Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Santeri Mahlamäki

Improving project efficiency and sustainability:

Expectations and rework

Master's Thesis
Espoo, February 13, 2020

Supervisors: Associate Professor Casper Lassenius
Advisor: M.Sc. Mira Heiskari

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Santeri Mahlamäki		
Title: Improving project efficiency and sustainability: Expectations and rework			
Date:	February 13, 2020	**Pages:**	115
Major:	Computer Science	**Code:**	SCI3043
Supervisors:	Associate Professor Casper Lassenius		
Advisor:	M.Sc. Mira Heiskari		

The objective of this Master's thesis is to examine the case project's software development process and find ways to make it more efficient and sustainable. Any suggested changes should be cost-efficient and relatively simple to adopt, to reduce the impact of having to learn the new processes.

Factors affecting the efficiency and sustainability of the project were investigated in a current state analysis, where five of the project employees were interviewed. Two of the identified factors were deemed the most relevant: expectations and rework.

To aid in formulating realistic expectations and to reduce the amount of unnecessary rework, an explorative literature review was conducted on the two topics. The output of this literature review was a set of suggestions that the case project could try to implement. A subset of the proposed suggestions were implemented and monitored for four two-week sprints. The selected suggestions were 'Estimation meetings before planning', 'Measure the velocity and hit-rate'. 'Measure the time spent on changes vs. new development', 'Benchmarks for story points', 'Include refactoring effort in estimates', 'Re-estimate incomplete stories', and 'Work more as a team'.

Due to highly unusual circumstances in the project, multiple changes being made to the development project simultaneously, and an unsuccessful selection of rework metrics, the results of this empirical part were left largely inconclusive. Further studies, preferably with a longer time-frame and after the circumstances have become more stable, are required for reliable assessments of the effects of the implemented suggestions.

Keywords:	agile, software, development, project, efficiency, sustainability, expectations, rework, predictability
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Santeri Mahlamäki		
Työn nimi: Projektin tehokkuuden ja kestävyyden parantaminen: Odotukset ja rework			
Päiväys:	13. helmikuuta 2020	**Sivumäärä:**	115
Pääaine:	Tietotekniikka	**Koodi:**	SCI3043
Valvojat:	Professori Casper Lassenius		
Ohjaaja:	Diplomi-insinööri Mira Heiskari		

Tämän tutkielman tavoite on tutkia erään projektin ohjelmistokehitysprosessia ja löytää tapoja, joilla tuosta prosessista saisi tehokkaamman ja kestävämmän. Kaikkien ehdotettujen muutosten tulisi olla kustannustehokkaita ja kohtuullisen yksinkertaisia toteuttaa, jotta uudelleen opettelun tarve voidaan minimoida.

Tehokkuuteen ja kestävyyteen vaikuttavia tekijöitä pyrittiin tunnistamaan nykytila-analyysin avulla. Tämän toteuttamiseksi tutkimuksessa haastateltiin viittä projektin työntekijää. Kaikkein keskeisimpien tekijöiden todettiin olevan odotukset ja lisätyö.

Realististen tavoitteiden asettamiseksi ja turhan lisätyön vähentämiseksi sopivien muutosten löytämiseksi toteutettiin kirjallisuuskatsaus. Tämän katsauksen lopputuote oli lista ehdotuksia, joita kyseisessä projektissa voitaisiin kokeilla. Osa näistä ehdotuksista otettiin käyttöön, ja niiden vaikutuksia seurattiin neljän kahden viikon mittaisen sprintin aikana. Valitut ehdotukset olivat ‘Arviointitapaamiset ennen suunnittelua’, ‘Kehitysvauhdin ja valmiusasteen mittaaminen’, ‘Muutoksiin ja uuden kehittämiseen kuluvan ajan mittaaminen’, ‘Kiintopisteet story pointeille’, ‘Refaktoroinnin huomioon ottaminen työmääräarvioissa’, ‘Keskeneräisten kehitysten uudelleenarviointi’, sekä ‘Yhteistyön lisääminen’.

Johtuen projektin sen hetkisistä, hyvin epätavallisista olosuhteista, monien muutosten samanaikaisesta toteuttamisesta, sekä epäonnistuneesta mittareiden valitsemisesta, tämän empiirisen osuuden tulokset jäivät hyvin epäselviksi. Olosuhteiden normalisoitua tarvitaan pidempiä lisätutkimuksia, jotta toteutettujen ehdotusten vaikutuksia voitaisiin luotettavasti arvioida.

Asiasanat:	ketterä, ohjelmistokehitys, projekti, tehokkuus, kestävyys, odotukset, lisätyö, ennustettavuus
Kieli:	Englanti

Aalto-universitetet
Högskolan för teknikvetenskaper
Magisterprogrammet i data-, kommunikations- och informationsteknik

SAMMANDRAG AV DIPLOMARBETET

Utfört av:	Santeri Mahlamäki		
Arbetets namn: Förbättra projektets effektivitet och hållbarhet: Förväntningar och omarbetning			
Datum:	Den 13 februari 2020	**Sidantal:**	115
Huvudämne:	Datateknik	**Kod:**	SCI3043
Övervakare:	Lektor Casper Lassenius		
Handledare:	Diplomingenjör Mira Heiskari		

Syftet med denna masteruppsats är att undersöka fallprojektets mjukvaruutvecklings process och hitta sätt att göra det mer effektivt och hållbart. Eventuella föreslagna ändringar bör vara kostnadseffektiva och relativt enkla att använda för att minska effekterna av att lära sig de nya processerna.

Faktorer som påverkade projektets effektivitet och hållbarhet undersöktes i en aktuell tillståndsanalys, där fem av projektanställda intervjuades. Två av de identifierade faktorerna ansågs vara de mest relevanta: förväntningar och omarbetning.

För att hjälpa till att formulera realistiska förväntningar och minska mängden onödig omarbetning genomfördes en undersökande litteraturöversikt om de två ämnena. Resultatet av denna litteraturöversikt var en uppsättning förslag som fallprojektet kunde försöka genomföra. En delmängd av de föreslagna förslagen implementerades och övervakades under fyra språng på två veckor. De valda förslagen var ”Uppskattningsmöten före planering”, Mät hastighet och träff frekvens”. Mät den tid som spenderats på förändringar mot ny utveckling”, rriktmärken för ‘berättelsepoäng’”, ”Inkludera refactoring-ansträngningar i uppskattningar”, Re-estimera ofullständiga berättelseröch Ärbeta mer som ett team”.

På grund av mycket ovanliga omständigheter i projektet, samtidigt som flera förändringar gjordes i utvecklingsprojektet samtidigt och ett misslyckat urval av omarbetningsmetriker, lämnades resultaten av denna empiriska del till stor del otydliga. Ytterligare studier, helst med en längre tidsram och efter att omständigheterna har blivit mer stabila, krävs för tillförlitliga bedömningar av effekterna av de implementerade förslagen.

Nyckelord:	agile, programvara, utveckling, projektet, effektivitet, hållbarhet, förväntningar, omarbeta, förutsägbarhet
Språk:	Engelska

Acknowledgements

I would like to thank my supervisor, Mira Heiskari, and my professor, Casper Lassenius, for all the encouragement and support I received during the conducting of this thesis. The process would not have been this smooth and pleasant without your aid.

I would also thank the case company of this study for providing the opportunity to conduct this study within one of their projects.

Espoo, February 13, 2020

Santeri Mahlamäki

Abbreviations and Acronyms

AQUSA	Automatic quality user story artisan
AR	Action Research
AWS	Amazon Web Services
BC	Burndown chart
CET	Combined estimation tendency
CFD	Cumulative flow diagram
CI	Continuous integration
EBV	Earned business value
ERP system	Enterprise resource planning -system
ET	Estimation tendency
FST	Fault slip through
MMRE	Mean magnitude of relative error
MRE	Magnitude of relative error
PO	Product Owner
QUS	Quality user story
RE	Requirements Engineering
ROI	Return of investment
SaaS	Software as a Service
SLOC	Source lines of code
SM	Scrum Master
SP	Story point
UI	User interface
UX	User experience
VCS	Version control system
WIP	Work in progress

Contents

Chapter 1

Introduction

As with any industry, the competition is tough in the software business. More software is expected faster and for less money, putting pressure on the software development processes.

To be able to expand to multiple markets, software systems need to be adjustable for different needs, customs, regulations, and legislation. Flexibility increases the complexity of the software, making it more difficult to maintain. A balance ought to be found between the development speed and sustainability.

Such is the situation also with the case product program of this study. The product is a cloud-based ERP system for a specialized industry in healthcare, begun in 2015. It has over 3000 daily users across Europe and employs over 60 people.

Multiple larger customers in different countries have taken an interest in the product. However, each of them requires modifications to the current functionality to make the product fit into their existing processes and local legislations. Some of these changes require massive overhauls of the current features, yet these changes should be possible to be made without affecting the user experience of the other customers.

Furthermore, due to the competition, it is deemed important to be able to expand to multiple markets simultaneously. While the resources are very limited, and additional resources are difficult to acquire, there is a need to find out ways to improve the efficiency of the development process to better serve the customers in different markets.

However, one can push the development speed only so much before starting to see some side-effects. The amount of defects has been increasing, requiring resources for better quality assurance (QA) practices. However, these practices require resources, which puts a tax on the development speed. Omitting proper documentation and having specified developers for given areas increases the speed on the short-term, but is creating bottlenecks and slowing down the development in the long term.

Yet, the need for even faster development speed is only increasing with the number of markets the product is trying to expand into. A balance between the development speed and how the speed can be sustained needs to be achieved to maintain a competitive edge within the markets.

1.1 Problem statement

The objective of this study is to examine the case product's development process and find ways to increase its efficiency while keeping the process sustainable. Due to constant schedule pressure, the proposed changes should be cost-efficient and relatively simple to adopt, to limit the downtime caused by having to learn the new processes.

The objectives of this study can be summarized with the following research questions:

RQ1 Which factors affect the case product's efficiency?

RQ2 Which factors affect the case product's sustainability?

RQ3 How the case product's efficiency could be improved cost-efficiently?

RQ4 How the case product's sustainability could be improved cost-efficiently?

Online dictionary Lexico defines the term 'sustainability' as "The ability to be maintained at a certain rate or level" (Lexico, n.d.). However, nowadays the term is often associated with environmental sustainability. In this thesis, the term sustainability is used to refer to the aspects that allow the software development to continue at a given pace.

While this study aims to bring tangible benefits for the case product program as improvements to the project's efficiency and sustainability, it also aims to contribute to the software development research scene. While agile software development (Highsmith and Cockburn, 2001; Lalsing et al., 2012; Leau et al., 2012) and project efficiency (Sedano et al., 2017) are frequently discussed topics in the literature, articles focused specifically on sustainability are harder to find. At the very least, this study aims to combine relevant findings of previous studies, with regards to efficiency and sustainability, to one concise study. This study also seeks to shed light on factors, their effects, and relationships, concerning these two topics.

1.2 Structure of the Thesis

The remainder of this study is structured as follows.

Chapter 2 elaborates on the context for the case product program and the company at large.

Chapter 3 describes the study process. The methods used during each study phase are presented in separate subsections.

Chapter 4 presents the results of a current state analysis, conducted at the beginning of the study. The factors affecting the efficiency and sustainability, identified by interviewing the employees of the case product program, are described in this chapter. In addition, a more in-depth look at the program's current development methods is provided at the beginning of this chapter.

The results from a literature review focused on the most important factors identified during the current state analysis are collected to 5. The output of this literature review is a list of suggestions that ought to have a positive impact on efficiency and sustainability.

A set of the suggestions proposed at the end of the literature review were adopted and monitored for four two-week sprints. The observations and results from these sprints are combined into chapter 6. A brief analysis of the possible effects of the implemented suggestions are also discussed in this chapter.

Chapter 7 discusses the results of the previous chapter a bit further and considers the limitations of the study. An assessment of whether or not this study achieved the set objectives is made in this chapter.

Finally, chapter 8 summarizes the conclusions derived from the findings of this study.

Chapter 2

Background

In this section, we will briefly present the case company and product program to provide some background and context to this study. The company, in general, will be described in section 2.1, and the specific product this study focuses on will be discussed in section 2.2.

2.1 Case company

The case company is a Finnish software company founded in 2001 and is one of three affiliates of a larger concern. It consists of five offices, four across Finland and one in the United Kingdom, and has roughly 80 employees at the time of writing. Most of the employees are located in the same office in Southern Finland, while the rest of the four are considerably smaller. The company has been growing quickly during the past few years: in 2017 there were roughly 50 employees (in May 2019 another 12 recruits started working).

Recently, the company has been becoming more international through hiring employees, especially customer supporters and sales negotiators, with international backgrounds: in addition to Finnish employees, the company now has at least one employee from Norway, Italy, the United States, and Spain. All non-Finnish employees are working with the same product.

The company focuses on SaaS development for two special industries as well as custom-tailored systems.

2.2 Case product program

The product is a cloud-based ERP system for a specialized industry in health-care. The product program was started in 2015 and is considered a remake of a purchased system with the same name. It has over 3000 daily users across

Europe and employs over 60 people in the company, making it the largest product within the case company, and the only product that is focused on markets outside of Finland.

While the product is intended for general use, major customizations are being constantly developed for larger customers. The product has primarily relied on own servers and hardware for hosting the service, but in late 2018 started gradually utilizing Amazon Web Services (AWS) (Amazon, n.d.). The utilization is a work in progress.

The product is being developed by one team that consists of 10 developers and a product owner (4 new developers began working in May 2019, 3 of them quit working after the summer, and one additional developer was recruited in September). The rest of the personnel consists of customer supporters, managers, sales negotiators, and accountants. The developer team adopted Scrum (Deemer et al., 2010) as their development framework in the summer of 2017.

Eight of the developers along with the Product Owner (PO) are located in the largest office, while the rest two work remotely from other offices. (the 4 summer employees are also situated in the largest office) The remote workers attend the meetings via conference call but are mainly communicating through text communication channels while working. The lead developer of the team fulfills the Scrum Master's (SM) role and is also located in the largest office. At the time of writing, two of the developers are working part-time, both located at the largest office. The distribution of team members across the offices is depicted in 2.1.

Scrum related events, such as daily standups and sprint plannings and -reviews are held in a separate, typically same meeting room. Otherwise, the team works in an open office along with employees from other projects.

The sprint length is two weeks and each release consists of two sprints. The releases are typically scheduled to be published one sprint after the end of its last sprint: when the first sprint of release N+1 ends, release N is published.

Sprint plannings are held on Wednesday mornings, and sprint reviews are held on Tuesday afternoons (recently the reviews have been on Tuesday mornings). Daily stand-ups are held every afternoon if no other Scrum events are scheduled for the day. Retrospectives were first held every sprint, but were recently changed to be held once every two sprints.

A more in-depth look at the program's development methods is provided in section 4.1.

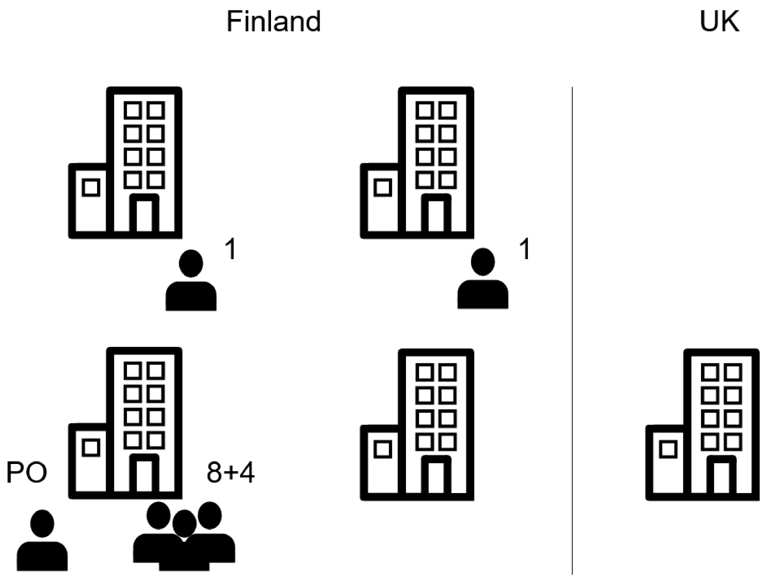

Figure 2.1: Distribution of developers across the offices

Chapter 3

Methods

The used research methods during this study are elaborated in this chapter. The study follows a process akin to the *Action research* -methodology. Coughlan and Coghlan (2002) discuss its general process, goals and, applications in their article. They summarize action research (AR) as "...an approach that aims both at taking action and creating knowledge or theory about that action." Referring to multiple authors, they list four key characteristics that define AR: "research in action, rather than research about action", "participative", "concurrent with action", and "a sequence of events and an approach to problem solving".

According to Coughlan and Coghlan (2002), AR follows a cyclical process consisting of the following steps:

- 1): Understanding the context and purpose
- 2): Gathering data
- 3): Acquiring feedback for the collected data
- 4): Data analysis
- 5): Planning the action
- 6): Implementing the action
- 7): Evaluating the effects
- 8): Monitoring the process

The steps 2-7 are cyclical, i.e. the step 7 is followed by step 2 if another iteration is executed. The authors categorize these steps as the "six main steps". Step one is categorized as a "pre-step", conducted only once at the beginning of the research. Finally, the authors categorize the step 8 as a "meta-step", and is carried out as a part of each main step.

The steps discussed above map to the chapters of this study as follows: Chapters 1 *Introduction* and 2 *Background* elaborate the context and purpose of this study, fulfilling the step 1. Steps 2, 3 and 4 are conducted during the chapter 4 *Current state analysis*, during which the current situation was being evaluated. Section 3.1 discusses further the steps taken at the beginning of the research when the current situation was being evaluated. From the findings of the current state analysis, two problem domains that were deemed most impactful were chosen for further study.

During chapter 5 *Literature review*, further information regarding the selected problem domains were seeked out from the literature to guide the action formulation, thus mapping it well to step 5. Section 3.2 discusses the literature review that was carried out for the selected issues. In chapter 6 *Evaluation*, a subset of the suggested actions were carried out per step 6. The process of this empirical step is elaborated in section 3.3. Finally, the effects of the selected actions are evaluated and discussed in chapter 6, and continued in chapter 7 *Discussion*.

While the AR studies typically consist of multiple iterations of the discussed steps, this study only conducted one such iteration. Based on the findings in chapters 6 and 7, another iteration of AR could be sparked with adjusted focus.

3.1 Current state analysis

At the beginning of this study, a brief pre-study was conducted on the case product program to provide a clearer picture and background of the current practices, strengths, and weaknesses. The goal of the study was to identify factors that affect the case product program's sustainability and efficiency, and some of those factors ought to be improved. The results of this analysis would also serve as the starting point for the primary study by assisting in selecting the primary focus. They were also utilized in writing the Background -chapter (see chapter 2), along with the author's insights while working within the product program.

The study was conducted through five semi-structured interviews (DiCicco-Bloom and Crabtree, 2006), each lasting roughly from one to one and a half hours. Despite being organized around predefined interview questions, semi-structured interviews emphasize the dialogue between the interviewer and interviewee: additional questions emerge during the discussions (DiCicco-Bloom and Crabtree, 2006). According to DiCicco-Bloom and Crabtree (2006), in-depth, semi-structured interviews, conducted for one interviewee at a time, "...allows the interviewer to delve deeply into social and personal matters...”.

Due to difficulties in finding a suitable time, one of the interviews were held in two separate sessions, with two hours in between them. The interviews were focused on the sustainability and efficiency of the product’s development. The interviews were begun by discussing the interviewees’ current role and tasks, and how they see the work is being done in the product program. After that, general discussion about efficiency and sustainability were held, mapping how the interviewees define these concepts and how they view the product in terms of these aspects. Potential problems in either of the two aspects were discussed further, aiming to arrive at a root cause for the issues. Finally, the interviewees were asked what they could do differently to alleviate the identified issues, in either their current role or if they had the authority to make decisions.

A rough outline of the interviews are provided in the first appendix (A), which contains the template that was used during the interviews.

To get a broad view and multiple points of view of the current situation, the selection of the interviewees was aimed to be as diverse as possible. Thus, the interviewees were mainly selected from different roles within the product program: the project manager, the product owner, a customer supporter, and two developers were selected. The developers were selected based on their experience, one senior and one junior developer.

Table 3.1 summarizes the durations of the interviews for each individual role.

Role	Duration
Product Manager	51min + 35min
Product Owner	66min
Customer Support	63min
Developer 1	51min
Developer 2	93min

Table 3.1: Interview roles and durations

The interviews were recorded for further analysis. Each recording was listened through shortly after the interview, and interesting insights and possible quotes were written down. During the listening, the focus was on identifying factors: which factors affect the sustainability and efficiency of the product program? Furthermore, any possible relationships between the factors themselves were also looked for. Finally, the insights of specific factors along with their causes and effects on other factors were grouped into categories of factors.

Once the insights had been collected from all interviews, the final groupings were developed from the preliminary groupings of each interview: smaller and similar groups were merged to keep the number of factors manageable. After that, the insights collected from the individual interviews were grouped under the final factors. Finally, a mind-map was drawn from the groupings by connecting the groups based on their causes and effects.

The results of the current state analysis are presented in chapter 4.

3.2 Literature review

The aforementioned current state analysis was conducted to provide visibility of the factors that affect the product program's sustainability and efficiency. The identified factors and their relationships were presented to the Product Manager, Product Owner and Lead Developer in a brief meeting.

After the factors and their relationships had been presented, a discussion was held about which of the factors would be important to investigate further. The selection was based on the amount of influence the factors had to other factors, how directly the factor influenced sustainability or efficiency, how much effort is already being put into resolving the issues related to the given factor, and general interest in the factor by the participants.

The discussion yielded four factors for further prioritization: Expectations, Predictability, Rework and Interruptions. The prioritization was done in rounds: during the first round, taking turns, each participant named their priority number one and brief justifications why it should be the number one priority. The priority numbers were written down for each factor. After the first round, each participant named also their second priority, again along with brief justifications.

Two of the factors that had received the most priority points were selected for further investigation. The selected factors were Expectations (5.1) and Rework (5.2). However, it was apparent that the issues regarding expectations had strong ties to predictability, and thus, predictability will likely be included within the expectations.

Finally, at the end of the meeting, two initial objectives were developed from the two selected factors. The aim related to expectations is to find a way to make them more realistic, and the aim related to rework is to try to minimize the amount of unnecessary rework. These goals for the literature review were formulated into the following objectives:

- O1: How to set realistic expectations?
- O2: How to reduce the amount of rework cost-efficiently?

Material for the literature review was searched from Google Scholar. Papers about expectations were searched with following search phrases: "realistic expectations in software development", "agile expectation management", "agile predictability" and "agile estimating and planning", "agile estimation accuracy". For papers about rework, the following search phrases were used: "software development rework", "software development unnecessary rework",

Due to the scope of a master's thesis and the nature of the researched issues, the literature review is more explorative rather than systematic: it was not known before-hand what practices or tools could be used to help achieve the assigned goals. Instead of going through all the resulting study papers with the given keywords, only some of them were analyzed for actors, methods, practices, tools, etc. that could be affecting expectations and rework. The papers to be analyzed were selected based on their abstracts and availability: only relevant papers that the author had access to were considered. The identified aspects were then pursued further in separate searches. Second-hand references were, when possible, resolved by seeking out the original paper: references to material that could not be accessed for free with the university's licenses were not resolved.

The results of the literature review are presented in chapter 5.

3.3 Empirical study

A set of suggestions were generated from the studied material during the literature review. Some of the suggestions were considered 'mandatory', and were adopted by default by the teams. For the rest of the suggestions, it was up to the teams which they would like to implement. The selected suggestions were implemented and monitored for four sprints.

The author worked closely with the teams during the implementation of the suggestions: to both provide support and to gain visibility of the effects. The selected suggestions were regularly evaluated in the teams' sprint retrospectives: in addition to the general process improvements, the discussions were also guided to the implemented suggestions. The team members shared their experiences with the implemented suggestions, evaluating their effects and usefulness. Despite the fixed length of four sprints, the suggestions were allowed to be modified and even dropped prematurely if they did not prove to be helpful.

Also, the results of the four studied sprints, especially in terms of hit-rate, scope changes, and velocity, were collected for further analysis. For identifying whether the suggestions had had an impact on these aspects, the data available for the previous sprints of the year 2019 was also gathered. The gathered metrics, along with comparisons to those of the previous sprints are presented and analyzed in chapter 6.

Chapter 4

Current state analysis

The conducted interviews revealed several factors that may have an impact on the case product program's efficiency or sustainability. Certain factors were assumed to have a direct impact, whilst others amplified the effects of other factors.

In this chapter, the results of the current state analysis are presented. Section 4.1 depicts the current development process in more detail. The factors identified during the interviews are discussed in section 4.2. The factors, along with their identified causes and effects, are summarised at the end of the chapter, in section 4.3.

4.1 Development process

This section elaborates on the case product programs current development process. The individual 'steps' are discussed in more detail in their respective subsections. Figure 4.1 offers a brief summary of the entire process.

4.1.1 Requirements Engineering

In general, new features and requirements are typically generated by the sales personnel and customer supporters through communicating with the clients: whenever a need emerges that cannot be satisfied with the current implementation, the employee will create a ticket in the product's backlog management tool, Jira. Some ideas are also generated by the company stakeholders themselves, including product manager and accountants: such ideas are typically aimed at gaining a competitive advantage, for example, features that competitors may have implemented poorly. The ticket is evaluated by the CEO and PO for its usefulness for general use: features that are deemed

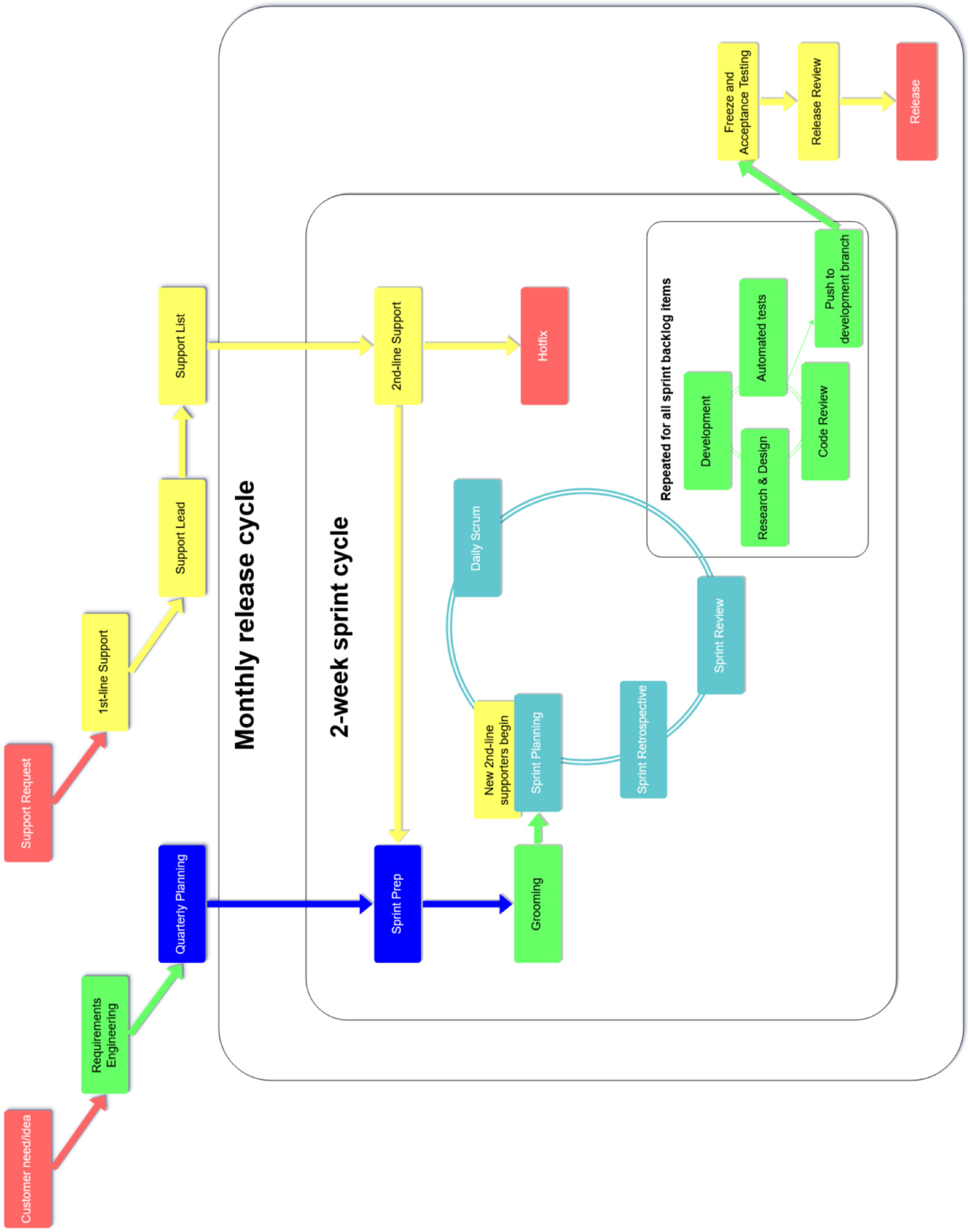

Figure 4.1: The development process

to provide value are refined with further details from discussions with the requesting client and related accountants and prioritized into the product backlog. Sometimes from one to two developers are also included in the specification discussions.

For two of the bigger enterprise clients, the Requirements Engineering (RE) process is a bit different. In these cases, the assigned accountant typically handles the requests and specifications without consulting the PO. Once specified, the accountant asks for an effort estimate from either the lead developer or the system architect and communicates that to the client (compare to general effort estimate in section 4.1.2) In more technical aspects, the accountant may also involve a developer in the specification discussions with the client.

Tickets that wouldn't provide enough value to compensate the costs are discarded. However, even if the feature or change may not be applicable for all users, they are typically implemented if the requesters are willing to pay for its development.

With larger features, especially with major user interface (UI) changes, the design team is often requested to design and create a prototype of the given feature. However, the utilization of the design team has not been systematic and lacks a real process.

4.1.2 Planning

Recently, the product program has started to develop a quarterly planning for the product's development. The accountants, sales personnel, the PO, the product manager, and other stakeholders meet regularly to discuss the future development items. Each accountant, sales personnel, etc. typically serves as the voice of a certain client group, either clients in a specific country or a specific enterprise, making sure that their wishes and needs are considered in the quarterly plans.

The stakeholders examine the generated tickets and prioritize them into upcoming quarters. The prioritization of items in the quarterly plans are affected by several factors: "How many clients will this benefit?", "Is this feature a pre-requisite for another important feature?", "Has it been requested by the bigger clients?", "Has the sales personnel given any deadlines for their implementation?", to name a few. For some bigger clients, an agreement of fixed development time has been agreed on: for instance, the company has promised one enterprise client 18 weeks of delivery time from a statement of work to release. Tickets with their deadlines closest to the next release typically take the highest priority.

Each quarter is divided further into sprints, which serve as the starting point for sprint plannings. The PO, in a way, serves as a gatekeeper and the voice of the development team, ensuring that the plans remain realistic considering the development speed.

Occasionally, before the sprint planning, a 'grooming' session is held between the PO and a couple of developers. In these sessions, the goal is to go through some of the more technical development items planned for the next sprint and assess if the specifications are of sufficient detail. If the use case is not properly communicated in the ticket, it is typically postponed to the next sprint, and the PO schedules a meeting with the stakeholder that had reported the ticket to discuss the specifications further. If the use case seems clear, the developers quickly discuss the essential technical aspects of the ticket, developing a rough implementation plan for the ticket: questions such as "What changes to the database schema are required?", "Are there pre-existing components that can be reused here?", "Which other areas will this change affect?" are considered during the development of the plan.

Sprint planning is held at the beginning of each sprint, on Wednesday mornings. The development team goes through the list of prioritized, planned tickets and discuss them together with the PO. The goal of the discussion is to get a shared understanding of what the ticket is about, how the feature or change could be implemented, and how much effort it will likely take.

The effort estimation is done by playing Planning Poker (Raith et al., 2013), where each developer gives their initial estimation in Story Points (Coelho and Basu, 2012; Hamouda, 2014). If the estimations vary largely among the developers, further discussion of the effort takes place. The developers with the highest and lowest estimate typically begin by sharing their perspective on why their estimates would be the most appropriate. After discussion, another round of estimates are given, and the process continues until the development team reaches sufficient consensus of the effort. This typically does not take more than two or three rounds of estimates.

At the end of the sprint planning, each developer selects a ticket that they will begin to work on after the planning, focusing on the top-prioritized tickets.

Sprint planning has typically lasted from two to three hours.

4.1.3 Development

The developers start the development with the ticket they selected at the end of the sprint planning. Only the first tickets are pre-allocated, rest are left unassigned and are available for selection by anyone within the team. The developers are expected to assign themselves new tickets only after their previous tickets are done, or if further development is not possible. For example, the ticket may be waiting to be code reviewed (See section 4.1.7).

Only one developer is assigned per ticket, and the developers work independently on them. However, asking colleagues for assistance is highly encouraged if something seems unclear. Occasionally, if there are not any unassigned tickets remaining when a developer finishes their work, they can offer their assistance to other developers in finishing their tasks. Alternatively, they could help the technical support in fixing bugs for the remainder of the sprint, if the sprint is almost complete.

The developers typically begin their development work by familiarizing themselves with the ticket's requirements and the existing features it affects. The constraints imposed by the requirements and current implementation guide the planning process of how the new features should be included. Integration of the new features may require refactoring the existing implementation, and sometimes the new requirements may turn out to be in direct conflict with them.

If such issues arise during the development, the developers are often directly in contact with the ticket's creator, often a customer supporter or a salesperson. Depending on the complexity of the raised issue, the PO, a UI/UX designer, and/or the lead developer may also be included in the discussion. If the issue remains unresolved after the discussion, it is often agreed that the requirements of the ticket require further elaboration, and is taken off of the sprint.

The order in which the implementation, planning and exploring the existing implementations are carried out is not fixed and the steps are often repeated throughout the development.

Once the new changes correspond with the descriptions of the ticket, the development is considered done and it will continue to quality assurance (QA)(See section 4.1.7).

4.1.4 Daily standup

During the sprint, a daily standup is held every afternoon to provide a brief status update and assist in coordination. The standup is held between the development team and the PO. Each developer communicates briefly what they had been working on since the last meeting, what they are currently working with, and if there are any obstacles. The rest of the team may give input or advice for any issues that are raised during the meeting. Sometimes, however, the issues are more complex, and a separate meeting is agreed upon with the participants.

The daily standups are also utilized for team-wide announcements and agreements, for example when the next hotfix or update will be run and who runs them.

The target duration for the daily standup is 15 minutes, but due to the rather big team size, it often lasts slightly longer.

4.1.5 Review

At the end of the sprints, on Tuesday afternoons, the team holds sprint reviews where the development team together with the PO go through the results of the sprint. The tickets in the "done" column in the backlog tool are gone through first. The developer responsible for the ticket's implementation will briefly summarize what the ticket was about and how it was implemented. With larger features, the developer may also give the team a quick demo on their local environment of the made changes. Done tickets are archived after their presentation.

After "done" tickets, the unfinished but started tickets are gone through. Again, the responsible developer briefly summarizes what the ticket was about and what has been done thus far, along with summarizing the work yet to be done. If there is only a little work to be done, the developer may conclude that the feature will be done before the next day's sprint planning, when the sprint officially ends. In these cases, the ticket may be archived in advance. Otherwise, the unfinished tickets are pushed to the next sprint.

The review usually takes from 1.5 to 2 hours.

4.1.6 Retrospective

Right after the sprint review, the team holds a sprint retrospective with the PO to reflect on the results of the past sprint. While the method of doing this varies, most typically the retrospective is held as follows: first, each developer writes down their thoughts anonymously on a piece of paper, focusing on questions "What did we do well?", "What we need to improve?", "What should we try doing next sprint?". After that, the papers are collected, shuffled, and read out loud by some of the developers. However, those participating remotely will read their thoughts out loud by themselves, compromising the anonymity of their answers. The anonymity, however, is not deemed sensitive by the team members, and it is not uncommon that the member discloses their answers during the discussions.

The points raised in the papers are discussed by the team, and further points may be raised during the discussions. The results are recorded into a team wiki, where they are available for later inspections. Unfortunately, the results of the retrospectives are hardly ever returned to once they are recorded.

Retrospectives have typically taken from 30 minutes to one hour. A short, from 5 to 15-minute break is scheduled in between the review and retrospective.

4.1.7 Quality Assurance

Once the development of the features or changes in a ticket are complete, they will go through a QA-routine before the changes are committed to the development branch of the version control system (VCS). The developer is responsible for writing automated unit-tests for any back-end changes they have made. The developers do not write tests for the front-end. While the product also has a couple of automated front-end test cases, they are only maintained by one tester (outside of the development team) and only target a small set of the most important features.

While the team wiki contains some advice on writing the tests, no real 'rules' regarding the coverage, inputs or methods have been developed. Thus, the level of reliability of the test cases varies across the features.

In addition to the newly created tests, the developer runs the existing automated tests either locally, or by committing the changes to a feature branch, in which case the continuous integration (CI) tool will test and build the software automatically.

Once all the tests pass, the developer creates a merge request to the development branch. Before they can merge the feature into the development branch, another developer has to do a code review to the made changes. The reviewer will do static analysis of the code, see if they can find any problematic, style violating, inefficient, unused, or confusing parts, and marks these sections for the developer to fix before merging. While some guidelines for conducting the code review exists, the level of detail and which areas are focused on depend heavily on the developer conducting the review.

The reviewer might also check out the feature branch and test the features in their local environment to see that it works. If no complaints from the reviewer are made, or once all of the complaints have been resolved, the developer can merge the feature branch to the development branch. Another pass of code reviewing is not conducted for resolved complaints. Code review pairs for the sprint are selected at random at the end of the sprint planning.

Roughly a week before each release, a separate acceptance testing is conducted for the tickets that were completed after the last release. A separate release branch is created from the development branch to prevent any new changes from entering the release. The acceptance testing is conducted on the release branch.

There are two primary testers responsible for the acceptance testing, sometimes assisted by either some of the developers or customer supporters. They will go through each ticket and compare the behavior to the description of the ticket. If any problems or confusion are detected, the testers will post a short description of the problem to the development team's communication channel. Any developer is authorized to correct the problem, though this is primarily done by the developer responsible for developing it.

Unfortunately, if the problems are more severe and there is little time to correct the problems, the faulty changes occasionally end up being released regardless. Any remaining problems are intended to be resolved as fast as possible in hotfixes.

4.1.8 Support process

The customer support for the product consists of three layers: the first-line support, the second-line support, and finally the development team. The support process is orchestrated by a dedicated Support Lead.

The first-line support consists of dedicated customer supporters: while all provide support in English, two of them also specialize in providing support in Swedish and Norwegian. Also, country-specific accountants and sales personnel provide general customer support for the given country, often in the country's official language.

Their tasks consist of answering the clients' questions directly, either through support chat, emails or phone calls. The questions typically relate to the actual use of the product. The client may not know, remember or understand how a certain feature works or is not aware of whether doing something is possible.

However, if the question or request proves to relate to a bug, technical issue or -task, the first-line supporters escalate the issue to the Support Lead. Most issues that do not require code changes (i.e. bugs) are often handled by the Support Lead directly. Examples of such cases include copying settings to another environment and setting up third-party -features. Bugs and more advanced technical issues are added to a list of open support issues, managed and prioritized by the Support Lead.

For every sprint, two of the developers are assigned to the role of second-line support. During this time, they only focus on urgent issues and going through the list of open support issues. These developers are essentially out of the actual sprint and do not commit to the sprint goals.

Whenever a bug is fixed from the list, the changes are added to a hotfix branch and the issue is marked as resolved from the list. The developer that resolved the issue might also ask the other supporter to quickly review the changes they have made. The CI takes care of running the automated unit-tests when the changes are committed, but user acceptance testing is not conducted on hotfix changes.

The second-line supporters are responsible for running the hotfixes. The frequency and size of the hotfixes depend on the severity of the issues they are fixing: if the changes that are pushed to the next hotfix are fixing a critical issue, they might be the only changes in that hotfix and the hotfix will be run soon after they have been committed. Correspondingly, less critical fixes are pooled into the hotfix and are run less frequently as a larger batch. Depending on the release, there are typically 10 to 20 hotfixes between each release.

If the bug fix would require more than an hour to resolve, instead of fixing them right away, the second-line supporters create a ticket of the bug to the team backlog. From that point onward, the bug is treated similarly as a new feature ticket: it will be prioritized by the stakeholders and eventually scheduled to a sprint, in which the development team makes the required changes to fix it.

The first-line supporters often get back at the client who had reported the issue if it was resolved by the Support Lead or in a hotfix. The issues resolved by the development team as a part of a sprint are pushed into the next release and will be communicated as a part of release notes.

4.1.9 Release review

Generally, the release cycle consists of two full-length sprints, resulting in roughly one release per month. This, however, has some flexibility, especially during holiday seasons.

A week after the second sprint of the release cycle, a release branch is created from the codebase. This branch is considered 'frozen': no changes to the code, other than bug fixes, are allowed to the frozen branch. The release branch is held frozen for approximately one week, during which the final release tests are carried out before the changes are released.

On the following Monday, after the freeze has been issued, a release review -event is held. During this event, the most relevant changes, including features and fixes, are presented to the support personnel and other relevant stakeholders. The biggest changes are often demoed by the responsible developer, and the demos are recorded for future reference. The meeting is very similar to the sprint reviews, except that they consider everything related to a full release cycle. The tone is also more focused on non-technical people and focuses more on how the features are used rather than how they were developed.

4.2 Identified factors

The factors, influencing the efficiency and sustainability, identified during the interviews are presented in this section. A separate subsection is dedicated for each individual factor. The factors are in the order of their discovery, without particular consideration.

4.2.1 Expectations

Expectations seem to play an important role in understanding the current situation, as it seems to have a direct effect on the perception of the program's efficiency, and thus, indirectly an effect in the program's sustainability. Discussions regarding expectations emerged in four of the five interviews. Both internal and external expectations are being laid on the development team, by the product product's management and the clients.

The company does not have explicit means of measuring the speed of the development, and thus, the internal expectations are based mostly on past development speed. The program does use story points for backlog items for getting a figure of how much work the development team can complete during a sprint, but the figure has not yet been reliable enough (see section 4.2.5). The development used to be very fast during the early years of the program, even with a considerably smaller team. According to the interviewees, the past development speed seems to affect the expectations of the current development speed. Furthermore, the clients seem to have also come to expect quick development due to the historic development speed, and for larger clients, a contract has been made that promises a fixed development time for their ordered features.

However, the product is much different today from what it was during its early years. The product is now much more complex, is becoming more global, and has big clients with very strict requirements. The number of developers has more than tripled, causing more coordination overhead. The program now also follows a rather laborious QA process, compared to the ad-hoc testing of the past. Considering all this, one of the interviewees feels that *"Stakeholders have too high or optimistic expectations"*.

Due to the high expectations, at least partly, more backlog items seem to be pushed to the sprints than the development team can reliably develop, making the velocity measurements unreliable (see section 4.2.5). Made promises and expectations, along with overloaded sprint backlog seem to cause rushing, taking a toll on the quality (see 4.2.2). Finally, because the sprints contain more work than the development team can reliably develop, delays for tasks that were promised for the clients by a certain date need to be communicated to the clients, causing an overhead.

4.2.2 Quality

The program has acknowledged deficiencies within its QA processes and has recently paid a lot of attention to improving them. At the time of writing, the product program has just hired a QA specialist to take a look at the processes and give suggestions on how it could be carried out better. Thus, some of the aspects we're going to discuss in this section may already be under improvement.

Three of the five interviewees occasionally feel that the program may be focusing more on fast development than quality, sometimes even at the expense of quality. The QA process focuses mainly on developing automated unit tests for the backend, the UI for manual testing. Also, there currently are no clear guidelines on what things to consider while developing unit tests.

Sometimes there isn't enough time to resolve all of the issues identified during the release testing, especially if the problems are bigger. If the needed fixes won't make it to the release, there currently are no processes or tools to exclude the feature from the release, and ends up being released anyway. The fixes are then released as part of the following hotfixes.

Newly released features regularly contain defects of some kind, and it is starting to irritate some of the customers. According to one interviewee, the clients occasionally find bugs through ordinary, most straight-forward use of the features, without really trying to break the feature. Some defects are also being found from older features when used in a certain way or combination with some newer functions.

The program is releasing from 10 to 20 hotfixes each release to resolve the found defects, depending on the number of acute issues. Fixing issues in production is laborious and risky and cause additional interruptions to development (see 4.2.6). The support queue is steadily growing while responding and resolving issue tickets leeches valuable time. According to one interviewee, *"we do a lot of 'putting-out' of fires, rather than making sure that it's fire-proof"*.

4.2.3 Technical debt

During the development of some parts of the product, some serious short-cuts have been made. Short-cuts make it difficult to integrate new features on top of the existing ones and deteriorates the specifications if quick fixes are not properly documented. Because of this, the resulting technical debt tends to deteriorate the maintainability of the product and directly slows down the development speed.

Due to the high expectations (see 4.2.1), the developers often feel pressure to get the features completed quickly, encouraging and occasionally even forcing them to take short-cuts. In contrast, features that were developed with more flexible schedule tend to contain less technical debt: *"I could name features, that these contain a lot (of technical debt), and okay, here has been time, this has been done quite well"*, according to one of the interviewees.

Also, due to the lack of long-term plans, the features are often developed without considering how they will likely need to be changed in the future. Combined with a tight schedule, there isn't always enough time to change the existing functionality to properly integrate the new features and often leads to developers trying to make it work on top of the old functionality.

Resolving the existing technical debt would require allocating resources for it, which is away from new development. One of the interviewees fears that if not resolved, the technical debt may be ever-increasing, ultimately leading to having to re-develop the entire feature (see 4.2.4).

4.2.4 Rework

Having to do the same things multiple times is a direct cut on the overall efficiency. Taking the time to resolve technical debt (see 4.2.3) requires resource allocation, which is away from developing something new. Time is being wasted by sloppy specifications (see 4.2.7), resulting in features that won't solve the problem that they were intended to, and has to be corrected. Besides, due to lack of long-term plans, the existing features might need to be changed before integrating new functions for them to work seamlessly, or risk accumulating technical debt (see 4.2.3).

Having to do little fixes and changes here and there can also deteriorate the specifications (see 4.2.7), if not documented properly, which also takes time. Also, *"There is the risk that it might not be known anymore what had been done, what this covers. And something will likely be left off."*, according to one of the interviewees.

4.2.5 Predictability

A predictable development process and steady development speed are a prerequisite for accurate planning. Unfortunately, the current state of the development does not seem to be predictable enough for long-term planning.

The sprints are often overflowing with tasks, scheduling more of them than the development team can reliably develop. Tasks may get started, but not completed before the end of the sprint. Thus, the task gets pushed to the next sprint, with some of the work already done. This can cause some of the story points to be counted on the latter sprint, although some of them would effectively be accumulated during the former sprint.

Moreover, while the tasks are selected to the sprints in order of priority, they tend not to be developed in the same order: lower priority items might get completed during the sprint before higher quality items, and due to the sprints being overly packed, higher priority tasks might not even get started during the sprint.

Finally, the development speed is impacted by various interruptions, such as quick fixes and meetings (see 4.2.6).

4.2.6 Interruptions

Since the same team is responsible for the development, fixing acute bugs, responding to support issues, performance issues and customer feedback, the interruptions to work become inevitable. As responding to these other things take time away from the development, they have a direct impact on the development speed and should be aimed to minimize. Switching between tasks and contexts also increases the mental load and disturbs the focus of the developers: *"...acute support requests that interrupted, at random to an extent, the developer's focused development work"* was mentioned by one of the interviewees. Furthermore, due to their various sources and random occurrences, they tend to make reliable predictions more difficult (see 4.2.5).

The product program has already implemented a separate support shift that runs parallel with the regular sprints: for each sprint, two developers are off from the regular development, and instead focus on the emerging support issues and bugs. The aim has been to protect the rest of the team from interruptions caused by these issues, and the absence of two developers can be taken into account during the plannings. However, as only one developer is typically assigned for each task, support shifts can cause longer-term interruptions: for example, if a larger development task does not get finished during a sprint, and the developer of the task is assigned on a support shift on the next sprint, there will be at least one-sprint-long gap between its development. Not only can it take time to pick up from where the development was left off, it also inevitably stretches the time before the task is finished by a full length of a sprint.

Furthermore, there are quite many meetings within the program that cause interruptions to the workflow. To mitigate the impact of these interruptions, only some of the developers are involved: larger subprojects often have their dedicated developers that attend the related meetings. As one of the interviewees puts it, *"...we have a few different roles, like (developer name) and (developer name) for example, the purpose of which is to let other work while they are focusing on some other things."* While the nominating of dedicated developers provide consistency, one of the interviewees is afraid that it might also create bottlenecks (see 4.2.8): without the knowledge being shared with the rest of the developers, the subproject will be dependent on the dedicated developer.

Due to the monthly release cycle, it might also take a while before the new features reach the clients. Thus, also the feedback from the clients will take some time before reaching the developers, especially if the feature was completed early during the first sprint of the release cycle. Revisiting the old work and making adjustments based on the feedback may cause additional interruptions, and are typically difficult to predict (see 4.2.5).

4.2.7 Specification

The level of detail in the specifications varies a lot and is sometimes insufficient. While with one bigger client a set of acceptance tests are developed, request processing from smaller clients is not as systematic.

Changes and new features have been developed based on clients' requests without considering how they will affect the product as a whole. Due to a large amount of client-specific differences, some emerging from different legislations in different countries, a massive amount of settings have been developed, making it near impossible to consider all the possible combinations when testing (see 4.2.2).

Specification meetings and grooming sessions typically only involve one to two developers. This has been to protect the rest of the developers from interruptions caused by the meetings (see 4.2.6), but likely at the cost of the quality of the specification: involving more developers will likely cause the specification to be viewed from multiple points of views and lead to fewer oversights.

One of the interviewees feels that some of the features have been developed without properly paying attention to the clients' workflow. Proper level of background descriptions are also often omitted due to the time pressure (see 4.2.1). Poor fit between the features and the clients' workflow and omissions in the specifications cause often avoidable rework later (see 4.2.4), and sloppy specification will take more time when planning how to develop the given feature. Three of the five interviewees mentioned that the acceptance tests have worked very well, and the hope is that they will be taken into general use.

4.2.8 Bottlenecks

Regarding sustainability and efficiency, the program has some major developer bottlenecks: while few people know a lot about different aspects of the product, most others know well only the areas they have either developed themselves or worked closely with. Also, some areas have only one or two developers knowledgeable about how they work.

There are 14 developers within the team, with varying degrees of experience with the product. This inevitably causes differences in the performance between the individuals. While performance differences are natural, it can cause discrepancies in the average development speeds, making predictions a bit more difficult to make (see 4.2.5).

Currently, the program assigns the same developers to similar tasks, and typically only one developer for each. In one instance, the same developer was assigned to all tasks of a bigger enterprise client. Assigning the same developers for similar tasks has been justified by speed: developing the feature is faster when the developer is already familiar with it, and having more developers on the same task does not seem to linearly increase the development speed of the given task (see 4.2.1). *"I know it costs more if two developers start developing one (task), probably at least at that time, it might come back later, I don't know"*, according to one interviewee. This has caused knowledge on certain critical functions to stack on only a few people and depending on the function, sometimes even to one person. While knowing who to contact with regards to certain feature provides consistency, it creates a bottleneck: the developer may be busy with something else, or otherwise unavailable.

The program has not been able to focus on knowledge sharing enough. The company is scaling rapidly and the processes have not yet been adapted to accommodate proper knowledge sharing. Knowledge sharing has also been neglected to save some time for development, to better meet the expectations (see 4.2.1). Lack of documentation (see 4.2.9) also plays a role in the lack of knowledge sharing.

While the product program has some training sessions, especially among the sales and customer support -personnel, rather few sessions are being held for the developers. One interviewee mentions specifically the lack of training for the support shifts: training sessions would probably make the support shift less risky, and improve the developers' attitude towards it. *"The only training I've received from the product was from the person that now came asking me questions about how a certain feature works"*, one of the interviewees soon after starting to work in the program.

Support shift, while developed for trying to mitigate interruptions (see 4.2.6), has helped in reducing the bottlenecks: it forces the developers to learn about other parts of the product and also provides little more visibility to the clients' workflow. Understanding how the clients do their work is valuable when developing specifications (see 4.2.7). Unfortunately, the developers do not seem to enjoy the support shifts: some support issues are very specific, and it can be intimidating to delve into if one lacks experience in the given functionality. Not only is the probability of further issues higher, it will also take more time to fix than from someone more knowledgable.

Furthermore, code reviews have been identified to support knowledge sharing, by providing useful guidance and advice from other developers. However, the usefulness depends heavily on the reviewer, as developers seem to focus on different things when conducting the review.

Bottlenecks might lead to having to wait for specific people to be available before moving on with certain issues, which is very inconvenient if the issues are urgent. Sometimes the problem domain may be so large and complex that it simply is not feasible to familiarize oneself enough to make the changes safely, especially without proper documentation (see 4.2.9). Bottlenecks pose serious risks, especially during vacation periods.

4.2.9 Documentation

The documentation within the product program is not systematic and is often very lacking. Information is spread across multiple knowledge bases, not everything is being recorded, and the information that is recorded is not regularly updated, making it difficult to assess if the available information is still valid or not. Furthermore, a lot of it is in Finnish, while there are now quite many people within the program that do not know Finnish.

The information is currently being dumped into the first place that seems relevant enough, although it is not always the most appropriate place, increasing the entropy of the knowledge bases. Perhaps due to the high expectations (see 4.2.1), the program has developed a bit neglective attitude towards the documentation.

4.2.10 Sustainability

Overall, even though there is much room for improvement, the program's sustainability has recently improved. Three of the five interviewees feel that the product program is not dependent on any specific person, even though in the past it might have been the case. While there are areas where only a few, sometimes even only one person is knowledgable (see 4.2.8), the team has survived in situations where the 'required' people were not present. However, there are certain people that, if lost, would cause real stagnation for a while, depending on if those areas will experience problems after that.

Some of the interviewees also worry that the current company culture may not encourage sustainability. The program is among the few that have clear QA -processes in place (see 4.2.2): *"Few projects have automated testing, only those made with Django"*, according to one interviewee. The attitude towards documentation (see 4.2.9) may also play a role in driving the stacking of knowledge to specific people.

There seems to be a real need for enhanced knowledge sharing, as the product is massive and very complex: according to the interviewees, no-one within the program knows everything about the product. One of the interviewees has actively been studying the product and its features for roughly a year now and does not feel that they have become familiarized with everything.

4.2.11 Efficiency

Despite the high expectations, three of the five interviewees view the development process quite efficient, at least within the domain's standards. However, the factors discussed above have a considerable, negative impact on the development team's productivity. According to one interviewee *"We have a quite fast pace, I would say, ...but not everything is as much with the quality as they could be"* (See 4.2.2).

Additionally, emphasizing development speed (see 4.2.1) increases the threshold of refusing to do something critical that the developer does not know how to do. Due to insufficient documentation and scarce training sessions, there is no effective way to quickly familiarize oneself before agreeing to do it. This can easily cause further problems when trying to fix the initial one.

Regarding the promised delivery dates for some clients, occasionally the product program promises shorter development times than usual, especially if the client pays a premium for faster development. The premium price is seen justified, as developing the feature in a shorter time likely requires additional resources. However, because the program typically only assigns one developer for each task, there is no room for additional resources, greatly limiting the speed with which the task will be completed.

As discussed earlier, the company has been expanding rapidly, and the need for stricter processes is a relatively new issue. The processes are still finding their forms while the teams are adapting to the increased size of the company. Clear processes are essential for recruits to be integrated into the existing team, and becoming productive quickly.

4.3 Summary

After identifying the factors from the interviews, another round of analysis was conducted to study their relationships. Discussing one factor during the interview often led to the identification of another factor. These relationships, that were identified during the interviews, are summarized in figure 4.2.

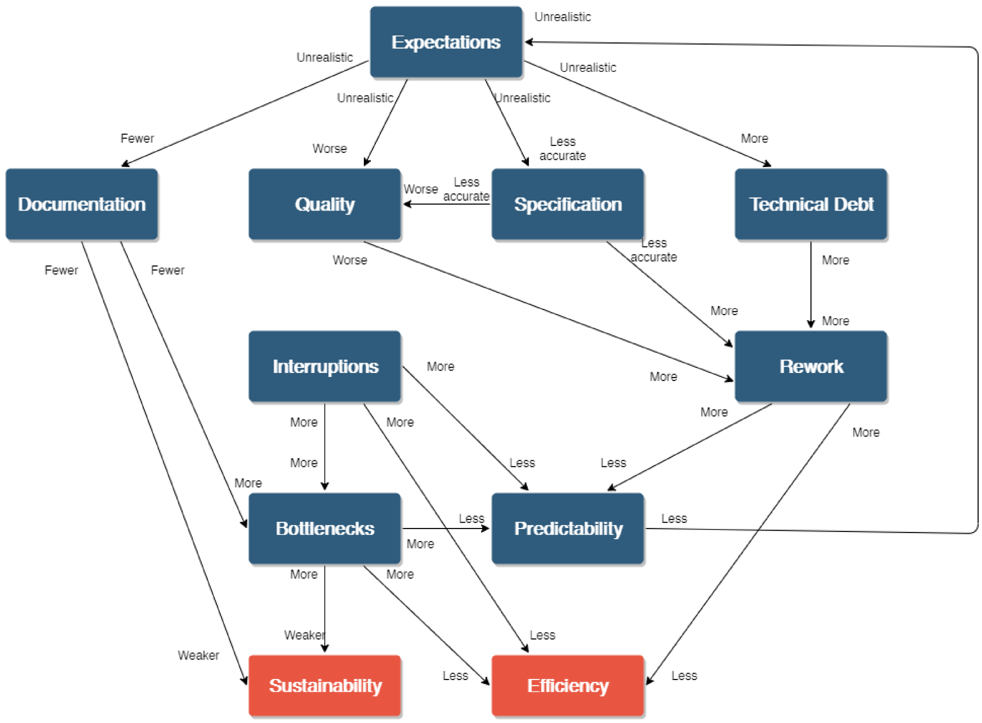

Figure 4.2: Factors affecting sustainability and effectiveness

It should be emphasized that only the relationships identified during the interviews, primarily by the interviewees, are depicted in the figure: some potential relationships may thus be omitted. For example, one could argue that quality and technical debt are tightly linked, but because such relationship was not brought up during the interview, no relationship between the two are presented in the figure.

The labels for the relationships were chosen to reflect the findings of the interviews. However, most of the relationships will likely work the other way as well: 'more technical debt→more rework' could probably also be depicted as 'less technical debt→less rework'.

As illustrated in the figure, the Expectations seemed to be the root cause for most of the other factors. Only interruptions seemed to be somewhat unaffected by the expectations, or at least no link between them was identified during the interviews.

However, one could argue that there is a link between the Expectations and Interruptions as well: the more customers the team needs to serve simultaneously, the more there will be technical tasks on top of the development work. These tasks include copying data between test- and production environments, which are typically handled by second-line supporters. Whenever there is too much work for the second-line to work on, these copying tasks sometimes spill to the rest of the developers, causing interruptions to the development work.

Interestingly, the only thing that seemed to affect the Expectations was Predictability: it's difficult to make realistic expectations if the work is highly unpredictable. Unrealistic expectations, in turn, seem to cause rushing, which has a tax on a lot of different factors: documentation is omitted, specifications are hastily made, and shortcuts are taken during the development. As a consequence, the quality gets worse, technical debt accumulates and a lot of effort goes into unnecessary rework. Furthermore, these issues make the predictability of the process even worse, completing a vicious cycle.

Chapter 5

Literature review

Based on the current state analysis, it was decided that the literature review would be directed at expectations and rework: how to set realistic expectations and how to reduce the amount of unnecessary rework. For a more detailed description of why these topics were chosen, refer to section 3.2.

In this chapter, the results of the literature review are presented. Findings related to expectations are described in section 5.1 and findings related to rework are described in section 5.2 Finally, results are summarized and the suggestions are presented in section 5.3.

5.1 Expectations and predictability

Each project has a mission, goals, which are being pursued, and the success of the project is assessed by the degree of which those goals have been achieved. Expectations, in turn, are tightly related to the defined goals and made agreements: for example, when the client is given a promise of a specific delivery date, the client will expect the delivery on the specified date. Likewise, when a software company makes a promise for a specific delivery date, they are expecting that the development team(s) will be able to make that delivery by the specified date.

To be able to give realistic promises, the software companies need to be aware of their capabilities: promising too little may dissatisfy the customers and make it difficult to win any contract bids, but promising too much will make it difficult, if not impossible to deliver the made promises. Thus, promises seem to be related to the companies' predictions: if a company makes a promise that the product will be delivered by a specified date, they are predicting that it will be deliverable by that date. In a way, the prediction is a precondition for expectations: promises are made based on predictions, and expectations stem from the made promises.

To study how expectations and predictability affect software development, it thus makes sense to study their origin. What kind of promises are being made? Which factors contribute to predictability? What are the effects of over/underestimation?

This section is divided into five parts: the first subsection (5.1.1) explores the origin of the expectations, and how 'success' is defined in software projects. The second subsection (5.1.2) investigates Effort Estimation: its different methods, how it relates to predictability, and its other effects on the software development process. The third subsection (5.1.3) is dedicated to discussing different measurements and metrics for monitoring the estimates and estimation processes. Fourth subsection (5.1.4) explores other relevant factors to be considered during effort estimations, and fifth subsection (5.1.5) presents the various effects of expectations and (un)realistic estimates. Finally, the sixth subsection (5.1.6) acknowledges some remaining limitations and challenges related to effort estimates.

5.1.1 Defining success

How is success defined in the software industry? When is a product considered a failure? Traditionally, the goals are related to functionality, schedule and budget: are the agreed features delivered on time and within the agreed cost limits (Coelho and Basu, 2012; Procaccino and Verner, 2006).

In their study, Procaccino and Verner (2006) studied the relevance of the above aspects with regards to defining success and found out that 'meeting requirements' proved to be the most relevant factor. Perhaps rather surprisingly, completing the project on time and completing it within budget did not seem to have much relevance when measuring success. As a possible explanation for their low relevance, the authors suggested that the budget and schedule considerations are merely estimates, and are often assumed to be flexible. Further, as one of their interviewees explains, "It depends on whether exceeding cost/dates was managed properly through change control, or mismanaged as part of unapproved overruns." For the highest-ranked, work-related factors, Procaccino and Verner (2006) found that "requirements of customer/users were met by the completed system", "the final system worked as intended" and "the project was delivered when needed by the customer/users (not necessarily within schedule)" were the top-three most relevant.It would seem that whether or not the product fulfills the customers'/users' needs is the most important factor when considering project success, accompanied by proper communication with the client. Cost and schedule overruns seem to have a smaller impact if they are handled properly. In any event, the definition of success seems to be tightly related to whether or not the project meets the expectations.

The size of the project seems to also have an impact on the likely outcomes of the project. Boehm and Turner (2003) found that, when measuring the projects by the number of people involved, the agile methods succeeded best on projects of 10 people or less, while traditional methods proved to work better with projects exceeding 100 people. For middle-sized projects, between 10 and 100 people, they suggest using a hybrid mix of agile and plan-driven methods. This would suggest that the projects might be best begun as agile projects, but elements of traditional software development should be incrementally integrated as the project size keeps increasing. The study, however, did not seem to make a difference between project and team size: would an agile project of 30 people still work well, if it was divided into three or more teams?

The importance of proper expectation management is highlighted in the literature. According to Valade (2008), "...the ROI of doing the proper expectation management upfront cannot be overstated". Likewise, according to Boehm and Turner (2003), "...the differences between successful and troubled software projects are most often the difference between good and bad expectations management."

Unrealistic promises may, at least partially, be attributed to the desire to please and avoiding confrontation (Boehm and Turner, 2003). It takes courage to say no to one's clients or managers, especially if they are being exceptionally pushy when trying to get more software for cheaper or in less time. Instead, the developers may agree to the demands and try to meet unrealistic expectations, potentially leading to unfulfilled promises.

Valade (2008) explains that the issues stemming from expectation management are alleviated if the customer understands the difficulties of software development and support, if even to a small degree. Acknowledging that unrealistic expectations do not benefit the company nor the client makes it easier to convince the customers and scale back their expectations (Boehm and Turner, 2003).

Heyer and Brereton (2008) also points out that one should be careful when using prototyping during client communications. According to them, it is easy to assume that a prototype is similar to the final, polished system. The clients may be disappointed with the quality of the prototype if it does not meet these expectations. Likewise, if the prototype is too polished, it could be easy to assume that the system is almost complete, making it difficult to justify longer development times or costs.

Agile software development methods seem to provide natural ways for managing expectations through close co-operation with the clients (Valade, 2008). Providing results in short iterations for the clients to see, and being able to predict how much can be achieved within each sprint provides a good starting point for managing software expectations (Boehm and Turner, 2003). When using prototypes in customer communication, it is advisable to ensure that the clients understand that they are only prototypes, for example by adding 'draft' -labels to them (Heyer and Brereton, 2008). One should also pay attention to the quality of the prototypes so that they do not alter the clients' expectations in an undesired manner.

Factors related to project failure have also been discussed in the literature. Lalsing et al. (2012), referring to the Standish 2009 Chaos report, point out that the failure rate of all IT projects is 24%, and 44% were challenged. In 2006, the same report indicated that nearly 60% of the software projects either failed or were challenged with cost or time overruns (Cao, 2008). These figures make it seem like the failure and the challenged rate is increasing: from roughly 60% in 2006 to 68% in 2009. However, these reports are over 10 years old, and thus, it should not be assumed to be similar today.

Lalsing et al. (2012) also investigated the success/failure rates of agile projects specificly and, referring to Scott W. Ambler's 2010 *IT project success survey*, found that 12% of agile projects were failures and 28% were challenged. Furthermore, they found that success rate was 83% for small agile teams, 70% for medium-sized teams, and 55% for large teams, confirming that the team size plays a role in project success and failure. Also, when comparing the combined failure and challenged rates of agile projects (40%) to those of traditional methods (68%), it seems that the agile projects fare better than traditional projects. However, this may not be fully attributed to the software development method, as the failure/challenged rate for large agile teams is close to that of the traditional methods (65%). It may be the size of the project that plays the biggest role.

Verner et al. (2008) examined the most frequent factors that contribute to project failure across 70 different case projects. They found that 93% of the failed projects were challenged because the delivery date impacted the development process and that 81% of the failed projects were underestimated. To summarize their findings, the "...failed projects have impossible estimation targets, with shortcuts taken, and good practices skipped". Verner et al. (2008) also points out, referring to Rober Glass' 2002 book *Facts and Fallacies of Software Engineering*, a relationship between project failure and poor estimates suggesting that estimation is generally done very badly, making the estimates more like wishes than realistic targets. These findings highlight the importance of realistic estimations, or predictions, to make realistic promises.

5.1.2 Effort estimation

If one can reliably predict how much work a certain feature will require, one can not only calculate how much money will be spent in its development but also when it will likely be ready for release. This information is important for clients when prioritizing features and making decisions: some features might be nice to have, but may not provide enough business value to justify the costs, and some features may be required sooner than others.

Britto et al. (2015) roughly defines effort estimation, referring to Emilia Mendes' 2007 book *Cost Estimation Techniques for Web Project*, as "... the process used to predict the effort needed to fulfill a given task." Techniques of size estimation and effort estimation are often used interchangeably, as effort estimates are derived from size estimates and velocity calculations, at least in agile contexts (Usman et al., 2014). Effort estimation is conducted to support project planning and tracking, in order to enable scheduling and calculation of costs to manage resources in an effective way (Britto et al., 2015; Raith et al., 2013). Accurate estimates make it more likely to reach the desired results within the defined time and cost constraints (Britto et al., 2015). Effort estimation requires some experience in estimating, along with access to some historical data to previously made effort estimates (Coelho and Basu, 2012).

5.1.2.1 Planning

In their article, Cao (2008) describe different levels of planning in an agile project, namely release plans and iteration plans.Iteration plans consider the tasks to be developed within a specified, typically 2 to 4-week-long period. The tasks are typically selected from a pool of estimated and prioritized tasks: the tasks with the highest priority are pulled to the iteration plan until the iteration is considered full. The agile team can estimate how many tasks they can complete in an iteration based on the estimates and their velocity (Javdani et al., 2013): if the average velocity of the team is 60 points, they can expect to be able to develop tasks with total estimate of 60 points.

The velocity can also be utilized in the release plans. If one estimates all items desired for a specific release, one can use the velocity to roughly calculate how many iterations are required for the items to be complete (Coelho and Basu, 2012; Hamouda, 2014; Javdani et al., 2013). Assuming, of course, that there won't be any changes to the requirements and/or prioritization between these iterations. Because iterations are typically of the same, fixed-length, one can also calculate the calendar time for the release (Coelho and Basu, 2012).

Javdani et al. (2013) also points out that velocity from previous projects can be utilized when scheduling a brand new project. However, this assumes that the team will remain the same and that the development items won't change much during the project.

Estimation in itself was pointed out to be a good way for developing a common understanding of what the development item is about. In their study, Tanveer et al. (2017) discovered that developing a common understanding among the team members turned out to be an even more important utility of effort estimating, rather than providing accurate effort estimates for scheduling. Convergence of the developers' impressions of the required effort might be a good indicator that everyone is utilizing the same set of information.

5.1.2.2 Methods

There are multiple different ways for going about the effort estimation. However, most of the techniques haven't been able to provide satisfactory results (Usman et al., 2014). Furthermore, Usman et al. (2014) points out that there is from little to no guidance in the literature as to which estimation technique or cost drivers to consider in a given context. Britto et al. (2015) also points out that, while specifically developed for co-located estimation, the same methods are being used in global software projects with little to no adaptions According to the authors, these methods are not readily applicable to global contexts.

The most dominant methods, according to the literature, seems to be expert-based assessments (Chua and Verner, 2010; Javdani et al., 2013; Tanveer et al., 2017; Usman et al., 2014). In a survey conducted by Raith et al. (2013), up to 80% of the companies used multiple expert-methods, such as Planning Poker and Delphi. Adoption and use of expert-based assessments are quick and cheap (Coelho and Basu, 2012), likely playing a role in its popularity. However, they require good domain knowledge from the participants (Coelho and Basu, 2012), and they do not objectively consider the potential impacts on existing features (Tanveer et al., 2017).The human judgment bias and the use of limited information decreases the prediction capability of these methods (Tanveer et al., 2017). As a result, estimates made by expert-based assessments tend to be too optimistic (Tanveer et al., 2017).

Fortunately, the literature suggests that the accuracy of estimates made by expert-based assessments can be improved. Coelho and Basu (2012) explains, referring to Peter Hill's 2010 book *Practical Software Project Estimation-A Toolkit for Estimating Software Development Effort and Duration*, that previous, already estimated and potentially completed items can be used as an analogy for estimating new items: a very similar task will likely require a very similar amount of effort to complete. While the software may have changed between the two tasks, it at least provides a starting point for further discussion. Tanveer et al. (2017) points out, referring to Usman et al.'s 2015 article *Effort estimation in agile software development: a survey on the state of the practice*, that combining the expert-based assessments with other estimation methods seem to increase the accuracy of the outputs. Accuracy can be further improved by making the estimations in groups rather than individually (Raith et al., 2013; Tamrakar and Jørgensen, 2012)

Coelho and Basu (2012) also points out that larger stories can be disaggregated into smaller stories, likely making the estimation easier. When the task's story point -value increases, so does the uncertainty (Zhong et al., 2011). This would suggest that to achieve accurate estimates, the estimated tasks should be relatively small. However, Coelho and Basu (2012) also warn that some aspects of the original story might be omitted in the disaggregation, and summing up the smaller parts to arrive at the final estimate might not be as straight forward as it seems. Cao (2008) also found that breaking the task into smaller subtasks does not fully prevent the development team from underestimating the required effort.

While there are other estimation methods, they are less common and do not seem to perform any better than the expert-based estimations. On the contrary, they may be more expensive to adopt and use, providing lesser value: for example, based on a survey conducted by Yang et al. (2008), model-based estimation methods provided from little to no value due to high costs of adoption. Another study by Moløkken-Østvold et al. (2004) found that using model-based estimation methods did not affect the accuracy of the estimates, but improved the manager's trust to the provided estimates. Increasing the trust without actually improving the reliability can potentially be dangerous, as the outputs are supposed to be used in scheduling and budgeting. Cao (2008) also found that model-based estimation methods are highly company-specific: the models require heavy adaptation to the specific company before providing expected results, increasing the cost of adoption.

Chua and Verner (2010) states, referring to multiple authors, that expert-based assessments seem to outperform other methods despite being susceptible to biases and limited information.

Among the expert-based assessments, Planning Poker is a very common estimation technique within agile projects (Raith et al., 2013). Tamrakar and Jørgensen (2012) describes it as "...a judgement-based, structured, group based estimation method with elements from the Delphi method". In a study conducted by Britto et al. (2015), Planning Poker is the most common estimation method in agile projects, being the choice of 72.55% of the companies.

It seems that, compared to mechanical averaged estimates, the results from Planning Poker tend to be more accurate and less optimistic (Raith et al., 2013). Also, being a group-based estimation technique, it tends to be more accurate than methods relying on individual estimates (Tamrakar and Jørgensen, 2012). However, Tamrakar and Jørgensen (2012) suggests that the participants make their initial estimates before further discussions to avoid anchoring effects (Tamrakar and Jørgensen, 2012). Moreover, Usman et al. (2014) found that Planning Poker yields more accurate estimates than unstructured group estimates if the team has relevant experience from the same or similar projects. A consensus should be sought between individual estimates, as it has been shown to provide less optimistic and more accurate estimates, rather than using simple averages (Usman et al., 2014)

Being an expert-based assessment, Planning Poker, too, is susceptible to over-optimism (Jørgensen, 2004; Mahnic, 2011). However, this optimism bias seems to recede to an extent as the participants become more experienced (Usman et al., 2014). The accuracy of the estimates seem to also increase with the experience of the participants (Usman et al., 2014). Javdani et al. (2013) also highlights the 'fun' factor of Planning Poker, as it might motivate the teams for participating actively in estimation practices.

Planning Poker uses a set of valued cards, each representing a suggested effort for the estimated task. The scale of values is often selected from famous, non-linear number sequences, like the Fibonacci sequence or powers of two. This makes the size of the functional effort increase more each step, reflecting the decrease of accuracy when the estimated item is larger (Raith et al., 2013; Tamrakar and Jørgensen, 2012). The used scale, both the values themselves and the amount of the cards, varies by the team.

However, a study conducted by Tamrakar and Jørgensen (2012) points out that one might want to carefully consider which scales to use in estimation. According to them, for example, the Fibonacci scale with the intervals of roughly 30% corresponds well with the average precision of the estimates (about +/-30%, according to Jørgensen (2004)), without too many gaps or overlaps. Also, in their experiment, the teams that used the Fibonacci scale systematically produced smaller estimates than the control-team using a linear scale. This disparity in their study, however, seems to decrease with more experienced estimators. Moreover, while the estimates were smaller, there is no evidence that they would be more accurate. After all, as discussed previously, the estimation methods tend to be prone to over-optimism rather than -pessimism. Tamrakar and Jørgensen (2012) warns that the use of the Fibonacci scale may strengthen the tendency towards over-optimism, if the estimation uncertainty is high or if the team is already prone to over-optimism.

Also, the length of the scale seems to have an impact on the estimations. In their experiment, (Tamrakar and Jørgensen, 2012) discovered that the estimates tend to get closer to the median value of the used scale, and is explained by "the central tendency of judgment". This meant that including bigger values to the end of the scale made the estimates made using that scale systematically larger. This phenomenon, however, was also found to be tied to the available information: when the uncertainty is higher, or information more limited, the estimates tended to weigh more towards the mean. Experienced estimators, equipped with appropriate background information about the item, will be less susceptible also to these effects.

5.1.3 Measurements

Agile ways of tracking progress and effectiveness differ from those of traditional methods (Javdani et al., 2013). Due to accepting unpredictable change requests in agile projects, most measuring practices in traditional software development models won't be readily applicable in agile projects (Javdani et al., 2013). Measuring the team's effectiveness in effort estimation is important to provide means for objective reflection (Tanveer et al., 2017). Without it, it would be difficult to adopt any improvements, as there won't be any objective evidence of their usefulness, essentially making it very difficult to continually improve (Tanveer et al., 2017). Even if the team is initially happy with the effort estimation accuracy, they won't become aware of its deterioration before it has already affected budgeting and schedules.

While many different aspects can be measured, it seems that their selection is not trivial. It becomes even more complicated if the team is being rewarded for improvements, identified by the measurements: for example, Jørgensen (2007) describes a case where the project leaders were rewarding the teams that had the most accurate effort estimates. While the estimates became very accurate, the teams' productivity fell and the effort estimates grew bigger. This was due to the team estimating the tasks big enough so that only a couple of tasks were selected to sprints, which they could easily complete and spend the remaining time for improving the product. While this made it very easy to reach accurate estimates, arguably making the development process predictable, it likely wasn't what the company had desired. This case illustrates well the 'you get what you measure' -effect of the measurements, and that the companies should be careful if they plan to issue rewards for improvements based on these measurements. Different measures may be required in different situations, and they must be selected to fulfill a clear purpose (Jørgensen, 2007).

Selecting the measurements and monitoring their development is only the first step. To facilitate continuous improvement, it is also important to identify the factors that cause the measurements to change. The change can be attributed to numerous reasons: estimation accuracy may have increased, the project's management may have become more efficient, the project is becoming less complex, etc. (Jørgensen, 2007). Having a couple sprints where the backlog items are much simpler than on average may appear in the measurements as improved estimation accuracy, but will likely return to its original level when more typical tasks are being scheduled again.

Raith et al. (2013) studied methods for identifying inaccurate estimates and came up with five measurements, utilized during the estimation process, that could aid in assessing the trustworthiness of the estimates. The suggested measures are accuracy, variance, duration, estimation tendency, and combined estimation tendency. If one or more of these measures signify issues, the more likely it is that the estimate is untrustworthy, and the project manager has a chance for further inspection (Raith et al., 2013). While multiple different ways of using these measures for identifying inaccurate estimates, and for proposing the final estimation values, were suggested in the paper, no empirical validation of their effectiveness was provided. The validation may have been planned for future research. Furthermore, these measures require software support for guidance in real-time, making their applicability to depend on the estimation tool used by the team.

These measures, along with some others, are discussed further in separate subsections.

5.1.3.1 Size metrics

Size metrics are used for quantifying the size of the estimated tasks. An estimation of the size could be considered as a prerequisite for effort estimation, although they are often used interchangeably.

The most common metrics for size seem to be use case points and story points and they are very common in combination with agile estimation techniques (Hamouda, 2014; Raith et al., 2013; Usman et al., 2014). Story point is a relative metric that can be assigned based on, for example, the required effort, complexity of the task, or the risk involved in the tasks development (Coelho and Basu, 2012; Hamouda, 2014). A task with two story points is twice as complex, risky, etc. than a task with only one story point. (Raith et al., 2013) According to Raith et al. (2013) (referring to Miranda (2001)) and Hamouda (2014), humans are better at perceiving differences in relative size, rather than in absolute size.

It is common to use a benchmark task for the definition of the story point (Hamouda, 2014; Zhong et al., 2011): a small, simple task is assigned one story point, and the estimation of other tasks is conducted by comparing the task to the benchmark. However, it is also easy to mistakingly associate the story points with how much time it takes to complete the task (Zhong et al., 2011). This is not advised, as in agile projects, any member of the team may end up developing the given task. There will inevitably be productivity differences between the team members, making it impossible to assign a fixed amount of time it will take to complete the task. This problem can be alleviated, however, if the whole team is developing all of the tasks together, but this is very difficult to achieve in larger teams.

Although the story point is the most frequently used size metric in agile projects, there are also other means. Referring to *ISO/IEC 20926: 2009* -standard, Coelho and Basu (2012) explains that while using the number of source lines of code (or SLOC) is arguably the simplest way of measuring the size, functionality does not correlate well with it: there are often numerous ways of achieving the same functionality, leading to situations where more functionality is provided with fewer lines of code.

In their article, Coelho and Basu (2012) also discuss the use of Function Point Analysis. According to them, function points are independent of the programming language, the used development method, and the platform, making them very adaptable for any context. This makes it also a pretty good measure for benchmarking across organizations, as the outputs ought to be objective: external inputs, external outputs, external inquiries, internal logic files and external interface files are all objectively quantified. Thus, it should not matter who is making the calculations, the results should remain the same. However, since the counting process cannot be automated, and have to be conducted for each task separately, making the use of function points very laborious. It is not obvious either that the required information for the calculations are readily available.

5.1.3.2 Velocity

Velocity is defined by summing the values (story points, for example) of completed user stories during an iteration, not to be mistaken with productivity (Coelho and Basu, 2012; Javdani et al., 2013). “A measured velocity of 15 means that the Scrum team realized a functional effort of 15 Story Points in the related sprint” (Raith et al., 2013). Due to being a combination of productivity and personnel placement (a team of this size can complete this many story points in this amount of calendar time), it can be used to predict when certain goals will be reached (Raith et al., 2013)

Although it could serve as a good measurement for the teams’ overall productivity, Javdani et al. (2013) warns that it might pose a threat to long-term productivity. Trying to improve the productivity figures in the short-term might make the development unsustainable. Javdani et al. (2013) explains, referring to Dave Nicolette's 2009 *Agile metrics*, that it should be considered an empirical observation, rather than a target to aim for.

For the velocity figure to be usable in the estimation, it should be calculated from several successive, fixed-length iterations, and use their average (Coelho and Basu, 2012; Javdani et al., 2013). According to Cao (2008), referring to Mike Cohn's 2005 book *Agile estimating and planning*, rapid adjustments to the process improve the effort estimation process, and the estimates become quite accurate after a couple iterations. Javdani et al. (2013) also reminds us that the velocity measurements rely on the consistency of the team: any changes in the team compositions will void the velocity measurements.

To provide value in effort estimation, the velocity needs to be rather stable: a very high variance in velocity entails very high unpredictability within the development process. Unstable velocity makes the estimation very difficult (Cao, 2008)

Cao (2008) describes an article *Embracing change: An XP experience report* 2003 by Fuqua et al. that found that the development teams commonly assume that their velocity is stable, even though the velocities seem to fluctuate with a 20% variation. Some of the developers might be unavailable for a couple of days, the tasks may be more or less complex than previously, and there may be unexpected technical issues in production that requires the developers' attention. Furthermore, some tasks may be left incomplete at the end of the iteration, spilling the related story points to the next iteration (Coelho and Basu, 2012). To counter this same effect, Mahnic (2011) advised their students to re-estimate the remaining work of the tasks that were spilled to consecutive iteration. While this won't completely solve the inaccuracy of the velocity calculation (the current sprint won't get any story points for the postponed task), it will reduce the variance. It also provides an opportunity to course-correct, if the required effort turned out to be much different from what was initially thought. Valade (2008) also realized during their case study that reserving a fixed amount of time each week for fixing unexpected errors made it easier to plan the iterations. If the fixed time stays the same each iteration, the velocity should not be impacted by unexpected issues.

In their article, Coelho and Basu (2012) present three different ways of estimating future velocity. In these presentations, the authors refer heavily to McDaid et al.'s 2006 article *Managing Uncertainty in Agile Release Planning.* The first technique involves the use of historical data: iterations executed before the last release, or even from previous projects. An average velocity is calculated and multiplied by 60% and 160% to get the likely upper and lower bounds: historical data may not properly reflect the current situation, and thus, requires margins for error. With these bounds, the team can create worst- and best-case scenarios for the next iteration, during its planning.

The second approach explained in the article is to execute a couple iterations and collect the velocity calculations. In addition to using the average velocity, Coelho and Basu (2012) also suggest that the range of the velocity values can be used as boundaries. This approach seems to be among the most common ways of predicting future velocities.

The last approach presented by Coelho and Basu (2012) is called forecasting. It begins by estimating how much time each team member has available for the next iteration. The sum of the available hours is used as guidance when selecting the tasks to the iteration: the work should not exceed the number of hours available for the iteration.

The three approaches have their specific strengths and weaknesses to be considered and are applicable in different situations. The first approach seeks to find an estimate from historical data, the second aims to discover the velocity through experience, and the third aims to guess a proper velocity value by looking at the available resources. It is not clear how the mapping from available hours to story points, use case points, etc. (unless using hours as the size metric) is achieved in the third, and the team would have to be somewhat experienced to make a realistic estimation. Then again, if the team is experienced, they may have historical data available, allowing the use of the first approach to use a more systematic approach. While it takes a couple of iterations to get the velocity figures using the second approach, it is likely the most accurate assessment of the team's velocity. The initial estimate may have to be done using the first or the third approach, depending on whether or not historical data is available, and then use the second approach for the remaining iterations.

5.1.3.3 Accuracy

In addition to knowing the velocity of the team, making reliable iteration plans requires that the effort estimates for individual items accurately represent the required effort. The literature review did not capture very many different methods for assessing the individual estimates, and even fewer that seem applicable in agile contexts.

Mean Magnitude of Relative Error (MMRE) and Magnitude of Relative Error (MRE) seem to be the most frequently used accuracy measures (Usman et al., 2014). The relative error (RE) is calculated by subtracting the estimated effort (E) from the actual effort (A), and dividing the result by

A: RE = (A-E)/A. MRE is simply the absolute value of the RE: MRE = abs(RE), and the MMRE the average of MRE's across all estimates: MMRE = (MRE1 + MRE2 + ... MREn) / n, where n is the number of estimates.

Making accurate estimates seem to be very difficult. Hamouda (2014) found that more than 40% of the requirements are mistakenly sized. Underestimating seems to be more common than overestimating (Cao, 2008). Estimating bugs seems to be even more difficult than stories: in their study, Cao (2008) found that average (MMRE) was 19% for stories and 28% for bugs.

The utilization of the MMRE and MRE does not seem to be straight forward though. The agile estimates give the tasks a story point, use case point, etc. value, which is the E of the calculation. However, the actual effort A is much trickier to asses, as the points do not have a systematic definition: how many points the task was?

One could measure the time it took to finish the task and develop a mapping between the points and the spent hours. Hamouda (2014) calls this the Productivity factor. However, it likely won't remain the same across all developers: the accuracy of the estimate would then be dependent on the developer that was selected to complete it. This issue is likely alleviated if the entire team works together on each task, but that is not easily accomplished, especially with larger teams.

Unless a way for assessing the actual effort is discovered, both MMRE and MRE calculations remain unusable. Until then, it might be that a simple hit-rate for planned story points and achieved story points will be the only applicable way of assessing the accuracy of the predictions. However, this only takes into consideration the accuracy of the iteration estimate, not the individual estimates for the items.

5.1.3.4 Variance

One of the measures for identifying inaccurate estimates, suggested by Raith et al. (2013), is to measure the variance between the individual estimates given by the team members. High variance across the different estimates suggests that the developers do not share a common understanding of what it is that needs to be done or might be operating on different assumptions. The variance should be aimed to be reduced by discussions, and when an acceptable threshold of variance is achieved, the most common estimate could be used.

Raith et al. (2013) explain in their article that since the increase in the estimation value already incorporates the increase in uncertainty, the probability for each value can be assumed to be the same, allowing the normalization of the scale by replacing the values by their positions. This makes the variance independent of the magnitude of the values, allowing a direct comparison between all estimates. Regardless, software support is required for the variance to be easily calculated on the fly, between each estimation round.

5.1.3.5 Estimation duration

Raith et al. (2013) suggest that the time required for making the estimate can indicate uncertainty. If the discussions drag on and it requires several estimation rounds to reach a proper consensus, there might be issues with the requirement. Differing assumptions on what needs to be done, and how the rest of the system is affected by the changes make an accurate evaluation of the task difficult.

Most of the tasks, according to Mike Cohn's 2006 book *Agile estimating and planning*, reach a sufficient consensus within two estimation rounds, making deviations easy to notice (Raith et al., 2013). Furthermore, if most of the experts come to the same conclusion, using the same assumptions and available information, the estimation is very likely to be correct (Hamouda, 2014).

In addition to the number of rounds required to reach consensus, one can measure the time spent on discussions. Raith et al. (2013) suggests that the timing should begin when the moderator begins to explain the estimation item and end when the final value is acquired. It might be a good idea to measure both, the spent time and required number of estimation rounds: the estimate may reach a consensus within two rounds, but might take a considerable amount of discussion before that. Another way would be to limit the discussions to a fixed amount of time, forcing a new estimation round after the time runs out. This allows us to focus solely on the estimation rounds: if more discussions are needed, the number of rounds increases.

Estimation duration, both the time spent on discussion and the number of estimation rounds, are easily measured even without software support. However, other measurements are required for assessing when sufficient consensus has been reached.

5.1.3.6 Estimation tendency

The final measurement suggested by Raith et al. (2013) for identifying unreliable estimates is the estimation tendency (ET). They define it as "...an individual factor concerning an individual estimator. It indicates whether this estimator tends to give oversized, undersized or suitable estimates". If one can asses the ET of the individual developers, the moderator can compensate when selecting the final values.

Following the same fashion as the variance, the scale is normalized by the values' positions (Raith et al., 2013). The ET is then calculated from the gaps between the final estimation value and the individual estimation value proposed by the individual estimator (Raith et al., 2013). Negative value indicates underestimation, a positive value indicates overestimation.

Furthermore, all of the ETs can be used to calculate their average, combined estimation tendency (CET) (Raith et al., 2013). It describes how much the team as a whole tends to over-/underestimate, and can be taken into account when planning the iterations.

It seems interesting that the tendency is calculated with regards to the final suggested estimation value. The tendency, in this case, actually describes how individual members estimate in comparison to the rest of the team, instead of in comparison to how much effort the task actually required. However, the real required effort is difficult to assess, and this decision might have been made because of that.

Calculating the tendency obviously requires software assistance. It might also be good to consider whether or not this information should be visible: if the team members know their estimation tendency, would they try to compensate for it? And how would such compensation affect the estimates?

5.1.3.7 Other tools and techniques

In their article, Javdani et al. (2013) present a couple of other techniques and tools for measuring and monitoring the overall progress of the project. Specifically, a burndown chart (BC), a cumulative flow diagram (CFD), and earned business value (EBV) will be discussed in this subsection.

BC, according to Mike Cohn's 2005 book *Agile Estimating and Planning*, is used as a representation of the amount of work that remains to be done (Javdani et al., 2013). It can be used in both the iteration and release levels. A comparison between the planned work and remaining work can help the teams in assessing whether or not all the planned tasks will get done, well before the end of the iteration/release. (Javdani et al., 2013). Alternative adaptation for the burndown chart is a burnup chart, which relies on the idea that an agile project's size remains an unknown, and thus, focuses on the incremental progress and completed work instead (Javdani et al., 2013)

CFD presents each individual work state and the amount of work associated with each given state (Javdani et al., 2013). Javdani et al. (2013) explain, referring to multiple authors, that it is useful in depicting work in progress (WIP), and thus, identifying bottlenecks within the process to achieve continuous improvement by eliminating them. Javdani et al. (2013) also note, referring to Mary and Tom Poppendieck's 2003 book *Lean Software Development: An Agile Toolkit*, that WIP figures can also be used in estimating: each item needs to pass through each process step, and if one knows how long it will typically take to pass through a given step, one can predict how long it takes to complete all items.

EBV is measured based on the ROI of each user story, tracking the valuable parts of the software (Javdani et al., 2013). It can also be used, according to Dan Rawsthorne's 2006 publication *Calculating earned business value for an agile project*, to assess how complete the product is, at least from the business perspective (Javdani et al., 2013). The ROI of each individual feature and user story are difficult to accurately predict, and thus, managers and product owners assign relative weights for them (Javdani et al., 2013): the higher the weight, the higher the ROI. Assessing the business value of each individual story seems useful in prioritization: having the team producing the most value all the time can be achieved through prioritization based on business value. However, the use of EBV requires the work to be quantifiable from the beginning, without much variation to the scope: "...if level of uncertainty related to the scope is high, it is completely useless" (Javdani et al., 2013) This is in line with the findings of a study by Tanveer et al. (2017): incorporating a single change to the requirements could have a big impact on other parts of the system, requiring the estimates to be adjusted for each sprint separately. Estimating everything at the beginning of the project may be wasted effort if the requirements can be expected to change.

5.1.4 Factors

There are also other factors identified during the literature review, that affect the accuracy of the made estimates, not directly related to the estimation process. While there is little to be done about them during the estimation process, they should be taken into account when making predictions. These factors, in no particular order, are briefly discussed in this subsection.

Procaccino and Verner (2006), referring to Steve McConnel's 1996 publication *Rapid development*, points out that the system quality has an impact on the required effort: according to them, high-quality code-base requires less time for fixing defects and re-testing the code. Refactoring during development will likely be less frequent, and unexpected incompatibilities with other features will likely be fewer. Thus, if the system quality leaves a lot to be desired, one will likely have to consider the implications when making estimates.

Due to tasks being rejected during the sprint review, work is transferred from one sprint to another. The task that was mostly done during the previous sprint, will be completed during the second, causing the story points of the task to be registered to the second sprint. This can make the velocity measurements unstable, and more difficult to use. To counter this, Mahnic (2011) created new tasks for addressing the short-comings, while accepting the original task. This will likely help in managing the variance in the velocity measures.

Verner et al. (2008) found, referring to Hihn and Habib-agahi (1991), that the project size affects the time estimations. According to Verner et al. (2008), smaller teams communicate more efficiently, making the problems more easy to deal with and making the estimation errors smaller. Britto et al. (2015) also point out that the communication overhead is higher for globally distributed teams, requiring extra effort that should be accounted for during estimation.

Jørgensen (2007) emphasizes the importance of defining the concept 'effort estimate' as a prerequisite for measuring estimation accuracy. They explain that without such definition, it will be difficult to assess whether the accuracy changes are caused by changes in estimation performance, or simply from different interpretations of what it means. Different interpretations include "planned effort", "budgeted effort", "most likely use of effort" and "the effort with a 50% probability of not exceeding" (Jørgensen, 2007) They also explain that the different interpretations cause problems mainly in expert judgment -based techniques, and that model-based approaches are less susceptible to it

Perhaps surprisingly, Cao (2008) found that "...the effort estimation is not improved over time as agile methods claimed". Because of this, other factors have a bigger impact on effort estimation. The factors they mention include resource availability, requirements change, insufficient refactoring, lack of unit tests, communication problems and distractions. While there may be a cap for how much the team can improve their estimation process, this will likely only affect mature teams: new and inexperienced teams will no doubt be able to improve from their initial level.

Usman et al. (2014) explain that the accuracy of the estimates will suffer without correct and appropriate cost drivers. In their study, the most common cost drivers include team's skill level (71%), communication model (65%), time zone difference (59%), team's prior experience (55%), cultural difference (45%) and software process model (41%). Lack of domain knowledge, lack of knowledge on the required technologies, lack of experience on the used effort estimation technique, lack of team cohesion, and misdocumented requirements were also reported in their study to have affected the effort estimates.

For the most relevant factors the developers consider during estimation, Tanveer et al. (2017) found that the developer implementation experience was considered the most relevant factor, along with the workloads of each developer. Perhaps surprisingly, they also found that assessing previously developed, similar backlog items, was not considered relevant by the developers. However, the authors provide a possible explanation for this: the case project did not store the estimates, and thus their relevance may have been overlooked entirely. Furthermore, Jørgensen (2007) suggests that the estimates should be based on a defensible strategy: estimating by an analogy to other items is a stronger argument than a gut-feeling. “For example, this story is like that story in the database (maybe a little more difficult, maybe a little less)”. (Hamouda, 2014)

In a study conducted by Britto et al. (2015), most of the projects are underestimated (54.90%). Acknowledging this the projects could try to compensate for this by making more careful estimates.

5.1.5 Effects

While the importance of accurate effort estimation was touched upon at the beginning of this chapter, this subsection will expand upon the causes for good and bad effort estimation, along with their immediate effects.

Procaccino and Verner (2006) warn about the effects of putting pressure to meet particular deadlines. They refer to Steve McConnel’s 1996 publication *Rapid development*, stating that excessive pressure can have an impact on system quality, as it encourages the team to take shortcuts. The pressure may be caused by a hastily prepared, or even politically-inspired, schedule/budget (Procaccino and Verner, 2006). As one of their examples, one company selected delivery dates at the beginning of the projects without regard to the actual time and resources required. According to the authors, this is a recipe for disaster, likely resulting in late deliveries, excessive costs, and frustrated development teams, managers and customers. Thus, it should first be assessed how much work there is to be done and how many resources there are available, to be able to select appropriate deadlines.

The importance of estimation before planning is also emphasized by Cao (2008): referring to Tarek Abdel-Hamid’s 1990 article *Investigating the cost/schedule trade-off in software development*, they explain that underestimates will cause schedule pressure, causing higher defect-rates and increased costs. Furthermore, they explain that overestimating will prevent the optimal use of resources, and makes it more difficult to win bids for contracts. According to Raith et al. (2013), referring to Nils Christian Haugen’s 2006 article *An Empirical Study of Using Planning Poker for User Story Estima-*

tion, "Accurate effort estimates are an essential factor for planning software projects to avoid budget overruns, delayed dates of delivery or lack of time, which often results in poor software quality". In their study, Sedano et al. (2017) also identified low team morale, rush mode, lack of empathy and poor technical decisions as effects of over-emphasizing the deadlines. The participants of this study felt that the clients should be able to fix either the scope, or the schedule: if there is a deadline, the features ready by that date are delivered, and if a set of features are requested, they are delivered when they are ready.

The impact of over-optimistic estimates is further illustrated by Jørgensen (2007). After the bid is completed, and it becomes evident that there is more work than initially anticipated, the managers have two options: face financial losses or act opportunistically to avoid the losses. The author mentions spending less effort on maintainability, usability, and robustness of the software as examples of the means to avoid the losses. While these means might make hitting the estimates more likely, improving the accuracy measures, the authors explain that it is not due to improved accuracy. This makes the accuracy measures redundant: even though the problem might have been with the accuracy of the estimates all along, it is no longer visible in the accuracy metrics.

5.1.6 Limitations

While short-term predictability in agile contexts is achieved through prioritization, long-term predictability seems to be a challenge. According to Agrawal et al. (2016), it is one of the agile methods' limitations, as the requirements are allowed to change, even long into the project. Coelho and Basu (2012) explain, referring to Lowell Lindstrom's and Ron Jeffrie's 2006 article *Extreme programming and agile software development methodologies*, that the requirements from the customers are influenced by changing technology and domain, budgets and political influence, to name a few. Because of this, constant re-prioritization and addition/removal of backlog items are inevitable, and this makes it very difficult to say what will be developed half-a-year from now. While this could be considered a limitation, it is considered one of the agile methods' strengths: being able to re-prioritize and reach to changes in value propositions will make it possible to reach for increased value (Boehm and Turner, 2003).

Since there typically aren't a detailed analysis or requirements specification at the beginning of the project, and they will likely evolve regardless, it is very difficult to make meaningful estimates in the beginning. According to Cao (2008), this characteristic makes underestimations almost a guarantee, and inevitably larger than in traditional approaches. Thus, one probably shouldn't try to make accurate estimates in the beginning, or at least base big decisions on those estimates. However, this can prove to be a challenge when competing for contracts.

Finally, Javdani et al. (2013) points out, referring to multiple authors, that there are no well-defined standards for effort estimation measurements. Due to differing project domains, company cultures, team sizes, etc., the companies will likely have to experiment on their own to find techniques and measurements that work for them.

As final suggestions for improving the estimation process, Verner et al. (2008) developed a guideline to "Invest in detailed planning in order to reduce estimation errors" (Verner et al., 2008). Furthermore, Tanveer et al. (2017) found that making factors to-be-considered explicit during estimation, the participants realized their relevance and impact on accuracy. Thus, it might be a good idea to create a list of the factors that ought to be considered during estimation and make it visible during the estimation process.

5.2 Rework

Referring to an article *The requirements change analysis for different level users* (2008) from Zhu et al., Khan et al. (2012) explains that a survey was conducted by the Standish group international, according to which 71% of projects have failed or are suspected, the main reason for project failures being requirement changes. Having to do the same work more than once causes a serious tax on the overall productivity, and should be avoided. However, due to changing requirements, technologies, markets, etc., some of the potential rework become unavoidable. Thus, it is important to identify what rework in unavoidable and what is unnecessary and focus on reducing the latter.

What are the causes of rework? When is it unavoidable or unnecessary? And finally, what could be done to reduce the unnecessary rework, as cost-efficiently as possible? In this section, the findings from the literature review related to rework will be discussed.

This section is broken down into multiple subsections, each approaching the topic of rework from their specific angle. The first subsection (5.2.1) discusses the different types of rework and their origins. Various measurements and metrics for monitoring the accumulation of rework are presented in the second subsection (5.2.2). Subsections *Changing requirements* (5.2.3), *Schedule pressure* (5.2.5), *Requirements Engineering* (5.2.7) and *Re-opening bugs* (5.2.8) discuss different ways of how required rework is introduced into the projects, and how they could be mitigated. And finally, subsections *Refactoring* (5.2.4), *Pair programming* (5.2.6), *Communication and feedback* (5.2.9) and *Quality assurance* (5.2.10) discuss specific means that could be utilized to counter unnecessary rework.

5.2.1 Types of rework

Rework, roughly, stands for work that has to be done again: changes to existing code may have to be made to support a new integration, the implemented feature does not satisfy the expectations, or the same function needs to be written in a different programming language, for example. According to Sedano et al. (2017), this includes rework caused by mistakes in writing acceptance criteria, mistakes in creating mockups, mistakes in writing tests and code, and technical debt introduced by taking shortcuts during development. They found that a high level of rework was often caused by, in addition to technical debt and defects, by unclear definitions of done, ambiguous mockups and rejected stories. They also make a distinction between mistakes that could have been avoided and mistakes that were possible to identify only in hindsight. One should not worry too much about mistakes that were 'unavoidable' at the time, but invest time in investigating ways to prevent it from happening again.

Javdani et al. (2013) and Cao et al. (2010) explain that embracing change is one of the agile principles, and each change can introduce re-work to the existing product. Flexibility to react to changes in the market gives the product an edge, and is one of the critical motivators for adopting agile methods (Cao et al., 2010). Thus, a certain amount of rework caused by change requests can be deemed unavoidable, and even necessary. However, if the potential benefits from reacting to the change do not outweigh the costs of required re-work, one should consider carefully whether or not to pursue them. Also, it is not always easy to assess the potential benefits, nor the costs, before the actual implementation.

One should also be careful not to let the amount of time spent on re-work to exceed the time spent on new, value-adding development: Verner et al. (2008) present in their article a case project that had failed due to rework costs exceeding the value the product could accumulate. They explain that inadequate planning and specifications had led to a myriad of change requests, which in turn led to cost and schedule overruns. According to the authors, "Rework is usually caused by requirements changes, so excessive rework costs are likely due to problems with the initial requirements". Change requests are most often related to error correction, changes in original requirements and changes to operational purposes (Chua and Verner, 2010).

Thus, while some of it being necessary, assessing an acceptable level of change requests seem necessary to identify when they are becoming excessive, which is an indicator that something needs to be done differently in the specification process. In Verner et al. (2008)'s case study of 70 failed projects, 73% of them had delivery decisions made without adequate requirements information as part of the critical factors. Furthermore, according to Jack Glazier's and Ronald Powell's 1992 book *Qualitative research in information management*, Chua and Verner (2010) explain that with each change request and modification of the existing functionality, the product becomes more and more complex and difficult to maintain, further increasing the costs.

The level of detail in requirement specifications is a bit unclear. As shown, too low-level specifications will likely result in misunderstandings, and as a consequence, more frequent change requests and rework. However, as pointed out by Mahnic (2011), "...the essence of the agile approach is not in writing detailed requirements specifications, but in acquiring missing details through communication with the Product Owner and end users". Especially tacit knowledge of the domain will be difficult to encapsulate in written specifications, and the best results will likely be achieved through continuous collaborations with the end-users: they can intervene immediately when the development is going in the wrong direction, potentially reducing the required rework later. Boehm and Turner (2003) points out that the agile methods emphasize the inclusion of a dedicated and collocated customer representative, available for the developers to ask for clarifications. However, such a commitment can be difficult to achieve, especially with a global product. Investing in good planning artifacts can alleviate the issue, making it possible to work with part-time representatives (Boehm and Turner, 2003).

According to Chua and Verner (2010), in addition to insufficient requirements, a high level of rework is often caused by the following: poor communication by the change requesters, lack of documentation to understand the change, unexpected problems, and lack of the maintainers' effort. While unexpected problems are, by definition, impossible to be prepared for, the rest of the causes seem avoidable, at least to an extent. Duka and Hribar (2010) claim that projects commonly spend at least 50% of their development time in rework, that could have either been completely avoided or fixed less expensively. According to them, most of the effort savings through process maturity, architecture and risk management come in fact from reducing the amount of avoidable rework. They propose that this is mostly attributed to the fact that defects and faults are cheaper to find and remove the earlier it occurs within the development process. For example, failing to correct a misunderstanding in a feature's specification, that is pointed out by the users after delivery, introduced massive amount of wasted effort: the design may have to be corrected, the functions need to be revised, and all automated unit tests need to be rewritten to accommodate the new, correct requirements. The cost of rework, according to Damm's and Lundberg's publication *Early and Cost-Effective Software Fault Detection*, could be decreased by up to 50% by identifying and correcting the faults earlier in the process (Duka and Hribar, 2010).

Finally, according to findings made by Sedano et al. (2017), trying to achieve 100% productivity may result in more rework, ironically making it more difficult to achieve full productivity potential. They found that engineers, when having to wait for something, would often try to find a way to spend this spare time in something productive, for example by trying to figure out ways how to reduce the future wait time. If not feasible, the engineers would work on something else. However, this context switching actually decreased productivity, and even increased the possibility of making mistakes, potentially introducing more rework Monsell (2003); Sedano et al. (2017). Instead of trying to fill in the wait-times with something productive, Sedano et al. (2017) suggest that taking a break may be a more productive way to spend it instead of working on other tasks.

5.2.2 Measures

Re-work can be defined as the man-hours spent in correcting flaws and defects (Javdani et al., 2013). With such a metric, one can compare the amount of time spent on rework rather than new development, offering a way to monitor that it does not spiral out of control (Javdani et al., 2013). By dividing the time spent in rework by the time spent on new development, one can calculate the rework level (Lalsing et al., 2012).

Lucassen et al. (2017) also discusses the use of the recidivism rate or the rate at which the user stories move backward in the development process: whenever the story moves from 'done' to 'in testing' or from 'in testing' to 'in progress', the recidivism rate goes up. The authors calculate the recidivism rate with the following formula: 200 * (Backward/(Forward + Backward)), resulting in a natural 0-100% range. They explain that this measure can be used in assessing the effectiveness of communication. As explained in the previous subsection, tight co-operation with the end-users would probably make course-correcting near-immediate, likely preventing the stories from moving backward in the process.

Although not specifically mentioned in the studied material, perhaps due to its simplicity, one should probably also keep track of the number of change requests and bugs in a given period, and monitor its development. Disproportional increase in the number of change requests and bugs related to the number of users is a sign of decreasing the quality of specifications and/or testing.

5.2.3 Changing requirements

According to Nurmuliani et al. (2004), the main causes for requirement changes are changes in customer requirements, improvements in the functionalities, change in managerial strategy, change in plans, omitted requirements, removing redundant features and making corrections to existing requirements. Unlike the reasons explained in previous subsections, these belong mainly to the 'necessary' -kinds of change requests: the development team has from few to no means of predicting changes in the customers' managerial strategy or changes in their workflow. Cao et al. (2010) and Chua and Verner (2010), referring to multiple authors, present a categorization of changes into adaptive, corrective, perfective and preventive changes. Corrective changes relate to fixing bugs and misunderstood requirements, while perfective changes are typically achieved through refactoring. Preventive changes, according to multiple authors, is undertaken to prevent future malfunctions or to improve the maintainability (Chua and Verner, 2010).

Adaptive changes are often treated as new requirements (Cao et al., 2010; Highsmith and Cockburn, 2001). However, even introducing new features or changes may require changes in the existing features, causing a potential need for rework. Due to the potential ripple-effects, proper investigations are required and estimates made for all required changes (Chua and Verner, 2010). Failing to take into account the full extent of the implications caused by the changes, the project runs a risk of unknowingly impacting the budget or scheduling (Lalsing et al., 2012).

Out of these four categories, adaptive and corrective changes seem to be the most frequent (Chua and Verner, 2010).

For properly managing the changing requirements, Khan et al. (2012) suggest that the companies need an effective Requirement Change Management (RCM) frameworks. Ineffective flow of changed information makes it very difficult to coordinate and serves as a vector for introducing more rework (Damian, 2007; Khan et al., 2012). Communicating the requirements and their changes becomes even more difficult in globally distributed projects, often caused by lack of common language, coordination systems, distance and strategic issues (Holmstrom et al., 2006; Khan et al., 2012).

5.2.4 Refactoring

Due to the principle of embracing change, the agile methods put little emphasis on up-front planning and designing. Instead, the design is constantly updated to be just at the acceptable level through refactoring (Cao et al., 2010). Cao et al. (2010) cites Martin Fowler's 1999 book *Refactoring: Improving the Design of Existing Code* to provide the following definition for refactoring: "change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". Continually and incrementally practicing refactoring, it is possible to incorporate frequent changes (Cao et al., 2010). According to Fowler, it can also be used to find certain bugs (Cao et al., 2010). Making the code easier to read likely makes it easier to spot mistakes in the code.

Due to the intensity at which refactoring is exercised in agile development, it is very critical and unique to those techniques (Cao et al., 2010). Refactoring seems to take from 6% to 39% of the total development effort, but since it is often exercised as part of new development, it is difficult to measure accurately (Cao et al., 2010). These percentages seem significant, and likely should be considered during the estimation of new features.

Cao et al. (2010) explain in their article that the required refactoring effort depends on three different factors: the quality goals, the starting status of the project and the coverage of unit tests. According to them, the unit tests provide a 'safety net', providing automatic validation of the made changes. Without them, considerably more time would be required to check that the system still works the way it did before. All developers may not even be fully aware of how the features should be working in each situation, making refactoring rather risky without proper test coverage. Cao et al. (2010) also notes that the relationship between the developers' confidence and test coverage is not linear: low coverage entails exceptionally low benefits.

According to Cao et al. (2010), developers rarely spend enough time on refactoring. This was confirmed by Cao et al. (2010)'s study, which also highlighted the effects of schedule pressure and pair programming on the refactoring effort. According to that study, developers will decrease their refactoring activity by about 50% if not paired with other developers. Furthermore, the study explains that when close to the deadline, the developers often choose not to refactor, as the benefits from it would not be realized before the deadline. According to Donald Robert's 1999 Ph.D. thesis *Practical analysis for refactoring*, developers will not perform refactoring while under schedule pressure (Cao et al., 2010). Furthermore, developers may not understand how refactoring is performed, might not care about the long-term benefits as they may not work within the project in the future, might view it as an overhead activity, and might be afraid of breaking the code (Opdyke, 1995).

Cao et al. (2010) explain that the need for refactoring is not always obvious. They also point out that programmers do not like the idea of modifying working code, due to the risk of breaking it. Developers may also get too focused on delivering new functionality, causing neglect for refactoring (Cao et al., 2010; Opdyke, 1995). As a result of these factors, according to Robert's thesis, the actual refactoring effort rarely meets with the effort that would be needed (Cao et al., 2010).

Neglecting the refactoring effort leads to a gap between the needed and actual refactoring effort, which can also be called "technical debt", according to Ward Cunningham's 1992 article *The Wycash portfolio management system* (Cao et al., 2010). According to Cao et al. (2010), this debt leads to unclean code, which further leads to corrupt designs. They explain that if the design is not appropriately updated to encompass the new requirements, the system becomes increasingly corrupt and brittle, eventually leading to a "major refactoring" to be able to progress. The longer the wait, the larger the required effort, as more and more features will be involved (Cao et al., 2010).

Cao et al. (2010) explain that the more the refactoring is neglected, the faster the quality of the system will decay. This results in reduced productivity, which in turn increases the schedule pressure. The schedule pressure, as they explained, makes the developers spend even less time on refactoring to dedicate more time on the development, which causes even more deterioration of the design. The result is a vicious reinforcing loop. The deterioration of the design makes the system more difficult to understand, and as a result, harder to incorporate changes, easier to make mistakes, and harder to detect defects and safely remove them (Cao et al., 2010). While short iterations and frequent releases might reduce the magnitude of changes that are incorporated into the system, the cost of these changes can increase rapidly if the design is not cleaned regularly. (Cao et al., 2010)

In their article, Leau et al. (2012) explain that larger systems may require even more refactoring than smaller systems. They explain that if the system has multiple user and client groups, there will likely be more change requests. Furthermore, these change requests may be conflicting, causing various incompatibility and integrity issues, further speeding up the deterioration of the software. They also discuss two dependency issues, rigidity, and mobility: the former referring to a situation where the change implies multiple other changes in other modules, and the latter to the inability of encapsulating components that can be reused, due to too much effort or risk. These dependency issues will require refactoring to correct, and speed up the development. Sedano et al. (2017) also points out that technical debt increases the number of things the developers need to remember and be aware of during development, increasing their cognitive load. Examples of these would be test cases that always fail, functions that are never used, and which areas should be inspected when making changes to a given area.

In their study, Cao et al. (2010) created a simulation that they used to estimate the effects of refactoring effort and its effect on the overall development. They found that the higher the level of refactoring, the lower was the overall development cost. However, as projects almost always suffer from insufficient refactoring, the major refactoring becomes unavoidable, and it should be planned for in the release plans. It should be kept in mind that this was only a simulation, and has was not evaluated in real settings.

5.2.5 Schedule pressure

To alleviate the effects of schedule pressure, Cao et al. (2010) provide a way of adjusting the scope. They explain that if the project has a fixed completion date and fixed resources, the scope needs to be adjusted by dropping stories or tasks if the project falls behind schedule. For adjusting the scope, they suggest that the functionalities are grouped into two categories: must-have and nice-to-have features. Must-haves would be features that the system cannot exist without, the core functions, and nice-to-haves would be conveniences and features that help the system in the competition, something that can be developed later. If the schedule pressure becomes too high, the nice-to-have features are dropped and re-scheduled for later. If the pressure remains, the stories can be decomposed into smaller tasks, and the same would be done to them: only critical tasks are developed, and the others are re-scheduled for later.

It may also be the case that the scope contains only must-have features, in which case the scope adjustment may not be feasible. In such a situation, the schedule needs to be adjusted, or the project needs to get a hold of more resources. However, the former is preferred, as including more developers late into the project has been seen to cause issues, and would increase the costs of development.

5.2.6 Pair programming

As discussed in subsection 5.2.4, pair programming can increase the degree to which the developers exercise refactoring. Also, according to Abdel-Hamid and Madnick's 1991 book *Software Project Dynamics: An Integrated Approach*, collaboration seems to improve the developers' performance and their enjoyment of solving problems (Cao et al., 2010). Pair programming was found to yield a better product with higher quality in less time (Cao et al., 2010; Williams et al., 2000). However, it also increases the labor costs, due to allocating two developers to the same tasks (Cao et al., 2010).

According to Cao et al. (2010), pair programming consumes about 30% more resources than programming alone. While perhaps more costly, they also found it to decrease the number of defects by more than 50%, and to increase the likelihood of engaging in unit testing. According to them, the coverage of unit testing decreases when the developers work individually.

Perhaps surprisingly, even though consuming more resources, pair programming was found to deliver more tasks than through developing individually (Cao et al., 2010). Furthermore, according to Cao et al. (2010), the cost per task delivered was 10% higher with no pair programming. They also point out that the cost of rework without pair programming is 36% higher, resulting from the lower defect rate with pair programming. Finally, the cost of making changes was found to be 10% higher and the cost of refactoring to be 12% higher in individual development. Because of higher engagement in refactoring in pair programming, the accumulation of technical debt is lower, and the cost of change increases more slowly.

These effects are also visible on student teams: Mahnic (2011) found in their study that teams that established good communication and team-work performed far better than their peers who acted as groups of individuals. Increasing the collaboration, thus, seems to increase the overall performance, while the immediate effects may show that it is a bit more expensive.

5.2.7 Requirements engineering

Even if one develops everything precisely according to the requirements, the project may end up a failure if the requirements themselves weren't adequate. According to Suma and Nair (2012), 50% to 60% of total defects occur during the requirement elicitation, followed by 15% to 30% in the design phase. Insufficient requirements that were noticed only after release cause a massive amount of rework: not only does one have to change the requirement, they also have to update the design, source code and test cases to accommodate the updated requirement. Thus, getting the requirements right is of utmost importance in preventing unnecessary rework.

Typical way of capturing requirements in agile contexts seem to be user stories, using semi-structured natural language (Lucassen et al., 2017). The structure often follows a certain template, for example "As a (role), I want (goal), so that (benefit)" (Lucassen et al., 2017). According to Mike Cohn's 2004 book *User stories applied: For agile software development*, the stories aim to capture and communicate the features to be implemented to better support the developers to produce software that the stakeholders want (Lucassen et al., 2017).

According to Duka and Hribar (2010), many of the faults originate from the early development phases, which suggests that design and specifications have not received appropriate attention. The use of user stories does not seem to have matured yet, as in their previous work Lucassen et al. (2016) found that 56% of the user stories contain easily preventable syntactic defects. Furthermore, Lucassen et al. (2017) explains that there is a relationship, albeit a weak one, between the quality of the requirements specification and the project outcome. Similarly, Knauss et al. (2009) found that student projects' success relates to the quality of the specifications they produce, to the point where the authors were able to define a threshold that could predict the risk of failure.

Getting the requirements engineering right is very important, as poor specifications are a massive vector for introducing unnecessary rework. In their article of software development waste, Sedano et al. (2017) explain that incomplete specifications may omit important functions, which need to be added later. This can be negated through unambiguous acceptance criteria and mock-ups. They point out that some stories may describe features that no-one needs, wants, or ends up using, which is an obvious waste of resources. They also found this to affect team morale, code ownership, and customer satisfaction. While building wrong features is not directly related to rework, the existing functions may still complicate the further development of related features, and thus cause more refactoring. Finally, Sedano et al. (2017) note that the stories should be simple: overcomplicated, long, unclear or full of pointers to necessary information elsewhere increase the cognitive load and overburden the working memory. As discussed earlier, this makes the developers more susceptible to making mistakes and to introduce defects, causing more rework effort.

5.2.7.1 Quality of the User Stories

In their study, Lucassen et al. (2017) present a way of measuring the intrinsic quality of the user stories, using Quality User Story (QUS) framework and Automatic Quality User Story Artisan (AQUSA) linguistic tool. The authors used five quality criteria: the stories need to be well-formed, atomic, minimal, uniform and unique, as in their previous work ((Lucassen et al., 2016)). These criteria were selected from a larger set of criteria provided by QUS, as they were the ones that could be automatically checked.

In their study, they used a Grimm method, formulated by themselves, to see how it affects the participants' user story quality. The results are compared from before and after training, both phases lasting for two months. The defects in this context relate to user story defects, i.e. defects related to the five quality criteria explained above, instead of actual software defects.

The study found that the defects per user story were reduced from 38.3% to 47.5% for two case companies, but were increased by 4.5% for the third, providing a considerable overall decrease in defects in a rather short time. Despite this, the participant's perception of the quality did not significantly change, nor did it seem to result in a meaningful, lasting process change. However, the participants did report to have become more aware of the relevance of capturing the user stories more systematically. The authors suspect that the criteria that are automatically collectible may focus on more trivial issues.

However, even though the technical quality of the resulting software did not have a significant change, the authors found that the goals of the user stories were clarified, and they made the development of the features easier and with less rework. The participants also reported that the frequency and effectiveness of the conversations around the user stories had been improved by the training. The authors also point out that the case companies had had multiple years of experience in writing user stories, which may at least partially explain the lack of perceived benefits.

5.2.8 Re-opening bugs

Because defects require additional work to resolve them, defects that keep re-appearing cause even more additional work. In their study, Shihab et al. (2010) examined re-opening bugs in an attempt to predict which defects will likely be re-opened in the future.

The authors explain that defects can be re-opened for a variety of reasons: for example, the initial fix may not have fully fixed the defect, or the ticket was closed due to not being able to reproduce it. Identifying defect-reports that will be re-opened in the future is important, as the authors found that it takes twice as long to fix a bug that keeps re-appearing than those that do not. Not only do they consume developers' time, the authors also note that re-opened bugs may deteriorate the users' trust in the quality of the product, negatively impacting the user experience.

In their study, they found that the bug description and comment text, the initial fix -time and the component in which the bug manifested were the best indicators for whether or not the ticket will be re-opened. In the work habits dimension, Shihab et al. (2010) found the month and time of the day the bug was closed to be the most important factors. For bug report dimension, they found the comment text -content to be the best predictors for re-opening, and was deemed the best performing predictor out of all studied dimensions: it achieved a precision of 62.9% and recall of 82.5% for re-opened defects, and precision of 96.5% and recall of 90.7% of not re-opened defects. For example, words such as “control”, “background”, “debugging”, “breakpoint”, “blocked” and “platforms” were associated with re-opened bugs, while “verified”, “duplicate”, “screenshot”, “important”, “testing”, and “warning” were associated with bugs that won’t be re-opened. However, it is hard to believe that the specific words would be relevant: certain words, such as “background” could be used as a part of the report template. Furthermore, the word “duplicate” would indicate that the ticket was automatically closed, even though the original ticket may be re-opening. Words such as “verified” and “screenshot” would indicate that the report contained enough information for reproducing the defect, but could also be contained in the contrary event: ‘Could not be verified’, or ‘Can you provide a screenshot?’ without ever acquiring one. Even if specific words may contain some predictive power, it will likely depend on the case company.

For the bug fix dimension, they found the most important factors to be the time it took to fix the bug and how many files needed to be changed in the fix as the most important predictors. And finally, in the people dimension, the most important factors were the name of the reporter and the name of the fixer. They explain that some of the reporters may not include all the relevant information, and the fixers will likely have a varying degree of experience. However, using the report dimension alone already produced a fairly accurate prediction model: using only one dimension makes the model easier to adopt, diminishing the relevance of the other three dimensions.

5.2.9 Communication and feedback

Communication and feedback play an important role in software development projects. In fact, according to Suma and Nair (2012), lack of communication, along with lack of training, oversight, lack of project methodology and inappropriate planning, are the most common root causes for defects. Lack of communication and fast and frequent feedback make the project susceptible to late change requests and thus, an increased amount of rework.

Lalsing et al. (2012) found that due to ineffective communication, namely forgetting to pass on information to the developers, some requirements may be improperly implemented, causing change requests and rework. Also, they found that changes to the requirements may not always be properly documented, causing a mismatch between the information used by the developers and testers: in their case, the developers would operate with the information provided by the business analysts, while the testers examined the conformance to the specification. Discrepancy between the information acquired verbally and what was documented in the specifications caused several defect reports.

Leau et al. (2012) also emphasize the importance of adequate domain knowledge, as it makes it easier to communicate with the customers and users. They explain that a team without such knowledge likely won't be able to produce a valuable product for the customers, which in turn negatively affects the customer trust with the team. Furthermore, they note that a lack of domain knowledge might make it difficult to understand what is required by the customers, likely resulting in modules that contain unwanted features. According to Suma and Nair (2012), from 50% to 75% of the defects are due to human actions, highlighting the need for training and education in the product and domain-specific knowledge. The authors suggest that the team can raise their understanding of the domain through training or by inviting domain experts to share their knowledge.

Leau et al. (2012) suggest that the features be left unpolished, and should not be fully integrated until it can be confirmed that the customers do not have additional requirements for the given feature. They explain that it is easier to include additional requirements to features that have not yet been fully integrated, as they will have fewer ripple-effects and thus, require less rework to include. Only integrating after a sufficient number of rounds of feedback has passed may make it less likely that there will be further changes to the module.

5.2.9.1 Customer involvement

Cao et al. (2010) explains, referring to multiple authors, that the agile development relies heavily on direct face-to-face communication between the customers and developers, and some methods even insist on having an on-site customer readily available to represent the needs of the customers. According to Kent Beck's 2000 book, *Extreme Programming Explained: Embrace Change*, on-site customers would also provide clarity in unclear situations, develop acceptance tests for the specified features, verify that the acceptance tests are run correctly, and participate in the planning meetings (Cao et al., 2010). According to Barry Boehm's and Richard Turner's 2004 book *Balancing agility and discipline: A guide for the perplexed, portable documents*, a good on-site customer manifests the following characteristics: they are collaborative, representative, authorized, committed and knowledgeable (also referred to as CRACK) (Cao et al., 2010).

According to Cao et al. (2010), having a tight customer involvement has several benefits, and overlooking the customer involvement could result in certain challenges. They explain that higher customer involvement tends to result in higher customer trust, which in turn has an impact on the quality and speed of the feedback for the development team. On the other hand, delayed feedback may increase the costs, as reacting to changes occurs later in the development and requires a larger amount of rework to achieve. Furthermore, they explain that the increased costs are reflected in the performance of the team, and the reduced performance or challenges in the implemented changes negatively affect customer satisfaction and trust. The customer involvement, thus, seems to create a reinforcing loop, making it even more relevant. Delayed feedback has a significant impact on the project outcomes (Cao et al., 2010; Duka and Hribar, 2010).

As discussed in the previous section, expectations are related to the scope and schedule: schedule pressure may require adjustments to the scope, or potentially risking the quality through omitting important quality assurance steps. Cao et al. (2010) explain that the unwillingness of the customers to make scope adjustments will increase the schedule pressure and that this willingness depends on customer trust.

However, acquiring an on-site customer representative may be challenging, and with global products with numerous customers, it may even be impossible. In such situations, the companies will likely have to look for other alternatives for increasing customer involvement.

5.2.10 Quality Assurance

Software testing is performed to identify defects to allow their resolving before they slip into the production environment. As explained previously, the earlier the defects are discovered, the less they will cost to correct. Preventing the defects from entering the production environments also protects customer trust (see 5.2.9.1).

Suma and Nair (2012), referring to numerous authors, explain that the testing can only detect defects, but cannot prevent them. They also point out, along with Duka and Hribar (2010), that it is the slowest technique for detecting defects, and that it is the last opportunity to try detecting them before they enter the production. According to Duka and Hribar (2010), the test at the end is merely a confirmation whether or not the feature adheres to the requirements: tests have no added benefits if the requirements themselves are inaccurate. In Lalsing et al. (2012)'s study, releases were delayed due to a high amount of rework, caused by identifying issues too late.

As explained previously (see 5.2.4), proper unit tests provide a safety net for refactoring, increasing the degree at which the developers engage in that activity. Furthermore, unit testing is supported by pair programming (see 5.2.6) through increased discipline.

5.2.10.1 Measures

For measuring and monitoring the effectiveness of the QA practices, Duka and Hribar (2010) propose that Faults Slip Through (FST) could be utilized. According to them, FST considers the phases during which each defect is discovered, and in which phase it 'should' have been discovered, or when it would have been the most cost-efficient to remove. As a general rule, they state that the earlier the defects are discovered, the cheaper they are to resolve, even though there are some exceptions. For example, a defect in the specification would be cheapest to correct after the review of the specification, as then it only causes rework on the specification. If the story had already been pushed to release, not only would one have to correct the specification, the design may have to be reworked, the source code needs to be altered, the automated test cases need to be changed and the user documentation updated. To determine which phases should detect which kind of defects, the test strategy, test processes and the overall development process need to be explicit (Duka and Hribar, 2010)

As potential benefits acquired through utilizing FST measuring, Duka and Hribar (2010) mentions fewer stopping faults, less redundant testing, less process variation, avoidance of doing same mistakes, improvement of product quality, and earlier and cheaper defect removal. Furthermore, they explain that it facilitates continuous improvement by providing feedback to the different actors and teams. If defects are identified to be slipping through a specific phase, it is an indication that something needs to be changed in that phase.

To take FST measurements into use, Duka and Hribar (2010) suggest a couple of new columns to be added to the trouble reports. The columns are 'Where the fault was detected' (DIDDET), 'Where the fault was supposed to be detected' (SHODET), 'Where the fault was introduced' (INTROD), 'Was there a test case in SHODET phase' (TC) and a 'Free text field for additional information' (INFO). They also mention that the FST can be measured in two different ways: as a fault slippage from a phase, and fault slippage to a phase. Furthermore, it can be used to identify phases with low slippage and high slippage. Duka and Hribar (2010) explain that phases with low slippages can be seen as good practices, whereas a root cause analysis should be executed for high slippage phases to identify the source of the slippage problem. Finally, the authors advise against the use of FST as a benchmarking measure, as the organizations and products differ in terms of definitions and processes.

As a part of testing measurements, one could also keep count on rejected stories. Sedano et al. (2017) define rejected stories as "stories that a product manager rejects delivered work because the implementation does not satisfy the acceptance criteria", and explain that they require rework to fix them. Monitoring how many stories get rejected during the reviews could be utilized as a means to track the testing performance.

5.2.10.2 Inspections

In their article, Suma and Nair (2012) discuss various quality assurance methods along with their strengths and weaknesses. However, they spend the most effort in explaining inspections. According to the authors, inspection examines the human at the early stages of software development to detect defects. They also point out that it is the most effective quality assurance technique.

The authors explain that by implementing the inspection technique provides a reduction of defects in the early phases of software development, thus reducing the overall development cost. They state that this, in turn, increases productivity and customer satisfaction. Their case study of 5 projects found that the companies that adopted defect prevention strategies (namely inspections) enhanced their product quality while their cost of quality reduced. Also, they found that with increased inspection time, the testing time decreased. However, they also point out that investment in inspection is initially high. Ultimately, the authors claim that it is possible to achieve a 99% defect-free product through implementing inspections, taking up from 10% to 15% of the total project time.

The inspections are conducted in several inspection rounds. According to the authors, the first-round targets the Product Requirement Definition, uncovering defects in it and providing a list of comments as its output. The assumptions made during the requirements definition are examined and validated in the second round. Suma and Nair (2012) explain that wrong assumptions may themselves lead to defects.

Inspections conducted during the design phase target the high-level and low-level designs. The authors explain that the assumptions made concerning interactions between the subsystems and other dependency factors are scrutinized. The outcome is the identification of flaws caused by a lack of clarity in the design. Finally, according to the authors, inspecting the test cases ensures that the code generation does not introduce defects. This, however, would require developing the test cases before the development.

The inspections in Suma and Nair (2012)'s paper are conducted in small groups instead of having one reviewer.

5.3 Suggestions

In this subsection, the findings from the literature review are summarized and formulated into suggestions for the case company's development team. While there will be overlap between the two topics, suggestions related to expectations and rework are discussed separately: for example, incorporating additional points for refactoring during estimation is an Expectations-category suggestion, but also (perhaps even primarily) affects Rework. The case company's current practices are considered during the suggestion generation.

5.3.1 Expectations

The first suggestion is related to measurements, and primarily serves this study and its validation phase: the impact of the implemented suggestions won't be quantifiable without appropriate measurements. However, this also allows the development team to quantify the same impact, along with any future changes. Thus, the measurements should facilitate the continuous improvement of the development team's processes.

It is suggested that the development team keep more diligent track of their velocity and user story hit-rate. A steady velocity should provide a solid argument in negotiating the scopes for new sprints, and the hit-rate should provide feedback on how accurately the team can estimate the workloads. The development team could also consider keeping track of the variance, estimation time and how many estimation rounds it required to reach consensus for each development item to assess the trustworthiness of the estimates.

The second suggestion is quite a big change. Currently, the sprints and their scopes are first planned by the stakeholders, after which their contents are estimated. Regardless of the estimation outcomes, barely any adjustments to the scope are made during the planning. Thus, the estimates themselves do not seem to have much of a purpose, if not to attempt to calculate the team's velocity.

It is suggested that this process is reversed: the stories are first estimated, and only afterward considered which of them will fit into the sprint. There probably should be a separate 'grooming' meeting where the future, most prioritized stories are estimated in advance. Then, at the planning meeting, it is decided which of the most prioritized stories will fit into the sprint, using the velocity as guidance. This would also free up the planning meeting for actual planning, instead of mere estimation.

The third suggestion is related to the estimation process. While the team has decided on two benchmark stories for certain numbers of story points, they are not readily accessible during the estimation. Furthermore, few arguments are ever heard that would compare the estimated stories to these benchmarks. It is unclear how each developer base their judgment on how many story points a certain story is. Expressions like "So you're going to do this fast?" would further suggest that some of the developers associate the story points with time instead of effort.

It is suggested that the estimation criteria are made explicit and the process more systematic. Benchmark stories could probably be updated, and they should be kept visible during the estimation meetings. Alternatively, one could develop a systematic model of what the estimation considers: a new model, a new view, and a new template is X amount of story points, effortful testing adds Y amount of story points, etc. Having a more systematic approach could potentially alleviate the effects of human judgment bias. Basing the judgment on something objective is more defensible, and will also facilitate the planning of the implementation. Making the factors to-be-considered visible was found to increase the degree to which developers considered them during estimation. This will also allow the verification after development, as one can then review how much effort was required, and translate that into story points. If possible, also make the previous stories that were estimated with a similar amount of story points accessible during the estimation meetings.

The fourth suggestion is related to deadlines. The material shows that schedule pressure tends to deteriorate the product quality, which in turn makes it more difficult to work efficiently with the code base, increasing the pressure even more. The situation is especially bad if the deadlines are overly optimistic. Also, emphasizing the deadlines was found to cause low team morale, rush mode, lack of empathy and poor technical decisions.

It is suggested that instead of deadlines, the emphasis ought to be put on prioritization. The team knows best how much work they can accomplish within the iteration. If something has a 'deadline', just make sure that there is enough room in the next sprint. By definition, everything included in the sprint plan ought to be completed during the sprint, making it a mere problem of prioritization. One should only fix two of the three: budget, schedule, and scope.

For the fifth suggestion, the team should aim to work more as a team, instead of individuals. Mapping hours to story points is not feasible due to productivity differences between the team members unless they work together. Working together on one task at-a-time also ensures that most of the items get done during the iteration, instead of most items being almost ready. This likely will reduce the knowledge gaps as well, and boost knowledge sharing. Also, the tasks won't be frozen if the dedicated developer is away.

The sixth suggestion is again related to the estimates. Due to the schedule pressure, and estimating instead of planning, refactoring effort is often omitted from the estimates. But then again, as stated above, the estimates have not been used during defining the scope, thus making it less important. The material revealed that neglecting refactoring will slow down development, make it more difficult to detect and resolve defects, and may require huge efforts later to resolve technical debt.

It is suggested that during the planning and estimation meetings, it is considered if there is any technical debt included in the target feature. The team should aim to incorporate the effort of refactoring into the estimates so that there will be enough time to resolve that as well. System quality was found to have an impact on the required effort. While taking more time during the development will increase the costs in the short-term, it will provide long-term benefits that easily outweigh the cost. Easy-to-read code makes it much easier to detect and correct defects.

Finally, the seventh suggestion is related to the velocity. Even though the goal is to get all selected items completed during the iterations, sometimes tasks will inevitably be left incomplete. Even if there is very little left undone, all of the story points of that item are currently marked for the next sprint, leaving none to the current sprint. This can make the velocity unstable, and thus, unusable in making predictions.

It is suggested that the team will re-estimate the incomplete stories during the next sprint planning, adjusting the story point -count to the current work to be done. While this does not mark any story points to the previous sprint, it will decrease the points marked for the next iteration, reducing the velocity variance.

5.3.2 Rework

As with the estimates, the first suggestion is related to the measurements. Not only will they assist this study in determining the impact of the suggestions, they will provide the means for continuous improvement for the team.

It is suggested that the amount of time spent on change requests and new development is tracked. This will provide the values required to calculate the rework-rate. Recidivism rate could also be tracked, to assess how many times the task is moving backward in the phases. A clear classification of stories into change requests and new development is required to track the time spent on them. Furthermore, one might have to split the change requests further into bug-fixes and paid changes.

The second suggestion relates to quality assurance practices. Currently, the company is investing heavily in actual testing: the developers implement automated test cases and provide testing instructions for dedicated test personnel. However, as the material highlighted, the earlier the defects are found, the cheaper their resolving will be. Testing captures the issues only right before the release, and won't be able to capture defects in other areas than development.

It is suggested that the company will introduce QA practices to other phases as well, including the requirements engineering and design phases. The material suggests that inspections could be a very good technique in assessing the quality of human artifacts, such as specification and design documents. FST measurements could be introduced to keep track of where the defects tend to originate and focus on those areas. Capturing defects earlier will likely reduce the amount of rework spent on change requests, and even reduce the amount of required, costly testing.

Thirdly, it is suggested that the team starts utilizing pair programming. While the material shows conflicting results regarding the efficiency, it has been shown to reduce the number of defects and even increase the throughput Furthermore, it has been shown to increase the degree to which the developers engage in refactoring and testing activities. Also, some of the studies indicate that the developers even enjoyed the problem solving more when doing it together with colleagues. Lastly, this would benefit the predictability of the project, as the feature won't be frozen if the developer goes vacationing.

Fourthly, it is suggested that the project spends a bit more time and effort in the requirements engineering. Better yet, it is advised to collaborate more with the clients, and especially increase the collaboration between the clients and developers, directly. Defects in the requirements require the most rework, as they cause rework in multiple levels: requirements, design, source code, test cases, user documentation. It is also possible to omit important details when communicating requirements to the developers, without actually involving them in the requirements meeting. The customers can also provide the developers with domain knowledge, which should help in making correct assumptions during the development.

5.3.3 Summary

As a summary, here is a list of the suggestions discussed in the above sections:

Expectations:

1.) Measure the velocity and hit-rate

2.) Estimation meetings before planning

3.) Benchmarks for story points

4.) Emphasize prioritization over deadlines

5.) Work more as a team

6.) Include refactoring effort in estimates

7.) Re-estimate incomplete stories

Rework:

1.) Measure the time spent on changes vs. new development

2.) Introduce QA processes into RE and design

3.) Utilize pair programming

4.) More RE and customer collaboration

Chapter 6

Evaluation

In late September, the development team was split into two smaller teams. The program held a special team-day, consisting of various workshops, to facilitate the forming of the two new teams. During the workshops, the team members would get to know each other and agree upon their shared work-practices.

The division into teams was pre-determined by the supervisors. The aim was to create as equal teams as possible, in terms of experience, variety of skills, and availability. Both teams ended up having four full-time and one part-time worker. One of the full-time workers, in both teams, worked remotely from another office. Unless agreed otherwise, both teams would dedicate one member to support duty for each sprint.

As a part of the team-day presentations and workshops, the suggestions proposed in section 5.3 were presented for the two new teams. The teams were given the freedom to select which suggestions they would be willing to implement for the next four sprints.

The selected suggestions were implemented and monitored for the next four sprints. Metrics were collected at the end of each sprint, along with feedback from the developers. Despite the fixed length, changes to the selections were allowed: improvements to the existing implementations were welcomed, and suggestions could be adopted or dropped at any given time.

The rest of this chapter is divided into two sections: The first section (6.1) describe the suggestions that were decided to be implemented, and the second section (6.2) present the acquired metrics and feedback, sprint-by-sprint.

6.1 Selected suggestions

The suggestions related to measurements, for both estimates and rework, were selected to be implemented regardless of the teams' choices. The measurements that could not be automated would be collected by the author, and thus, the act of measuring would not impact the normal work-flow of the teams. In addition, the reversing of the estimation- and planning meetings was also selected by default, and would be implemented by both teams.

The previous planning meeting was split into two separate meetings: planning and 'grooming'. In the grooming meetings, the teams would go over the prioritised list of tasks and determine whether they are of appropriate detail. Inadequate stories are enhanced with further details and implementation steps, and finally given an estimate. Dedicating a specific meeting for grooming ought to naturally increase the amount of effort spent for specifications.

The first team, team *Alpha*, selected the following suggestions: 'Benchmarks for story points', 'Work more as a team', 'Include refactoring effort in estimates', and 'Re-estimate incomplete stories'. A meeting was held where the team members would select benchmark tasks for 1, 2, 3, 5, 8 and 13 story points, and summarise the required changes of those benchmarks into a bullet-list. These benchmarks would then be referenced during the team's grooming sessions, to assist in making story point estimates. The team agreed to pay more attention to the potential refactoring efforts when creating effort estimates, although it was concluded to be difficult. Any unfinished tasks were decided to be re-estimated after sprint reviews, so that the remaining work could be taken into account when planning the next sprint's commitments. In addition, two members of *Alpha* indicated their interest in trying out pair-programming.

The second team, team *Beta*, did not initially choose any additional suggestions.

6.2 Results

The teams' velocities and hit-rates, along with created and resolved bug-ticket counts, were monitored for four sprints. Additional metrics included scope changes, estimation completeness, and resource allocation. The results for each sprint, along with brief notes made during the sprint, are presented in their own subsections. To provide some base for assessing the effectiveness of the suggestions, similar data was acquired for 18 previous sprints, starting from the first sprint of 2019. These figures are presented in the first subsection, and will be referenced when discussing the studied sprints.

6.2.1 Past sprints

Sprint 14 and 15 was a combined sprint, but for simplicity's sake, they are treated as two different sprints: the metrics are divided by two, so that when combined, the rows do reflect the real values for that period.

Since there has been some changes to the product program roster during the 2019, the past data has to be adjusted for it to be comparable with the new metrics. Thus, the total amount of working hours was acquired from the program's time-tracking system. A 'Developer factor' was calculated from the total by subtracting the support shifts, dividing the result by the number of sprints, and finally divided by the theoretical amount of hours per full-time worker (10 working days * 7.5hrs). The resulting factor was roughly 8.35.

The past sprint data is split further into two sub-sections. The first subsection presents the expectation metrics and summarises a couple key observations. The second subsection in turn discusses the rework -metrics and conclusions drawn from them.

6.2.1.1 Expectations

The hit-rate, velocity, scope creep and estimation percentage for the past 18 sprints are represented in table 6.1. The table is divided into four columns, each representing a different sub-set of the 18 sprints: the first column contains averages for all 18 sprints, the second column for the sprints of regular length, the third column for the latest 4 sprints, and the final column for the latest 4 sprints of regular length.

Contrasting the latest 4 sprints to the overall averages should give visibility of the current developments and which way it has been advancing.

	All sprints	Regular sprints	Last four sprints	Last four regular sprints
Hit-rate SP	58.65%	49.05%	51.22%	30.45%
Hit-rate task	51.71%	43.68%	52.02%	32.04%
Velocity SP	130.72	103.23	149.75	81.25
Velocity task	21.89	18.62	22.50	14.50

Velo Std. Deviation SP	76.68	40.63	123.09	21.00
Velo Std. Deviation task	8.34	4.94	12.50	2.65
Scope creep % SP	16.90%	19.06%	0.43%	8.00%
Scope creep % task	0.53%	0.18%	-6.23%	-1.63%
Estimation % initially	70.77%	68.95%	80.47%	77.61%
Estimation % finally	85.05%	83.96%	90.04%	86.36%
Added SP	22.28	23.92	11.75	19.25
Added task	5.39	5.38	4.75	3.00
Removed SP	20.56	21.00	36.25	23.50
Removed task	5.17	5.31	7.63	3.75

Table 6.1: Expectation metrics, regular vs. irregular sprints.

Comparing the results with regards to all and only regular sprints, certain observations can be made.

Firstly, both hit-rate and velocity are higher when irregular sprints are included. This is partially explained by the fact that irregular sprints were always longer than regular sprints, allowing more time for completing tasks. However, as can be seen from the deviation metrics, having longer sprints here and there will make the velocity unstable.

With regards to the sprint scope, roughly the same amount of tasks, both in terms of count and story point values, on average, are being added and removed from the sprint. Thus, the sprints' scopes are, on average barely affected by the changing contents. However, this does not tell the whole truth: while the scope creep % with regards to tasks is under one percent, individual sprints vary a lot. In fact, the scope creep percentage with respect to tasks varies between 16.67% and -19.15% across the sprints. So, while the average is very close to zero, scope creep has had a big impact on the sprint scopes. Also, the lack of initial story point estimates causes further variation in the scope during the sprints. In total, the estimation percentage increases roughly by 10 to 15 points during the sprints.

Finally, as observed from the hit-rate metrics, the sprints have consistently contained almost twice as much work as the team could reliably deliver within a sprint. However, these metrics, especially with regards to story points, may in fact paint too rosy a picture. Since only 70% of the stories were estimated initially, and 15% still remaining un-estimated at the end of the sprint, the hit-rate won't consider the 'actual' amount of story points within the sprint. As the story point -values were often assigned after the task had been completed, the tasks left incomplete that did not have a story point would not bring the hit-rate down.

The discrepancies between all and only regular sprints is even more pronounced in the metrics for the last four sprints. While the velocity during last four regular sprints have been the most stable, counting in the irregular sprints shows the highest standard deviation: over 80%.

The scope and its changes have been managed better during the last four sprints: the estimation percentage, especially at the beginning of the sprints, is notably higher than the average across all sprints. Consequently, also the scope creep has been reduced. However, the hit-rate shows that the tendency to take in way too much work has not been alleviated, but even made worse.

6.2.1.2 Rework

Similar to the expectation metrics, some figures with regards to rework were acquired from the same set of 18 past sprints. However, as the program has not tracked the hours spent on bug-fixes, nor has it assigned story point estimates for them, the only available option was to compare the completed amount of bug-tickets to story tickets. However, as the tickets vary in size, the following figures are only rough estimates of the rework-rates.

As with the expectation metrics, these metrics are also divided into all sprints, all regular sprints, last four sprints, and last four regular sprints. These metrics are combined to table 6.2.

In addition to this, the metrics were also viewed from the perspective of release sprints, non-release sprints, last four release sprints, and last four non-release sprints. These metrics are presented in table 6.3.

	All sprints	Regular sprints	Last four sprints	Last four regular sprints

Re-work rate %	34.89%	31.06%	45.78%	46.07%
New bugs	11.2	8.0	19.6	5.0
Bugs fixed	9.8	7.0	14.3	10.3
Bug growth	1.3	1.0	5.4	-5.3

Table 6.2: Rework metrics, regular vs. irregular sprints.

As indicated by the metrics, roughly third of the development efforts go to the fixing of defects. Slightly fewer bug-tickets are both recorded and resolved during the regular sprints. After all, the quality issues are one of the key factors affecting the decision to prolong sprints. Overall, the total amount of bugs is increasing slightly every sprint.

During the last four sprints the re-work rate has been higher: half of the completed tickets are related to defects. The contrast between all and regular sprint is also much steeper, as regular sprints have had roughly 75% fewer defects generated. Consequently, even though fewer bugs were also resolved during the last four regular sprints, the total amount of bugs were actually decreasing.

However, the metrics for the last four sprints also point out the weakness of measuring the rework -rate by the ticket counts. Even though regular sprints had fewer bugs resolved, the rework -rate was still higher in comparison to all sprints. Fewer development tasks were finished during the last four regular sprints, suggesting that they had been larger in size.

	Release sprints	Non-release sprints	Last four release sprints	Last four non-release sprints
Re-work rate %	39.88%	25.07%	51.85%	17.27%
New bugs	12.6	9.4	19.1	9.6
Bugs fixed	13.5	5.3	19.3	3.0
Bug growth	-1.0	4.2	-0.1	6.6

Table 6.3: Rework metrics, release- vs. non-release sprints.

While the rework rate did not differ notably between regular and irregular sprints, there is a clear difference between the release- and non-release sprints. More bugs are both recorded and resolved during the release sprints. This might be expected, as any defects that had slipped by the quality assurance processes would inevitably caught by the users after the update. In addition, bug-tickets are rarely generated for un-released features: instead, the aim is to fix the issues right after they have been found.

Interestingly, the total amount of bugs is decreasing only during release sprints. This suggests a negligence towards issues between updates, where the development team focused more in developing new features. Possible explanation for this could be that the issues identified right after updates are often more severe, and the developers may still remember the implemented features well, making the fixing easier.

The difference between release- and non-release sprints is, again, much steeper. Over twice as many bugs are generated, and over six times as many bugs are resolved during the release sprints. Consequently, the rework -rate for the release sprints climbs to over 50%, while it remains below 20% for non-release sprints.

The amount of generated bugs does not differ much between all non-release sprints and the last four release sprints, but slightly fewer defects have been fixed recently. Due to this, the total amount bug-tickets have been increasing more rapidly during the last four sprints. However, roughly 50% more bug-tickets have been generated during the last four release sprints, when compared to all release sprints. This suggests that the latest releases had been more problematic than usual.

6.2.2 Sprint 1

Sprint 1 was shorter than usual: while the sprints typically last for two weeks, this was only one week in length. This was due to problems during the latest release, which caused the sprint to be prolonged by one week. In order to return to the normal rhythm, the first sprint set to end at the pre-planned date, despite it beginning one week later than expected.

The sprint 1 could be considered as a "migration" sprint: a lot of new practices were introduced simultaneously, and some of the personnel were migrated to different roles. The planning had to be kept the same fashion as the previous sprints, as there were no estimation meetings held prior to this sprint. In other words, the stories did not have estimates prior to sprint planning, making it impossible to select stories without giving an estimate first. Furthermore, as some of the tasks had already been assigned to specific developers, the tasks were automatically assigned to the team that the developer belonged. Due to this, there were likely some deviations in the scopes of the two teams.

One of the team *Alpha*'s members was solely dedicated for one larger feature during the studied sprints. Since it had little to no effect to the rest of the developments, its development was considered "outside" of the regular efforts. Thus, whenever a task related to that feature was completed by a developer, they selected another one and started working on it immediately.

Since the tasks were selected on-demand, and new tasks were generated on the go, the opportunities to estimate the tasks before including them into sprints were very limited. Most of the tasks were estimated only after the development was complete, primarily to measure velocity. However, this inevitably caused some scope creep for team *Alpha*'s commitments.

In addition, since new tasks were pulled to the sprint whenever the previous task was completed, even if the sprint was almost at its end, it wouldn't have been fair to consider all of the added tasks as commitment. Thus, some of these tasks were considered 'stretch goals': the story points of the completed tasks were counted towards the velocity, but these tasks did not have an effect on the hit-rate.

The tables below represent the metrics acquired from Sprint 1.

	Alpha	*Beta*
Hit-rate SP	100.00%	61.76%
Hit-rate task	83.33%	55.56%
Velocity SP	16	42
Velocity task	5	5
Resources	2.05	1.33
Scope creep % SP	0.00%	0.00%
Scope creep % task	20.00%	0.00%
Estimation % initially	66.67%	80.00%

Estimation % finally	75.00%	80.00%
Added SP	0	0
Added task	1	0
Removed SP	0	0
Removed task	0	0
Developer factor	3.60	3.27

Table 6.4: Sprint 1 expectation metrics.

Despite being considered as a ‘migration’ sprint, Sprint 1 already shows promising results. The combined hit-rate, with regards to story points, had increased from 58.65% average to 69.05%. While the scope remained unrealistic, it was more realistic than it had been before.

When normalising the combined velocity to 4 developers, and taking into account that Sprint 1 was only one week in length, the adjusted velocities become 68.64 and 11.83, in terms of story points and tasks, respectively. The corresponding values for the 2019 average is 62.62 and 10.49. This would suggest that the performance had increased roughly by 9.6% in terms of story points, and by 12.8% in terms of tasks, already during the first sprint.

Estimation percentage remained largely the same. However, this can be expected, as there were no estimation meetings held prior to Sprint 1.

Re-work rate %	69.57%
New bugs	25
Bugs fixed	16
Bug growth	9

Table 6.5: Sprint 1 rework metrics.

The effects of the issues during the latest release are still visible during this sprint.

As indicated by table 6.5, the re-work rate was disproportionately high compared to the 2019 average. Despite being only one week in length, 16 bugs were resolved during this period. Given the shorter sprint length, also fewer stories could be completed during the given sprint, bringing the re-work rate up.

Despite the increased effort in solving defects, the bug-growth remained high: only one sprint during 2019 had had a higher bug-growth value.

6.2.3 Sprint 2

Unfortunately, sprint 2 was also atypical. The problems briefly referred to in the previous subsection had long-lasting effects on the support queue, requiring additional resources to resolve. Because of this, the whole team *Alpha* were on support duty, making their contributions to the actual sprint goals non-existent.

In addition, for the sake of solidarity, the support of the next sprint will be solely team *Beta*'s responsibility. Two developers from that team will be on support duty for sprint 3. For team *Beta* the sprint 2 was mostly typical, albeit that they had one extra developer to contribute towards the sprint goals.

The planning meeting, however, followed the expected course: the development tickets had been pre-estimated beforehand, and team *Beta* could relatively easily select stories they thought they could finish during the sprint. Team *Alpha* did not participate in the planning, as they wouldn't contribute towards the development effort. Summaries of the developments that occurred during the sprint were elaborated on the sprint review -meeting instead.

The tables below represent the metrics acquired from Sprint 2.

	Alpha	*Beta*
Hit-rate SP	-%	86.46%
Hit-rate task	66.67%	86.67%
Velocity SP	10	83

Velocity task	16	13
Resources	0	4.27
Scope creep % SP	-%	0.00%
Scope creep % task	242.86%	7.14%
Estimation % initially	0.00%	100.00%
Estimation % finally	50.00%	100.00%
Added SP	0	0
Added task	18	1
Removed SP	0	0
Removed task	0	0

Table 6.6: Sprint 2 expectation metrics.

The *Beta* team's results look promising. The hit-rate in terms of story points was over 85%, almost 30% higher than the 2019 average. All of the story tickets had an estimate before the beginning of the sprint. Consequently, the scope had remained relatively stable, with only 7.14% increase in story points during the sprint.

The velocity was also slightly improved: the normalised velocity for team *Beta* was roughly 77.75. When compared to the 2019 average, normalised velocity of 62.62, it would suggest an increase of roughly 24% in effectiveness.

Re-work rate %	59.38%
New bugs	25
Bugs fixed	19

Bug growth	6

Table 6.7: Sprint 2 rework metrics.

Despite having assigned an entire team to the support duty, two of the full-time developers were routinely needed elsewhere. As a result, the resourcing for support duty was only 2.5 developers (two full-time, one part-time developer), half-a-developer more than usual.

The re-work rate was roughly 10% smaller than during the previous sprint. However, the issue-list kept growing, indicating insufficient resourcing: 6 more tickets were added to the list than were removed.

6.2.4 Sprint 3

Sprint 3 was relatively close to the expected, typical sprints. One major exception was that team *Beta* had two of the five developers on support duty, effectively halving the available resources for the sprint. Conversely, team *Alpha* had one extra developer available, due to not having to dedicate a support person. However, team *Alpha* also had one developer dedicated on a separate sub-project, doing development outside of the normal sprint rhythm. Their contributions were still followed on the team's sprint board, and their contributions were included in the team's velocity.

The concept of 'stretch goal' was utilised heavily during sprint 3. The top-priority development tickets turned out to be quite big: the top priority ticket was deemed larger than what could be reliably achieved during one sprint, by one person. Thus, the item was indicated as a 'stretch goal', and would not be considered as real commitment.

In addition to this, some other tasks were also indicated as stretch goals. It was discovered that some of the developers ran out of tasks to do during the previous sprints, initiating a need to pull more to the sprints. The ideal was that the developers would help the others finish their tasks once they were done with their own, but thus far there has been a lot of resistance to do so. Since the developers tended to run out of things to do more towards the end of the sprints, it was not likely that the newly pulled tickets would get completed during the remainder of the sprint. Thus, these too were indicated as 'stretch goals', and considered outside the team's commitment.

Allowing the developers to continue with stretch goals once done with their original tasks had its pros and cons. On one hand, the available resources were utilised to their maximum. Starting on a bigger task already on the previous sprint made it more likely that it will get completed during the next one. However, having work in progress during the shift of the sprints turned out to be a problem, since the developer may end up in support for the next sprint. In addition, it inevitably causes deviation in the teams' velocities, making it counter-productive towards the goal of more predictability.

The tables below represent the results of Sprint 3.

	Alpha	*Beta*
Hit-rate SP	76.09%	38.24%
Hit-rate task	76.47%	44.44%
Velocity SP	69	42
Velocity task	14	6
Resources	4.5	2.17
Scope creep % SP	12.20%	0.00%
Scope creep % task	30.77%	0.00%
Estimation % initially	100.00%	100.00%
Estimation % finally	83.33%	100.00%
Added SP	0	0
Added task	4	0
Removed SP	0	0
Removed task	0	0

Table 6.8: Sprint 3 expectation metrics.

For team *Alpha*, this was the first relatively regular sprint, in terms of allocated resources and time. And the results are quite promising: the hit rate, both in terms of tasks and story points, were as high as 76%, with fairly stable goals.

Team *Beta*, the figures look less satisfactory. The hit-rate in terms of tasks was roughly 45%, and even lower for story points at roughly 38%. The dip is very steep from the 86% of the last sprint.

However, as discussed previously, team *Beta* had two developers in support, whereas during Sprint 2 they didn't have any. While on the Sprint 2 the team's developer factor was 4.27, this sprint it was only 2.17. Normalising the velocities of Sprint 2 and 3 to a development factor of 4, we get adjusted velocities of 77.75 and 77.42, respectively. Thus, the development speed had remained largely similar. This suggest that the dip in the hit-rate was a result from not taking the lack of resources properly into account when making the sprint commitment.

Re-work rate %	62.96%
New bugs	29
Bugs fixed	17
Bug growth	12

Table 6.9: Sprint 3 rework metrics.

Despite the return to a normal amount of technical supporters, the number of fixed bugs had not decreased by much. However, the number of new bugs has not reduced either, indicating that the quality issues are still very much present. The number of total bugs increased by 12 tickets this sprint, suggesting that technical support still isn't getting enough resources to match the need.

6.2.5 Sprint 4

Sprint 4, at last, could be considered a typical sprint: both teams had one developer dedicated to support work, and the sprint was of normal length. This with only one, rather minor exception: one developer from team *Alpha* was still dedicated on separate developments. In addition, a 'Hackday' was held during sprint 4, where both teams would dedicate to fix bugs. This meant that even though the sprint was of normal length, only 90% of the normal resources were available during this sprint. This, however, was taken into account during the sprint planning. Hackdays are issued every now and then, and are considered a normal, although irregular, occurrence during sprints.

The tables below represent the results of Sprint 4.

	Alpha	*Beta*
Hit-rate SP	46.67%	41.49%
Hit-rate task	50.00%	54.55%
Velocity SP	58	39
Velocity task	7	6
Resources	3.2	2.97
Scope creep % SP	39.76%	0.00%
Scope creep % task	36.36%	0.00%
Estimation % initially	87.50%	100.00%
Estimation % finally	100.00%	100.00%
Added SP	20	0
Added task	4	0
Removed SP	0	0
Removed task	0	0

Table 6.10: Sprint 4 expectation metrics.

Unfortunately, the hit-rate, especially with regards to story points, was lower than the 2019 average. Team *Alpha*'s hit-rate was lowered from 76%

to 50%, only half of the committed work being done by the end of the sprint. While *Beta*'s hit-rate slightly improved from the previous sprint, it is still way below the results of the second sprint.

However, the hit-rate measures during this sprint may paint a bit too gloomy picture. A lot of tasks ended up being developed, but lacked a code review or the mere merge to the development branch. In other words, there were essentially completed tasks that the teams just did not get pushed to the external testing before the end of the sprint.

This is better illustrated when taking the tickets in 'In Review' state into account: the hit-rate, in terms of story points, would have been 90.52% for team *Alpha*, and 100.00% for team *Beta*. The teams seemed to consider the tasks as 'developed' regardless of the merge/code review status, as there are (usually) no further development required.

The involvement of one developer in a separate sub-project has a clear impact on the scope metrics. The scope had increased by almost 40% for team *Alpha*, due to tasks being pulled into the sprint by the given developer. The pulled tasks had no initial story point estimates, and were given by the developer once the tasks were complete. Their effect is best articulated when comparing the scope metrics of the two teams: team *Beta* did not have the same situation, and their scope remained fixed during the entire sprint. However, there were some smaller tasks that were added to team *Alpha*'s sprint, even though they weren't related to the sub-project.

The velocities, when normalized for four developers, were relatively similar to the previous sprints. Team *Alpha*'s adjusted velocity increased to 72.5 story points, and team *Beta*'s reduced to 52.53 story points. The average for 2019 was 62.62. These adjusted velocities, however, did not consider the 'almost' complete tasks.

Re-work rate %	80.60%
New bugs	28
Bugs fixed	52
Bug growth	-26

Table 6.11: Sprint 4 rework metrics.

The effects of the Hackday really show: because of it, this sprint was the only one during which the total amount of bugs actually decreased. Almost twice as many bugs were fixed as new ones were introduced.

However, this is also very visible in the re-work rate: roughly 80% of the completed tickets were bugs. This illustrates very well why a naive comparison of tasks is not sufficient way of measuring rework, as the hackday only took 10% of the available resources. The regular technical support would also take their own share of the resources, but nowhere near this level.

6.2.6 Summary of the studied sprints

As a summary, the teams' results for the four sprints are combined into the table 6.12. Both Expectation and Rework related metrics are presented here, along with some additional metrics. In addition, the averages of these sprints are compared to the 2019 averages in table 6.13.

	Sprint 1	Sprint 2	Sprint 3	Sprint 4
Hit-rate SP	69.05%	87.74%	50.00%	46.19%
Hit-rate task	66.67%	74.36%	65.38%	50.00%
Velocity SP	58	93	111	97
Velocity task	10	29	20	13
Commitment SP	84	106	148	210
Commitment task	15	39	26	26
Resources	3.38	4.27	6.67	6.17
Adj. Velocity SP	68.64	87.12	66.56	62.88
Adj. Velocity task	11.84	27.16	12.00	8.44
Adj. Commit. SP	99.40	99.28	88.76	136.16
Adj. Commit. task	17.76	36.53	15.59	16.84

Scope creep % SP	0.00%	10.42%	3.50%	18.64%
Scope creep % task	7.14%	85.71%	18.18%	18.18%
Estimation % initially	75.00%	71.43%	100.00%	94.12%
Estimation % finally	77.78%	85.71%	92.86%	100.00%
Re-work rate	69.57%	59.38%	62.96%	80.60%
New bugs	25	25	29	28
Bugs fixed	16	19	17	54
Bug growth	9	6	12	-26

Table 6.12: Summary of the studied sprints.

	2019 average	Average
Hit-rate SP	58.65%	58.76%
Hit-rate task	51.71%	65.09%
Velocity SP	130.72	89.75
Velocity task	21.89	18
Commitment SP	222.89	137.00
Commitment task	42.33	26.50
Resources	8.35	5.12
Adj. Velocity SP	62.62	70.12
Adj. Velocity task	10.49	14.01

Adj. Commit. SP	106.77	107.03
Adj. Commit task	20.28	20.70
Scope creep % SP	16.90%	9.60%
Scope creep % task	0.53%	34.18%
Estimation % initially	70.77%	86.54%
Estimation % finally	85.05%	91.38%
Re-work rate	34.98%	71.14%
New bugs	11.2	26.75
Bugs fixed	9.8	26.50
Bug growth	1.3	0.25

Table 6.13: Comparison of the averages.

The results of sprint 2 need to be taken with a grain of salt. As team *Alpha* was solely on support, they did not have any commitments during the sprint, essentially boosting the hit-rate to 100%. Also, as they were only bugs that they were resolving, the amount of tasks completed during that sprint is disproportionate to the others. This also explains why there are fewer story points completed per task, and why the scope creep is so massive in sprint 2.

The standard deviation of the velocity for the four studied sprints is thus 8.45 (46.92%) in terms of tasks, and 22.53 (25.10%) in terms of story points. The corresponding values for the previous sprints of 2019 were 8.34 (38.09%) and 76.68 (58.66%).

Chapter 7

Discussion

In this chapter, the results presented in the previous chapter are discussed a bit further, and a mapping between the observed effects and the probable causes is attempted to be made. A brief recap of the selected suggestions, along with their potential effects, are discussed in section 7.1. The challenges during the research are reflected in section 7.2, where the limitations of this study are considered. Finally, potential avenues for further research, stemming from the findings of this study, are presented in section 7.3

7.1 Effects of the implemented suggestions

At the beginning of the evaluation part, both teams were given the freedom to implement any of the suggestions proposed in section 5.3. Reversing the estimation and planning was considered a mandatory change, and was implemented by both teams. Acquiring metrics to study the progress of the teams was also implemented by default.

Having specified and estimated the stories before the sprint planning had clear benefits. The planning meeting was much smoother, requiring roughly half the time than previously. Of course, the time saved in the plannings was spent on the separate grooming sessions, but having two shorter meetings instead of one long one made it more bearable and the developers seemed to be able to focus better. The grooming was also done with fewer developers, potentially increasing the time spent on development. However, including the whole teams on the grooming meetings could have made the specifications and estimates more accurate.

The reversal of the events also made it possible to focus on making more realistic commitments. There was very little scope creep, except for that which was caused by the separate sub-project. The high scope creep percentage task-wise was primarily caused by the 'support sprint' of team *Alpha*: there was no real commitment at the beginning, and tasks were added to the sprint as the previous tasks were complete. Furthermore, the scope creep values for the previous 2019 sprints give too rosy picture: the deviation was very high, with values ranging from 16.67% to -19.15%. With the studied sprints, no tasks were removed from the initial commitments, and the initial and final estimation percentages were increased by 16% and 6%, respectfully.

Static targets are arguably easier to take seriously, as the target is not running away at the same speed one is pursuing it. While the set goals were not exactly achieved, the mindset shift was very visible: the teams are now aiming to complete everything they take into the sprints. The hit-rate with regards to tasks was increased by 15%, while it remained roughly the same for story points. However, as stated before, the hit-rate values for the previous sprints of 2019 likely presents a bit over-optimistic view. Although the velocity had slightly decreased with regards to story points (but increased in terms of stories), the velocity had become much more stable, from a standard deviation of 58.66% down to 25.10%. But then again, the standard deviation of velocity with regards to tasks had increased from 38.09% to 46.92%, due to the aforementioned factors.

As estimating before planning is the order in which it is supposed to be done, it is difficult to find reference material to compare these results with.

Team *Alpha* had initially chosen 'Benchmarks for story points', 'Work more as a team', 'Include refactoring effort in estimates', and 'Re-estimate incomplete stories'. Team *Beta* did not implement any specific suggestions but gravitated a bit towards working more as a team by the split into two teams.

The benchmarks were considered helpful, although the developed benchmarks will likely still require some adjusting. While it gives a defensible strategy for arguing how many story points a certain story is, the developers sometimes felt that the benchmarks would suggest a different story point estimate than they would with the given information. Determining what would be the appropriate story point value based on the benchmarks was deemed rather easy, but the developers worried that this may not be the correct estimate. Finally, the benchmarks would be more useful if they were more visible during the grooming meetings: the developers had to open the document on their computers, and there was no guarantee that everyone did consider the benchmarks during the estimation.

While defining what a story point means with a benchmark story is quite common (Hamouda, 2014; Zhong et al., 2011), having defined example stories for all relevant story point values within the selected scale did not come up in the studied material. Thus, it is difficult to compare the effectiveness of this practice. It may have been sufficient to just inspect the latest stories that were given similar values, had they been readily available.

Working more as a team turned out to be quite difficult. Even though two developers stated their interest in trying pair-programming, that never became a reality. However, the circumstances for trying it were not too great either: a lot of big features were under considerable schedule pressure, and pair-programming is known to take a bit more development time (Cao et al., 2010). Also, one of the developers who would have been interested in pair-programming was working part-time, reducing the number of opportunities even further. While one of the developers worked part-time, one in another office, and one in a separate sub-project for the duration of the four sprints, teamwork was quite minimum.

Regardless, splitting into two teams had also had positive effects for both teams: it was deemed easier to ask for help from the own team members, and the meetings got a lot shorter. Perhaps not directly caused by the division into smaller teams, the teams also changed the way they started working on larger features. While previously only one person would be dedicated to a single feature, bigger features were now developed in parallel, if there weren't many dependencies between the stories. On occasion, the teams also started the work by a short 'workshop' where they would develop together the essential things that would be shared across the stories, to prevent conflicts later. To an extent, coordination and co-operation had thus increased after the change. Better teamwork is a known characteristic of smaller teams (Hoegl, 2005), and these findings seem to be inline with that.

While team *Alpha* aimed to take the refactoring effort into account during the first grooming sessions, it somewhat stopped prematurely. Perhaps it was the fact that a lot of entirely new features were prioritized at the time, containing no technical debt yet, or that team *Beta* did not specifically take it into account either. After all, either of the teams could end up developing the tickets groomed by either of the teams. Also, being able to take the refactoring effort into account would require a rather deep knowledge of the existing feature to be able to do that. Since most of the existing features have been developed by a single developer, perhaps some changes to it being made by some other developer, it was really up to chance whether or not the given developer got to groom the related story.

Finally, team *Alpha* gave a go at re-estimating the stories that were started but left incomplete at the end of the sprint. However, due to the atypical sprints at the beginning, only one story that had a prior estimate was left incomplete during the first three sprints. Also, the team felt that subtracting story points from incomplete stories did not serve the team: the team did not consider the standard deviation of the velocity but based their judgments on the average velocity. Re-estimation was aimed at improving velocity's stability, at the cost of reducing the average, which the team did not find pleasant. In extreme cases, a task only missing a code-review, with a big story point estimate, would only be roughly 1 story point in the next sprint, bringing down the average by a lot. Thus, re-estimation was dropped after the third sprint.

The practice of re-estimating incomplete stories was adopted from an article by Mahnic (2011). However, even in that article, it was only a suggestions: no real data on its effectiveness was discussed. Thus, it is difficult to compare the experiences regarding the usefulness of this practice.

Unfortunately, none of the suggestions specifically targeting the rework were selected by the teams. Real pair-programming was never attempted, despite the initial interest of two developers in team *Alpha*. Indeed, finding a good time to do that was difficult do to one of the developers working part-time, during regular office hours, and the other preferring to start their days later. The measuring of re-work based on a naive comparison of bug tickets to stories was also found lacking. The one-day-long hack day was enough to bloat the re-work rate to over 80%, even though it took a mere 10% of the actual time.

However, due to a lot of bigger developments being prioritized for the last four sprints, and due to having separate grooming sessions, a considerable amount of additional effort was spent in RE. Unfortunately, the time was too short for most of the features to be released: it's impossible to say if the investment in RE had had a positive effect on the required re-work.

7.2 Limitations

The timing of this study was far from optimal. In addition to the implemented suggestions, the team was split into two smaller teams, and some of the personnel changed their roles. Not only that, but the release before the split had been quite faulty, causing serious problems to the normal workflow. All this combined, there were a lot of things going on at the same time, requiring serious effort to get under control.

Since multiple suggestions were implemented at the same time, meanwhile the whole development process was being re-organized, it has become difficult to map the causes to their effects. Some changes may also have combined effects, making these distinctions even more difficult. Thus, the above assessments of the effects of the individual suggestions must be taken with a grain of salt.

Due to the problematic release, the number of bugs had been disproportionately higher during the last four sprints than during the sprints before them. To combat that, more resources were put on technical support than usual, leaving much less resources for new development. Indeed, over twice as many bugs were recorded, and over thrice as many bugs were fixed per sprint during the studied period than during the previous 2019 sprints.

Due to these issues, three of the four sprints had to be considered atypical: sprint 1 was only one-week-long, the entire team *Alpha* was on technical support for sprint 2, and two instead of one developer was on technical support from team *Beta* for sprint 3. Besides that, one developer from team *Alpha* was primarily working on a separate sub-project, following an entirely different development rhythm, which skewed the metrics even further. After all, one developer is a lot in a team of 5, where one developer is supposed to be on technical support at a time, and one works part-time.

Due to the aforementioned challenges, and a relatively short study period, the results likely won't be very reliable. A longer study-period, preferably after the biggest problems had dissipated, would be required for accurate metrics.

This study focused solely on this case product program's context: the topics were chosen based on the relevance to this program. The suggestions, both their proposal and final implementation, were also tailored for this program specifically. Because of this, the results likely won't be generalizable to other projects, or other software development contexts.

Also, due to the scope of a Master's Thesis, going for a systematic literature review was deemed unrealistic. Thus, some relevant papers may have been omitted. Since the set of studied articles is quite limited, chance plays a role in what suggestions were discovered. Doing the research again may potentially result in a different set of suggestions.

Due to the unavailability of proper rework data for the past sprints, and the inability to track it during the studied sprints, the rework metric is mostly unusable. However, due to the atypical circumstances caused by the faulty release, any positive effects likely wouldn't have been visible anyways.

7.3 Future research

Should a similar study be conducted for the case program in the future, certain things should likely be considered, based on the results of this study.

Firstly, the study time should be longer, preferably at least by two sprints. The more sprints that were studied, the more stable the averages, and more accurate stability metrics can be acquired. One should also evaluate the current situation carefully before beginning the research, to try to avoid highly unusual circumstances.

While the hit-rate had not increased as much as was expected, especially the last sprint got *almost* all tasks completed. Had the code reviews been conducted and the changes merged, the hit-rate would have been very close to 100%. Thus, it would probably be beneficial to find out why these things were not completed by the end of the sprint and find out ways to make it easier to achieve in the future. For example, finding good guidelines for splitting the tasks into smaller chunks would make the velocity more stable, as huge tasks, when left incomplete, take a very big tax on the hit-rate. Ingraining a process to check the open merge requests before continuing with other tasks could also decrease the number of incomplete tasks if it is only an issue of not remembering them. However, there could also be problems with commitment, where the team does not find it that important to get the tasks 'done' done during the sprints, especially if no release is made after them.

If the story point benchmarks turn out to be a good practice, one could also start to measure the estimation accuracy. Since the benchmarks offer a rather objective way of assessing how much work a certain task has, this can be repeated after the task is complete. Thus, the estimated effort can be compared to the actual effort, giving feedback to the estimation process: if the estimates turn out to be overly optimistic, the teams can try to compensate for it by giving a bit higher estimates than they normally would. Currently, whenever the goals of the sprints are not met, the teams can only guess whether it was caused by inaccurate estimates, or by taking too many story points to the sprints.

Finally, a new system for measuring the rework effort should be put in place. Naive comparison of completed bugs to completed stories did not produce reliable results, making any potential advancements in the rework effort undetectable. Preferably, the teams would be able to track the spent hours on bug fixes, support, new development, miscellaneous things, and perhaps even meetings, to give an accurate view of where the time is being spent.

While these are general pointers for future research, stemming from the results of this study, the new research should also start from a current state analysis and find out the most pressing issues to focus on. Depending on how far in the future that research would take place, the situation might be very different.

Chapter 8

Conclusions

The goal of this study was to improve the efficiency and sustainability of the case product program. To find the most relevant problem areas, a current-state analysis was executed: five employees from the program, working in different roles, were interviewed.

Based on the analysis, expectations seemed to play the biggest role with regards to efficiency and sustainability and were concluded to be guided by the low predictability. Thus, a literature review was conducted, primarily aiming to find ways to improve the predictability of the case product program's development process. On top of that, reducing rework was chosen as a secondary objective of this study, as it was considered the second most important factor affecting efficiency and sustainability.

Based on the studied articles, the output of the literature review was a set of suggestions, aimed at improving the predictability and reducing unnecessary rework. The product teams were given freedom to select which of these suggestions they would implement for four sprints. The implementation of the selected suggestions, along with the sprint results, were monitored during this period.

Unfortunately, due to atypical circumstances, the results were left largely inconclusive. Estimating and specifying the development tasks in a separate session before the planning allowed the teams to better plan the contents of the sprints. Due to this, the scope remained more stable throughout the sprints. Also, the velocities of the teams became more stable, although the data is somewhat unreliable. However, being able to plan the sprints better had a smaller impact on the hit-rate than initially expected.

Most of the additional suggestions were not deemed that useful. Only the benchmarks for story point estimates were considered particularly useful, even though they still require some adjustments to be accurate. This may be due to more pressing matters during the evaluation period and may have to be tried again later.

As a summary, the implemented suggestions left much to be desired with regards to improving the predictability, despite having provided some benefits. The effects on rework turned out to be impossible to determine by comparing the number of completed bugs to the number of completed stories. Further research with access to the work-time loggings is required to determine the effects on the required rework.

Bibliography

A. Agrawal, M. A. Atiq, and L. S. Maurya. A current study on the limitations of agile methods in industry using secure google forms. *Procedia Computer Science*, 78:291–297, 2016.

Amazon. AWS documentation. `https://docs.aws.amazon.com/index.html`, n.d. Accessed: 16.12.2019.

B. Boehm and R. Turner. People factors in software management: lessons from comparing agile and plan-driven methods. *Crosstalk-The Journal of Defense Software Engineering*, 2003.

R. Britto, E. Mendes, and J. Börstler. An empirical investigation on effort estimation in agile global software development. In *2015 IEEE 10th international conference on global software engineering*, pages 38–45. IEEE, 2015.

L. Cao. Estimating agile software project effort: an empirical study. *AMCIS 2008 Proceedings*, page 401, 2008.

L. Cao, B. Ramesh, and T. Abdel-Hamid. Modeling dynamics in agile software development. *ACM Transactions on Management Information Systems (TMIS)*, 1(1):5, 2010.

B. B. Chua and J. Verner. Examining requirements change rework effort: A study. *arXiv preprint arXiv:1007.5126*, 2010.

E. Coelho and A. Basu. Effort estimation in agile software development using story points. *International Journal of Applied Information Systems (IJAIS)*, 3(7), 2012.

P. Coughlan and D. Coghlan. Action research for operations management. *International journal of operations & production management*, 22(2):220–240, 2002.

D. Damian. Stakeholders in global requirements engineering: Lessons learned from practice. *IEEE software*, 24(2):21–27, 2007.

P. Deemer, G. Benefield, C. Larman, and B. Vodde. The scrum primer. *Scrum Primer is an in-depth introduction to the theory and practice of Scrum, albeit primarily from a software development perspective, available at: http://assets. scrumtraininginstitute. com/downloads/1/scrumprimer121. pdf*, 1285931497:15, 2010.

B. DiCicco-Bloom and B. F. Crabtree. The qualitative research interview. *Medical education*, 40(4):314–321, 2006.

D. Duka and L. Hribar. Fault slip through measurement in software development process. In *Proceedings ELMAR-2010*, pages 177–182. IEEE, 2010.

A. E. D. Hamouda. Using agile story points as an estimation technique in cmmi organizations. In *2014 agile conference*, pages 16–23. IEEE, 2014.

C. Heyer and M. Brereton. Reflective agile iterative design. 2008.

J. Highsmith and A. Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

J. Hihn and H. Habib-agahi. Cost estimation of software intensive projects: A survey of current practices. In *Proceedings of the 13th international conference on Software engineering*, pages 276–287. Citeseer, 1991.

M. Hoegl. Smaller teams–better teamwork: How to keep project teams small. *Business Horizons*, 48(3):209–214, 2005.

H. Holmstrom, E. Ó. Conchúir, J. Agerfalk, and B. Fitzgerald. Global software development challenges: A case study on temporal, geographical and socio-cultural distance. In *2006 IEEE International Conference on Global Software Engineering (ICGSE'06)*, pages 3–11. IEEE, 2006.

T. Javdani, H. Zulzalil, A. A. A. Ghani, A. B. M. Sultan, and R. M. Parizi. On the current measurement practices in agile software development. *arXiv preprint arXiv:1301.5964*, 2013.

M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.

M. Jørgensen. A critique of how we measure and interpret the accuracy of software development effort estimation. In *First international workshop on software productivity analysis and cost estimation*. Citeseer, 2007.

A. A. Khan, S. Basri, and P. Dominic. A propose framework for requirement change management in global software development. In *2012 International Conference on Computer & Information Science (ICCIS)*, volume 2, pages 944–947. IEEE, 2012.

E. Knauss, C. El Boustani, and T. Flohr. Investigating the impact of software requirements specification quality on project success. In *International Conference on Product-Focused Software Process Improvement*, pages 28–42. Springer, 2009.

V. Lalsing, S. Kishnah, and S. Pudaruth. People factors in agile software development and project management. *International Journal of Software Engineering & Applications*, 3(1):117, 2012.

Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan. Software development life cycle agile vs traditional approaches. In *International Conference on Information and Network Technology*, volume 37, pages 162–167, 2012.

Lexico. "sustainability". `https://www.lexico.com/en/definition/sustainability`, n.d. Accessed: 15.12.2019.

G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, and S. Brinkkemper. Improving agile requirements: the quality user story framework and tool. *Requirements Engineering*, 21(3):383–403, 2016.

G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, and S. Brinkkemper. Improving user story practice with the grimm method: A multiple case study in the software industry. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 235–252. Springer, 2017.

V. Mahnic. A case study on agile estimating and planning using scrum. *Elektronika ir Elektrotechnika*, 111(5):123–128, 2011.

E. Miranda. Improving subjective estimates using paired comparisons. *IEEE Software*, 18(1):87–91, 2001.

K. Moløkken-Østvold, M. Jørgensen, S. S. Tanilkan, H. Gallis, A. C. Lien, and S. Hove. A survey on software estimation in the norwegian industry. In *10th International Symposium on Software Metrics, 2004. Proceedings.*, pages 208–219. IEEE, 2004.

S. Monsell. Task switching. *Trends in cognitive sciences*, 7(3):134–140, 2003.

N. Nurmuliani, D. Zowghi, and S. Powell. Analysis of requirements volatility during software development life cycle. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 28–37. IEEE, 2004.

W. F. Opdyke. Refactoring object-oriented software to support evolution and reuse. In *Proceedings of the 7th Annual Workshop on Institutionalizing Software Reuse*, 1995.

J. D. Procaccino and J. M. Verner. Software project managers and project success: An exploratory study. *Journal of Systems and Software*, 79(11): 1541–1551, 2006.

F. Raith, I. Richter, R. Lindermeier, and G. Klinker. Identification of inaccurate effort estimates in agile software development. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 67–72. IEEE, 2013.

T. Sedano, P. Ralph, and C. Péraire. Software development waste. In *Proceedings of the 39th International Conference on Software Engineering*, pages 130–140. IEEE Press, 2017.

E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *2010 17th Working Conference on Reverse Engineering*, pages 249–258. IEEE, 2010.

V. Suma and T. Nair. Defect management strategies in software development. *arXiv preprint arXiv:1209.5573*, 2012.

R. Tamrakar and M. Jørgensen. Does the use of fibonacci numbers in planning poker affect effort estimates? 2012.

B. Tanveer, L. Guzmán, and U. M. Engel. Effort estimation in agile software development: Case study and improvement framework. *Journal of Software: Evolution and Process*, 29(11):e1862, 2017.

M. Usman, E. Mendes, F. Weidt, and R. Britto. Effort estimation in agile software development: a systematic literature review. In *Proceedings of the 10th international conference on predictive models in software engineering*, pages 82–91. ACM, 2014.

R. Valade. The big projects always fail: Taking an enterprise agile. In *Agile 2008 Conference*, pages 148–153. IEEE, 2008.

J. Verner, J. Sampson, and N. Cerpa. What factors lead to software project failure? In *2008 Second International Conference on Research Challenges in Information Science*, pages 71–80. IEEE, 2008.

L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE software*, 17(4):19–25, 2000.

D. Yang, Q. Wang, M. Li, Y. Yang, K. Ye, and J. Du. A survey on software cost estimation in the chinese software industry. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 253–262. ACM, 2008.

S. Zhong, C. Liping, and C. Tian-en. Agile planning and development methods. In *2011 3rd International Conference on Computer Research and Development*, volume 1, pages 488–491. IEEE, 2011.

Appendix A

Interview template

The following template was used when conducting the semi-structured interviews for the current state analysis.

Due to the semi-structured nature of this interview, any emergent topics, that the interviewer deemed relevant, were discussed between the pre-defined steps. Otherwise, the below template was followed rather linearly: topics that emerged during step N were discussed further right away, before continuing with step N+1.

Introduction

- 1: Brief introduction to the topic. Explanation why the interview is being conducted. Asking for consent for recording, agreeing on definitions, etc.
- 2: "How would you describe your role and duties within this program?"
- 3: "How do you execute your tasks, and what does executing them mean in practice?"

Efficiency

- 4: "Based on your evaluation, how efficiently is the program progressing?"
- 5: "Which areas do you think would require the most attention? Why?"
- 6: "What do you think is contributing to X?", where X is a problem identified by the interviewee. Each identified problem are discussed individually.

Sustainability

- 7: "Based on your evaluation, how sustainable is the program?"

- 8: "How is the know-how distributed among the employees?"
- 9: "Do you feel that there are bottlenecks, or people that the program could never afford to lose?"
- 10: "What do you think is contributing to Y?", where Y is a problem identified by the interviewee. Each identified problem are discussed individually.

Actions

- 11: "What could you do in your role to enhance the efficiency and/or sustainability?"
- 12: "What should the program as a whole do differently to enhance the efficiency and/or sustainability?"
- 13: "If you were given the power to make any one change to the program, what would it be?"