, or lack of communication between all parties, including developers, project managers and customers

Especially the Verification and Internal & Validation issues can be countered (at least partially) by some methodologies, especially the agile methodologies. For this reason these will be a focus of some of the following sections in this and future chapters. The External issues are a bit harder for a methodology to act upon. These points will serve as a guideline when analysing methodologies, in order to keep these common causes of failure in mind.

2.4 Literature study and review

In this section I will present a variety of methodologies, the State of the Art, and give a brief run-down of how they operate. I will reason why I have chosen these methodologies, and in subsection 2.4.9 I will select a subset of these for deeper analysis. This section is based on multiple articles and webpages from reputable authors, which I will list in the appropriate subsections for ease of the reader.

2.4.1 Waterfall

As discussed earlier, Waterfall is the traditional software development methodology. I have chosen to address this methodology since, as described in section 2.2, it is the basis of many other methodologies, it is still widely used, and the activities it defines provides a good understanding of software engineering in general. This subsection is based mainly on the article by Petersen et al [28] and the thesis by Jacob [21], and secondarily on lectures given at The Technical University of Denmark in the courses, 02291 System Integration, 02161 Software Engineering 1, and 02162 Software Engineering 2. The waterfall model consist of five consecutive phases always done in the same order. According to the updated Waterfall model, it is possible to go backwards in the process, but then the artefacts produced would have to be updated and go through formal review again. The phases are; Requirements Engineering (or Requirement Analysis), Design, Implementation, Testing (or Verification & Validation), and Maintenance. For the reasons given in section 2.1 maintenance will not be discussed. The waterfall model emphasises Big Design Up Front, giving the reason that it is much cheaper to fix an issue early in the process, during requirements or design than during implementation or even verification. The modified waterfall model can be seen in Figure 2.2 with the blue arrows indicating the normal flow of development, and the red arrows indicating backtracking.

Requirements engineering The first phase of the Waterfall model is Requirements engineering. In this phase the customer's needs are identified, understood and documented, usually on a higher abstraction level first. This step is vital for the waterfall model, since all following phases will be built upon the work done here, so any mistakes or issues not found here can become very expensive to fix later in development. The customer requirements are then analysed, and further specified so that they can be used later on in design. It is important to check whether all requirements are aligned with what the customer expects,

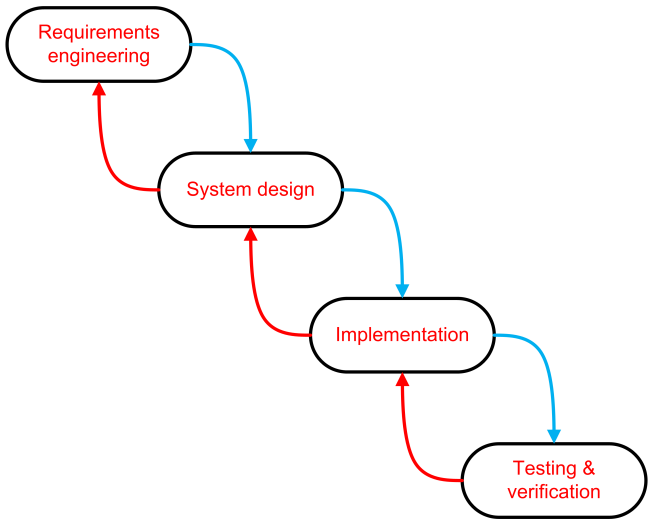

Figure 2.2: The modified waterfall model which allows for backtracking

and that their success criteria is clearly defined and verifiable. The main artefact of this phase is the Requirement Specification, which is the formal result of the work, and is the input for the next phase.

Design The following phase is Design, which take the Requirement Specification as input. The phase includes both system design, component design and non-functional design. The architecture of the system is defined on different abstraction levels, so the programmers will know exactly what to do once the coding starts. The main artefact of this phase is the Design Description. In this phase it is especially important to verify and document that the architecture fulfils the requirements, if there are any deviations in scope or resource management, and that the documentation is sufficiently specified so that programmers can make it.

Implementation The phase of actually writing software is Implementation. It takes as input the Design Description, and the programmers will start coding as to the specifications given therein. Once coding is complete, a finished system should have been produced, which satisfies the customer needs as given by the Requirements Specification, and the architectural and non-functional design given by the Design Description.

Testing Once the implementation team has finished and they hand over the system, testing and verification is done by the testers. Different levels of testing can be done, including unit testing (testing the specific modules), system testing (testing the overall system) and acceptance testing (to see if the system satisfies the customer). These three test types can be seen as first testing that implementation was done correctly (unit testing), then that the design was good (system testing) and finally that the requirements were met (acceptance testing).

2.4.2 Agile

An agile process differs on some important points from the traditional waterfall process, so a brief introduction to Agile development is provided. You can hardly talk about agile development without mentioning the Agile Manifesto, and it's Twelve Principles, here quoted from The Agile Manifesto [5]:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions *over processes and tools*

Working software *over comprehensive documentation*

Customer collaboration *over contract negotiation*

Responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

The Agile Manifesto can seem a bit simplistic and abstract, so to give a more concrete and accurate representation of agile development and it's values, I quote the agile principles from The Agile Manifesto [5] which put it eloquently:

[AP1] *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

[AP2] *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

[AP3] *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*

[AP4] *Business people and developers must work together daily throughout the project.*

[AP5] *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

[AP6] *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

[AP7] *Working software is the primary measure of progress.*

[AP8] *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

[AP9] *Continuous attention to technical excellence and good design enhances agility.*

[AP10] *Simplicity - the art of maximizing the amount of work not done - is essential.*

[AP11] *The best architectures, requirements, and designs emerge from self-organizing teams.*

[AP12] *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Many of the following methodologies are based on the agile principles, and thus knowing these core values are important.

2.4.3 Extreme Programming

Among the agile processes, the most often discussed is Extreme Programming. As **[AP7]** states, code is the measure of progress on any given project, which aligns very well with the very code-centric approach of Extreme Programming. Many small releases are incremented out, and each time a working system is delivered. XP is such named since it aims to take the agile principles "to the extreme", and as such will be a focus among the chosen agile methodologies. It was spearheaded by Kent Beck during the Chrysler Comprehensive Compensation (C3) project, and has evolved into a complete suite for software development. This subsection is based mainly on the webpages by Don Wells [35, 34], one of the main developers on the C3 project, the Extreme Programming flagship, and the book by Beck and Andres [4]. Secondarily this is based on the thesis by Jacob [21] and the web articles by Fowler [16, 17]. A basic overview of the overall XP process can be seen in Figure 2.3.

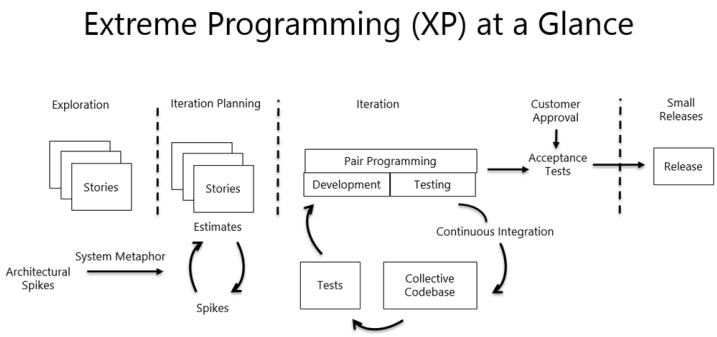

Figure 2.3: The basic XP process. Image source: blogs.msdn.com/b/jmeier/archive/2014/06/06/extreme-programming-xp-at-a-glance-visual.aspx, 18/03/15

Planning One of the underlying assumptions of XP is that an on-site customer will be present, which is made clear by **[AP4]** and **[AP6]**, and already from the start of the project this is evident. XP practices planning on multiple distinct levels. First, the customer will write *User Stories*, which are written in the customers terminology. They are not limited to the user interface, or any particular part of the system for that matter, and make up the requirements of the system, in the form of a story about a fictional user wanting to do something in the system. These User Stories are based solely on the customers needs, as per **[AP1]**, and thus do not define any technology or implementation. In accordance with **[AP10]**, User Stories should contain just enough detail to be able to make a low risk (low comparatively to the risk of the story) estimate of how long it would take to implement, and nothing more.
When the project starts up, a *Release Plan* is created, roughly sketching out which User Stories to develop in each iteration. Each iteration should give the customer a system providing acceptable business value, so User Stories should be planned accordingly. This is also important given **[AP7]**.
Once the project is under way, each iteration will start-off with a *Iteration Planning* meeting, between the developers and the customer. During this meeting the customer select which User Stories from the Release Plan should be included in the iteration.Then the customer, with the help of the developers, create acceptance tests for each User Story, so that their completion is well-understood and testable. Each chosen User Story is then translated from the customer's business language into a more technical language the developer can work with. These meetings highlight **[AP4]** and **[AP6]**. Once this is done, the developers sign up for programming tasks associated with each user story, and then make a fine-grained estimate of how long it will take to implement.
The final part of planning in XP is the *Daily Stand Up Meeting* which is held

at the start of each day among the developers. In this meeting, problems and solutions are communicated among the developers, and three main questions are answered by each team member; *"what was accomplished yesterday? What will be attempted today? What problems are causing delays?"*. Detailed technical solutions are not discussed during the daily stand up, but are rather done among the involved parties afterwards, to minimize wasted time.

Design One of the most common points of criticism in XP, is that the methodology lacks proper design. XP relies on *Evolutionary Design*, where the design emerges over time, which seems very much akin to "cowboy-coding" or "code-and-fix", which often leads to either failed or challenged projects according to Ambler [2]. However, as Martin Fowler points outs [16], this misunderstanding happens because some important practices of XP is not taken into consideration. The reason given by opponents of XP why Evolutionary Design does not work, is because of the exponential software change curve. This curve tells that the longer a project is in development, the more expensive it becomes to make changes. As mentioned in section 2.2 and section 2.3, the whole point of moving from Waterfall to Agile was to be able to adapt to change, and thus the core of XP exists to flatten the change curve, and later part to exploit that flattening. Martin Fowler talks about the enabling practices which makes Evolutionary Design, indeed most of XP, work. He postulates that the criticism stems from the fact that people tried Evolutionary Design, without also doing the enabling practices, thus not flattening the change curve before exploiting it.
First of, XP advocates *Simple Design*, as per **[AP10]**. This is supported by *Refactoring*, as Don Wells puts it *"Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. Make sure everything is expressed once and only once. In the end it takes less time to produce a system that is well groomed."* - [35]. But this alone is not enough to flatten the change curve. To accomplish this, XP relies on possibly it's two most important practices, *Test Driven Development* and *Continuous Integration*. Test Driven Development (TDD) ensures that changes made late in the project does not negatively affect the already developed features, since the entire automated test suite is run, and Continuous Integration ensures that conflicts are discovered early by keeping all developers in sync with each other. Continuous Integration as a practice has become a standard used in most projects, thanks to tools such as git and svn.
Once the change curve has been flatten and Evolutionary Design is possible, we can delve deeper into how XP uses design. Class, Responsibilities and Collaboration Cards, or *CRC Cards* are used by the development team to model how different classes within the system interacts. They contain the name of the class, what responsibilities it has to, and how it collaborates with other classes within

the system. As with all else in XP, simplicity is advised when using CRC Cards as well, so only fill out the card completely if it is necessary. While filling out the CRC Cards, the system is simulated by treating the cards as objects, and moving them around to simulate messages being sent. Going through how the system works step-by-step as a team will uncover potential problems or weaknesses, and solutions can be quickly simulated right away, saving time.
When the overall functions of the system, or whatever part of the system was modelled using CRC Cards, is understood the team moves on to find a good implementation. *Prototypes*, or "spikes", are used to learn more of which design will and will not work. These are the code equivalent of a sketch, and are expected to be thrown away once the developers learn what they need from them. They are used mainly for architectural purposes, or for very complex parts of the system. And even if a certain Prototype is agreed to be the way to go, it can still be changed later if it would make the system more clear or concise.
Finally, to allow developers to easily talk about the design, a *Metaphor* for the system is created. The Metaphor should be simple and easily understood without any expert knowledge. The Metaphor has two main functions: First, it helps explain the system to new developers or stakeholders without the need for huge amount of documentation, and secondly it provides a clear naming convention for classes and methods. Using a clear naming convention allows developers to easily understand parts of the system they have never seen before by a glance, and less time is wasted figuring out what a good, descriptive name would be. However, a good Metaphor might be difficult to come up with until the team gains a better understanding of the project, and as such it does not need to be created in the beginning of the project.

Implementation As mentioned above, some of the most important practices of XP are Test Driven Development, Continuous Integration, and Refactoring. Bringing testing to the forefront is what TDD is all about, and never writing any code unless a test fails. So to implement a feature, the developer would first have to implement an automated unit test that tests for that functionality. Then the test is run and should fail. Then enough code is written in order to make the tests pass, and nothing more. This process continuous until the entire system has been developed. An important thing to note here, is that all unit tests should be run each time anything new is done, thus making sure that nothing breaks elsewhere in the system once a new feature is being developed. If a bug is found, first write a test that shows the bug, so that it will never emerge again, and then fix it. This is the very core of XP, and is the greatest contributor to quality code and on time and budget XP projects. To make sure that the test suite and code base is up to date among all developers, so that conflicts are found and resolved as early in the process as possible and to facilitate code re-use , XP uses Continuous Integration. Developers should commit

and integrate their work every few hours if their work allows it (all tests passes, some part of functionality is complete), or at the very least once a day. Doing it this way prevents large amount of time spent near the end of the project when everyone's work has to be merged and integrated together.
Something else which is considered important to XP, is *Collective Ownership*, in the sense that all team members "owns" the entire code base, and there is not a certain part of the system only a single developer understand. This eliminates the risk that if a developer leaves, some part of the code can not be maintained or further developed. It also means that each developer is allowed to change, fix, improve or refactor any piece of code. It seems non-trivial to have the entire team be responsible for the entire system, and not having a dedicated lead programmer or architect to keep the vision of the system. But as complexity rises in a system, it will quickly become too much for a single person to have a good vision of the system, especially given the rapid change which might occur. This is why the responsibility is spread among the entire team, and all parts are understood by multiple developers, which also removes the bottle-neck that an architect often becomes in a rapid changing environment. How XP achieves Collective Ownership is by *Code Review*, *Moving People Around*, and Code Review taken to the extreme, *Pair Programming*, where code is being reviewed continuously. All non-trivial code should be written in pairs on a single computer. The idea is that two people working in a pair will add as much functionality to the system as they would apart during a day, but the code will be of higher quality, thus saving time in the long run. It is a social skill, and mastering it will take a considerable amount of time for most. It's important to note that Pair Programming should not become a teacher-student relationship, but rather two equals working together, even if one of the developers are considerably more experience. The second part of achieving Collective Ownership is to Move People Around, which simply furthers everyone's understanding of the system, since no one person keeps working on the same part(s). Collective Ownership also provides the benefit of flexible workload balance, since no developer becomes a bottleneck. Finally, all code should be written in accordance with both the Metaphor, and some agreed upon *Coding Standard*. The Coding Standard, is very important, since it allows developers to understand the code at a glance, and essentially just read the code as if it was plain text.

Team care As per **[AP8]**, XP puts a high value on the welfare of the project's team members, both developers, managers, and customers. To avoid burnout, stress, and other human issues, XP promotes a *40-Hour Work Week*. Of course overtime might occur once in a while, but two weeks in a row with overtime is considered a serious problem that needs to be addressed. *Team Empowerment* is also important in XP, as given in **[AP5]**. An engaged and motivated developer takes pride in what he does, and produce better results. By extension, teams

also should be *Self-organizing* in accordance with **[AP11]**. Every member of the team should be co-located and the workspace should be open to further communication, since this is vital to a team practising XP.

2.4.4 V-model

The V-model (Vee-Model, V-Shaped Model) is in many ways an extension of the traditional Waterfall model. It follows the same basic steps as the waterfall model, but on each stage of development, testing is also considered. Furthermore, the testing phase at the end of the project is more elaborate. Since the V-model produces the same amount of documentation as the waterfall process, but has more focus on validation and verification, it makes it suitable for more formal environments such as the pharmaceutical industry, where certifications are needed by official bodies. It is also the recommended methodology in the international standard IEC 61508, titled *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*, and for these reasons it has been included. The V-model is shown in Figure 2.4, where the blue arrows indicate the development flow, red arrows indicates backtracking through the feedback loop as with the Waterfall model, and the black arrows shows when test plans are created and executed. This subsection is based on the article by Deuter [13] and the webpages by Coley Consulting [12].

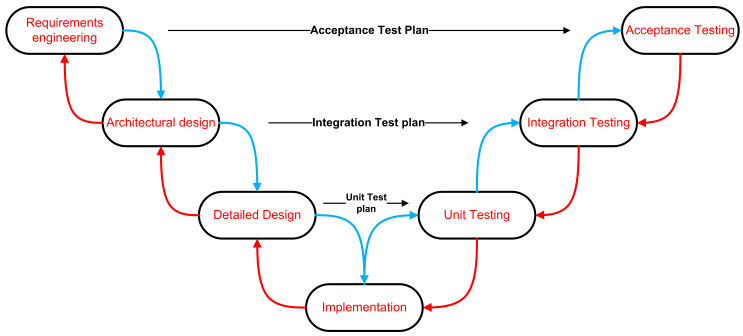

Figure 2.4: The V-model. Blue arrows indicate the development flow, red arrows indicates backtracking, and the black arrows shows when test plans are created and executed

The V-model starts with creating the *Requirement Specification*, just like the waterfall model, where all requirements of the customer are recorded, analysed and documented. Once these have been understood by the involved developers and the customers agree on the requirements put forth, the first deviation from the waterfall model appears. Alongside the Requirements Specification, the test

plan for the acceptance tests are also developed. All details can not be filled out yet, but much of the initial work is done now, while the requirements are fresh in memory, so that testers may benefit later in the project. These tests may also be checked during other stages of development (design for example), to see what has been developed so far still passes the tests outlined in previous stages.

Once both the specification and test plans are done, the V-model moves on to the *design* phase, which is divided in two separate phase, for the sake of making test plans. First the overall architecture is designed along with the per waterfall model appropriate documentation, and the plans for integration testing. Then the design is further specified on a more modular level, and the overall structure of the unit tests are planned as well.

In the *implementation* phase all coding is done using the documents produced earlier, just like the waterfall model. No new test plans are created, but as this phase moves along, the finishing touches can be put unto the test plans from previous iterations, so that they are in a more ready state once the appropriate phase arrives.

What the waterfall simply labels as "testing", the V-model has split into three distinct phases which handles *verification and validation* of the system. First functionality is tested by the unit tests, then all parts are put together, and their integration is tested, and finally the (hopefully) finished product is run through acceptance testing, to verify that it indeed satisfies the customer's requirements.

From this description it is clear that this is a step-up from the traditional waterfall model with it's added focus on testing in the forms of verification and validation. However, the remaining problems in the waterfall model is also present here such as; it does not respond well to change, it is expensive to backtrack if bugs, issues or missing functionality are found, and a general lack of customer involvement once the requirements phase is done. But this methodology is still worth mentioning for it's rigid process which makes it attractive for formal environments with a focus on documentation such as the medicinal industries.

2.4.5 Scrum

Scrum is not a development methodology per se, but rather a management methodology, and is often combined with "generic agile" practices since it supports these very well. It is also by far the most used agile methodology according to the surveys done by the annual State of Agile survey [33]. Figure 2.5 shows

the basic activities and how they relate to each other in the flow of a single iteration, or sprint in the Scrum terminology. This process is repeated for each sprint until the project runs out of time, or all features has been implemented. Since the list of features is prioritized by the customer and the result of each sprint is a potentially shippable product, running out of time is not as huge a problem for Scrum projects as it is for traditional projects. This subsection is based on the work of Rising et al [30] and Paasivaara et al [27], as wells as the webpage by The Scrum Alliance [31] and Wells [34], combined with my own personal experience working in two Scrum teams in different organizations.

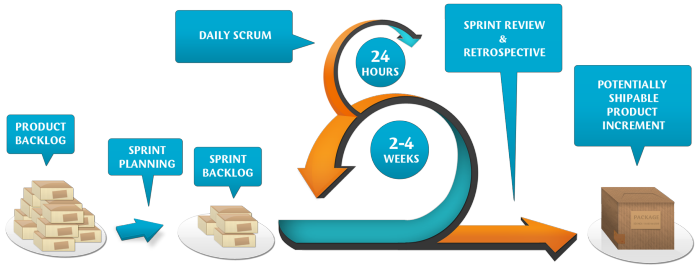

Figure 2.5: The process of a Scrum project. Image source: galleryhip.com/scrum-methodology.html, 18/03/15.

The Scrum process is quite simple. The only thing that can be puzzling is the terminology which might not be readily understandable. For this reason I will present a description of the main actors and artefacts to give an understanding of the basic terminology, and then explain the flow of the process by presenting the main events.

Actors In the development team there are two important roles beside the developers, the Product Owner and the Scrum Master. The *Product Owner* is essentially the customer, or the one who represents the customer. The job of the Product Owner is to ensure that tasks are well understood, and that the sprints are filled with the most important features first. The *Scrum Master* is a developer, but with some extra responsibility. He acts like a coach and ensures that the team follows the values of Scrum. Other tasks include to help the team perform at their highest potential, dealing with issues or impediments within the team, working and handling communication with the Product Owner. He can be referred to as the "Process Owner", since he essentially controls the Scrum process, but not the team itself.

Artefacts The *Product Backlog* is created by the Product Owner and Scrum Master at the start of a Scrum project. They also maintain it, since it can be updated after each sprint if new requirements appear. It is a prioritized list of User Stories with associated tasks, and the priority can be based on risk or value. An interesting approach to prioritization presented by The Scrum Alliance [31], is that instead of selecting the most important features, since this can be hard if the customer thinks all the features are important, then select the least important feature, and move up from there. It is important to note that it is not a goal of the Product Backlog to be complete before the project starts, since everyone learns more of the product as development continues, and thus more tasks will be added during development. The *Sprint Backlog* is, like the Product Backlog, a prioritized list of tasks. Those tasks are selected in the beginning of a sprint, and the team commits to completing them during the sprint. Within the sprint, the team does not need to follow any specific order on the list, as they have committed to completing all the tasks within the sprint. Changes to the Sprint Backlog during the sprint should be avoided, since that runs the risk of the team not completing what they are committed to do. Finally we have the *Burndown Chart*, which is maintained by the Scrum Master. The Burndown Chart shows the progress the team has been doing compared to remaining and estimated work. It is updated continuously, and is a effective visual tool to show outside stakeholders.

Events A project is started by creating the initial Product Backlog which is a one-time event. Then at the beginning of each sprint a *Sprint Planning Meeting* is held, usually with the entire team, including developers, Product Owner and Scrum Master. The team, without the Product Owner, decides how much can be done within the coming sprint. The Product Owner describes the different tasks in the Product Backlog so that developers get a better understanding of the task and are able to estimate the time needed. In the beginning of each day of a sprint, the *Daily Scrum* is held for all team members. Three main questions are answered by each person attending; "What did you do yesterday?", "What will you do today?" and "Are there any impediments in your way?". These meetings need to be kept short, and potentially long discussion about technical issues are done only by the involved parties after the meeting. The Scrum Master will help deal with impediments, either directly or will put the affected team member in contact with someone who can. The importance of the two first questions is that it shows the commitment of each developer to each other, and not to some far-off non-present customer, making it more pressing to uphold. Each sprint is concluded with a *Review* of the product, where the state of the product is evaluated. The overall goal of each sprint is a product of sufficient quality, so that it could be considered finished. This overall goal, along with the backlog items the team committed to complete, are reviewed. This meeting is the only

meeting where outside stakeholders, such as upper management, customers and developers from other projects, are often invited so that the progress of the team can be shown. Usually following the Review, is the sprint *Retrospective*, where the process itself is evaluated by the team. A simple approach to this is to have everyone write their personal answer to the three questions "What should we start doing?", "What should we stop doing?" and "What should we continue to do?". Then these answers can be grouped together by topic, and the team can select the most important of these topics to improve upon.

It is evident that a main theme of Scrum is commitment, something that agile processes are often criticized for lacking. In State of Agile [33] two important concerns about agile processes are "Lack of management control" and "Lack of Predictability", which 53% of respondents said. These could roughly be categorized as lack of commitment, or at least something a commitment heavy approach might help solve. It is also for this reason that Scrum is quickly becoming one of the main players in agile development.

2.4.6 Kanban

Another management methodology, Kanban, is inspired by Toyota Production System and Lean manufacturing. In software engineering it is used mainly as a component of other methodologies such as ScrumBan, but is still among the more popular agile practices according to the State of Agile survey [33]. A short mention of it will be presented here based on the work of Polk [29], Hiranabe [20] and personal experience in a large development team using Kanban.

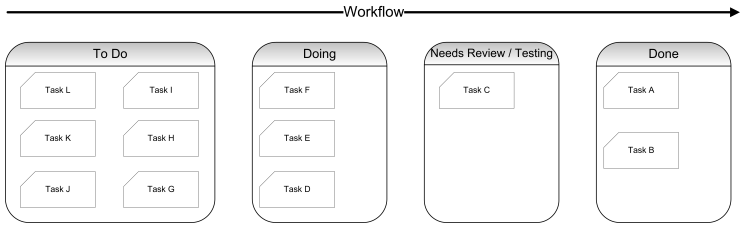

Figure 2.6: Kanban board with tasks in different stages

Figure 2.6 shows an example of a Kanban board used in software development. Tasks are placed in the *To Do* queue, once created, and developers then move these tasks to *Doing* once work has started. In this example a *Needs Review / Testing* queue has been added as well, where all tasks undergoes testing or review. When a developer has time he should take a look at the review queue and take the first item in the queue and review it. It is then moved if it is

approved, otherwise it goes back into To Do with a note of what still needs to be done. This additional queue were used were the author worked, but this could differ from organisation to organisation. Finally a task is moved to *Done* once it has been approved. This approach emphasises just-in-time completion of tasks, while maintaining balance in the different activities and not overloading specific personal with tasks. It works as a visual aid to track the progress of the team, and helps by providing at-a-glance overview.

2.4.7 Agile Modelling

Another common point of criticism that most Agile methodologies faces is that they lack documentation, which 24% of respondents mentioned as a concern about adopting an agile process in State of Agile [33]. However, Agile Modelling propose a way for modelling, and documentation in general, to be agile and thereby easier to integrate into the agile methodologies, thus countering the criticism. It is not a methodology by itself, but a useful component. This subsection is based on the work of Ambler [1] and Wells [34].

The main premises of Agile Modelling is that it should follower the same core principles as agile development of code. For example, Agile Modelling says to deliver documentation and models that are "Just Barely Good Enough" "Just-In-Time", which is consistent with **[AP10]**. "Just-In-Time" means that documentation should not be made until it is actually needed, as to avoid having to re-write the documentation because of requirement or design changes. "Just Barely Good Enough" is easily misunderstood as something of low quality, since it is only barely good enough, but the important thing to remember is that if it is good enough, then it adds no extra value for the customer, to put more work into it. This brings us to another point, that documentation should be produced because it provides value to the customer, not documentation for documentation's sake. Therefore each diagram, or other piece of documentation, should be treated as a User Story, and the customer can then choose if the documentation has a high enough priority to be included.

The target audience of documentation is also very important in Agile Modelling. Mostly in agile project, documentation is produced to non-developers since all developers are using more efficient ways to understand the system, such as face-to-face communication, and collective ownership. For this reason, what should be documented and how it should be documented needs careful consideration, so that the documentation provides real value to the customer. For example in XP, the philosophy is to write clean, easy-to-understand source code, since that is the only thing is always in sync. While this might be true, if the target audience is management or the end users then clean source code is not a good

fit. If documentation is created without the explicit wish of the customer, the value it provides to the team should also be carefully evaluated, and lots of effort should not be put into these models, as developers often gains more value from the process of creating the model, than the model itself actually provides, which CRC cards from XP is a good example of.

Agile Modelling propose to use "Multiple Models", so that practitioners do not get stucked using only UML for instance. Ambler argues that while UML is a good foundation, it is simply not able to model all scenarios, and thus practitioners needs to use multiple models, and multiple ways to model. An agile twist on this that Ambler gives, is that often the best tool for the job is the simplest. No need to draw a huge UML diagram, if what you want to show can be modelled with a sketch on a whiteboard. Wells goes as far as saying *"Having a model printed out with straight lines and square corners in a specially labelled binder does not mean you have a good design, it just means you have a well documented one. I have found that a huge UML document is an excellent way to hide a complex design."*.[1]

2.4.8 Spiral Model

The Spiral Model developed by Barry Boehm was one of the earlier methodologies to move away from the traditional Waterfall methodology and into a more iterative approach, but is not part of the agile framework. It is less code-centric than most agile approaches, and thus seems closer to what traditionalists would use. This subsection is based on the work of Boehm on his original Spiral Model [7, 6] and the refinements made later [8]. This methodology is described as a risk driven process model generator, since it let risk analysis choose what specific process to choose at any given iteration, or cycle as it is called in the Spiral Model, and for any important choice during development. This Subsection will focus on the main points of the Spiral Model, which are present in all processes generated by it, and not on the process generation itself.

Figure 2.7 shows Boehms original diagram of the the Spiral Model, where the system is incrementally being designed and build, while the associated risks are being reduced. At each cycle, the project is evaluated so that cost, schedule and other estimates can become more and more precise and realistic the further along the project is. The two main features of the Spiral Model is it's Anchor Point Milestones which ensures that development team, management and customer only commits to feasible and acceptable solutions for all parties, and it's six Invariants. I will start by presenting the Invariants.

[1] Don Wells, 21-03-2015, http://www.agile-process.org/model.html

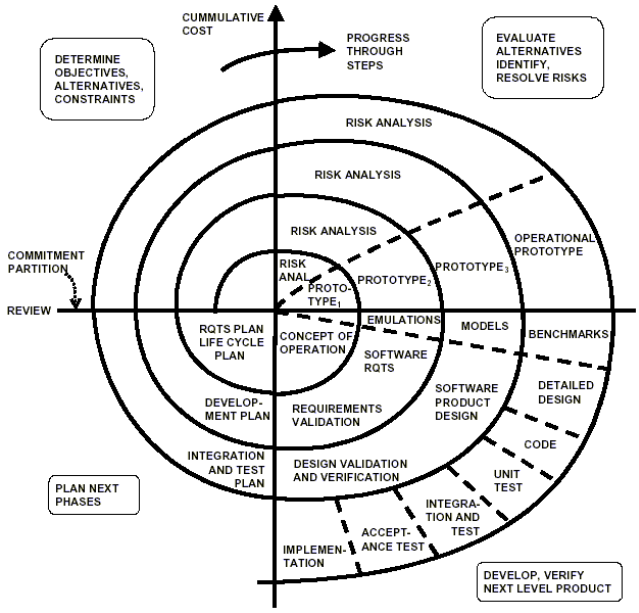

Figure 2.7: The Spiral Model by Boehm [8]. Each full cycle in the spiral is an iteration.

Invariant 1 The first invariant of the Spiral Model, is *"Concurrent Determination of Key Artifacts (Ops Concept, Requirements, Plans, Design, Code)"* Boehm [8]. This invariant ensures that the project is not committed to key artefacts, such as those listed by Boehm, too early since that would run the risk of causing incompatibility or infeasibility later in the project. If the team has already decided to use component A, then they have to choose component B later on for it to work with A. Finally the team will have to settle on C as the last component for it work with A and B, even though C is not an acceptable choice compared to D. This will then either result in a lot of re-work or a system that does not satisfies the stakeholders. Instead these decision should have been done concurrently, so that it would have been discovered that A would lead to B and lead to C, which would result in something unacceptable.
The example given by Boehm [8] which shows the lack of Invariant 1, is of a large database system. The customers said they want less than 1 second response time on the system, and without looking at other consequences the development team accepted that. The results was then that after two thousand pages of requirements, it was discovered that such an architecture would cost $100 million to create, whereas a 4 second response time architecture would take only $30 million to make. After doing a prototype with the slow response time and testing it with users, it showed that 90% of the time a 4 second response

time was good enough.

Invariant 2 Invariant 2 are to some degree the same as Invariant 1, *"Each Cycle Does Objectives, Constraints, Alternatives, Risks, Review, Commitment to Proceed"* Boehm [8]. The difference here is that this invariant deals with stakeholder participation and not directly key system artefacts. It ensures that the team considers key stakeholders objectives and constraints, evaluate suitable alternatives to constraints with clearly defined risks, and that key stakeholders review these critical objectives and constraints and their alternatives.
Boehm gives the example [8] of committing to a Windows-only component, even though large parts of the user base consisted of Mac and UNIX users. This happened because key stakeholders, in this case someone from the UNIX community, was not involved with the process, and caused a lot of rework to change the core component to be multi-platform compatible.

Invariant 3 The third invariant *"Level of Effort Driven by Risk Considerations"* Boehm [8]. This invariant is achieved by risk consideration, so that an activity does not take too much or too little effort. This invariant, along with Invariant 4, is very important to the core of the process model generator of the Spiral Model, since they ensures that the Spiral Model adapts properly to different projects. For example, a safety-critical project involving a nuclear plant requires a lot of testing, but a first-to-market sensitive project should focus less on testing and more on speed. This prioritization is done by the risk analysis.

Invariant 4 As can be seen, Invariant 4 looks very much like Invariant 3 *"Degree of Detail Driven by Risk Considerations"* Boehm [8]. The difference here is that this invariant concerns the artefacts of the system, and their level of detail, and as such can be seen as the product counterpart of Invariant 3. An example given by Boehm is that a complete, highly specified requirements specification is not a good idea for a graphical user interface. The risk of specifying the interface too early and locking it into a contract have a high probability of resulting in a bad and awkward user interface, since not all is known of the system and how the interface would pan out. On the other hand, it is a low risk not specifying the interface until much later, since flexible GUI tools are widespread. Thus Invariant 4 ensures that each artefact has the right amount of detail put into them, according to what poses the smallest risk.

Invariant 5 The fifth invariant is simply *"Use of Anchor Point Milestones: LCO, LCA, IOC"*. The milestones will be explained further down this subsec-

tion, but in short; LCO represent commitment to architecting, LCA represents commitment to the full software development life cycle, and IOC represent commitment to operations of the system. This allows for incremental commitment so both customer and developer can withdraw early without great losses.

Invariant 6 The final invariant is *"Emphasis on System and Life Cycle Activities and Artifacts"* Boehm [8]. This invariant ensures that the project scope does not focus too heavily on code, but considers the entire business process that the system should support / improve. This ensures that the project has a system-level business case which shows that the system actually does fulfill the objectives of the customer, before resources are committed to the project.

LCO Anchor Point Milestone The first milestone that a project will reach is the Life Cycle Objectives milestone. In this milestone the key stakeholders, including the development team, review the operational concept description, prototyping results, requirements description, architecture description, and life cycle plan. To sum these up, a feasibility rationale is made answering the question *"If I build this product using the specified architecture and processes, will it support the operational concept, realize the prototyping results, satisfy the requirements, and finish within the budgets and schedules in the plan?"* Boehm [8]. If the answer to the feasibility rationale is 'no' then the development team needs to rework the current work so far, or abandon the project. Once the LCO milestone has been passed, it means at least one architecture is viable from a business point of view. Boehm [8] compares the LCO to getting engaged, you have agreed that this might work out and it is currently looking good.

LCA Anchor Point Milestone Life Cycle Architecture is the second milestone. It reviews the same artefacts, but in much greater detail, and answers the updated feasibility rationale and if it succeeds all parties are now committed. In addition all significant risks must have been dealt with, either by finding an acceptable alternative, or by having a risk-management plan. Boehm [8] compares the LCA to getting married. Significant commitment is done on both sides and you are very sure this will work out in the long run. As with getting married, if you pass the LCA too hastily, the development team and stakeholders is likely to drift apart, and the project will die.

IOC Anchor Point Milestone The final milestone is the Initial Operational Capability, and it focuses a lot on preparation of the launch, including software, site and user preparations. It is also the point in time where the users will first

see a functioning system, and thus needs to be well prepared, as to not alienate the user base. If the preparation is deemed sufficient the milestone is passed and the system will be launched. Boehm [8] compares the IOC to getting your first child. Breaking apart now is (ideally) out of the question, and both sides are heavily committed to making it a success.

In conclusion of the Spiral Model, it's high emphasis on risk analysis and management and the feedback loop when it comes to planning and expectations of all involved parties, shows that this model is ideal for project with a considerable amount of risks, which is typically for safety-critical and large projects.

2.4.9 Selection of methodologies

In section 2.5 a subset of the reviewed State of the Art methodologies will be analysed, and this subset will be selected here. First of all, Agile Modelling and Kanban is excluded as is also hinted at in their respective subsection, since they are considered tools in the context of this thesis. The V-model is excluded since it is effectively just a slightly modified Waterfall model, with a higher focus on testing. The *Waterfall* model is included for it's instructiveness and it's clear overview of phases, *Scrum* is included for it's widespread use, *XP* is included since it is the most agile methodology showing a lot of promise, and finally the *Spiral* model is included since it represents a model highly suitable for large, high-risk projects, and that is an area where the agile approaches are considered lacking according to research done by Barlow et al [3].

2.5 The chosen methodologies

The chosen methodologies will be analysed for their strengths and in what areas they are challenged. Where each methodology is best applied and the tools used will be identified and their application analysed.

2.5.1 Strengths and challenges

The tables in this section are structured as follows; First the strengths and challenges of *Planning and Analysis* activities are listed, followed by *Design*, *Implementation* and *General*. The following subsections are based on the same sources as those listed for their respective methodologies in section 2.4.

2.5.1.1 Waterfall

From subsection 2.4.1, it is clear that the Waterfall model has some serious drawbacks, but also some strengths, and these will be elaborated here.

Strengths First of all, the Waterfall model is an old model, which means that there are a lot of knowledge and written experience concerning it. It's gated and very rigid structure forces developers to be disciplined, since there is not much room in the model for deviations. The very planning-centric focus of the Waterfall model allows the developers to catch some problems early in the process, such as uncovering possible design problems during requirements or identifying implementation pitfalls during design, and then counteract these problems. And finally from a manager and customer viewpoint, the Waterfall model provides the excellent strength that it provides clearly defined project length and cost.

Challenges Documentation of different kind is an important aspect of the Waterfall model, but also one of it's greatest weaknesses. Documentation of this kind ranges from comments, to diagrams, to formal documents, and they all have some things in common. They are static meaning that if code or larger parts of the system is changed, all of this documentation has to be checked that it is still up-to-date, and if not updated. This is an expensive operation, and often does not result in something that produces direct value to the customer. In line with this, some of the produced documentation might never be looked at again after it has been approved, and thus truly has been a wasted effort. The planning dependency of Waterfall results in poor handling of unforeseen problems, and such problems are likely to appear since it is unlikely that all requirements are known in the beginning. Furthermore requirements might change after initial planning phase is done, again resulting in unforeseen problems. When these problems then appear and gets addressed, the project is likely behind schedule, and thus has to gain time. Since testing is the last phase of Waterfall, this is usually what is cut if the project falls behind schedule and development is not extended.

Conclusion Table 2.1 shows the summation of the challenges and strengths of the Waterfall model during the different activities in development. Note that the challenges outnumber the strengths, and that the importance of some it's strengths is a bit questionable. It is evident that the Waterfall model is very vulnerable to any kind of change during development and, as described in section 2.2, change is very much a constant factor in software development.

Challenges	Strengths
Not likely that all requirements are known in the beginning and requirements might change after initial planning phase is done. Unforeseen problems are poorly handled. Some documentation is produced but might never be used.	Clearly defined project length and cost. Design problems can be discovered early and counteracted.
	Implementation challenges can be discovered early and counteracted.
Testing is done after all analysis and design, so problems affecting these phases is very expensive to rework. Delays earlier in the process will hurt testing since this is the last phase of the project. Stakeholders can be concerned of progress since nothing product-related can be shown.	Since implementation is gated behind other phases, the chance that each feature is clearly defined is high.
Best applied on well understood and non-changing project, which is rare. Documentation is static and gets outdated. Documentation is expensive to produce, and often produce no direct value for the customer.	Extensive documentations. Forces discipline through an experienced, gated and rigid development scheme.

Table 2.1: The challenges and strengths of the Waterfall model

2.5.1.2 Extreme Programming

Subsection 2.4.3 outlines how XP works, but also brings up both some points of critique and some of it's strengths, which will be elaborated here. Most of the points discussed here will be present in other agile approaches as well, but are usually more pronounced in XP, whether good or bad.

Strengths The three absolute main strengths of XP is that it produces higher quality code, completes projects faster and the finished product is often time more suited for the business due to high customer involvement. Closely after that is how adaptable XP is to changing requirements, and it's built-in protection against feature creep due to Test Driven Development. All of these is some very powerful strengths, and are the main reason XP is held in such high regard by it's practitioners. Some of the supporting strengths of XP also includes Collective Ownership, Daily face-to-face communication, self-documenting code, Continuous Integration and Refactoring. Maintenance is also supported quite nicely in XP, since automated test suites ensures that changes post-development do not break functionality somewhere in the system.

Challenges The main source of challenges XP faces as a methodology is from misuse, either using it in a scenario in which it is not suited, or usage by people with poor understanding of it. I will start with applying XP in inappropriate scenarios. Many organizations are not used to the very agile approach of XP, and this conflict can cause problems and severe slowdowns. XP prescribes face-to-face communication across the team, which is an issue in large teams as Barlow et al shows in Figure 2.8 due to the exponentially increasing amount of social interactions needed in a team, where each member communicates with each other member. Another problem that relates to applying XP to the wrong scenario, is when it comes to documentation. Some customers might expect and / or require a certain amount of documentation, either as "proof" that the development team knows what it is doing, or to fulfil some industry standards, such as those for safety or pharmaceutical use. There indeed exists ways to counteract this in agile projects, but these will be considered in subsection 2.5.3.
Lets now take a look at challenges that arise from not having a proper understanding of XP. Only partially implementing XP can cause severe problems, since multiple parts are required for optimal use, as Fowler [16] describes with the change curve. Scope creep can occur if the development team is not disciplined when dealing with the customer, since it is easy for the customer to keep requesting more features. The evolutionary design of XP can degenerate into "code and fix", if automated testing, refactoring and continuous integration are not applied properly. To avoid these points, an XP project requires experienced

programmers and XPers. Relying on Test Driven Development and automated testing can also lead to disaster if performed blindly by an inexperienced developer, since the strength that these provide are only a false sense of security if not handled correctly.
The only challenge that cannot be avoided by having a proper understanding of XP and only applying it to appropriate projects is related to documentation. Having no documentation to show what code or modules are used for can make it difficult to re-use parts of a project in another project. While not directly a problem for a single project, this is still something to keep in mind.

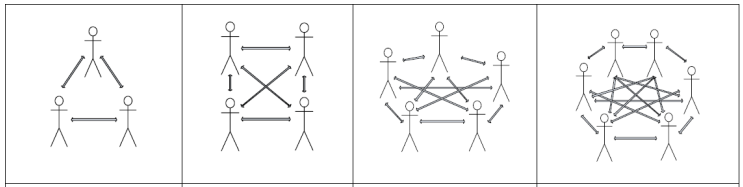

Figure 2.8: The exponentially increasing complexity of a communication network [3]

Conclusion The strong focus on code is evident in Table 2.2 which summarizes the challenges and strengths of Extreme Programming. The challenges of XP are almost exclusively based on misuse from not understanding it, or applying it to inappropriate scenarios. Solving these issues by making XP more suitable for it's bad cases and improving or removing the areas that is the cause of misuse will be important going forward.

Challenges	Strengths
Lack of up-front planning can conflict with the rest of the organization. Daily stand-up is only really useful on small to medium teams, and with committed team members. Customer might require a certain amount of documentation.	The daily stand-up meeting focuses on being short and efficient, while still communicating problems, solutions, and promote team focus throughout the team.
Evolutionary design can lead to "code and fix". Can cause problems in large teams, due to communication concerns.	
Lack of documentation can make it difficult to re-use code across projects.	Collective ownership of code prevents bottlenecks. Daily face-to-face communication promotes shared knowledge of code base. Code is self-documenting, and thus is always up-to-date compared to comments and other documentation. Continuous Integration, TDD and refactoring to ensure quality. Somewhat prevents feature creep.
Only partially implementing XP can cause severe problems, since multiple parts are required optimal use. Requires experienced (senior) programmers / XPers to prevent falling into pitfalls. Scope creep if lacking discipline and customer keeps requesting more features. Requires on-site customer. Requires large amounts of trust between all involved.	High customer involvement leads to good domain knowledge and thus solution suited for the customer and business. Able to use knowledge from previous iteration to improve upon current iteration. Is very adaptable. Have no problem with changing requirements. Produces higher quality code. Completes projects faster.

Table 2.2: The challenges and strengths of the Extreme Programming

2.5.1.3 Scrum

As mentioned in subsection 2.4.5, Scrum is often combined with some kind of generic agile development method but this will not be covered here. Instead this section will focus entirely on the management aspects of Scrum so that these might be highlighted.

Strengths While Scrum is a management methodology, it is still very much an agile approach. It uses short sprints with a review and retrospective in the end to improve the process for the next sprint. These two meetings are great for showing progress to stakeholders, which is important according to section 2.3. Other agile principles that Scrum uses to great effect is high customer involvement ensuring that solutions are business-appropriate, and the standup meeting, called dailt Scrum, which helps focuses the team and show their commitment. Scrum is also very adaptable to new and changing requirements, which further benefits from the customer involvement. Also of great importance for Scrum is that projects have a higher chance to finish on time and budget, and developers are usually more engaged and satisfied with their job, thus performing better.

Challenges As with XP, there is a chance that more items are continuously being added to the product backlog. This is especially a problem if more time and/or resources are not given to the project, so this balance needs to be controlled. Scrum is often best with small, committed teams, and large teams are often considered problematic. This can be counteracted a bit by some changes (such as Scrum of Scrums) but I will not delve further into this. For a Scrum team to work optimally, it requires mutual trust between Product Owner, management and the development team, that the Product Owner is knowledgeable, and Scrum Master is good at keeping the process running smoothly.

Conclusion The location of the challenges and strengths of Scrum in Table 2.3 quite clearly reflect that this is not a development methodology like XP or Waterfall, but a management methodology. The widespread use of Scrum can be attributed to the power of it's strengths versus it's challenges, and the ease of implementing it and transitioning to it.

Challenges	Strengths
Chance of scope creep, since the Product Owner can keep adding more items to the backlog unless controlled. Daily Scrum is only really useful on small to medium teams, and with committed team members.	The daily Scrum meeting focuses on being short and efficient, while still communicating problems and solutions. Also promotes team focus throughout the team.
	Adds visibility to stakeholders of development progress.
Requires a good Scrum master to keep the process running smoothly without taking too much time away from development. Requires mutual trust between Product Owner, management and the development team. Works best when team members are committed to the project, and not splitting their attention across multiple tasks in other projects. Requires knowledgeable Product Owner on-site.	High customer involvement leads to good domain knowledge and thus solution suited for the customer and his business. Able to use knowledge from previous iteration to improve upon current iteration, through sprint review and retrospective. Is very adaptable. Have no problem with changing requirements. Projects finish on time and budget. Improved employee engagement and job satisfaction.

Table 2.3: The challenges and strengths of Scrum

2.5.1.4 Spiral

The Spiral Model relies heavily on risk analysis as described in subsection 2.4.8. This makes it a rather expensive methodology to follow, which might be suited for some few projects, but the process of risk analysis is often overkill.

Strengths The strengths of the Spiral Model is mainly based around it's Invariants and Anchor Point Milestones. The milestones ensures that the project has reached a certain goal before moving too far, and the invariants makes sure that the project stays on course, and the appropriate tools and methods are applied.

Challenges At a glance, the challenges of the Spiral Model seems sparse compared to the other methodologies. While some of the other methodologies have some surmountable challenges, those of the Spiral Model is severe and stems from the core of the methodology. Expert knowledge of risk analysis in software projects is needed, and most likely also expert knowledge in the business and technologies used for proper risk assessments. This will inevitability results in a higher final cost for the project [19]. While the Spiral Model adapts to the project using Invariant 3 & 4, risk analysis would still be overkill for a lot of smaller and / or low-risk projects.

Conclusion At first glance on Table 2.4 , the Spiral Model looks promising. But the additional cost associated with risk analysis is too often not worth it. However for some projects, namely those that often is considered troublesome for agile approaches, the risk analysis provides a clear advantage.

Challenges	Strengths
	Cost, Schedule and other estimates become more and more precise and realistic the more cycles have been done. Invariant 6: Looks beyond software, to a more system-level business case, which might reveal other problem areas. Have a clearly defined milestone (LCO) to determine when the project is specified and well understood enough that design can start.
	Have a clearly defined milestone (LCA) to determine when the project is completely specified and understood, and all risks are found and managed, so that actual implementation may begin.
	Software is produced early and incrementally. Have a clearly defined milestone (IOC) to determine when to show-off progress to customers.
Contains many pitfalls that invalidates the Spiral model and turn it into a "Hazardous Spiral Look-Alike" which is likely to fail. Can be *very costly* due to the amount of risk analysis involved. Requires knowledgeable and experienced risk analysts. Even though it adapts to the project, the risk analysis is overkill for smaller or low-risk projects.	Invariant 1: Concurrently determine a compatible and feasible combination of artefacts and requirements to avoid premature commitment. Invariant 2: Avoids commitment of resources in any phase to unacceptable or overly risky plans, designs or implementations caused by lack of key stakeholder participation. Invariant 3 & 4: Level of effort driven by risk considerations. Invariant 5: Allows for incremental commitment so both customer and developer can withdraw early without great losses.

Table 2.4: The challenges and strengths of Spiral

2.5.1.5 Comparing to the main points of failure

How these four methodologies hold up against the most common points of failure presented in section 2.3 will now be discussed. XP and, to a lesser extend, Scrum can have trouble with A(1) and A(2), since many businesses is more traditionally structured. However, both of them focus on counteracting B(4), C(1), C(2) and C(3). Scrum also put emphasis on B(2) and B(6), making it a very strong candidate against the main points of failure. XP, on the other hand, goes a long way to improve B(5).

Waterfall have trouble with B(2), C(1),C(2) and C(3). If any unexpected change occur Waterfall can easily have trouble with B(6), and the lack of implementation focus can result in troubles with B(5). The Spiral Model excels at B(1) for obvious reasons, and it's milestones and invariants does good work on B(2), B(3), B(4), C(1), C(2), C(3). The Spiral Model again looks like a very promising methodology were most issues are handled with no obvious downside, but the main problem of the Spiral Model is still the cost associated with risk management.

2.5.2 Team and project properties

In this section I will present easily identifiable properties of both project and team, and how each methodology relates to those. Each methodology assigns each property a degree of applicability, between 1 (do not use) and 5 (highly recommended to use). This is done so that a new pick-and-chose framework, which can be optimized for a concrete project, and a solid core, can be designed in the following chapter. The properties are grouped in two categories; project types (safety critical, time to market, etc.) and team types (size, experience, etc.) - and the reason for the score in each property are identified. As mentioned, the properties are selected because they are easily identifiable, so that the right choices can be made without loss of time. The following subsections are based on the same sources as those listed for their respective methodologies in section 2.4. If these scores should be used for choosing a methodology, they would need to be weighted and then the methodology maximizing the scores should be chosen (given that the required knowledge / experience is present) and not considered without the descriptions, since that could lead to misunderstandings (see the note on XP and stable requirements for instance).

2.5.2.1 Team properties

Table 2.5 shows the degree of applicability or benefit for each team property which is considered. What each property means, and any notes on regarding the property and a methodology, is explained in the following paragraphs.

	Waterfall	XP	Scrum	Spiral
Experienced	4	5	4	5
Inexperienced	3	1	2	1
Tiny (1-3)	2	4	1	1
Small (4-8)	2	5	5	1
Medium (9-14)	4	5	5	4
Large (15+)	5	3	4	5

Table 2.5: The benefit or degree of applicability for each methodology and team property

Experience This point covers both experience and inexperience in developing and other expertise, such as Risk Analysis for Spiral or the single Scrum Master for Scrum. Even though Waterfall certainly benefits from having specialists perform the different activities (requirement engineering, architecting, etc.), these are not as vital as having experience in XP for an XP project, where evolutionary design and TDD can lead to disaster if not handled carefully.

Size The given team sizes are not set in stone, but merely an indication. Scrum as a management methodology does not make much sense to use in a tiny team, since management is not as necessary with such as small team. For Spiral, you need at the very least one risk analyst and preferable more, and thus a team need a certain size to be viable. XP is often said to be only viable for small to medium sized teams, which was true in the beginning, but according to Beck & Andres [4] it has been proven to work in large teams, but smaller team sizes are still preferred. The concept of "Scrum of Scrums" helps with using Scrum in large teams.

2.5.2.2 Project properties

Table 2.6 shows the degree of applicability or benefit for each project property which is considered. What each property means, and any notes on regarding the property and a methodology, is explained in the following paragraphs.

	Waterfall	XP	Scrum	Spiral
Documentation required	4	3	3	5
Testing required	1	5	3	4
Time-to-market	1	5	5	3
On-site customer	1	5	5	2
No on-site customer	5	1	3	5
Stable requirements	5	2	4	4
Unstable Requirements	1	5	4	3
High risk	2	2	2	5
Low risk	5	5	5	1
Maintenance requried	2	5	3	2
Small budget	2	4	4	1

Table 2.6: The benefit or degree of applicability for each methodology and project property

Documentation required This point covers compliance to a standard, and the more general case of documentation as proof that the team knows what it is doing, which is a point that can be required by the customer. The reason XP and Scrum does not receive a lower degree of applicability is thanks to Agile Modelling, which would treat these documents as any other work item. They still receive a lower score than Waterfall and Spiral since these have some tried and tested document structures embedded. Due to the risk analysis aspect of Spiral, it should produce higher quality documentation.

Testing required This point is important for safety-critical projects, such as the nuclear plant example from Spiral. Scrum achieves it's score by being iterative and allowing customers to test the system. This is of course not as structured and formal as automated testing in XP, which is why XP achieves

the highest. Spiral achieves it's rank by prioritizing testing in such as project, and by carefully analysing the risks to know where to focus it's efforts. Waterfall does include testing, but does so in the end of the process, and delays early might result in reduction of the testing phase, and is otherwise not known for solid testing.

Time-to-market In some businesses quick development is key, and this point covers this. Both Scrum and XP focuses on short iterations and providing a viable product after each iteration. This makes them ideal for projects where speed is of the essence, since a product could be developed very quickly, and then if more time was available development could continue. If the customer then abruptly decides that it is time to finish the product, the system is always in a deployable state and ready to go. Spiral will adapt to the speed requirement, but the risk analysis and milestones will still make it less than ideal. The very rigid is structure of Waterfall makes it appeal very little to this kind of project.

On-site customer This point covers both on-site customer and knowledgeable customer representative. XP is very dependent on the on-site customer. Scrum also needs either an on-site customer or knowledgeable customer representative for functioning optimally. However, with external software developers the role Product Owner is often filled by the external party. This is not ideal, but in practice has been found to work, as long as the Product Owner understands the customer, the business and the market well enough. This can be gained by having ongoing communication with the actual customer. Waterfall mainly uses the customer up-front when doing requirements, while spiral have a bit more ongoing communication with the customer due to it's milestones. Neither is really hurt by not having an on-site customer, and they do not benefit very much from it either.

Requirements Waterfall is very ill-suited for unstable requirements, and benefits greatly from stable requirements. While XP do not really benefit from stable requirements, it is important to note that the low score reflects just that, it does not gain anything. It should not be understood as a project with stable requirements should never do XP. Scrum is easily applicable on projects with unstable requirements but it's more structured nature compared to XP's "orderly chaos" makes it a bit less capable than XP. Scrum does in fact gain some benefit from having stable requirements since the Product Owner then has less work to do. Spiral handles requirements better than Waterfall but worse than Scrum and XP, since it does rely on it's milestones but it's iterative nature helps improving it compared to Waterfall.

Risk This point covers not just the risk of rewriting code and documents when requirements change, but also more high level risks as those presented with the Spiral model, such as committing to an inappropriate component or architecture. The only really suitable methodology for high risk projects are the Spiral model. The other methodologies would need something to negate the risk, such as high experience in market or platform, to make sure that a high risk project would be feasible undertaken. On the other hand, low risk projects are very much suited for Waterfall, XP and Scrum, but should Spiral due to it's risk driven nature.

Maintenance This point is rather general since it covers both the quality of the code base (the chance that an unknown bug lurks deep within), how easily the system can be changed by an outsider, and how easy it is for outsiders to understand the actual implementation. Agile is empirically shown [33, 2] to produce code of higher quality, having automated testing with a large coverage, and having self-documenting code. These are all very strong points for maintainability since automated testing ensures that no part of the system is broken by doing changes elsewhere. Furthermore, the code should be easily readable by itself and the system metaphor, instead of having to compare possibly complex code with huge documents and diagrams.
Scrum, being an agile approach, should produce code of high quality, but it does not mandate any of the extra practices of XP. This does not mean that they could not be done within the Scrum framework, but their are not mandatory by default.
Waterfall and Spiral produces documentation which could help with understanding the system before making any changes, but this is by no means an guarantee since the documentation might be as complex to go through as the code base, and changing requirements might render some of the documentation outdated or straight up wrong.

Small budget None of the methodologies can be considered ideal for small budget projects. The focus on short iterations providing working, deployable software, makes XP and Scrum better, but they both require expertise that probably is not cheap to acquire. The documentation heavy Waterfall model contains a lot of work which does not directly provide value to the customer The expensive risk analysis of Spiral makes it very much unsuitable for small budget projects.

2.5.3 Tools and practices

This section will give an overview of the tools and practices which will be considered as the building blocks of the pick-and-choose framework presented in the next chapter. I will first present the tools and practices from the four main methodologies in Table 2.7, and then briefly touch upon other tools.

Waterfall	XP	Scrum	Spiral
Use case	User Stories	User Stories	Invariant 1-6
Requirements Specification	Acceptance test	Acceptance test	LCA milestone
Design Description	CRC cards	Goal commitment	LCO milestone
Modelling frameworks	TDD	Sprint Review	IOC milestone
	Automated testing	Retrospective	Risk analysis
	Continuous integration	Daily Scrum	
	Pair Programming		
	Code Review		
	Collective Ownership		
	Daily standup		
	Self-documenting code		
	Metaphor		
	Simple Design		

Table 2.7: The tools, concepts and practices, ordered by their originating methodologies

Even though modelling frameworks such as the UML is not exclusive to Waterfall, it is more frequently used there than in the other methodologies. User stories and Acceptance tests appear both in XP and Scrum, since both methodologies apply it.

Except the tools listed in Table 2.7, there is also Kanban and Agile Modelling. These are both applicable on iterative methodologies, and helps some of the troubles these might face such as, lack of understanding from organisation and executives, documentation, and gives a nice and easy to use visual guide.

2.6 Conclusion

Many software development methodologies have been developed over the years, and some of them have been presented in this chapter, along with the overall software life cycle. A subset of these methodologies have been further analysed

for their strengths and where they are challenged. A list of common causes for failures in software development has been identified and presented, and how the subset of methodologies hold up against the list has been analysed. The tools and practices used by the chosen methodologies have also been listed, making it easier to design a new framework using these as components. Finally, some general team and project properties are identified, and how each methodology work with them is analysed. This section has all the information needed to proceed in the next chapter to design a adaptive framework for software development, where a solid core is provided along with a toolbox with components that can enhance the core when certain properties are present in the team or project.

Chapter 3

Design

In this chapter I will design a framework for software development. The framework will have a solid core which consist of a proven methodology, with some components added to strengthening the methodology where appropriate. Along with the core of the framework I will supply a toolbox of additional components, categorized in packages, which will allow the team to adapt the core to their specific needs, be it customer or team dependent. The framework will also contain ways to transition to the framework, which is also applicable for other agile methodologies. As mention in subsection 2.4.3, it has become common practice to use systems for Continuous Integration, such as svn or git, so applying Continuous Integration is taken for granted during this chapter, as most people already do this, whether they know it or not.

3.1 Selecting a starting point

Selecting the core for the framework is obviously important, and the decision is based on several factors presented in chapter 2. I have chosen Scrum as the core for the framework, and the reasoning will be presented in the following.

First of all, Scrum is already widely used and accepted (55% of the respondents of the State of Agile survey [33]) making it a familiar sight to many. It does not

prescribe any engineering practices, focusing solely on the management-side of software development. This makes Scrum easily extendible without causing conflicts with established rules or norms when it comes to development. Scrum also handles many of the most common causes for failure in software development identified in section 2.3 , replicated here.

A **External issues**

(1) Outside pressure (Commercial, rivals, first-to-market, organizational)

(2) Lack of support from executives and/or organization

B **Internal & Validation issues**

(1) Insufficient risk management

(2) Poor status reporting

(3) Use of immature or unsupported technologies

(4) Team not able to handle the complexity of the project

(5) Bad development practices

(6) Poor resource management and estimates

C **Verification issues**

(1) Lack of customer involvement

(2) Unspoken, unknown or unrealistic project goals and requirements

(3) Poor, or lack of communication between all parties, including developers, project managers and customers

The issues which Scrum takes care of are B(2), B(4), B(6), C(1), C(2) and C(3), leaving us to focus on the remaining five issues. A(1) and A(2) deals with external issues, and while these can be mitigated to some degree, the focus will be on the other three. In subsection 2.5.2 it is clear that Scrum is generally a strong choice for most properties, making it an ideal starting point, where practices from other methodologies might be applied to improve it against properties where it does not scores among the top. Finally, Scrum does not have any inherent weaknesses, as seen in Table 2.3, besides the team lacking experience. Granting experience in Scrum is outside the scope of this framework, but industry experience and / or becoming a certified Scrum Master will go a long way.

3.2 Strengthening the core

In this section I will focus on improving the core of the framework, so that the core in itself provides some solid advantages during all development activities. Some weak points are pointed out in section 3.1 which are important to improve upon, but other areas that could be improved will be identified here as well. Only if an activity or method provides significant advantage and is generally applicable will it be added to the core, otherwise it will be an optional component that can be added from the toolbox.

3.2.1 Guarding against the common points of failure

From the list of common issues in software development in section 2.3, the following points are not sufficiently handled by Scrum; A(1), A(2), B(1), B(3), B(5).

As mentioned before, A(1) and A(2) deal with external factors, and are not easily countered by a development methodology. However, the commitment focus of Scrum fits neatly into a more traditional company structure, which should help a bit. A practice that is well known by most managers and executives, and not expensive for a development team to implement, is Kanban as described in subsection 2.4.6. It is an established lean practice which have been around for decades and thus should appeal managers and executives, signalling that the team is working in a lean and efficient way. It also gives these two stakeholders the option of easily keeping tabs on the project without interfering with development, again something that might be needed in a more traditional control-and-command company structure. Furthermore, Kanban will prove to be a good way of transitioning to an agile approach, as will be described in section 3.5.

B(1) and B(3), while definitely important, are best handled by components in the toolbox since these are often not relevant on smaller projects or project types which the team is experienced in completing, and is expensive to guard against.

While B(4) is somewhat helped by Scrum's management structure of dividing work items into small tasks, and the additional focus provided by the small, time-boxed sprints, this is definitely a point that could still be improved, while also improving B(5). A key practice in reducing complexity and improving code in XP is Test Driven Development. Applying TDD to the core of the framework might seem to go against the previously stated *"... and is generally*

applicable". Simple projects or projects with a focus on time-to-market do not seem to be suitable targets for the additional work of TDD. However, XP practitioners argue that in the long run TDD actually is a net gain in time spent, since developers will spend less time on backtracking, changing, and debugging code. And even for simple projects, there will be parts of the system which are more complex, critical and / or risky. These parts would then be developed using TDD, while the rest could be developed without if the project was simple enough. So, the core will include TDD, but not mandate TDD throughout the system. Instead, the most complex, critical and / or risky components of the code should be developed with TDD. As the complexity of a project increases, the amount of components developed with TDD should increase as well. To further strengthen both B(4) and B(5), collective ownership and discussion of code is also encouraged, along with coding standards to make the code easily readable. Collective ownership can be accomplished by Moving People Around [35] and by simply discussing different technical solutions during or after the Daily Scrum.

Specifically improving B(5) will be done by implementing some practices from XP. Do note that the entire XP methodology is not part of the core, and Fowler states [16] that using evolutionary design without flattening the change curve, which is done by implementing a large majority of the XP practices, is risky and not recommended. The core contains TDD, which goes a long way, but something more is needed to guide design. Using CRC Cards to understand how the system should work, and documenting it with diagrams, seems a good solution, that makes it possible to understand and design complex systems, and avoiding unsupported evolutionary design. For this we draw upon some core principles of Agile Modelling, *"Depict Models Simply"*, *"Use the Simplest Tools"* og *"Discard Temporary Models"*. *DMS* states that you should only model the necessary parts of the system for what you are trying to tell with the diagram. Do not model the engine if all you want to tell is that the car can move forward. *UST* advocates that there is no reason to use a complex UML designer if what you want to model can easily be made on a whiteboard as a sketch. *DTM* tells that developers should not be afraid to model just so that they can learn to understand the system, and then throw away the model afterwards. In other words, do not spend hours on perfecting a huge diagram just so that you can understand the component and then never look at it again. Rather make a quick temporary model, learn what you need from it, and then discard it.

3.2.2 Improving the scores

Taking a look at the scores of Scrum from Table 2.5 and Table 2.6 reveals the following properties with a score below 4, which I will look to improve as part

of the core:

- **Inexperience**. For scrum this mainly focuses on Scrum Master and, as stated previously, this is outside the scope of the framework.
- **Tiny**. For a team of 2 or 3 it might be too time consuming to spend large amounts of time discussing code, and for a "team" of 1 it is even impossible. This might result in a system of lesser quality containing more bugs. For these reasons TDD is even more important for tiny teams with 1-3 members, and applying TDD should improve this score.
- **Documentation required**. This will be improved by applying agile modelling.
- **Testing required**. This will be improved by using TDD.
- **No on-site customer**. More on this below.
- **High risk**. This will be somewhat improved by TDD, but the real gain will come from the toolbox, so that it is only applied when deemed necessary.
- **Maintenance required**. Implementing TDD and easily readable code will bring this up to par with Extreme Programming.

The big risk of using Scrum, and Extreme Programming for that matter, is if the customer can not be on-site to support the development efforts. While it will definitely be preferable if the customer can be on-site, an alternative is now presented in the form a specific interview process. This interview will take the form of a process of gathering information and reviewing it with the customer to gain enough knowledge to act as product owner, and is presented in Table 3.1. While a customer interview in the beginning of the Scrum process is not unusual, this is a more structured and extended version.

1) Create CRC cards in collaboration with the customer. These should not be created to necessarily simulate code, but rather to give an understanding of the objects of the system form the customer's point of view, and how they interact.

2) Have the customer show you some of the key interactions of the system using the CRC cards.

3) Create User stories with the customer as normally in Scrum.

4) Review the user stories by replaying the story using the CRC cards to make sure that it is correctly recorded.

5) **Optional**. If a story is especially complex or important to the customer, consider creating a use case.

6) Create acceptance tests for each story, to make sure that the success criteria of a user story aligns with the customer's wishes.

7) Review the acceptance tests in combination with the user stories.

Table 3.1: The process for interviewing a customer to gain enough valuable insight to act as a product owner

3.2.3 The final core

Given this information the core will work as illustrated in Figure 3.1, which is an extension of Scrum. First an interview is held with the customer as is normal in Scrum, but this is extended with activities to further imprint the knowledge of the customer on the Product Owner. During this interview, the CRC cards are created with the customer using his business terminology first. Then these are used to illustrate key interactions in the system, and these stories are recorded. If they are complex or especially important, a use case is created as well. Acceptance tests are created for each story to determine when the task can be considered complete. This is all then reviewed with the customer to detect if any misunderstandings have happened, or some inconsistency is present.

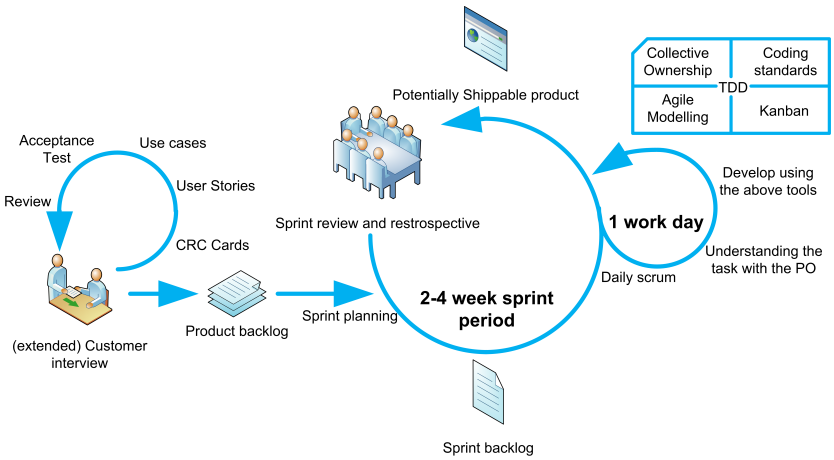

Figure 3.1: The core of the framework which is built upon Scrum

The user stories and their acceptance tests created during the interview will form the Product Backlog. During Sprint Planning, the most important of these items are included in the Sprint Backlog. Each workday then starts off with the Daily Scrum, where today's commitments are announced along with progress from yesterdays commitments. Problems are brought up as well, but their solution is discussed among the relevant developers afterwards. This along with changing tasks promotes collective ownership. When a task has been given, the developer will talk to the Product Owner to understand the task properly, and an architect is possibly consulted and design is discussed with some simply, agile models. When coding starts, some agreed upon standards are followed when using Test Driven Development. As the day progresses, the Kanban board is updated so that the entire team, managements and other stakeholders have easy access to the status of the project.

3.3 Pick-and-Choose toolbox

In this section, the toolbox will be presented. It will contain a mapping from easily identifiable properties to possible ways to enhance the core in regards to the specific property. The toolbox could easily be extended by simply adding another such mapping if a practitioner of the framework found an unidentified issue and it's solution. It is not expected that this toolbox will cover every possible property or solution, but it will enhance the core when it comes to the properties where the core is performing below expectation. Furthermore, this

section will present guidelines on how other properties and their solutions could be found, documented and added to the toolbox. The following properties have been chosen as the focus of the current iteration of the toolbox; *Documentation required*, *Testing required*, *High-risk*, *Small budget*, *Tiny team*, and *Large team*.

3.3.1 Extending the toolbox

When extending the toolbox, the first step is to identify which property is causing problems. These properties should be as simple and easily identifiable as possible, so that a team working on a new project can see right away if this property is present in their project. Other property categories could be added, such as organization or market. Whenever possible, break down the property into smaller, more specific pieces to allow for more customization when it comes to the solution and having a highly targeted solution.

The next step is finding a solution. Where the solution fits into the core needs to be pointed out so that it can be correctly applied. If the solution has any synergies these should be mentioned as well, such as TDD, Merciless Refactoring and Continuously Integration makes evolutionary design a possibility [16]. If the proposed solution has any opposition it should be mentioned as well, such as the example with Risk Analysis as a solution does not go well with the properties Small Budget and Time-to-Market. Even if a solution is not discovered along with the property, it should still be added so that a solution might be found in the future, and other team can be aware of the problem.

In Table 3.2 a form is proposed to represent elements in the toolbox.

Property name	
Property description	
Solution description	
Solution synergies	
Solution opposition	
Placement in the core	

Table 3.2: The form to be filled out, so that a property-solution pair can be added to the toolbox

3.3.2 The toolbox

Using the form in Table 3.2, the focus properties that are part of the initial toolbox will be presented on the following pages. Each of the tables Table 3.3 through Table 3.8 represent a toolbox element for a specific property.

In the case where the solutions of two properties conflict with each other, such as 'Small budget' and 'Large team', the developers will have to weight the conflicting properties against each other to determine which of them is the most important. This would then result in the solution of the most important property to be used.

Property name	*High risk*
Property description	High risk projects are those that contain multiple components the team is not experienced with developing. A team specializing in creating banking systems would not consider a new banking project as high risk, but other teams might.
Solution description	Apply both LCO, LCA, IOC and the Invariants of Spiral. These are the corner stones of the Spiral model, which is specialized for high risk.
Solution synergies	Milestones will also help with the Documentation required property.
Solution opposition	The additional work required by risk analysis for milestones and invariants makes this unsuitable for the Small Budget and Time-to-Market properties.
Placement in core model	LCO after the Product Backlog has been created, but before first sprint. LCA will be created iteratively before each sprint as part of the sprint planning. IOC during the Retrospective

Table 3.3: The toolbox element describing the property High risk, and it's solution.

Property name	*Documentation required*
Property description	Documentation might be required by a customer or the organization, either as part of some standard compliance, or as a proof that the development team is on the right track.
Solution description	Create models efficiently and just in time, as per Agile Modeling. For non-diagram documentation use the milestones of Spiral, LCO, LCA, and IOC.
Solution synergies	Milestones will also help with the High risk property.
Solution opposition	Creating additional documentation will slow down the process and thus will not work well with Time-to-Market and Small Budget properties.
Placement in core model	Modeling before coding. LCO after the Product Backlog has been created, but before first sprint. LCA will be created iteratively before each sprint as part of the sprint planning. IOC during Retrospective demo.

Table 3.4: The toolbox element describing the property Documentation required, and it's solution.

Property name	*Testing required*
Property description	An additional focus might be placed on testing for very critical systems, or systems with very little fault tolerance.
Solution description	Do full TDD. Use both Pair Programming and Code Review to ensure that as many mistakes as possible are found. Apply to both the code AND the test suite.
Solution synergies	None
Solution opposition	Using both Pair Programming and Code Review, along with full TDD, will slow down the process and thus will not work well with Time-to-Market and Small Budget properties.
Placement in core model	During coding in the work day loop.

Table 3.5: The toolbox element describing the property Testing required, and it's solution.

Property name	*Small budget*
Property description	When the project is running on a tight budget, there is no room for mistakes, so getting it right the first time is important.
Solution description	Make sure the initial interview is very thorough, and if possible have multiple customer representives review the result of the interview. Be very critical when to apply TDD, skipping as much as possible while still making sure the important parts are done.
Solution synergies	If requirements are known to be very stable, using this approach might also be beneficial, since getting it right the first time is plausible.
Solution opposition	Spending additional time on the initial interview might not be worth it if requirements are unstable.
Placement in core model	The initial interview and in the work day loop during development.

Table 3.6: The toolbox element describing the property Small budget, and it's solution.

Property name	*Tiny team*
Property description	When a team have 1-3 members.
Solution description	Having very few (or no) team members might make it hard, or impossible, to discuss the system, specific issues and review code. To combat this, TDD should be applied throughout the project, as well as an added focus on the interview, to ensure quality and avoiding costly rework. Furthermore there needs to be some role overlap due to the limited team size.
Solution synergies	None
Solution opposition	While these practices could be applied in most cases, unstable requirements often make this added focus wasted.
Placement in core model	Throughout the model, especially during the work day loop.

Table 3.7: The toolbox element describing the property Tiny team, and it's solution.

Property name	*Large team*
Property description	When the team working on the project is of 15 or more. A team might even benefit from this when at sizes of around 10-12.
Solution description	Apply Scrum of Scrums as described by Mike Cohen from Scrum Alliance [11].
Solution synergies	If a project could be divided into several fairly de-coupled subproject, it could provide additional benefit to split the team and apply Scrum of Scrum as well.
Solution opposition	Additional management should only be applied when needed, otherwise it just creates an additional layer of meetings.
Placement in core model	After the daily Scrum meeting, a Scrum of Scrum meeting would be inserted. This might not have to be held daily, but 2-3 times a week is sufficient.

Table 3.8: The toolbox element describing the property Large team, and it's solution.

3.4 The methodology

3.4.1 My Contributions

The developed methodology contains many existing practices, and a new practices inspired by an existing one. The new practice is the way CRC Cards are used during the initial customer interview. Normally, CRC cards are used by the developers to design the system, with each card representing an object [35]. The idea of using the cards to design the system is retained, but in this methodology it is used partly as a design guideline, but mostly as a way for the developers and Product Owner to gain a thorough understanding of the customer's vision of the system, before development starts. The reason for using this practices is to help answer the validation question of "Are we building the right product?".

How the rest of the practices have been combined is another contribution. What these practices does for their native methodologies have been analysed and this knowledge has been used to select how to strengthen Scrum to make it more capable of producing products which sufficient answers to the questions of Validation and Verification.

This methodology is based on Scrum with practices from many different methodologies, and as such some similarities are present. What makes this process distinct from other processes is it very intentional focus on *both* Validation, Verification and delivering on time and budget. This is achieved by cherry-picking practices from many methodologies, making sure they can function together, and presenting a framework for their use. Another problem that this methodology aims to solve is that many methodologies are not applicable in some circumstances. To fix this, the framework of the developed methodology contains a toolbox, such that each time it is used, it can be adapted to that specific project.

3.4.2 Using the methodology

To use this methodology, one must first determine the properties of the team and project, using the elements in the toolbox. Once this has been decided, any conflicts between the solution of these properties should be resolved to determine which of the conflicting properties are the most important. Now the framework will start to diverge depending on which elements of the toolbox are used, but generally the customer interview is held first. Here the customer's vision of the project and it's interactions are described using CRC cards. High-level user stories are then created, and the most complex or important user stories

are created as a use case as well. Finally, for each user story an acceptance test is created, defining the success criteria of each story. Then every item is reviewed by attendants of the meeting, and the Product Owner makes sure that his understanding of the system is aligned with that of the customer. The user stories, use cases, and acceptance tests form the product backlog as known from Scrum.

Then the first sprint starts and, as every other sprint, a sprint planning meeting is held. During the meeting, elements of the product backlog are selected for the upcoming sprint, and these are further defined by the developers so that they are ready to be developed. During the sprint, different practices are applied, again depending on which elements from the toolbox are used. Generally, each day starts with the daily scrum, followed by any clarification that the developer might need from the Product Owner. The Product Owner should have excellent understanding of the system due to the initial interview. Then development of the story starts, using mainly Test Driven Development. Agile Modelling might be applied if the story is non-trivial, have a lot of interactions, or is especially important. A Kanban board is used to track the progress of each story, beginning at 'To do' and moving to 'Doing'. Once a story is completed it could be move to an optional queue called 'Testing' or 'Needs review', otherwise it is moved directly 'Done'. Remember to use a system such as svn or git for Continuous Integration, and be sure to synchronise often. Finally it is important that some agreed upon coding standards are used by the entire team, so that any developer will be able to easily understand the code written by any other developer. This will help with the last important practices, Collective Ownership, along with making sure that developers continuously work within different areas of the system.

For a visual representation, the reader is referred to Figure 3.1. For the elements of the toolbox, the reader is referred to the tables in subsection 3.3.2.

3.5 Transitioning

As mentioned in section 2.2, transitioning from traditional development methodologies to an agile methodology such as this can feel like a big jump and be met with resistance. This is also confirmed by State of Agile [33], where respondents cited "Inability to change organization culture" and "General resistance to change", as the two main barriers to adopting an agile approach. A solution to this is presented by Sahota [32], in which Kanban is used to ease the transition; *"The gateway drug theory states that softer drugs (Kanban) can lead to harder drugs (Scrum, XP) ⋯ With Kanban, there are documented cases*

of teams spontaneously collaborating, learning, and noticing/solving problems. This has been my experience as well and would confirm the hypothesis of Kanban as a Trojan horse (containing Agile on the inside)." - Sahota [32]. This section aim to help with transitioning to this methodology, or any agile methodology in general, and is based mainly on the work of Sahota [32].

According to Sahota, it is important to identify the culture of the company first. If the culture seems ripe for agile, the methodology can be gradually implemented without much issue. The real issue is if company has a control culture, which is at odds with many core agile values. To work around this, it is important to not try a radical, over-night overhaul of the culture, but instead focus on doing it very incrementally by "$\cdots$ *identify practices that support the dominant culture of the company or group rather than to try and change it.*" - Sahota [32]. This is where Kanban comes in. Kanban is a very control focused process, which would align with a traditional control-and-command company structure. Interestingly enough, Kanban is also by many considered an agile practice, which seems at odds with it's control focus. So by following both advices of Sahota, using kanban as a trojan horse and supporting the dominant culture, a slow transformation to agile can begin, even in a hostile environment.

The next step would be to show agile solutions to existing problems, and here Sahota refers to Elssamadisy's *Agile Adoption Patterns: A Roadmap to Organizational Success* [14]. The approach chosen here is that of pain-driven adoption. This approach aims to identify problems on different levels, such as business (features are not used) or process (lack of noticeable progress), and then provide a set of agile practices which could help solving the problem. A good example of this is given by Sahota:

Problem:	Hardening phase is needed at the end of the release cycle.
Applicable Practices:	Automated developer tests, continuous integration, functional tests, done state, and release often.

So by first introducing the use of Kanban, making the organization comfortable with it, and then by finding the issues and presenting solutions to them, can agile slowly be introduced into an otherwise resistant company using a traditional control-and-command structure. Sahota suggests that the solutions should be presented as out-of-the-box components that can be used, instead of parts of an agile framework, since this could make the management worried that they are loosing control of the process.

An additional benefit of having Kanban as a part of the core of this framework,

is that it is a familiar sight, which both team, management and executives recognize from the transitioning phase. The solutions suggested during the pain-driven adoption should, if possible, correspond to the practices presented in the core or the toolbox, for the same reason.

3.6 Conclusion

In this chapter a framework for software development has been designed. Scrum was chosen as the core of the framework. Using the knowledge gained from the literature study in chapter 2, additional practices was added to Scrum, forming the foundation of the framework. These practices were chosen for their general applicability. Then a method to extend the framework was presented, in the form of components in a toolbox. A few of these components were created to fix problems specific to some projects or teams, but often coming with enough drawbacks to warrant their exclusion form the core. Finally, a process to ease the transition into the framework, or any other agile methodology in general, is presented. It uses Kanban followed by pain-driven adoption to smoothen the transition from a traditional command-and-control company structure.

Chapter 4

Implementation

In this chapter I will first give a high level description and analysis of the case, in which the developed methodology has been tested. I will then go on to describe how the testing proceeded to allow for easy reproduction, what the focus was and what limitation is present in this test case. In the same section, the results of the test are also presented. Following that, I will present the feedback from the customer, along with a discussion of the feedback and testing process. The methodology will also be discussed, after having tried it in action, so that it may be improved in the future. Finally a project lead and manager of IT Minds has been asked to evaluate the methodology and provide some feedback, which is presented and analysed in section 4.4.

4.1 Case analysis

For testing the methodology designed in chapter 3, a test case is needed. For optimal testing it should be a case of reasonable size and complexity but due to the nature of this project, being a one man team with limited time for development, this could not be achieved. Instead, the case would be of a smaller size and because the the owner of the case, referred to as the customer, had limited time only a subset of the case would be treated in-depth according to

the methodology. The chosen case was provided by IT Minds, a development and consulting company based in Aarhus and Copenhagen with more than 100 employees, which provided development for the customer.

4.1.1 Description

The case is about developing an online service to help administrate fairs and markets, but also provide services to multiple layers of customers. First off, it should allow organizers of fairs to easily have other people book stalls and other accessories. Accessories could be chairs or tables, and is owned by the organizer who rents it out. Stall owners use an interactive map of the fair to book stall areas, which updates according to current bookings. Everything related to bookings of stall areas and accessories are handled automatically by the website, including payment, such that an organizer only needs to provide a map for the fair and pricing details of the different areas and accessories. Since this service aims to ease the administrations of fairs, it is paramount that it is easy to use.

Part of the service is also advertisement through social media, to attract both stall owners and fair guests. Furthermore, fair guests should be able to search for fairs fitting certain criteria such as geographical location, types, stalls, or date.

4.1.2 Analysis

As described in subsection 4.1.1, there are multiple users of this service, each of them dependent on each other. All of these users needs some kind of value to want to use the service since it is a multisided platform[26]. This makes it an interesting case to work on, since there are many important points that each needs to provide value for a customer. The developed methodology is intended to help developers provide value to their customers so in that aspect this is a fitting case. The author is not fluent in front-end development, and for this reason the parts of the case relating to front-end development, such as the interactive map and easy to use interface has not been considered, only the back-end system.

The properties of the case, in regards to the methodology, are the following;

- Small budget
- Tiny team

Since this is a start-up project, the amount of funds available will be limited. According to the toolbox element in Table 3.6, an added focus should be placed on the interview to improve the chances that the developers get it right the first time. The other part of the solution is to be critical on when to use Test Driven Development, as resources will be sparse and a large test coverage on trivial parts of the system might not be the best way to spend these resources. The second property of the case relates to the team. The team consists of one person and is thus unable to discuss any issues with other team members. To combat this, the toolbox element in Table 3.7 suggests to use thorough Test Driven Development to make sure that the system is as solid and robust as possible. This is of course at odds with the advice of conservative Test Driven Development usage from the Small budget property. In a case such as this, the two properties will have to be weighted against each other to determine the level of Test Driven Development that should be used. In this case the Tiny team property was found to be the most dominant due to the nature of the service, being a multisided platform. Alienating just one of the user groups due to bad design pose a serious risk to the service, and the quality theoretically provided by Test Driven Development thus takes precedence. Tiny team also advises added focus on the interview which aligns with Small budget. So in this case, the choice is to extent the core with an extended interview including use cases, and full Test Driven Development.

The customer and author did not know each other before this project. The customer had, however, had dealings with IT Minds before.

4.2 Testing the framework

Since the customer only had limited time to spare, the focus has been on testing the interview part and how the results from this have translated into Test Driven Development. What was developed was then judged by the customer in how it holds up to his wishes and expectations. For the same reason, and since this MSc project is focused on the theoretical work, only a few features have been through the entire process. The features were selected by the customer as the most critical for the service. Other parts of the service have been handled in

the interview and will still be presented but they are incomplete. The reader is reminded that the development part of the testing is solely back-end and no User Interface was created.

4.2.1 The interview

The meeting with the customer was at the Copenhagen office of IT Minds, since a lot of space was needed to play out the key interactions of the system. The customer was asked to explain his idea, and while doing that, the entities that were mentioned were noted down on CRC cards. These entities could be viewed as objects as well, but since everything was done in the customers business terminology the word 'object' was not used. Once the initial explanation was done, the CRC cards were updated with collaborators, and finally with responsibilities where it was appropriate. During this part of the interview, the customer was asked both some practical questions and some critical questions about the system. These helped shape the understanding of the system, and often led to changes in the CRC layout, and possibly giving new perspectives or ideas to the customer, who with the help of the CRC cards could see the issues raised.

This part of the interview resulted in the cards laid out according roughly to their interaction with each other as shown in Figure 4.1. This session laid the foundation of the rest of the interview, and was frequently referred to and updated as the interview continued. Having a representation of the system at hand that was easily changeable and understandable by both developer and customer was a huge boon.

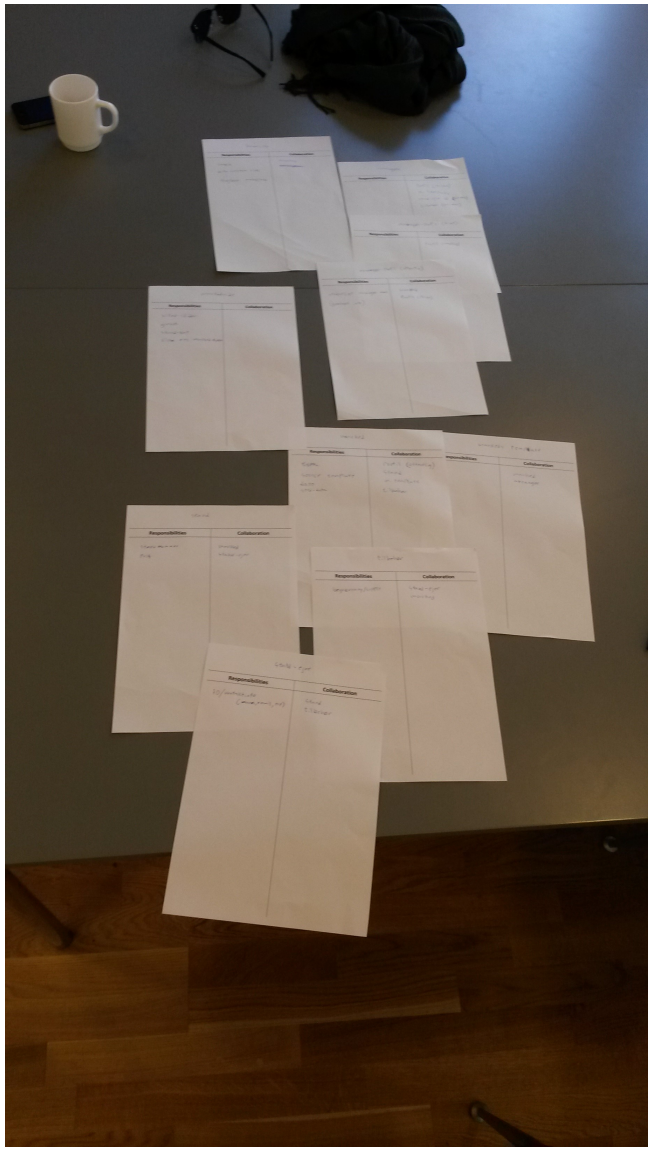

Figure 4.1: The result of the CRC session of the interview.

Once the CRC session was complete, the interview moved on to User Stories. These took the form of simple sentences and were prioritized in order of importance by the customer from 1 to 6, with 1 as the most important.

- As an organizer I would like to
  - create a fair. (Priority 1)
  - avoid the trouble of administrating a fair. (Priority 2)
  - offer accessories for rent to stall bookers. (Priority 3)
  - advertise my fair and stall areas to potential stall bookers. (Priority 4)
  - advertise my fair to potential fair guests. (Priority 5)
- As a stall booker I would like to
  - have an overview of stall area placements and price. (Priority 2)
  - have precise control of the placement of my stall. (Priority 3)
  - be able to rent accessories. (Priority 3)
  - advertise my stall to potential fair guests. (Priority 5)
- As a fair guest I would like to
  - have an overview of upcoming fairs. (Priority 6)

After the User Stories, time was running short and it was decided only to go forward with a single User Story which was the most important - creating a fair. Since this was a critical part of the system, and the development team was tiny, a Use Case was drawn roughly following the UML standard. Getting all the details just right according to the UML was not worth the time, and would also have resulted in the customer waiting for a long time, thus the principles of Agile Modelling were applied. A digital version of the use case is presented in Figure 4.2 which is a replica of the original use case found in Appendix B. As it is seen in the use case, two more features were actually part of the use case as described by the customer, namely login and profile creation. This could have been split into separate use cases, but the number of interactions was low enough, that it played little role.

To create a fair you need to be logged in and have a profile. A profile consist of a private part, containing the information the system needs for billing and such, and a public part which is shown to stall bookers and fair guests. A profile is created along with the user on the website. When an organizer is creating a fair the system is silently creating a template based on that fair which contain

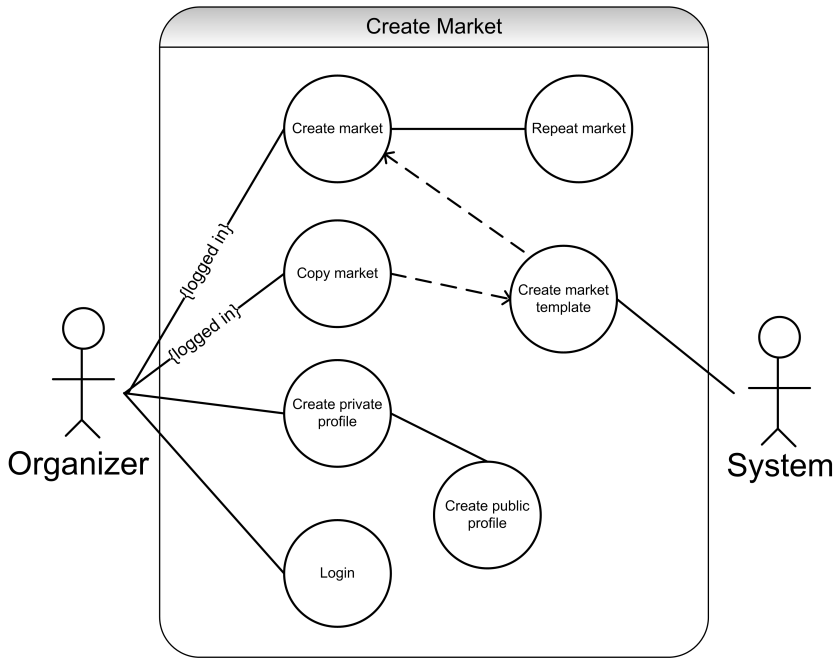

Figure 4.2: Use case containing fair creation and it's requirements, logging in and creating a profile

all information except the date. The same organizer can then repeat the fair on any number of other dates based on the template just by providing the new dates. This process happens when the original fair is first created. However, an organizer can also make a copy of an existing fair by getting the needed information from the template.

Once the use case was created the acceptance tests were made, defining the success criteria of the use case. Most of the acceptance tests were front-end and UX based. Outlining the success criteria helped getting some additional details about the system documented, such as what a page for a fair should contain, and how a profile was created. Looking at the use case, one could think that it was two separate actions to create first the private and then the public profile, but the acceptance test clarified that it actually should be combined into a single creation page. A cleaned up version of the acceptance tests can be seen in Figure 4.3, while the original can be found in Appendix B.

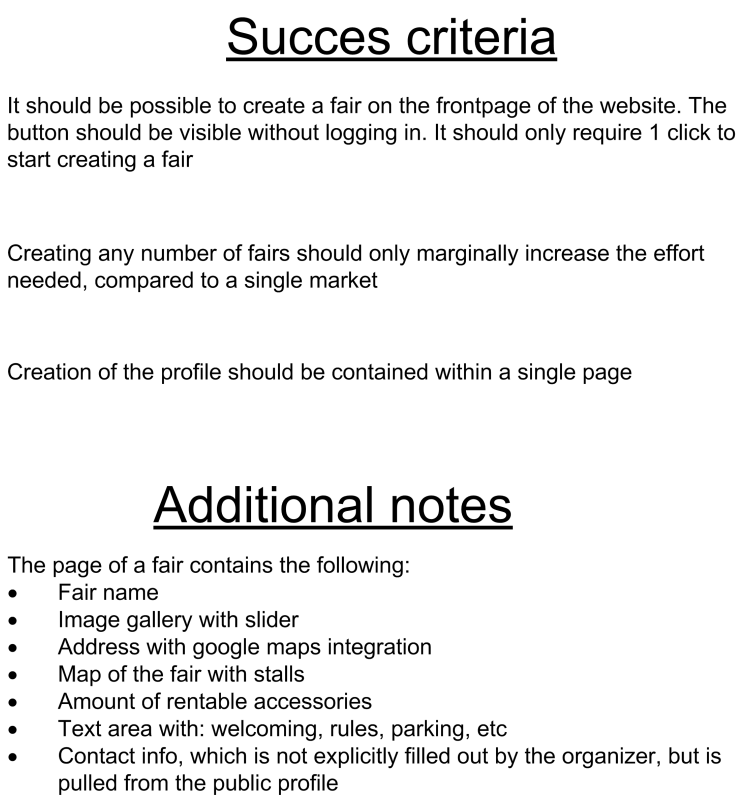

Figure 4.3: Notes from the acceptance test session of the interview

Once the acceptance test session was completed, every artefact created was reviewed by the customer. Furthermore, the developer explained how he understood the information in each artefact to ensure that everything was aligned with the customer's vision of the system. The main vehicle of the review was the CRC cards. The workings of the system was replayed by the developer, which referenced the other artefacts when relevant.

The interview took between 1.5 and 2 hours in total. Some time was spent talking about the process and methodology which would not have been relevant, had this not been a test. It would have been optimal if the customer had 1 to 2 hours more, since it would have allowed for most, if not all, of the system to go through the entire process.

After the interview, the class diagram shown in Figure 4.4 was created based mainly on the CRC session. This diagram is not meant as a representation of the actual developed system, as it represents the system seen from the customer's perspective. This provides some very solid guidelines for developing the actual system, since the idea behind the system is clearly defined in a way a developer understands, and yet no specific implementation details are enforced.

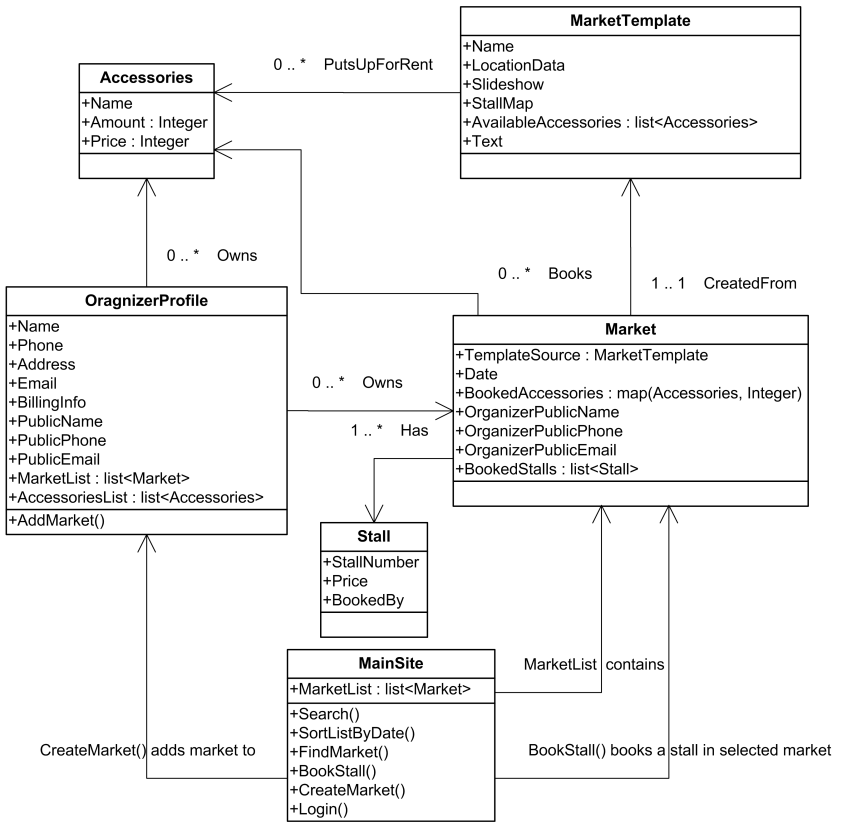

Figure 4.4: The result of the CRC session turned into a diagram for development purposes.

4.2.2 Development

The system was developed using Test Driven Development. As mentioned earlier, only the parts of the system that went through the entire interview process were developed, which of course resulted in an unfinished system. The development itself was straight forward and was based on the diagram in Figure 4.4. It provided an easy overview of how the different components fit together, and the tests were made on this basis. As mentioned earlier, the diagram in Figure 4.4 represents the system from the customer's point of view, and is thus not a design for the software system, but rather a guideline. The actual implementation differs slightly from those given in the diagram, but the overall interactions and structure remains the same. Most of the practices that the core of the framework prescribes, such as Daily Scrum and collective ownership, are not very suitable

for one-man teams and thus was not used.

The test suite is divided into three subgroups; tests for login, tests for profile creation, and tests for fair creation. The tests are developed using the NUnit Test Adapter for Visual Studio 2012/2013. Since only the parts of the system which have been through the entire interview process have been developed, there is no current way to run the code. Rather, the tests should be run to show the functionality. The idea is to have one test file for each component, and each file then contains a collection of test cases, called a TestFixture in NUnit. These TestFixtures then have several test cases which are named using the following convention: action_input. A small example of a two test cases and the resulting code can be found in Appendix C.

4.3 Case feedback and discussion

This section will present the feedback from the customer which was obtained after the work was completed. The section will also contain an evaluation of the methodology after having tried it in action, and a discussion of the feedback from the customer.

4.3.1 Feedback from the customer

After development was completed the customer was asked to provide feedback on the interview process, and evaluate if the developed test cases represented the system he had in mind. The following questions were asked:

1. Did you feel that the developer received sufficient information to understand the concept and develop the product through the interview process?
2. Did you feel that your project was in good hands following the interview?
3. Did you feel the time was well spent on the interview?
4. Would you consider using the interview process again in another project?
5. How would you compare the interview process to the start up of other projects in your experience?
6. Did you feel that the CRC session provided
   (a) high-level insight into the system for the developer?

(b) high-level insight into the project for you as the project owner?

7. General feedback and suggestions?

Question 1 This question was asked to determine if there were some areas of the project that were left untouched by the interview. The customer felt the interview in the end got around the project. He felt the CRC session gave a good, if a bit rough, overview of the system. It was good getting into the details of fair creation during the use case and acceptance test portion of the interview.

Question 2 This question was asked to determine if the customer trusted that the developer were well equipped to handle the project after the interview, since trust is an important factor. The customer felt that especially the CRC session provided him with a lot of trust in the developer. Some specific details were not covered by the CRC session, such as dates.

Question 3 This question was asked to determine if the interview had redundant elements. The customer felt that since the project was understood by the developer the time was well spent. After the developed test cases had been explained the customer noted that some front-end related subjects were not fully managed, which resulted in some slight misunderstandings between the developer and the customer.

Question 4 This question was asked to determine how the customer's overall experience with the interview process had been. The customer was very positive towards using the interview process again in another project. Having it proceed in a very structured way very useful. It felt very suited for a product owner, and for small development teams in a start-up companies.

Question 5 This question was asked to compare the interview process with the early phases of other projects that the customer have had. The customer said it would be *"a huge advantage"* and *"obviously a better idea"* to use this interview process for the start up phase of development, compared to earlier experiences. He also mentioned that the process was good for teams where new people were coming in during development, since there was a lot of artefacts that newcomers could use to get an idea of the overall project. The lack of such artefacts have, in the customer's experience, resulted in wasted time when new developers came to the project. Formally in Scrum and XP the only artefacts

created during initial planning is user stories, but these additional artefacts seems to be a good compromise between only user stories and Big Design Up Front of waterfall.

Question 6a This question was asked to to gather feedback specifically on the CRC session, since this part is a new contribution. The customer felt that the CRC session was a good tool to make the project more tangible, and provided a good means of communication between the developer and customer.

Question 6b Same as 6a, but specifically if the session provided any value for the customer. The customer felt that this process provided project owners with a critical and practical view of their system, which helped reveal certain details early on, instead of them being discovered during development.

Question 7 The customer liked that notes were taken and artefacts were created, saying "making things more tangible are very important". He felt that the customer did not need too many details of the hows and whys of the interview process itself. A suggestion was made that a short follow-up meeting was held a few days later to review the artefacts, and the developer should then pitch the idea back to the customer to show his understanding. This would also be an opportune time for any follow-up questions that might have arisen.

Test Cases The test cases for 'create fair' and 'login' were judged to be completed by the customer without any misunderstandings. The test cases for the 'create profile' had some misunderstandings mainly based around the front-end. The test cases showed signs that a user for the website was created separately from the profile, but that was not the intend of the customer. Since it is only organizers who have a user on the website, there is no reason to split the organizer profile from the website user, and they are thus supposed to be combined.

4.3.2 Discussion

As mentioned earlier, the author found that the development practices, except for Test Driven Development, was not very useful or even applicable for a one man team. This points to a conflict in the assumption that the core should be applicable to any project and team. Thus a redefinition of the core might be needed, allowing elements of the core to be removed when certain properties are

present. In that case the items in the toolbox should be given an additional field which describes what elements in the core that should be removed. Allowing the core to be modified would open up for adding Code Review as a practice of the core. This was originally excluded since it heavily conflicts with the Tiny Team property. According to Wells [35], the practice of Pair Programming increase code quality immensely, but it is a hard and difficult skill to master. However, code review is a lot more simple to learn, and as mentioned earlier, pair programming can be seen as a form of continuously code review, so adding the practice of Code Review to the core should improve code quality and thus helps with the verification aspects of development. Another improvement to the core when it comes to the Tiny Team property is to take the Daily Scrum and make it relevant for 1 man teams. This is very much a corner case and thus will not be discussed, although it could be interesting to investigate further. Another option instead of allowing core elements to be removed would be to have multiple versions of the core, depending on team size. It would also solve the problem of having practices not suited for tiny team in the core.

The feedback from the questions brought forth some good points to discuss as well. Overall, the CRC session received positive feedback. It resulted in the customer trusting the developer, and was a handy tool during the interview to make the project more tangible and improved communication by providing common grounds between developer and customer. The CRC session was mostly focused on back-end, mainly as a result of time constraints in the thesis. It would be good to have some tools akin to the CRC session but for front-end uses, especially in a project like this. Something like wireframes, as described by Klein [23], could be useful since their role is also to make a front-end design more tangible and go through a rapid prototyping process, just as the CRC session does for back-end. The class diagram which was constructed based on the layout and interaction of the CRC cards also proved very useful in simplifying implementation, so adding this as a recommendation in the core also seems like a good choice. The interview process as a whole was also fairly well received. As the CRC session, the rest of the interview could be improved by adding more front-end focused tools and processes. The artefacts were well received as well, so the core should definitely define some guidelines on artefacts to help developer conduct the interview in a structured way, instead of the spontaneously usage that was the case in this interview. Finally, going through multiple loops of the interview process was suggested. This seems well aligned with Scrum, and could either be something done a few days after the initial interview, or something being done continuously, for example after each sprint.

The author had a brief email conversation with a consultant from the Software Improvement Group about their experiences when it comes to software quality from different methodologies and practices. The Software Improvement Group is an international company that specializes in analysing and improving existing

software solutions. Here is an excerpt of the correspondence, written by Mark Hissink Muller of SIG:

> *"You ask very interesting and good questions...! Please note that you could probably set up a research programme with 3-5 PhD students based on these. I believe you are shaping a MSc project?*
>
> *Good software is typically delivered by small teams that know what they are doing and have some Agile-like methodology, e.g. SCRUM and put all kinds of good engineering practices in place. Measuring (product) quality and acting iteratively on this during the process is more important than the methodology itself. Which criteria are you considering to compare and evaluate process/methodology success? We've seen many teams that work according to e.g. CMMi level 5, but still deliver poor product quality."*

This raises two very good points. The first is taking an iterative approach to both the product and the process itself. Evaluate your product and process continuously and from that make improvements as necessary. Scrum already has this in the form of the Retrospective and Review held after each sprint. However, there are no clear guidelines on how to act on these, or even how to evaluate both product and process in a structured manner. The second point is what criteria is used for evaluating a methodology, which ties into the first point. An example given in the excerpt is the Capability Maturity Model Integration, CMMI [24], which appraises the process itself. This a process oriented approach and, as stated in the excerpt, it does not guarantee that the product is not of low quality. Instead the aim here will be to evaluate the process based on it's ability to produce software:

- Of sufficient quality (sufficient in regards to it's application area. Quality as in maintainability, extendability, robustness, UX, performance, low amounts of bugs, depending on it's area of application)
- On time and budget
- That produces real value for the customer.

Having a module of the framework that deals with ways to evaluate a product according to these criteria would be very useful. As mentioned in the excerpt, it is important that this process happens continuously throughout the lifespan of the project, so improvements can be made before the product is released. It would also be helpful if this evaluation process could be used to compare

methodologies with each other, but the experience using any given methodology would also have to be taken into consideration. Note that according to the excerpt this is a large area of research, thus it is out of the scope of this project to truly go into depth with it.

4.4 External evaluation of framework

A project lead and manager of IT Minds has been asked to evaluate the methodology and provide some feedback, which is presented and analysed in this section. This was done to get some real life perspective on the developed methodology and see how it holds up to the requirements of real life development. Like the author suggested, the project lead suggested that some guidelines on artefacts were included. He also suggested that some guidelines were presented when it came to some of the more loosely defined practices like Coding Standards. Both how to define and manage these, and how to introduce them to developers.

New elements were also suggested for the toolbox based on experience of the project lead. The Indecisive Customer element was suggested, which would represent the case where the customer does not have a clear vision of his project, a common case in start-up companies. The solution would be to go through multiple loops of the interview process, preferably with different combinations of managers, leads, developers, and customer representatives if possible. The project lead also suggested refinements of the existing element for the High risk property, which in his view had a misleading name. In it's current state it refers to a team taking on a project which they are not experienced with or lacking competence within the given field or technology. But high risk could also refer to software security, like a payment system, or software robustness, like a public transportation system. To handle this issue, the High Risk property will be renamed to *Project Type Inexperience* and the two properties *Robustness Required* and *Security Required*, shown in Table 4.1 and Table 4.2 respectively, will be added to the toolbox.

Property name	*Robustness required*
Property description	A project might have certain robustness requirements such as high up-time and reliance, and an upper limit on response times.
Solution description	Apply thorough TDD and code review. Also use the principles of Spiral Invariant 1, to ensure that the requirements are realistic.
Solution synergies	Works well with Testing required since both solutions focuses on TDD and code review.
Solution opposition	The principles of Invariant 1 requires additional work, and thus it does not work well with the Small budget property.
Placement in the core	TDD and code review during development, and Invariant 1 during the interview and early stages of development.

Table 4.1: The toolbox element for the property 'Robustness required'

Property name	*Security required*
Property description	When the system are to handle sensitive information.
Solution description	Use proven security libraries instead of developing everything from the grounds up. Heavy use of code reviews and systems testing to find vulnerabilities. Find appropriate coding standards for high-security projects.
Solution synergies	Works well with Testing required since both aim to eliminate issues and as such there is some tools overlap.
Solution opposition	The Small budget property does not work well with a project doing extensive systems testing, since this is often a manual process and thus time consuming.
Placement in the core	During development.

Table 4.2: The toolbox element for the property 'Security required'

Overall, the project lead thought that the methodology looked promising, and he was especially interested in the modular and adaptable approach, which he thought was something that could be highly useful. He felt that a standardized yet adaptable approach to development is much needed. This is especially true for consulting and development companies, due to the large variance in project types that they undertake.

4.5 Conclusion

The result of the testing and feedback gathered is that the methodology is well received. However, more testing is definitely required before anything truly can be said. The lack of time the customer was available to the author resulted in a smaller than intended test.

Feedback from the customer and the project lead from IT Minds have resulted in quite a few improvements and refinements for the framework of the methodology. While not all of these have been implemented, sufficient information is discussed and presented such that further work could be done to implement these improvements. It was also confirmed that there is a need for an adaptable development process, and a more clear and structured way to gather information and acquire understanding of the project. This methodology aims to fulfil both of these needs.

The CRC session of the interview was key in ensuring the success of the interview and allowing the developer to obtain the necessary understanding of the system. This could possibly be considered as a stand-alone practice which could be applied regardless of overall methodology. Combining it with wireframes, or some other front-end related design tool, could be an interesting and useful extension to it.

A separate project could be started to create an evaluation module. This module would allow for the methodology, or any other methodology, to gauge it's capabilities to produce software of sufficient quality on time and budget. Acting on this information is key to developing and delivering software.

Chapter 5

Conclusion

In conclusion of this thesis, this chapter presents the limitations of the projects, what kind of future work could be done relating to this, and finally the main conclusion.

5.1 Limitations

A few areas which could be considered relevant for this thesis was outside it's scope. Firstly, It was not part of the developed methodology to teach it's users how to properly use Scrum. Having a proper understanding of Scrum and it's practices is very important to both this methodology and Scrum itself. It was however a topic which is sufficiently covered in other literature and this thesis instead focused on it's own contributions. Secondly, an evaluation module for the methodology used to evaluate the produced software was briefly discussed in section 4.3, but that is already a large area of research in and of itself, and was thus also outside the scope of this project. The third scope limitation was that the customer had very limited time to spare for the project, and thus the implementation and testing phase of this thesis was limited that way.
Having the developed methodology tested more thoroughly would have been very beneficial. Optimally, this would have to be done with a larger team working on a larger and more complex case, given a realistic amount of time

according to the project. This would much better simulate the methodology's usage in a real world scenario, and thus provide valuable feedback and possibly further improvements. As it stands, a lot of valuable feedback was already received, but having access to more is always preferably.
A limitation of the developed methodology that was found during testing was that parts of the core was not suited for one person teams. This led to the conclusion that the core should be modifiable by the toolbox, and not static as was first proposed.

5.2 Future work

The focus of future studies related to this thesis, or the general topic of an adaptable development framework, would have to include the proposed evaluation module, so that product quality can be continuously measured and actions can be taken during development to improve lacking areas. It should also include more focus on testing the ideas of the framework, using a larger sample size of different projects to provide more empirical data and improvements from the people of the industry. The suggestions and general views discussed in chapter 4 should be implemented and tested.

5.3 Conclusion

Decreasing cost overrun and increasing quality, while ensuring the software completes the job it is meant for and thus provides value to it's owners, are essential in IT projects. A new methodology has been designed during this project, combining many existing practices into a framework which can adapt to many project and team types. The existing practices have been studied in their native methodology, and how they contribute has been analysed. From this study, the methodology was designed with a core containing practices that were applicable in most circumstances. A toolbox was also developed for the framework, which contains the practices needed to specialize the core to different project and team types. All of these practices have been chosen for their contribution to validation, verification and deliver products on time and budget.

The framework was tested using a real life case provided by the development and consultancy firm IT Minds. Feedback was gathered from the owner of the case, and from a project lead from IT Minds. The feedback was mostly positive and contained valuable refinements to the developed methodology. These

results point towards the notion that an adaptable framework for software development could be both viable and wanted. The CRC interview session which was developed during this project could easily be adopted by other methodologies, and should prove a valuable tool for software companies using Scrum without a product owner from their customer.

I believe that these findings are very relevant for software development teams. A standardized approach to development provide teams with the tools and practices they need to succeed, but since each project is different this approach needs to be adaptable. This is accomplished in this project and while more thorough testing would have been nice, the feedback received so far has been good and the developed methodology looks promising. Further studies should focus on testing the toolbox using realistic cases, and develop a way to evaluate a product during it's development so that correcting action can be taken when necessary.

Many methodologies focus mostly on some aspect of development, such as Scrum mostly focusing on delivering on time and budget. The developed methodology builds upon Scrum, so it should still deliver products on time and budget. This methodology also attempts to deliver on product quality and creating customer value, which answers the questions from the introduction; "Are we building the product right?" and "Are we building the right product?".

Appendix A

Project plans

Week 1	Project planning. Literature study generally on Software Engineering and development.
Week 2	Literature study on selected methodologies.
Week 3	Literature study on selected methodologies, practices and tool.
Week 4	Analysis: Intro, review section, and pros / cons.
Week 5	Analysis: Focus areas and common grounds.
Week 6	Analysis: Tools.
Week 7	General refinement of analysis chapter.
Week 8	Design: Work on general idea of framework.
Week 9	Design: Work on general idea of framework. Start work on framework.
Week 10	Design: Work on framework.
Week 11	General refinement of design chapter.
Week 12	Meeting with customer.
Week 13	Work on case. Document work as an ongoing process.
Week 14	Work on case. Document work as an ongoing process.
Week 15	Work on case. Document work as an ongoing process.
Week 16	Evaluate work.
Week 17	General refinement of implementation chapter.
Week 18	Discussion section.
Week 19	Discussion section.
Week 20	Introduction and conclusions.
Week 21	Read through thesis and refine.
Week 22	Read through thesis and refine. Hand in.

Table A.1: How the project was planned to proceed, week by week.

Week 1	Project planning. Literature study generally on Software Engineering and development.
Week 2	Literature study on selected methodologies.
Week 3	Literature study on selected methodologies, practices and tool.
Week 4	Literature study & Analysis: Introductory sections.
Week 5	Literature study & Analysis: Literature and review study section.
Week 6	Literature study & Analysis: Literature and review study section.
Week 7	Analysis: Strengths and challenges section.
Week 8	Analysis: Properties and tools section.
Week 9	Design: Work on general idea of framework. Start work on core section.
Week 10	Design: Strengthening the core, toolbox and transitioning sections.
Week 11	General refinement of design and analysis chapter.
Week 12	Awaiting customer. Introduction chapter.
Week 13	Awaiting customer.
Week 14	Awaiting customer. Customer interview. Create artefacts of interview.
Week 15	Work on case.
Week 16	Evaluate the process with IT Minds. Start work on implementation chapter.
Week 17	Case analysis, result and test sections.
Week 18	Interview with customer and feedback.
Week 19	Discussion sections and conclusion chapter. Read through thesis and refine.
Week 20	Read through thesis and refine.
Week 21	Final refinements of introduction and conclusion. Hand in.
Week 22	

Table A.2: What was actually done in each week.

Appendix B

Items created during interview

Use case

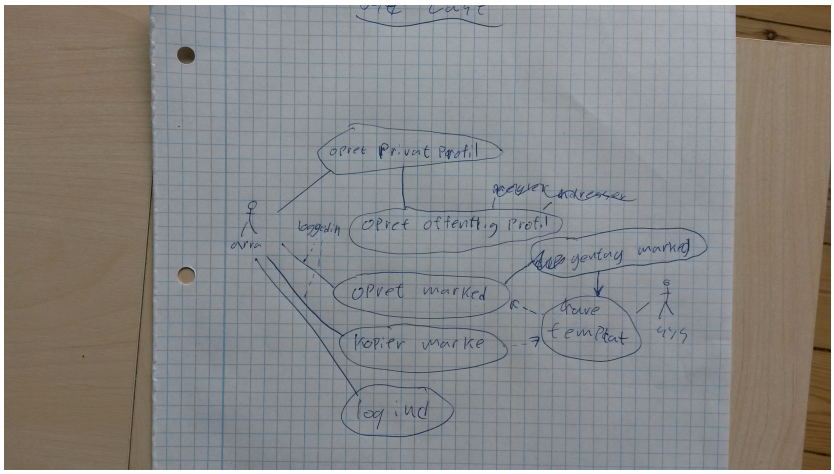

Figure B.1: The original use case made during the interview

Acceptance tests and notes

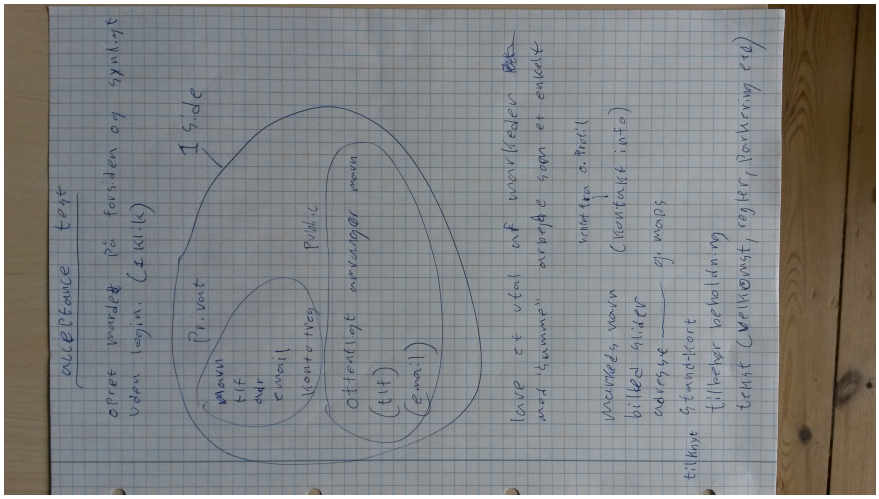

Figure B.2: The original acceptance test and additional notes made during the interview

Appendix C

Code examples

Some test cases

```
[Test]
public void CreateMarket_NotLoggedIn()
{
        Assert.IsTrue(site.MarketList.Count == 0);
        Assert.IsTrue(profile.MarketList.Count == 0);

        site.LogOut(username);
        site.CreateMarket(username, marketName, date1,
            locationData, stallMap, text, slideShow, new
            List<Stall>(), new List<Accessories>());

        Assert.IsTrue(site.MarketList.Count == 0);
        Assert.IsTrue(profile.MarketList.Count == 0);
}

[Test]
public void CreateMarket_FromTemplate()
{
        int stallnumber = 42, price = 1750;
```

```
        site.CreateMarket(username, marketName, date1,
            locationData, stallMap, text, slideShow, new
            List<Stall>() { new Stall(stallnumber, price) },
            new List<Accessories>());
        MarketTemplate templateOfAnotherMarket =
            site.UserToMarketTemplate[username][0];

        site.CreateMarket(username, date2,
            templateOfAnotherMarket);

        Assert.IsTrue(site.MarketList.Count == 2);
        Assert.IsTrue(profile.MarketList.Count == 2);
        Assert.IsTrue(site.UserToMarketTemplate[username].Count
            == 1);

        Assert.IsTrue(profile.MarketList[0].OrganizerPublicName
            == profile.MarketList[1].OrganizerPublicName);
        Assert.IsTrue(profile.MarketList[0].OrganizerPublicEmail
            == profile.MarketList[1].OrganizerPublicEmail);
        Assert.IsTrue(profile.MarketList[0].OrganizerPublicPhone
            == profile.MarketList[1].OrganizerPublicPhone);
        Assert.IsTrue(profile.MarketList[0].TemplateSource ==
            profile.MarketList[1].TemplateSource);
        Assert.IsTrue(profile.MarketList[0].Date == date1);
        Assert.IsTrue(profile.MarketList[1].Date == date2);
        Assert.IsTrue(date1 != date2); //tests for a mistake
            in the test case
}
```

The corresponding developed code

```
internal void CreateMarket(string username, string date,
    MarketTemplate marketTemplate)
{
        if (!IsLoggedIn(username))
        return;
        User user = null;

        foreach (User u in UserList)
        {
                if (u.Username == username)
                user = u;
```

```
        }
        if (user == null)
        return;

        Market m = new Market(user, date, marketTemplate);
        MarketList.Add(m);
        user.Profile.MarketList.Add(m);
}
```

Bibliography

[1] Scott Ambler. Agile software development. `http://www.agilemodeling.com/`. 2014, Accessed: 11-03-2015.

[2] Scott Ambler. Answering the "where is the proof that agile methods work?" question. `http://agilemodeling.com/essays/proof.htm`. 2014, Accessed: 11-03-2015.

[3] Jordan B. Barlow, Mark Jeffrey Keith, and David W. Wilson. Overview and guidance on agile development in large organizations. *Communications of the Association for Information Systems*, 29(29):25, 2011.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2005.

[5] Kent Beck, Martin Fowler, Ward Cunningham, and Alistair Cockburn et al. The agile manifesto and it's twelve principles. `http://agilemanifesto.org/`. 2001, Accessed: 13-03-2015.

[6] B. W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):22–42, 1986.

[7] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer Society*, 21(5):61–72, 1988.

[8] B. W. Boehm. Spiral development: Experience, principles, and refinements. *Carnegie Mellon University, Software Engineering Institute*, 2000.

[9] B.W. Boehm. Verifying and validating software requirements and design specifications. *Software, IEEE*, 1(1):75–88, Jan 1984.

[10] Robert N. Charette. Why software fails. http://spectrum.ieee.org/computing/software/why-software-fails. 2005, Accessed: 11-03-2015.

[11] Mike Cohen. Advice on conducting the scrum of scrums meeting. https://www.scrumalliance.org/community/articles/2007/may/advice-on-conducting-the-scrum-of-scrums-meeting. 2007, Accessed: 17-04-2015.

[12] Coley Consulting. From waterfall to v-model. http://www.coleyconsulting.co.uk/from-waterfall-to-v-model.htm. 2015, Accessed: 18-03-2015.

[13] Andreas Deuter. Slicing the v-model - reduced effort, higher flexibility. *2013 IEEE 8th International Conference On Global Software Engineering (ICGSE 2013)*, pages 1–10, 2013.

[14] Amr Elssamadisy. *Agile Adoption Patterns: A Roadmap to Organizational Success*. Addison-Wesley Professional, 1 edition, 2008.

[15] J. Estublier and S. Garcia. Concurrent engineering support in software engineering. *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 209–220, Sept 2006.

[16] Martin Fowler. Is design dead? http://martinfowler.com/articles/designDead.html. 2004, Accessed: 16-03-2015.

[17] Martin Fowler. Using an agile software process with offshore development. http://www.martinfowler.com/articles/agileOffshore.html. 2006, Accessed: 29-03-2015.

[18] Robert Frese and Vicki Sauter. Improving your odds for software project success. *IEEE Engineering Management Review*, 42(4):125–131, 2014.

[19] D. Hillson. The cost of managing risk. *Project Manager Today*, 19(5):30, 2007.

[20] Kenji Hiranabe. Kanban applied to software development. http://www.infoq.com/articles/hiranabe-lean-agile-kanban. 2008, Accessed: 11-03-2015.

[21] Jennifer Dorette Jacob. Comparing agile xp and waterfall software development processes in two start-up companies, chalmers university of technology, 2011.

[22] Gottfried Kellner. Software engineering. *11th CERN School of Computing*, 1989.

[23] Laura Klein. *UX for Lean Startups*. O'Reilly Media inc, 2013.

[24] CMMI Institute of Carnegie Mellon University. Cmmi institute. http://cmmiinstitute.com/. 2015, Accessed: 14-06-2015.

[25] US Department of Justice. System development life cycle guidance document. http://www.justice.gov/archive/jmd/irm/lifecycle/ch1.htm. 2003, Accessed: 09-03-2015.

[26] Alexander Osterwalder and Yves Pigneur. *Business Model Generation*. Addison-Wesley, 2010.

[27] Maria Paasivaara, Sandra Durasiewicz, and Casper Lassenius. Using scrum in a globally distributed project: A case study. *Software Process Improvement and Practice, Softw. Process Improv. Pract*, 13(6):527–544, 2008.

[28] Kai Petersen, Claes Wohlin, Dejan Baca, Kai Petersen, and Dejan Baca. The waterfall model in large-scale development. *Lecture Notes in Business Information Processing*, 32 LNBIP:386–400, 2009.

[29] Ryan Polk. Agile and kanban in coordination. *Proceedings - 2011 Agile Conference, Agile 2011, Proc. - Agile Conf., Agile*, pages 263–268, 2011.

[30] L. Rising and NS Janoff. The scrum software development process for small teams. *IEEE SOFTWARE*, 17(4):26–+, 2000.

[31] Ken Rubin, Mike Cohn, Pete Deemer, Steve Denning, and Michele Sliger. Using scrum. https://www.scrumalliance.org/. 2015, Accessed: 11-03-2015.

[32] Michael Sahota. *An Agile Adoption and Transformation Survival Guide*. InfoQ, 2012.

[33] VersionOne. State of agile. http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf. 2013, Accessed: 21-03-2015.

[34] Don Wells. Agile software development. http://www.agile-process.org/. 2009, Accessed: 11-03-2015.

[35] Don Wells. Extreme programming. http://www.extremeprogramming.org/. 2013, Accessed: 11-03-2015.