Remodelling the Software Development Life-Cycle using combinations from Scrum and Extreme Programming

A case study about using components from two agile frameworks to shape development processes

Author:
Marcus Hilliges
marcus@hilliges.com

Advisor:
Ulf Asklund
Department of Computer Science
ulf.asklund@cs.lth.se

Examiner:
Lars Bendix
Department of Computer Science
bendix@cs.lth.se

ABSTRACT

The purpose of the study is to have increased clarity from a real-world case in what the possible benefits and risks are when combining two popular agile development frameworks. Many studies have been focused towards evaluating agile methodologies individually, but few have looked at the benefits and risks of combining them.

The paper starts by seeking an understanding as to why a combination would be more beneficial from a theoretical standpoint by referencing the software development life-cycle model (SDLC). Two combinations built on elements from Scrum and Extreme Programming are defined. The components are chosen in relation to how they complement each other in the SDLC. A baseline combination and evaluation framework are set up in which the new combinations will be measured against. In the evaluation framework, the author describes how the combinations could be assessed from both a functional and performance perspective.

The data collection is performed on a medium-sized SME within a couple of teams working in the same project. Scrum Masters from each development team where a new combination are being assessed are interviewed and data around the team's performance is gathered. The author tries to prove whether the agile combinations performs better than a limited stand-alone implementation and to identify key components that make a combination successful.

The result was that the development teams that participated in the study were strongly positive to the new processes. The teams thought that some of the processes gave them a new approach to problem-solving and despite having a learning curve, the teams were able to complete their work in time. Statistics showed that the time spent resolving defects versus time spent on development were close to half after the introduction of Scrum and Extreme Programming combinations.

PREFACE

I would like express my sincerest gratitude to the possibility to carry out this thesis with the case company, psHealth. Special thanks to Krassimir Dimitrov, Head of Engineering and my colleagues at psHealth who were always available and helpful throughout the entire project.

I would also like to thank my supervisors Ulf Asklund and Lars Bendix for their great input and support. Lastly, I would also like to thank Anton Fredriksson and Arpit Bohra for proof-reading and coming with valuable input to the study.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 BACKGROUND OF THE STUDY

1.1.1 Theoretical Background

In 2001, representatives from Extreme Programming, Scrum, DSDM, Feature-Driven Development and other well-known agile development frameworks signed what is known as the "Agile Manifesto". This document was an agreement of a number of shared principles among the agile community. It became a milestone for agile development and showed that even though many agile frameworks differed in approach, they had shared the same vision and goals.

Agile development methodologies have ever since they were introduced fundamentally changed how software developers and teams work. Increased complexity of requirements meant that software companies had to move away from linear development strategies to be more modular and responsive to change. The introduction of agile frameworks allowed developers to gain the flexibility and tools they needed in order to deliver within tight timeframes and meet customer satisfaction.

Two well-known agile development frameworks are Scrum and Extreme Programming (also known as XP). Many studies have been focused towards evaluating these agile methodologies individually, but few have looked at the benefits and risks of combining them.

1.1.2 Background of psHealth

psHealth Limited (psHealth) is a healthcare technology company founded in London, 2009. psHealth core business is to offer automated referral management and care coordination software solutions. Their main product is tailored case management solutions in which healthcare providers can triage and coordinate services to patients. In the recent years, psHealth have expanded their product range to offer e.g. eRS (Electronical Referral Services for NHS) integration and triaging based on machine learning.

psHealth consists of approximately 40-50 employees with offices both within and outside of the UK. Among psHealth's client list are large companies such as BUPA, Remploy, OH Assist and G4S. Many of these whom are among the biggest healthcare providers in the UK.

For the time being (2017), psHealth only offers their products to clients within the UK.

1.2 CHALLENGE

In the recent years, psHealth have had an exponential growth from what used to be a handful of engineers to full-scale development teams spread across a number of projects. During this period of rapid growth, the engineering teams have had to adjust in their approach to development processes multiple times.

In order to evolve and scale as a team of developers, it is important to decide on common engineering processes. This is a natural crossroad for many IT-companies when their development teams grows. It is also common that an IT-company want to try out different processes before investing fully in one.

Up until this report was written, the engineering team at psHealth had not fully taken a decision on how these set of processes should look like. Their current process were a combination of linear and agile development with a limited Scrum implementation. The engineering team were unanimous on

that their next development processes need to step in an agile direction. The agile direction is an important step in become more competitive and flexible to sudden requirement changes. While contemplating in which agile methodologies to adopt, the author of this paper saw a research opportunity. Could a combination of agile development frameworks be more beneficial than an individual implementation? If so, are there key components that make them more successful together than others?

1.3 Purpose

The purpose of this paper is to gain increased clarity from a real-world case in what the possible benefits and risks are when combining multiple agile development frameworks. Based on the challenge described above, this will be achieved by looking into two research questions:

1. Investigate if a combination of the agile development frameworks Scrum and XP could be more beneficial than an limited implementation of Scrum
2. Try to identify which key components that make a combination more favourable than another

1.4 Delimitations

Since the project was performed by a single student, limitations in terms of time constraints and research depth applied. There was also a constraint on the availability of the developers since the data collection was done during live projects within the company.

During the period in which the study ran, the data collection came from four sprints of approximately two weeks each. This was measured against baseline data made up of four sprints with the same length. Since the study had to fit within the projects that the company was currently working on, the level of complexity for the development done within the sprints varied.

There were also limitations in terms of how much data that could be presented. Because the company that was studied is in health-care technology, any further details than what is presented in this report are classified.

Even though key persons were interviewed during the process, the results should not be taken as conclusive evidence whether combining agile development frameworks is beneficial or not. It should rather serve as a guideline and example of how agile development frameworks can be combined and how these combinations can be evaluated.

1.5 Outline of the Report

Introduction

The initial chapter starts by describing the background of the project and the company which the study was made on. Subsequently, the problem and purpose is outlined together with research questions and delimitations of the project.

Theory

This chapter explains the current development process and gives a brief description of the methodologies that will be used. The author also motivates the structure of the combinations and relates this to the software development life-cycle.

Method
This chapter explains the methodology used for the data collection and evaluation strategy. The evaluation strategy is defined within a framework and further described for each combination.

Results
This chapter presents the results from sprints using the SXP combinations in context with the baseline. The results are divided into a quantitative and a qualitative result section.

Discussion
This chapter analyses the results of the SXP combinations and each agile component individually. It ends with a conclusion of which factors are important when modelling SXP combinations and which steps that could lead to a successful SXP combination.

2 THEORY

2.1 TRADITIONAL PROCESS MODELS

Process models are not software development specific and exist in a variety of other industries. A process model simply describes processes of the same nature which are then classified together as a model. As a result, software companies initially inherited process models which originally came from the hardware industry, such as the Waterfall model.

2.1.1 Waterfall

Waterfall within software development is a non-iterative design process in which each stage of the software development life-cycle is done only once and without the possibility to reverse, similar to a waterfall. The waterfall model was initially and formally introduced by Winston W. Royce in his paper "Managing the development of large software systems" where each step of the software development process is done sequentially. (Royce, Dr. Winston W. 1970)

The waterfall model descends from the manufacturing and assembly industries where cost grew exponentially as a product was processed. This is an approach in which software development came to adapt from a highly-physical manufacturing model. (Benington, Herbert D. 1983)

2.1.2 Spiral

The spiral model is risk-driven process model generator first presented by Barry Boehm in his paper "A spiral model of software development and enhancement" (Boehm, Barry W. 1988). The spiral development model focuses on risk patterns, which can lead a team to pick components from other software development approaches like incremental, waterfall and evolutionary prototyping. (Boehm, Barry W. 2000). Because of this trait, Boehm started calling the spiral model a process model generator because it generated process models depending on risk patterns of the project.

The spiral model uses four phases: Planning, Risk Analysis, Engineering and Evaluation. The idea is that a software project moves iteratively through these four phases, which in this model is known as "spirals". In the initial planning phase, the requirements are gathered and builds on with the risk analysis where alternative solutions and risks are identified. By the end of each risk analysis phase, a prototype of the product is produced. During the engineering phase the software is produced and the product undergoes testing. In the last phase, the customer get to evaluate the product and determine the goals for the next spiral. (Munassar, Nabil Mohammed Ali. Govardhan, A. 2010)

Boehm's goal with the spiral model was to come up with an incremental approach to grow an implementation while concurrently decreasing the risk. Risks would be defined as potential events that could cause a project to fail. The spiral model attempts to take this into account by introducing *anchor point milestones* to ensure the stakeholders are going towards appropriate system solutions.

Because of a number of reasons, the spiral model is not universally understood. Boehm identifies a few significant misconceptions:

- The spiral represents a sequence of waterfall steps
- The spiral only goes through one single iteration
- There is no backtracking to revisit previous decisions

(Boehm, Barry W. 2000).

2.1.3 Big Bang

The big bang model is a software development approach where all project resources are put into development. Named after the cosmological model, the idea is that with a lot of resources being put into expansion, a finished product will emerge faster.

The model is unique in the way that it requires little to no planning, organization, leadership, best practices or procedures (Powell-Morse, Andrew. 2017). The requirements are loosely recognised and accepted by the team and customer. All effort is spent on developing and getting the product ready as fast as possible. Because of the great risks this approach introduces, this model is best suited for small and short-term projects with very few people working on them.

Even in small projects, the risks with the big bang model are significant. If the requirements are somehow misunderstood or the project needs to go on for a longer period of time, this approach can be very costly. Advantages with this model is for example that it gives all the flexibility to the developers and is a good learning tool for people new to software development (Powell-Morse, Andrew. 2017)

2.2 Agile Development Processes

2.2.1 Scrum

Scrum is an agile development framework for software product management. In 1993, Jeff Sutherland, John Scumniotales and Jeff McKenna hold what they refer to as “the first Scrum” which they described in their paper “Agile Development: Lessons learned from the first Scrum” (Sutherland, Dr. Jeff. 2004). Sutherland would later introduce the paper “Scrum Methodology” in 1995 together with Ken Schwaber which became the beginning of Scrum methodology as we know it.

Scrum serves mainly as a strategic development framework based on short, incremental iteration loops and emphasizes frequent customer engagement. With a potential gap between a customer’s initial product idea and final expectation, Scrum tries to tackle requirement volatility by maximizing product delivery and team efficiency.

Four formal events are well-known within Scrum (Sutherland, Jeff. Schwaber, Ken. 2016):

- Sprint Planning
- Daily Scrum
- Sprint Review
- Sprint Retrospective

These events are conducted by the *Scrum Team*, which consists of a *Product Owner*, the *Development Team* and a *Scrum Master*. Together they use these events and artifacts like the product and sprint backlog to manage software product development.

2.2.2 Extreme programming

Extreme Programming is an agile methodology framework for software product development. XP was created by Kent Beck and first extensively explained in his book “Extreme Programming Explained”, which he released in 1999 (Beck, Kent. 1999).

Similar to Scrum, XP also tries to meet customer satisfaction through short, incremental development cycles. However, XP is more focused towards methodology and further describes

concrete methods and practices within the developer teams. Common XP practices are (Wells, Don. 1999):

- Continuous Integration
- Pair Programming
- Collective Code Ownership
- Automated Unit Testing
- Code Re-factoring

Using these practices among many more, XP tries to encourage its four values; communication, simplicity, feedback and courage (Beck, Kent. Andres, Cynthia. 2004).

2.3 Current Development Process

The current development process within the development teams at psHealth have been a mix of Waterfall and components taken from Scrum. The agile components that have been used from Scrum are *Daily Scrum* and *Sprint Planning*.

Daily Scrum is a stand-up activity which the scrum master and the development team uses to track progress. Sprint Planning is a session where the scrum master, the product owner and the development team decides what work goes into the upcoming sprint. Both of these components are further elaborated later in this report.

Looking at the four phases of the SDLC, the current development process have been to start off the first sprint as soon as the initial requirements analysis has been completed. Then while the sprint progresses, the business analyst team are working through the second iteration of the requirements analysis. As soon as the first sprint finishes, the second sprint starts based on the second iteration of the requirements analysis. These iterations continue on until the customer and development team have reached an agreement of what is going to be the complete requirement specification. Depending on how long and extensive the sprints are, the quality assurance team might perform the testing after either the first or second sprint.

One risk with this approach is that the requirements might change significantly while the business analysts are in the middle of their requirement analysis. In these cases, the development teams have continued on and finished their sprint before looking at the changed requirements.

2.4 Future Development Process

When one of the founders of Scrum analysed the use of the waterfall model within software development, he found a number of common scenarios were this approach was unsuccessful. A few of these were:

- Requirements would change throughout the development process
- Requirements weren't fully understood in the beginning of the project
- The users wouldn't know exactly what they wanted until they could see an initial version of the software

(Sutherland, Dr. Jeff. 2004)

Using the waterfall model meant that changes could become very costly and revenue forecast became harder to predict. As a result, 31 % of software projects, many which are driven by some kind of waterfall methodology is terminated before completion (Sutherland, Jeff. 2001).

From the waterfall model, some companies adapted the spiral model. The idea with the spiral model was that the high amount of risk analysis and incremental development would cover help to cover for unforeseen events, such as requirements changes in mid-development. However with the high amount of time spent on risk analysis and prototyping that came with the spiral model, it could be an expensive model to use. The model success was also dependent on the risk analysis phase being accurate. (Munassar, Nabil Mohammed Ali. Govardhan, A. 2010)

Another approach that was reviewed were the big bang model. As described earlier in 5.1.3, this methodology meant putting the majority of the resources into development, many times executed by a single individual. One example of this was when Matisse object database was written to drive the US $10 billion nuclear reprocessing plants around the world. A single individual spent two years writing just about 50,000 lines of code to drive what would become one of the fastest and most reliable databases ever benchmarked for nuclear plants. Even in this case where the big bang model would be successful from a technical stand-point, the issue arose quickly when only a single individual would know how database responsible for nuclear plants worked. It took the nuclear engineers years before they would understand the model that the Matisse Object was built on. This approach did not scale well for larger projects. (Sutherland, Dr. Jeff. 2004)

By analysing these three models, constraints can be derived such as minimal flexibility, high costs, speed-to-market, high risk and low-level dependencies. These constraints would have the greatest impact on smaller projects and start-ups with limited resources. Many models were developed around the same time but often shared similar constraints, which meant that many software companies were in need of new innovative methodologies.

When the agile frameworks first started appearing, they seemed to be able to solve these problems by controlling scope and keeping close customer contact. By working incrementally and iteratively they could reduce risk and be prepared for any sudden change in requirements. After the launch of Scrum in 1993 and XP in 1999 came the introduction of the agile manifesto in 2001. The agile manifesto proposed a common of set of principles and values for agile methodologies to work within. The four core values are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

(Fernandes, Joao M. Almeida, Mauro. 2010)

Even though Scrum was considered more of an agile management framework and XP an agile engineering methodology, the founders had officially agreed that they shared the same core principles and values. From this came the idea that a hybrid approach using components from both frameworks might be better than a stand-alone implementation.

One of the main reasons why a hybrid approach using Scrum and XP (SXP) could be beneficial is because they fundamentally complement each other in relation to the SDLC. In the paper “Classification and Comparison of Agile Methods”, both Scrum and XP are evaluated towards how well their components cover common software development attributes within the SDLC. The attributes used in the evaluation were: software requirements, construction of software, software testing, software engineering management and agile principles – proposed practices relation. The result of the evaluation is presented in the table below.

Table 1: Describing how well the agile methodologies cover the software attributes on a scale using AS (Adequately Satisfies), PS (Partially Satisfies) and NS (Not Satisfy). Each attribute consists of a multiple sub-attributes in the original paper

Attribute	XP			Scrum		
	AS	PS	NS	AS	PS	NS
Software Requirements	60 %	40 %	0 %	60 %	20 %	20 %
Construction of Software	75 %	25 %	0 %	0 %	0 %	100 %
Software Testing	100 %	0 %	0 %	0 %	0 %	100 %
Software Engineering Management	80 %	0 %	20 %	80 %	20 %	0 %
Agile Principles – Proposed Practices Relation	75 %	8 %	17 %	50 %	17 %	33 %

(Fernandes, Joao M. Almeida, Mauro. 2010)

By analysing the table above and how components from Scrum and XP relates to the SDLC, the hypothesis is that SXP combinations can be made to complement each other. If the components picked overlap and have poor coverage of the full SDLC, the benefits will decrease. Analysis of how each component relates to the SDLC is needed in order to take full advantage of using both agile frameworks.

2.4.1 SXP in context with SDLC

The SDLC gives an overview of the workflow for software development and can be used to characterise both waterfall and agile methodologies. The SDLC used in this report is a simplified version, since the SDLC can vary between development methodologies. The SDLC for a waterfall model might start off with requirement analysis followed by design, while test-driven development (TDD) starts with requirement analysis followed by writing test scripts.

The SDLC in this report describes a linear flow starting with requirement analysis, followed by design, implementation and then testing. This version of the SDLC can be used to describe both waterfall and agile process models, where the main difference is that the SDLC is iterated multiple times in a project for an agile methodology but only iterated once for waterfall.

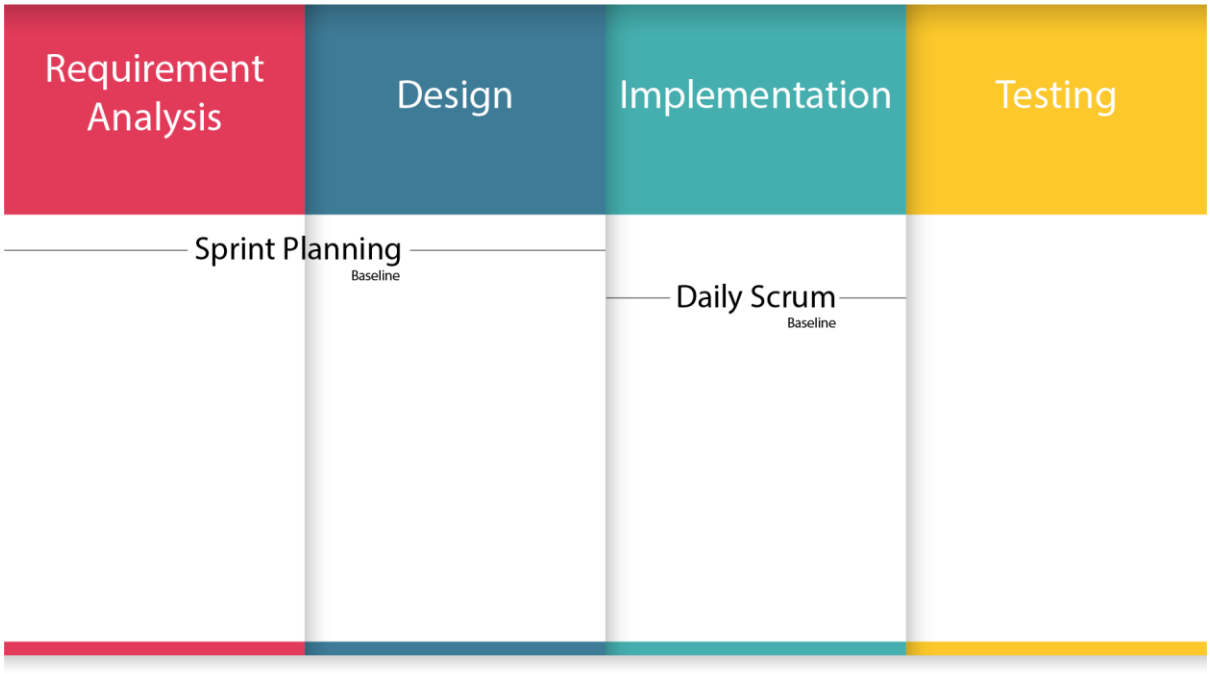

Figure 1: The development teams' coverage of agile methodologies in relation to the SDLC before the SXP combinations were introduced

The current coverage of agile methodologies in the company in relation to the software development life-cycle is limited. The idea with the suggested SXP combinations in this report is that together they will be able to cover more of the SDLC with agile elements and thus have a greater impact.

With the new combinations, an increased area of the SDLC will be covered:

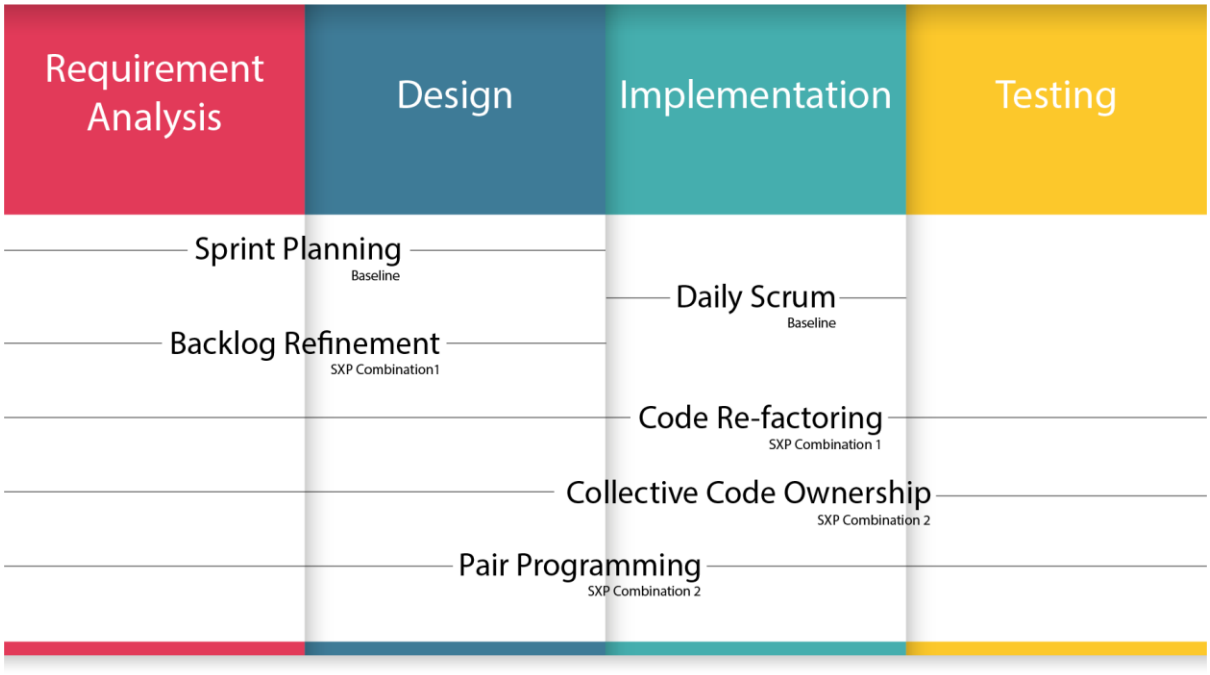

Figure 2: The development teams' coverage of agile methodologies in relation to the SDLC after the SXP combinations were introduced

One might question how components such as code refactoring, collective code ownership and pair programming could stretch over the whole SDLC and more specifically, how they could also cover the testing phase. Taking code refactoring as an example, the reason doing code refactoring could be more than one. If a new feature is to be covered, the system might need to be refactored in order to support a design which could cover the new feature. However, code refactoring could also be done as part of a code review. If feedback from a code review indicates that the system is overly complex or engineered, it might need code factoring in order to simplify the design without changing its functionality.

2.5 Similar Studies

In 2008, a study was conducted to see how useful companies thought components from the agile frameworks Scrum and XP were. This was based on how useful the companies considered each component to be and how well they were used within the companies' software projects. The study took 11 XP practices and 6 Scrum practices and surveyed 35 software projects in 13 software organisations from 8 different countries.

The study also evaluated how useful the surveyed software projects thought each individual component were. Comparing this with components picked for the SXP components, the results showed that the most appreciated XP practice were collective code ownership and the most appreciated Scrum practice were backlog refinement. The least appreciated XP component were pair programming, whilst the least appreciated Scrum component were sprint planning. (Abrahamsson, P. Salo, O. 2008)

The outcome of the study was that nearly 90 % of responses where XP practices had been applied were positive and only 5.8 % could be considered negative. Similarly, 77 % of the responses where Scrum had been applied were positive and 11 % could be considered negative. (Abrahamsson, P. Salo, O. 2008)

The structure for the SXP combinations were reviewed against the study which is described above and while it was tempting for this study to pick the components which had the highest appreciation rate, one had to consider other factors as well. These factors could be for example the components relation to the SDLC or how suitable the components are for the project both from a cost and company preference perspective. All of these factors were taken into account when constructing the SXP combinations.

2.6 Structure of SXP Combinations

The components that are used to form the baseline and SXP combinations from the two frameworks are the following:

From Extreme Programming:

- **Pair Programming** – Two developers working together on a single computer produces code more efficiently and with higher quality without taking twice as long than if they were to work individually. One person *the driver* writes code while the other *the observer/navigator* reviews each line of code and inputs feedback simultaneously. After a set amount of time, the developers swap roles.

- **Collective Code Ownership** – All team members are responsible for the full codebase. Any developer on the project should be able to contribute, change, improve or refactor a line of

code anywhere within the system. No individual developer should become the bottleneck for new changes.

- **Code Re-factoring** – Refactoring in software development means that redundant or overly complex design decisions is improved and made simpler without changing the systems external behaviour. This is usually done when by external developer who was not involved in the sprint.

(Beck, Kent. 2004)

From Scrum:

- **Daily Scrum** - For each day of the sprint, the team will hold a daily stand-up to discuss progress and potential issues. Everyone from the development team including the Scrum Master should attend the daily scrum every day to accurately measure progress and adjust scope if necessary. The Product Owner may attend the Scrums if deemed necessary.

- **Sprint Planning** - A planning session arranged with the Scrum Master, the Product Owner and the Development Team. The Product Owner describes which features of the product are of the highest priority and a mutual decision is made to decide what should go into the sprint.

- **Backlog Refinement** - Throughout the project there will be a backlog of product items consisting of e.g. ad-hoc requirements and defects that for some reason were not part of any sprint. Before each sprint, this backlog will be reviewed by the team who will decide if any of the items should be included in the upcoming sprints.

(Sutherland, Jeff. Schwaber, Ken. 2016)

The elements taken from Scrum focuses more on product and team management, whilst the XP ones drives development methodology and values. To get an appropriate measure of how these processes will impact the development teams, they will be compared in the context of the software development life-cycle. Figure 3 shows a visual example of how a basic flow of the SDLC looks like.

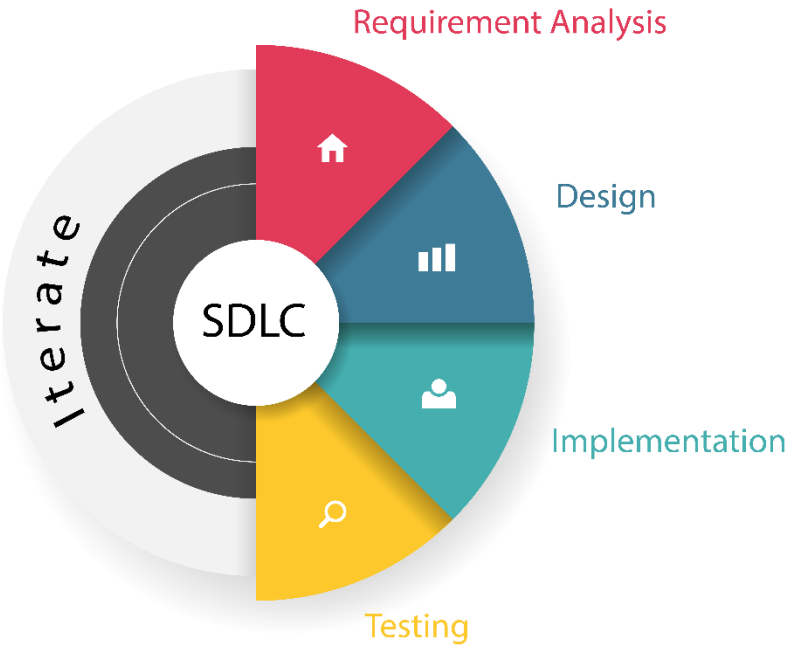

Figure 3: Basic flowchart of the Software Development Life-Cycle

3 METHOD

3.1 RESEARCH STRATEGY

The research strategy throughout the project has been a quantitative and qualitative data collection together with an abductive reasoning approach.

Abductive reasoning is a reasoning approach which starts with an observation and seeks to find the most simple and likely explanation. Unlike inductive reasoning, the abductive process starts off with a theoretical framework or by discarding a theory (Kovács, G. Spens, KM. 2005). Using a theoretical framework and real-life observations, the goal of abductive research is to understand a new phenomenon. If a pre-existing theoretical framework is unable to explain the observation, an iterative process of "theory matching" or "systematic combining" starts (Kovács, G. Spens, KM. 2005). This iterative process helps reaching an outcome which might lead to new theory developments or conclusions.

3.2 DATA COLLECTION & RESOURCES

The data collection is divided into three areas: JIRA, interviews and observation.

3.2.1 JIRA

JIRA is an advanced commercial product for project management and monitoring developed by Atlassian. JIRA provides an online platform for issue tracking and sprint planning which could be used for agile software development. According to Atlassian, JIRA has over 60,000 customers world-wide (Atlassian. 2017).

Because of its issue-tracking functionality, reports could be run on outputs such as how much time was spent on design, development and defect-resolving in relation to user estimate.

JIRA was the main source of quantitative data acquisition and was used on all sprints throughout this study.

3.2.2 Interviews

The interviews were conducted in a semi-structured fashion which allowed the interviewer to ask both open- and closed-ended questions. The interviews were targeted to the Scrum Masters of the development teams, since they were thought to be able describe the experience of the full team. The questions were tailored to what SXP combination that were currently being evaluated, and the answers were to be given in comparison to the baseline configuration.

There is a risk that interviewing only one member of a group will not give an accurate view of the full group's opinion, but the author decided that targeted interviews were the preferable choice due to the time constraints.

3.2.3 Observation

Similarly to interviews, the observation was also handled in a semi-structured way. For every day of development and during the time the SXP combinations were tested, the author held a *Scrum of Scrums*. A Scrum of Scrums is a technique to scale Scrums by having one team member from each agile team who is practicing Scrum to join a Scrum meeting with the purpose of summarizing the

work of all teams (Sutherland, Jeff. 2001). The observation is later reflected in the discussion part of this report.

One of the risks with observation is that the activity itself might alter the behavior of the participants being observed. This might affect the quality of any data being collected during the period of observation. However, taken into account that the author was hosting Scrum of Scrums prior to the introduction of the SXP combinations, hence why the risk of the observation affecting the data collection is considered low.

3.3 Evaluation Framework

In order to be able to appropriately assess and rate the different SXP combinations, the evaluation process will need to cover the challenge in both a quantitative and a qualitative way.

The first step in an evaluation process will be to define a baseline. The initial baseline will be based on Scrum and contain the two components sprint planning and daily scrum. This baseline will be used to track the progress of the different SXP implementations against the challenge as well as the research questions.

Table 2: The table below describes the SXP that will form the baseline for the experiments

SXP Baseline Configuration		
Agile Framework	Component	Implementation Level
Scrum	Sprint Planning	Implementation Attempted
Scrum	Daily Scrum	Implementation Attempted
Scrum	Backlog Refinement	No Implementation
Extreme Programming	Code Re-factoring	No Implementation
Extreme Programming	Collective Code Ownership	No Implementation
Extreme Programming	Pair Programming	No Implementation

Table 3: The table below describes the structure of SXP combination 1

SXP Combination 1		
Agile Framework	Component	Implementation Level
Scrum	Sprint Planning	Implementation Attempted
Scrum	Daily Scrum	Implementation Attempted
Scrum	Backlog Refinement	Implementation Attempted
Extreme Programming	Code Re-factoring	Implementation Attempted
Extreme Programming	Collective Code Ownership	No Implementation
Extreme Programming	Pair Programming	No Implementation

Table 4: The table below describes the structure of SXP combination 2

SXP Combination 2		
Agile Framework	Component	Implementation Level
Scrum	Sprint Planning	Implementation Attempted
Scrum	Daily Scrum	Implementation Attempted
Scrum	Backlog Refinement	No Implementation
Extreme Programming	Code Re-factoring	No Implementation
Extreme Programming	Collective Code Ownership	Implementation Attempted
Extreme Programming	Pair Programming	Implementation Attempted

Since it could be imprecise to set a level of how well a framework component is implemented, only two levels of implementations are defined:

- **Implementation Attempted**, means the developers have made a significant attempt to embrace the agile framework component into their planning, design and work on a sprint
- **No Implementation,** means that there has been no effort to incorporate the agile framework component in the sprint

In order to isolate how much extra value the new components will add compared to the baseline configuration, the baseline components are kept in the new configurations.

3.3.1 Quantitative Evaluation

The quantitative evaluation is conducted by running different reports in JIRA. These reports are run in the same way for the baseline, SXP combination 1 and SXP combination 2. For each sprint, the following data attributes will be generated:

- Estimated time spent on development
- Time spent on development
- Time spent on design
- Time spent on resolving defects for the sprint

These numbers will then be put into the table 5.

Table 5

Sprint	**Combination**	**Original Estimate DEV (h)**	**Time Spent DEV (h)**	**Time Spent Design (h)**	**Time Spent Defects (h)**
1	SXP Baseline				
2	SXP Baseline				
3	SXP Baseline				
4	SXP Baseline				
5	SXP 1				
6	SXP 1				
7	SXP 2				
8	SXP 2				

Using these data attributes, the following relationships could be derived:

- **Delta Estimate:** Time spent on development vs time estimated

- **Delta Design:** Time spent on development vs time spent on design
- **Delta Defects:** Time spent on defects vs time spent on development

These relationships were chosen because they are used to produce the most common JIRA reports for the development teams participating in this study. This could benefit the development teams if they wish to use this evaluation framework for future tests. These relationships are then filled into the table 6.

The expectations for these relationships is to have the delta estimate close to a 100 %, since this will indicate that the developers were able to accurately estimate the effort to complete their work in time. Anything higher than a 100 % could show that more time was needed than planned and anything lower could mean that the estimate was higher than the actual time required.

For delta design, there is not a "correct" number but a guideline number often set up by someone more experienced in the team. In the company participating in this study, it is recommended to spend approximately 20 % to 40 % on design versus time spent on development. A higher number could indicate that the work might be too complex to fit into one sprint, while a lower number could mean that the solution has not been thought through enough before implementation.

Lastly for delta defects, the aim is to have as close to 0 % as possible. Since this is difficult even for experienced programmers, a threshold is usually set from either the quality assurance (QA) team or a lead engineer. In this study, the threshold were set to 20 % where anything higher could mean that there are fundamental design flaws in the solution.

Table 6

Sprint	Combination	Delta Estimate (%)	Delta Design (%)	Delta Defects (%)
1	SXP Baseline			
2	SXP Baseline			
3	SXP Baseline			
4	SXP Baseline			
5	SXP 1			
6	SXP 1			
7	SXP 2			
8	SXP 2			

3.3.2 Qualitative Evaluation

The qualitative evaluation is carried out throughout the sprint lifecycle and ends by an interview with the Scrum Master who led the sprint. The structure of the qualitative evaluation is as follows:

- The assigned Scrum Master might need to perform a planning activity and/or answer a few questions prior to the start of the sprint. This varies depending on which SXP combination that is being evaluated. These questions are tailored to the SXP combination that they are trying out.
- While the sprint is in progress, the Scrum Master and the team is asked to perform activities which also correlates to the SXP combination they are trying out.
- After the sprint is complete, the Scrum Master is asked a number of questions around sprint performance and how well the new SXP combination integrated with their work. The questions are laid out to first make the interviewee consider how the whole sprint went and then to score the new processes together with a motivation.

3.3.2.1 SXP Combination 1

Prior to the sprints using SXP combination 1, the Scrum Master will need to review the sprint backlog together with the team to determine which items that will be included in the upcoming sprint. There are two types of backlog: the sprint backlog, which consists of items that are outstanding work for the sprint (e.g. defects) and the product backlog, which are items specific to the whole module (e.g. a new feature). As part of the backlog refinement review activity, the team can choose to include items from either backlog.

The Scrum Master will also be asked the following questions:

- Are there any items in the backlog that could be clarified? (Yes/No)
- Are there any items in the backlog that can be included in the next sprint? (Yes/No)
- Did any of the backlog items overlap the new user stories from a requirements perspective? (Yes/No)
- Are there any re-factoring that will be done as part of previous code reviews? (Yes/No)

After the sprint has started, the developers are asked to make brief notes of what parts of the system that could use refactoring. This information is saved into a consolidated sheet which will be reviewed alongside an independent code review done by someone who was not involved in the sprint.

By the end of the sprint, the developers are asked the following questions:

- What went well?
- What could have been done better?
- What should we do differently the next time?
- Did the new processes that were introduced help in any way?
    - If yes, then in what way did they help?
    - If no, then could they have been introduced better or should not they have been there at all?
- Do you think reviewing the backlog before the sprint started helped you meeting the requirements? *(On a scale from 1 to 5 with five being the most positive)*
- Do you think reviewing the backlog before the sprint started helped you get a better picture of the overall state of the software module? *(On a scale from 1 to 5 with five being the most positive)*
- Do you think re-factoring parts of the programme while developing will help others understand the code you wrote? *(On a scale from 1 to 5 with five being the most positive)*
- Do you think re-factoring parts of the programme while developing created a large overhead? *(On a scale from 1 to 5 with five being the most positive)*
- Would you recommend backlog refinement? (Yes/No)
- Would you recommend code re-factoring (Yes/No)

Similar to a sprint retrospective, the developers will have the opportunity to reflect over what went well and what could be improved. The difference is that the developers are also asked questions that will show the variations within the SXP combinations.

3.3.2.2 SXP Combination 2

Prior to the sprints using SXP combination 2, the Scrum Master will need to determine which tickets in the sprint that might be suitable for pair programming. These tickets might be more complex than the average ticket in the sprint or have functionality that will affect the whole system design. In

addition to this, the Scrum Master needs to make sure everyone in the team is aware of the suggested design implementations in the upcoming sprint.

The Scrum Master will also be asked the following questions:

- Are there any items in the upcoming sprint that are suitable for pair programming? (Yes/No)
- Are there any parts of the design implementations in the upcoming sprint that not every team member is aware of? (Yes/No)

After the sprint has started, the team members continuously brief each other on the design of the high-level functionality that they have implemented. Every team member needs to be able to understand the design and functionality in case for example someone gets sick or they need to resolve a future bug in a part of the system they did not develop.

By letting a developer explain their design to other team members, logical errors can be spotted at an early stage. Pair programming naturally helps with these kind of issues by having each line of code to be reviewed by at least two developers.

By the end of the sprint, the developers are asked the following questions:

- What went well?
- What could have been done better?
- What should we do differently the next time?
- Did the new processes that were introduced help in any way?
    - If yes, then in what way did they help?
    - If no, then could they have been introduced better or should not they have been there at all?
- Do you think your awareness of the system design as a whole increased by having other developers explain their design to you? *(On a scale from 1 to 5 with five being the most positive)*
- Do you think developing together with another programmer helped improve design and code quality? *(On a scale from 1 to 5 with five being the most positive)*
- Did it feel like it took longer to develop code with another programmer took longer than if you were to develop the code by yourself? *(On a scale from 1 to 5 with five being the most positive)*
- Would you recommend collective code ownership? (Yes/No)
- Would you recommend pair programming? (Yes/No)

4 RESULTS

The data comparison is based on how the baseline performs against the two SXP combinations. As seen in the tables below, the quantitate data collection is based on time spent on development, design and resolving defects.

Table 7: Listing the results for the original estimate, time spent on development, time spent on design and time spent on defects

Sprint	Combination	Original Estimate DEV (h)	Time Spent DEV (h)	Time Spent Design (h)	Time Spent Defects (h)
1	Baseline	89.50	87.67	25.83	11.00
2	Baseline	208.00	203.00	16.00	8.75
3	Baseline	144.00	141.50	65.00	16.00
4	Baseline	236.00	239.00	82.00	23.00
5	SXP 1	102.00	129.75	5.75	5.50
6	SXP 1	41.50	60.00	6.50	4.25
7	SXP 2	38.58	45.00	24.50	2.08
8	SXP 2	72.50	95.50	16.00	3.73

Table 8: Listing the results for the time spent on development over original estimate (delta estimate), time spent on design over time spent on development (delta design) and time spent on resolving defects over time spent on development (delta defects)

Sprint	Combination	Delta Estimate DEV (%)	Delta Design (%)	Delta Defects (%)
1	Baseline	97.96	29.46	12.55
2	Baseline	97.60	7.88	4.31
3	Baseline	98.26	45.94	11.31
4	Baseline	101.27	34.31	9.62
5	SXP 1	127.21	4.43	4.24
6	SXP 1	144.28	10.83	7.08
7	SXP 2	116.64	54.44	4.62
8	SXP 2	131.72	16.75	3.91

Since methodologies can absorbed in many different ways, it is important to put these numbers into context by also giving a brief description of how the components were used.

ID	Component	Observation
O1	**Backlog Refinement**	The team would start with a core set of features that were to be included in the upcoming sprint. They would then use backlog refinement to go through the backlog and try to include features that would were directly or relatively related to the core set of features they were planning to implement.
O2	**Collective Code Ownership**	For any new feature that were to be implemented into the system, the team would set aside some time to discuss approaches and design. This would happen on a daily basis and a more in-depth understanding would follow after the team had done a code review.

O3	**Code Refactoring**	Refactoring was a continuous process which could happen after the team had discussed how to implement a new feature or after a code review. The team would need to mutually agree on design changes which would affect larger parts of the system.
O4	**Pair Programming**	The team would form pairs consisting of one experienced and one junior developer. They would then set aside an hour or two per session to discuss mainly new core features and how to implement them. The experienced developer would also bring up alternative approaches and trade-offs between them. The team would have these pair programming sessions between two to four times per week.

It was observed during the sprints using the SXP combinations 1 and 2 that the component which required the most effort to apply was pair programming. The component which required the least amount of effort to implement was code refactoring.

For the qualitative data collection, the results are provided in relation to the SDLC. The answers from the interviews have been summarised in the tables below:

General Questions			
ID	**Question**	**SXP Combination 1** *(Backlog Refinement, Code Refactoring)*	**SXP Combination 2** *(Pair Programming, Collective Code Ownership)*
A1	**What went well?**	The developer teams thought that the quality of code was higher than in previous sprints. The scope wasn't majorly affected, one of the teams met their scope and one of the teams had to drop a piece of functionality. However, they have commented that this was unrelated to the introduction of the new SXP components.	The developer teams managed to meet the scope and their estimates, even though the complexity was high and new processes were involved.
A2	**What could have been done better?**	Both teams agreed on what they could have done better. Both teams wish they would have spent more time on design and planning activities.	Requirements could have been clearer, but that was outside of the developer team's control.
A3	**What should we do differently the next time?**	For the next time, the teams think more time should have been allocated to planning and design. It is also worth trying to minimize the risk of unplanned work coming up, even though this is hard to predict.	For the next time, more time needs to be spent on the complex tickets.

A4	**Did the new processes that were introduced help in any way?**	Doing code reviews and making notes of which bits of code to re-factor increased not only the quality of the code but also helped gaining a more general understanding of the functionality.	Pair programming helped by working through the problem together, providing the ability to explain the thought process of how to approach specific problems. That opened up for alternatives ways to tackle the problems.

SXP Combination 1 *(Backlog Refinement, Code Refactoring)*			
ID	**Question**	**Answer**	**Average Score**
B1.1	Do you think reviewing the backlog before the sprint started helped you meeting the requirements? *(On a scale from 1 to 5 with five being the most positive)*	Backlog Refinement helped keeping other areas of the code fresh in mind, but did not help if you have entirely planned your development already. It helped meeting the requirements because you gained a clearer sense of the full functionality. Reviewing the backlog helped you in having good overview of your planned development and your backlog.	**3**
B1.2	Do you think reviewing the backlog before the sprint started helped you get a better picture of the overall state of the software module? *(On a scale from 1 to 5 with five being the most positive)*	Backlog refinement by itself did not give a sense of maintainability of the design, but it helped giving a sense of what is in scope and the full-scale product. It helped getting a better overview and seeing which parts of functionality is done and which is not.	**4**
B1.3	Do you think re-factoring parts of the programme while developing will help others understand the code you wrote? *(On a scale from 1 to 5 with five being the most positive)*	It depends on if the developer who is re-factoring the code also made the code clear in terms of comments or design documents. Going from complex code without documentation to re-factored code with no documentation would not help much. As long as the refactored code is better than the original code, which is a risk when refactoring.	**3**
B1.4	Do you think re-factoring parts of the programme while developing created a large overhead? *(On a scale from 1 to 5 with five being the most positive)*	Not if the parts that are getting refactoring are fairly well contained and the extent of the refactoring estimated well. However, for larger systems the complexity can increase exponentially.	**3**
B1.5	**Would you recommend Backlog Refinement?**	Yes	
B1.6	**Would you recommend Code Refactoring?**	Yes	

SXP Combination 2 *(Pair Programming, Collective Code Ownership)*			
ID	**Question**	**Answer**	**Average Score**
B2.1	Do you think your awareness of the system design as a whole increased by having other developers explain their design to you? (On a scale from 1 to 5 with five being the most positive)	It definitely helped increase my awareness, especially if you only have an abstract view of the rest of the system and are unsure of the full logic in the module.	**4**
B2.2	Do you think developing together with another programmer helped improve design and code quality? (On a scale from 1 to 5 with five being the most positive)	By having two minds on issues instead of one it allowed us to bounce ideas off each other in order to come to best design possible. During designing together we would discuss possible ways to break the code which allowed us to build more robust code and improve the code quality.	**4**
B2.3	Did it feel like it took longer to develop code with another programmer took longer than if you were to develop the code by yourself? (On a scale from 1 to 5 with five being the most positive)	It undeniably took longer to develop together with someone else than by yourself, since you have to explicitly communicate your intended design which takes more time. However I think there is a trade of here in the sense that it may have taken longer to develop but the benefits mentioned in the previous question enabled better design and code quality, which will hopefully minimise bugs and fix work in the future.	**3**
B2.4	**Would you recommend Pair Programming?**	Yes	
B2.5	**Would you recommend Collective Code Ownership?**	Yes	

5 DISCUSSION

5.1 PERFORMANCE

By looking at the numbers, one can see that the average number for the delta estimate were 98.77 % for the baseline, 135.89 % for SXP 1 and 124.30 % for SXP 2. This shows that when the development teams were using the SXP combinations, their ability to estimate the time required was not as accurate as before the introduction of SXP. A reason for this could be the overhead of learning new processes which could initially increase the time needed to the same amount of work.

The average numbers for delta design were 29.40 % for the baseline, 7.63 % for SXP 1 and 35.60 % for SXP 2. While the numbers for the baseline and SXP 2 were within agreed levels, the number for SXP 1 were significantly lower. It does not necessarily mean that the design for the sprints using SXP 1 were not well thought through, but could mean that the complexity of the requirements were not as high as usual.

The average numbers for delta defect were 9.45 % for the baseline, 5.66 % for SXP 1 and 4.26 % for SXP 2. While all the numbers were under the agreed threshold, the numbers for SXP 1 and 2 were approximately half of the baseline number. This displays a positive trend and indicates that SXP components could help developers write high quality code.

There are a number of variables to consider when evaluating the time spent on defects compared to the total time spent on the sprint. A few of these are:

- The complexity of the work in each sprint
- The level of developer experience in each team
- Origin of the defects reported (in some of the sprints, the defects were reported by internal testing and in others it was both from internal testing and UAT)

Comparing Scrum and XP from a pure performance stand-point using only the data captured in this report could be considered risky. The reason for this is that the source from where the defects were raised varies, which makes the reliability of the testing as a whole volatile.

With that in mind, there are positive trends that can be derived from these numbers as well. Even though the developers estimated stories with newly introduced processes, they were still relatively close compared to the baseline estimate values. However, there is a risk that the developers padded their tickets with extra time because they were going to use the new processes. This could mean that the time required to do the work using the new processes is significantly higher than before.

5.2 OVERALL EXPERIENCE

Comparing the results from the performance section above with the answers given in A1 for both SXP combinations, the developers themselves thought that the introduction of the new processes did not create a large overhead as a whole. More importantly, they thought that that the new processes did not affect scope. Comparing the developers' answers with the delta estimate numbers given before and after the SXP combinations, this might seem contradictory since the delta estimate average when using an SXP combination where significantly higher than the baseline average.

However, these numbers could have been affected by the fact that the complexity and amount of R&D needed for the sprints using the SXP combinations were significantly higher compared to the baseline sprints. This was not intentional but came as a natural step as the project progressed and

new requirements demanded utilities using technology the company had not used before. The aim with using agile methodologies such as SXP combinations is to improve the software development process as a whole. This means writing higher quality code and spending less time on resolving defects, something that was improved after the introduction of SXP combinations.

When introducing new processes in any company, it is tempting to make big changes in big steps. However, having to change direction while taking small steps compared to big steps can be significantly less costly (Beck, Kent. Andres, Cynthia. 2004).

Looking at the answers given in A2-A3, the teams recognised that the one of the largest issues were that there were not enough time allocated to design and planning. This is something that could be improved regardless if the team is using SXP or not. Lastly when looking at A4, the teams were unanimous that the processes had helped in their development and positive to the newly introduced components from Scrum and XP. From an observation stand-point, pair programming was the component which required the most amount of effort and code refactoring the least amount of effort.

5.3 SXP Components

5.3.1 Backlog Refinement

Analysing the results deeper and starting off with SXP combination 1, B1.1 asked the developers if reviewing the backlog helped them meet requirements. The teams agreed that this activity helped gaining a better understanding of the functionality, but thought that reviewing the backlog after you already planned your sprint was not very valuable unless you have room for contingency. For this particular purpose, the teams gave this activity an average score of three out of five.

When asked if reviewing the backlog helped getting a better understanding of the overall state of the software module, one of the teams said that backlog refinement by itself does not give a sense of the maintainability of the design. However, both teams agreed that backlog refinement helped gaining a good understanding for what is in scope and what development that is outstanding. For this particular purpose, the teams gave this activity an average score of four out of five.

Both teams said they would recommend the activity backlog refinement.

5.3.2 Code Refactoring

In B2.2 the developers were asked if refactoring parts of a programme will help others understand the code they wrote better. Both the teams were positive to refactoring as a whole, but identified risks that could come with refactoring. If the refactoring does not include any new comments or documentation, it might be as complex as the un-refactored code. Furthermore, another risk when refactoring complex designs is that the refactored solutions might not be much more improved than the original design and justify the time invested. For this particular purpose, the teams gave this activity an average score of three out of five.

However, refactoring does not always mean making a more simple design out of existing code. It could also mean changing the design in order to handle new requirements. Since refactoring code could be a significant time investment, the developers were asked in B2.3 if they thought that refactoring while developing created a large overhead. The teams agreed that refactoring while developing did not create a large overhead if the functionality that were due to be refactored were contained to a certain extent. Even though the definition of refactoring means that the functionality should be exactly the same as before the refactoring, altering the code design could mean that other

parts of the system could be affected. For this particular purpose, the teams gave this activity an average score of three out of five.

Both teams said they would recommend the activity code refactoring.

5.3.3 Collective Code Ownership

Keeping every team member up to date with the full design of the codebase can be challenging. In B2.1 the developers were asked if their awareness of the system design increased by having other developers of the team continuously brief their part of the design to the rest of the group. There is a possibility with collective code ownership that keeping every team member up to date with the full system design can be time-consuming. The benefits are many though, e.g. that every developer will be able to look at a defect anywhere in the system, and the chance of breaking other parts of the system when refactoring is lower. The development teams thought that the benefits of collective code ownership outweighed the downsides and that it was very helpful. For this particular purpose, the teams gave this activity an average score of four out of five.

Both teams said they would recommend the activity collective code ownership.

5.3.4 Pair Programming

In B2.2, the developers were asked if developing with another programmer helped improve design and code quality. One would naturally think that if given the time and resources, two programmers working side by side will always produce code of higher quality. This might not always be the case, as described by Kent Beck in his book “Extreme Programming Explained: Embrace Change”, second edition (Beck, Kent. 2004). Kent Beck discourages pair programming if not at least one of the programmers can be considered to be an experienced developer on the project. However, the teams found pair programming significantly useful, especially when discussing design and potential security vulnerabilities. For this particular purpose, the teams gave this activity an average score of four out of five.

One of the biggest arguments against pair programming have naturally been that the stakeholders think the benefits of two programmers developing side by side will not outweigh the time overhead. In B2.3, the developers were asked if they thought pair programming took longer to develop code than if they were to develop code by themselves. When developing, one has to include the time to maintain the code and resolve any defects that might come up in a later stage. The teams agreed that it did feel like it took longer, but thought that there was a trade-off in terms of fix-time and increased code quality which outweighed the extra time spent. For this particular purpose, the teams gave this activity an average score of three out of five.

Both teams said they would recommend the activity pair programming.

5.4 Analysis of SXP Modelling - Lessons Learned

When analysing what contributed to a successful SXP combination in this report, one could derive a number of important factors such as:

- How well do the components complement each other
- How the combination is implemented into the project
- How the components are used within the project

The first point on how well the components complement each other goes into the structure of the SXP combination. In this study, SDLC analysis was performed in an early stage to determine potential

areas of overlap. This was to make sure the combinations did not contain components which tried to serve the same purpose.

One example of this would be pick the XP component daily stand-up, which similarly to the Scrum component daily scrum is a practice where the development team hosts a daily stand-up to checkpoint how a sprint is progressing. One could question what the motivation is to implement a SXP combination and not a full Scrum or full XP combination when they both have very similar components. With the example of the XP daily stand-up and the daily scrum, even with seemingly identical components, there are differences in how they should be implemented by the frameworks and which rules that apply. While both XP and Scrum stress the importance of co-location, only XP makes it a deal-breaker (Bowes, Jim. 2015). This makes daily scrum more suitable when working with remote development teams.

The second point address how the combination is implemented into a project. Going back to the SDLC phases, it is important that the SXP combination is balanced. Looking at Figure 2, where both SXP Combination 1 and 2 are outlined in relation to the SDLC, one could draw the conclusion that "Testing" was the area which the combinations had the least coverage in. With XP considering automated unit testing (*AUT*) as one of its cornerstones (Wells, Don. 1999), this might seem like a mistake during the SDLC analysis phase. However, because of restrictions in the company's current development environment it was not possible during the time the study ran for to implement advanced testing practices such as AUT.

The third and last point takes up how components are used within a project. When introducing new processes into a company, it is natural to have a period of learning overhead as previously mentioned in 8.2. It is also common that the change of processes are taken in smaller steps, which means one team might have come very far into introducing one agile component, while just trying out another agile component and thus not using it to the same extent yet. However, even though a team is not spending an equal amount of time on all the agile components it is using, it is important that they follow the rules and best practices for the agile practices as much as they can. There is always a level of customization to a process for a team, but it is also important that the individuals who are meant to implement the new processes are well informed. This could mean either having years of experience in the area or perhaps taken a certification in a particular methodology, for example a Scrum Master certification.

It became clear how important the usage of the components are in this study were based on the feedback that was provided by the development teams. One example is in B1.3, which asked the developers if re-factoring parts of the programme while developing will help others understand the code they wrote. They answered that it all depends on how good the developer who is re-factoring the code both at refactoring but also providing clear comments and documentation which is essential when refactoring code. They also mentioned that there is always the risk that the re-factored code is as unusable as the original code if no clear documentation is provided.

6 CONCLUSION

Comparing software development methodologies is difficult. There are many challenges to consider, such as finding application contexts that are similar for the considered methods (Fernandes, Joao M. Almeida, Mauro. 2010).

The challenge was to investigate if an SXP combination could be more beneficial than components only picked from Scrum. It was also to investigate if there are key components from the two combinations that work better than others.

The answer to the first research question lies partly in the second research question. How successful the SXP combination will depend on how well the combination is structured. Using SXP components that greatly overlap each other in the SDLC are more likely to increase redundancy in a project. It also depends on how the components will be implemented. Taking pair programming as an example, Kent Beck recommends pair programming between either two experienced developers or one experienced and one beginner, but not two beginners (Beck, Kent. Andres, Cynthia. 2004).

As stated in the beginning of 8.4, there are a number of factors that could determine how successful a SXP combination turn out. Lessons learned from this study has shown that in order to model a successful SXP combination, the following steps are recommended:

1. **Perform SDLC analysis** - How well do the SXP components relate to the software development life-cycle? Do they overlap or/and serve the same purpose?
2. **Study the components of the SXP combination well** - Learn from the inventors and community how to best implement the Scrum and XP components
3. **Start with the least amount of effort** - Change is difficult and best taken in small steps
4. **Review and improve** - Analyse and gather feedback from the team to help develop the processes

6.1 FUTURE WORK RECOMMENDATIONS

In order to further validate the results from this study, a more extensive study could be made in order to verify the correlation between the factors in 8.4 and the steps in 8.5. This could be divided into further research within two areas: validation with more resources and further analysis of SXP modelling.

Validation with more resources could mean testing with more SXP combinations and variations in a higher number of sprints. The number of components in a SXP combination is not limited to what is used in this report and both fewer and more components could form a combination. Having more resources could help improve the coverage of each phase in the SDLC.

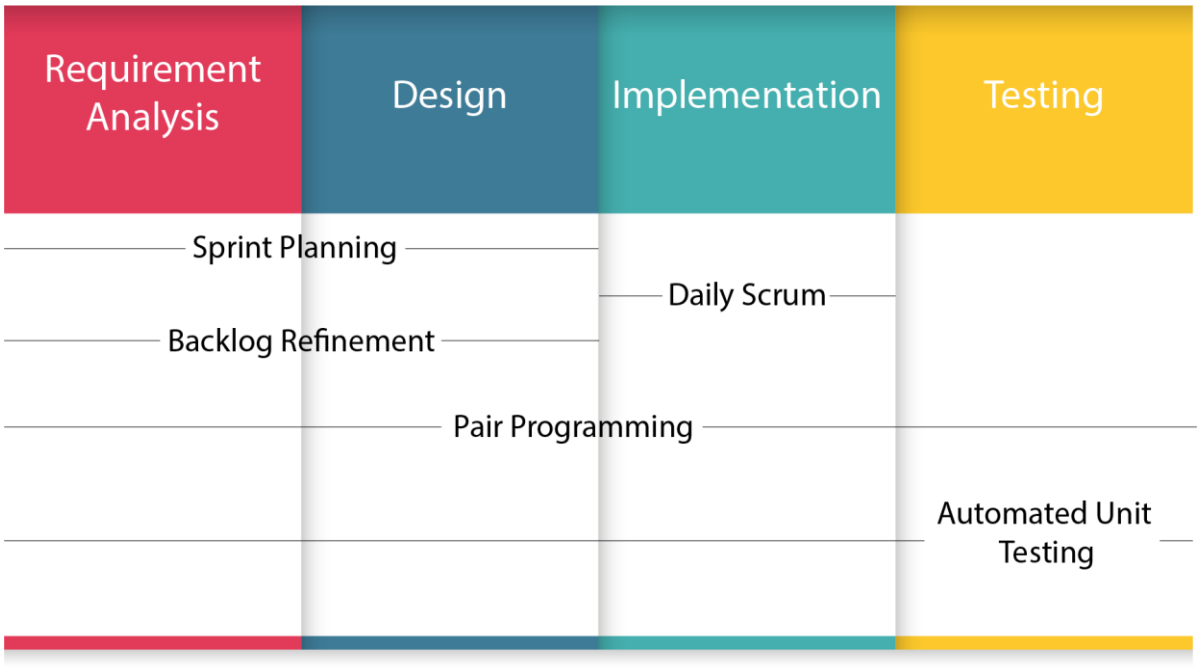

Figure 4: Suggestion of a SXP combination with improved SDLC coverage in the testing phase compared to the SXP combinations used in this report

Validation with more resources could also mean trying to align sprints to land within similar context and effort required in order to reduce any irregularities. Otherwise there is always a risk that some sprints might take longer simply because the work required is more complex.

Further analysis of SXP modelling could be to compare SXP combinations to full implementations of Scrum and XP, or even other agile methodologies. With this case study as a guideline, one might discover additional factors which could be relevant when trying to create successful SXP combinations.

7 REFERENCES

1. **Munassar, Nabil Mohammed Ali. Govardhan, A. 2010. "A Comparison Between Five Models of Software Engineering". IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 5. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.8120&rep=rep1&type=pdf#page=115 (Retrieved 13/05/2017)**
2. **Kovács, G. Spens, KM. 2005. "Abductive reasoning in logistics research", International Journal of Physical Distribution & Logistics Management, Vol. 35 Iss 2 pp. 132 - 144. Doi: "10.1108/09600030510590318"**
3. **Sutherland, Dr. Jeff. 2004. "Agile Development: Lessons learned from the first Scrum". https://www.scrumalliance.org/resources/35 (Retrieved 30/04/2017)**
4. **Sutherland, Jeff. 2001. "Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies" http://static1.1.sqspcdn.com/static/f/447037/6486358/1270929593650/Sutherland+200111+proof.pdf?token=joX7Ioq1qQEx2zTcS56oICCfv%2FU%3D (Retrieved 30/04/2017)**
5. **Abrahamsson, P. Salo, O. 2008. "Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and Scrum". Doi: 10.1049/iet-sen:20070038**
6. **Boehm, Barry W. 1988. "A spiral model of software development and enhancement". Doi: 10.1109/2.59**
7. **Powell-Morse, Andrew. 2017. "Big Bang Model: What Is It And How Do You Use It?". https://airbrake.io/blog/sdlc/big-bang-model (Retrieved 06/05/2017)**
8. **Fernandes, Joao M. Almeida, Mauro. 2010. "Classification and Comparison of Agile Methods". Doi: 10.1109/QUATIC.2010.71**
9. **Atlassian. 2017 "Companies around the world rely on Atlassian" https://www.atlassian.com/customers?page=1&sortParam=date_created%20desc (Retrieved 30/04/2017)**
10. **Cohn, Mike. 2007. "Differences Between Scrum and Extreme Programming" https://www.mountaingoatsoftware.com/blog/differences-between-scrum-and-extreme-programming (Retrieved 30/04/2017)**
11. **Beck, Kent. 1999. "Extreme Programming Explained: Embrace Change", First Edition. Addison-Wesley Longman Publishing Co.**
12. **Beck, Kent. Andres, Cynthia. 2004. "Extreme Programming Explained: Embrace Change", Second Edition. Addison-Wesley Longman Publishing Co. E-book.**
13. **Bowes, Jim. 2015. "Kanban vs Scrum vs XP – an Agile comparison". https://manifesto.co.uk/kanban-vs-scrum-vs-xp-an-agile-comparison/ (Retrieved 30/04/2017)**
14. **Royce, Dr. Winston W. 1970. "Managing the development of large software systems". https://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf (Retrieved 30/04/2017)**
15. **Benington, Herbert D. 1983. "Production of Large Computer Programs". http://csse.usc.edu/TECHRPTS/1983/usccse83-501/usccse83-501.pdf (Retrieved 30/04/2017)**
16. **Boehm, Barry W. 2000. "Spiral Development: Experience, Principles, and Refinements". http://www.sei.cmu.edu/reports/00sr008.pdf (Retrieved 06/05/2017)**

17. **Sutherland, Jeff. Schwaber, Ken. 2016. "The Scrum Guide".** **http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100 (Retrieved 30/04/2016)**
18. **Wells, Don. 1999. "The Rules of Extreme Programming"** http://www.extremeprogramming.org/rules.html **(Retrieved 30/04/2016)**