

Master of Information Systems: Management and Innovation

How to get away with technical debt: An explorative multiple-case study on autonomous teams and technical debt management

Karl Omar Skeimo – Student no. 703957

A report submitted in partial fulfillment of the requirement for the degree of Master of
Information Systems: Management and Innovation

Supervisor: Ranvir Rai

Restricted: Yes No

Kristiania University College
Prinsens Gate 7-9
0152 Oslo
Norway

Abstract

Technical debt (TD) is constantly accumulating throughout software development processes. In many autonomous teams this technical debt will damage and injure the process, prohibiting them from adding new functionalities to their products. Tech companies must therefore understand how they can manage TD to avoid getting stuck fixing bad code. In the research on technical debt management (TDM), there seems to be a lack of empirical studies that examine how TD is managed in autonomous teams. Some frameworks are developed with the purpose of investigating TDM but lack the empirical validation and reliability.

This study investigates how autonomous teams actively manage technical debt, by conducting a multiple-case study in a Norwegian fintech company. The teams are studied by utilizing the TDM framework, measuring autonomous teams' degree of maturity within different TDM activities in order to understand their current state of practice and how to further improve these.

The study found that all autonomous teams practiced TDM, but to various extents. Some teams had structured processes, while others had no clear strategies. Most of the teams were ranked with what the framework call "received level of maturity", and conducted TDM activities occasionally based on their current needs. The study also found challenges related to the TDM frameworks maturity levels relation to TDM success, and identified that TDM activities ranked as highly mature did not necessarily translate into higher TDM success.

The study identified a need for the TDM framework to be further empirically tested and iterated on for it to work as a an accurate tool for understanding and improving autonomous teams' TDM processes.

Keywords: agile software development, autonomous teams, technical debt, technical debt management, case study

Acknowledgements


Firstly, I would like to express my gratitude to my supervisor, Ranvir Rai, for giving me great guidance and for stepping in as my supervisor on short notice.

I am also grateful for fellow students, friends, and family showing their support throughout this research project.

I would also like to thank the case company and respondents for providing me with excellent insights into their inspiring work with software development.

I also wish to offer special thanks to my initial supervisor, Knut H. Rolland, who helped me initiate and organize this research project. Although he is not with us anymore, he opened my eyes to the agile universe with his passion, and will continue to inspire me in my future work.

I certify that the work presented in the thesis is my own unless referenced

Signature: .....

Date: 05.25.2021.....

Total number of words: 19 809

Table of contents

1.0 INTRODUCTION	6
2.0 LITERATURE REVIEW	8
2.1 AGILE SOFTWARE DEVELOPMENT AND AUTONOMOUS TEAMS	8
2.2 TECHNICAL DEBT	9
2.3 TECHNICAL DEBT MANAGEMENT	11
2.3.1 TDM activities	11
2.3.2 Challenges related to TDM	15
2.4 TDM FRAMEWORK	16
3.0 RESEARCH DESIGN	19
3.1 MULTIPLE-CASE STUDY	19
3.2 CONSTRUCTION OF RESEARCH INSTRUMENT	21
3.3 DATA COLLECTION	21
3.3.1 Transcription of video recordings, coding, and data analysis	22
3.4 VALIDITY AND RELIABILITY	23
3.5 CASE DESCRIPTION	24
4.0 RESULTS AND ANALYSIS	26
4.1 SUMMARY OF FINDINGS	26
4.2 TDM ACTIVITIES	29
4.2.1 Level 1 – Unorganized	29
4.2.2 Level 2 – Received	30
4.2.3 Level 3 – Organized	36
5.0 DISCUSSION	40
5.1 DISCUSSION OF TDM ACTIVITIES	40
5.1.1 Organized activities	41
5.1.2 Received activities	43
5.1.3 Unorganized activities	45
5.2 RELIABILITY OF TDM FRAMEWORK	46
6.0 IMPLICATIONS	48
6.1 LIMITATIONS	48
6.2 PRACTICAL IMPLICATIONS	48
6.3 IMPLICATIONS FOR FUTURE RESEARCH	49
7.0 CONCLUSION	51
REFERENCES	52
APPENDIX.....	61
APPENDIX A: ETHICAL APPROVAL AND NSD APPROVAL	61
APPENDIX B: RESEARCH INSTRUMENT	63

Tables and figures

Table 1: Overview of TDM activities suggested by Li et al. (2015).....11

Table 2: TDM framework.....17

Table 3: Overview of case study classifications.....19

Table 4: Overview of interviews.....22

Table 5: Summary of findings.....26

Table 6: TDM framework and team evaluation.....29

Figure 1: Data collection process.....23

1.0 Introduction

Due to complexity and uncertainty influencing how companies work towards software development, companies have recognized human capital and agility as fundamentally crucial to create organizational success. Thus, the agile project management methods became the new way of working towards software development (Dybå et al. 2014, 281). The agile way of approaching software development has nearly become a synonym for success, and methods such as Scrum (Kniberg 2015), eXtreme Programming (Beck 1999), and Kanban (Huang and Kusiak 1996) have become industry standards caused by their ability to shorten development cycles, focusing on iterative work and quicker product releases. In many ways, the "agile way" of working has become a shift in how companies approach software development, where its practices are based upon principles from The Agile Manifesto (Agile Alliance 2001).

Companies constantly needs to increase their speed of innovation, which demands them to understand how to balance the ambidexterity of prioritizing different factors that can affect their pace of responsiveness. This often results in software teams paying less attention to a product's design, programming practices, and test coverage and more attention to the product's visible functionality to release it (Codabux and Williams 2013, 8). As a result, many autonomous teams are trying to find short-term beneficial shortcuts in software development. They constantly make trade-offs where faster value delivery to their customers is prioritized over internal product quality (Klotins 2018, 75). Consequently, many software teams are left with accumulated technical debt (TD) they will have to repay in later runs.

Originally, TD was introduced in 1992 as a metaphor to communicate the consequences of poorly developed software (Cunningham 1992). Since then, the concept has become widely known in agile software development (ASD) practices. Choosing a shortcut in the code writing can provide the team with short-term benefits from having a quicker product release, please the customer, and give the company clear advantages in the time-to-market competition (Kruchten et al. 2012; Yli-Huumo et al. 2016). However, the accumulated debt can quickly turn hurtful towards the quality of the product and the ASD process itself (Zazworka et al. 2011). As autonomous teams tend to lean towards building up debt, the complexity of the product will also increase accordingly (Yli-Huumo et al. 2016, 195).

Because of the severity TD can have on ASD processes and product releases, it can be argued to be crucial for companies to know how autonomous teams should manage the down payment of accumulated debt. One could easily argue that a simple solution for doing this is to repay the known TD as soon as possible. However, autonomous teams are often restricted in terms of team size and resources, as well as shorter deadlines and tight schedules. This often makes it difficult to focus on repaying the debt, which will not provide customers with any noticeable results or expansion of product functionality.

Therefore, companies must know how to manage TD early and ensure that autonomous teams have a strategy to repay the debt as soon as possible.

TD has been under the spotlight of IS research for a longer time. Studies have reported that technology companies struggle with managing their TD. Because of TD's growing interest in size, this has resulted in autonomous teams not using their time on new feature development and has severely impacted companies' speed of innovation (Verwijns 2018). Challenges like these have raised attention to IS and software engineering practitioners, and thus the field of TD management (TDM) became its own topic within research. As a result, researchers have tried to develop TDM frameworks for understanding and improving companies' TDM processes. Still, most of these frameworks seem to mostly live within the borders of research journals and have not received adequate empirical validation. In addition, there seems to be a general lack of qualitative studies on autonomous teams and their work towards managing TD (Nielsen et al. 2020, 12). Because of this gap in research, it is essential to further explore the field of TDM and test TDM frameworks created by researchers to find whether these could be used by practitioners for understanding how they can get away with TD. This study, therefore, aims to use Yli-Huumo et al.'s (2016) TDM framework as a theoretical lens in order to answer the following research question:

How do autonomous teams actively manage their technical debt?

This dissertation will answer the research question by conducting a multiple-case study on how one of Norway's biggest actors within the fintech industry is managing TD today. This study aims to further study the concept of TD management, empirically test the TDM framework by Yli-Huumo et al. (2016), and contribute to the field of research and practice of TDM by investigating how autonomous teams actively manage TD and whether the TDM framework could work as a reliable tool for practitioners to use in order to understand and improve TDM processes.

The study will have the following structure: First, a synthesized literature review will occur, presenting and synthesizing relevant literature surrounding TDM and the TDM framework. Secondly, the research method will be presented, explaining how the project got executed. Thirdly, the findings and analysis will occur, where the findings will be organized according to the theoretical framework. Fourth, a discussion will be presented, where the findings will be discussed against the theoretical framework and previous research. Lastly, the conclusion will briefly present the study's implications, limitations, and suggestions for further research.

2.0 Literature review

This chapter will present key topics and concepts which is relevant for this study. The first three subsections will present theories and research on autonomous teams and TDM, while the fourth section will present the TDM framework, used as the theoretical proposition.

When researching the topic of agile software development and technical debt, I initially approached collecting literature through Kristiania University College's library catalog Oria and using different databases such as Wiley and Web of Science. In order to broaden the parameters of my search, I found the need to use more open search engines for exploring the field of research and used Google Scholar to explore the field further. In order to effectively find relevant literature, I specified keywords I wanted to find in articles, such as "agile," "technical debt management," and "autonomous teams," to narrow the search results down. To ensure that collected literature can be perceived as legitimate and peer-reviewed, I also aimed to direct my searches towards well-reviewed journals in the field IS research referred to as the Basket of Eight-journals, such as MIS Quarterly, Information Systems Research, and Journal of Information Technology.

This assignment will followingly present a concept-centric structured literature review to present the literature in a logical and reader-friendly way. Webster and Watson (2002) suggest organizing literature by concepts rather than conducting an author-centric literature review, caused by that author-centric literature reviews present a summary of relevant articles and fail to synthesize them properly (16). Therefore, the following presented literature and findings will not be sorted after author, but by concept.

2.1 Agile software development and autonomous teams

Agile software development (ASD) is fundamentally based on *The Agile Manifesto* (Agile Alliance 2020) in which takes a point of departure from a set of principles focusing on (1) individuals and interactions, (2) working software, (3) customer collaboration and (4) responding to change in software development. As a response to better the traditional software development methods, focusing more on linear workflows, ASD methods improves this method by addressing changes through iterative development cycles, focusing on the creation of incremental deliverables, characterized by a continuous integration of changes (Dybå et al. 2014, 281). ASD methods such as Scrum (Kniberg 2015), eXtreme Programming (Beck 1994), and Kanban (Huang and Kusiak 1996) are some of the most practiced methods to date, and are practiced by innumerable companies worldwide in their work towards creating innovative digital services in a plethora of different industries.

For companies to success in the agile work practices, they have to find ways to approach and regulate their teams' degree of autonomy accordingly with the environmental dynamism. Looking at team

autonomy in the context of ASD, it is reported as key in order to achieve agility (Lee and Xia 2010). Autonomous teams, often referred to as *self-organizing teams*, *empowered teams*, or in this dissertation just as *teams*, is a central part of working agile towards software development, and teams are approaching ASD through a higher level of self-driven work (Moe et al. 2008, 76). Guzzo and Dickson (1996) explains autonomous teams as employees who often performs interdependent jobs, are identified and identifiable as social unit in an organization, and are given authority and responsibility for many aspects of their work. Due to their high degree of autonomy, independence, leadership, dedication, and collocation, they are argued to often be better suited for innovation and new product development (Patanakul et al. 2012, 734).

The teams are often composed of team members coming from different work practices, and usually consists of one product manager, one tech lead, and developers. However, there are several other roles companies often include in these teams, such as testing engineers, engineering managers, UX designers, and agile coaches. Their responsibilities can be highly technical, or more human-process-centric, or both. The possibilities of team compositions are many, and as a result, these teams has the ability to react and adapt quickly in dynamic environments, without the disruption from higher levels in the organization, such as from reviews or resource bureaucracy (Patanakul et al. 2012, 734).

2.2 Technical debt

Technical debt (TD) is a concept used by both scholars and practitioners in which refers to sub-optimal technical solutions expressed in code (Rolland and Lyytinen 2021, 6723). Even though TD has gotten significant attention in IS research within the last two decades, the concept was first introduced by Ward Cunningham in 1992 to communicate consequences emerging from poorly developed software to non-technical product stakeholders. He stated in his article that “Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt” (Cunningham 1992, 30). The word “debt,” borrowed from financial terminology, is a metaphor used to symbolize the relationship short-term benefits from inadequate software development maintenance tasks, and its consequential long-term costs (Guo and Seaman 2011, 31).

TD was earlier perceived as a metaphor for bad code and compromises on code level of software development but has throughout the last decade been refined and extended within research in order to describe a plethora of other variations of debts related to hindering deployment, selling, or evolving anything software development related (Cunningham 1992; Kruchten et al. 2012, 18; Tom et al. 2013; Yli-Huumo 2016, 196). The metaphor describes a situation where software developers accept

compromises in one dimension (e.g., maintenance in backend-code) in order to meet an urgent need in another dimension (e.g., delivery deadline) (Cunningham 1992). Looking at existing research on TD, the metaphor has also shown its presence in several forms, where some of the most common forms of TD are requirements (Brown et al. 2010), design (Zazworka et al. 2011), architectural (Nord et al. 2012; Martini et al. 2012), process (Lim et al. 2012), documentation (Kruchten et al. 2012), and people debt (Kruchten et al. 2012). These all share the same demonstrational effect of “cutting corners” in different stages in a software development process. For example, when developers are in a hurry, they can end up writing code of lower quality unintentionally due to lack of experience. Therefore, by delaying certain maintenance tasks or less carefully, the developer can focus more on functionality rather than the quality of code. This enables teams to deliver their product quicker to customers, which benefits them in the fast-paced market. However, it will result in the team having to repay the debt they have accumulated in the future, which can be higher costs due to their code being more complex and less understandable (Guo and Seaman 2011, 31).

Martini et al. (2014, 87) present several ways that TD can be accumulated in their literature. Even though it can be overwhelming with a walkthrough of all of them, it can be beneficial to understand a few of them to understand its emergence. The most prevalent TD accumulation factors entail different business factors. One of the most typical ways is the uncertainty of use cases at the beginning of ASD projects, where teams often define designs for their products that do not consider potential variability. Other business factors accumulating TD can be time pressure with its deadlines and its penalties. Martini et al. (2014, 87) exemplified this as being time constraints in contracts where delayed product delivery can result in penalties, making teams paying less attention to managing TD. Another typical case of TD accumulation is through design and architecture documentation, where architectural requirements are not well defined in the documentation. This can cause misinterpretations made by the software developers deploying code and making it not match these defined requirements to the system (88). This type of accumulated TD can also be threatening refactoring activities and their estimation.

Despite TD’s often negative perception, creating TD can also be used strategically if done correctly. Caused by TD’s function of time-saving, many teams choose to use TD as a strategy to transform from a highly beneficial ASD strategy into a counter-productive way of value-delivery if the accumulated debt is too high. Therefore, one could argue that product managers need to find ways to balance the ambidexterity of handling incurring TD and facilitate further innovative iterations of ASD (Guo and Seaman 2011, 31). Despite “debt” meaning something one has to pay in the long-term of a project in financial contexts, debt in software development causing no defects or harms in the system will not do anything else than saving software developers’ time as they are developing (Guo and Seaman 2011, 31). By producing smaller amounts of debt, teams can speed up their development processes in the short term (34). Also, if teams aim to produce a system or a module not needing future updates and

maintenance, choosing to produce quicker codes for prioritizing functionality in front of quality and design can be highly beneficial without any long-term consequences (31). Therefore, many teams use TD as a common strategy for shortening development processes and create faster deliveries. It can be beneficial if the teams know that the TD will not affect their project in a long-term perspective. Unfortunately, product owners rarely have this knowledge beforehand, and TD can therefore act as a crucial penalty for finding shortcuts in software development processes (31). Literature seems to agree that creating TD is never seen as an optimal solution for anything (Yli-Huumo 2016, 197). Therefore, teams must be sure that they will be able to repay it for the strategy to be beneficial, and know how to manage their TD processes.

2.3 Technical Debt Management

Technical debt management (TDM) can be explained as ways to manage, prevent, track, measure, and reduce technical debt (Yli-Huumo et al. 2016, 197). TDM can be conducted in many ways, and the concept has gained a raised degree of attention within IS research. Generally, research seems to show clear indications of TDM providing teams with several benefits. For example, Guo et al. (2011) found that not carefully analyzing identified TD items could aggravate the harmful effects of TD. However, a study conducted by dos Santos et al. (2013, 10) identified that TD monetization and representation worked as motivational factors for teams. In addition, Martini et al. (2016, 165) also found a set of benefits of actively tracking and managing TD in software projects, where having a dedicated TD backlog provided teams with long-term perspectives on their development process as a whole, and not a short-term as only having a feature backlog would provide.

2.3.1 TDM activities

Li et al. (2015, 204) found that TDM's current state-of-the-art seems to identify eight different activities present in TDM research, as well as in practice in software engineering practice and autonomous teams: (1) identification, (2) measurement, (3) prioritization, (4) prevention, (5) monitoring, (6) repayment, (7) documentation, and (8) communication. An overview of them is presented in table 1, and each TDM activity is more elaborately explained in the following paragraphs.

Overview of TDM activities
TD identification entails detecting TD from intentional or unintentional technical decisions through techniques, such as testing code coverage.
TD measurement quantifies the benefit and cost of identified TD in software projects. This can happen through estimation techniques.
TD prioritization ranks known TD accordingly after defined rules and techniques used for ranking. Here, the purpose is to investigate which of the identified TDs should be repaid sooner than others.

TD prevention is a set of actions or techniques used to hinder the accumulation of TD in a software system.
TD monitoring entails visualizing TD and observe its cost and benefit over a time-span.
TD repayment refers to techniques in which has as purpose to resolve accumulated TD, for example through refactoring.
TD documentation is used as an activity in TDM that represents TD in a unified manner, addressing potential challenges of relevant stakeholders.
TD communication regards making TD visible to stakeholders in order to discuss and manage them further.

Table 1. Overview of TDM activities suggested by Li et al. (2015)

TD identification entails detecting TD in code through using techniques or tools. Research shows that the activity happens differently in teams. In some teams, TD identification has earlier been conducted through structured processes. Findings seem to show the use of both dynamic code analysis techniques, such as unit testing and code coverage tests, and dynamic code analysis techniques, such as investigating code complexity, code duplication, and design properties (Gat and Heintz 2011). Several tools have also been created to actively identify TD, whereas the open-source application The Sonar Tool has been used by companies in order to get indications of TD in their codebase (SonarSource 2021). However, research shows that identification also is conducted manually by developers inspecting code (Yli-Huumo et al. 2016). Research has investigated whether using tools for identifying TD is better than human elicited TD identification. Results seem to show tools being helpful in identifying defect- and code-related TD. In contrast, human elicited identification works better for many other types of TD caused by its increased accuracy and contextual understanding, which is challenging to get from analysis tools (Zazworka et al. 2013).

TD measurement entails quantifying the benefit and cost of identified TD through estimation techniques. Measuring TD can be conducted in a number of ways, where one of the most common methods are measurement conducted through informal discussions based on a hunch and simple data. Here, the ways of measuring TD can range from basing measurement on simple data gathered from management tools, for example, looking at the number of TD-related issues in Jira (Yli-Huumo et al. 2016). Other teams has also conducted measurement through data analyses tools, such as SonarQube (Yli-Huumo et al. 2016; SonarSource 2021) in order to gain various data, and use this to quantify their TD-issues based on this data. However, research seems to identify this as a challenging aspect of TDM caused by the enormous differences in sizes of TD-issues. As Kruchten et al. (2012) points out, TD could be smaller bugs as well as architectural and structural issues.

TD prioritization refers to ranking identified TD based on different factors to understand which TD-issues should be repaid sooner than others. TD is viewed by many as having an equal or higher severity level compared to new feature tasks (Bavani 2012; Codabux and Williams 2013). There are several ways of conducting TD prioritization. Prioritization often happens using tools or by discussing it with the team. The decisions in discussed TD prioritization is often based on a hunch, or by the team prioritizing based on discussing the TD-issue's potential impact factors, such as severity level, customer satisfaction, and surface impact (Yli-Huumo et a. 2016; Ramasubbu et al. 2015). Other teams have practiced prioritization based on cost-benefit analyses of TD in order to prioritize their TD (Zazworka et al. 2011). Research has also found that some teams conduct TD prioritization based on technical calculations, such as by conducting test coverages in order to quantify the TD and thereafter use this to understand the TD-issue's severity level (Seaman et al. 2012).

TD prevention entails actions practiced by the team for avoiding TD accumulation. This can be conducted through numerous practices to implement into development routines. For example, some of the most common and easiest ways of executing TD prevention is by teams having coding standards and code reviews throughout their development process (Yli-Huumo et al. 2016). This entails developers more cautious of maintaining quality in the code they write, and brings several perspectives on the code being deployed. Test automation is another common prevention strategy and has proven to reduce TD accumulation (Bavani 2012; Gat and Heintz 2011, Codabux and Williams 2013). By automating tests entailing checking code coverage, developers can easily search for indication of bad code and, after that, effectively go back and fix it. However, teams must understand and execute these test automation processes correctly, and the team should collaborate in the designs of test automation scripts (Bavani 2012). Behutiye et al. (2016) also found that many teams practice several programming practices as prevention practices. These practices range from pair programming, test-driven development, and continuous integration (Stolberg 2006; Birkeland 2010, Nord et al. 2012). Through these practices, developers can also better their communication and develop an understanding of TD and its required management.

TD monitoring is all about visualizing TD to present its cost and benefits over a time span, and can help the team effectively with maintaining control over their TDs evolution, as well as to communicate its evolution to other stakeholders. There are several ways this can be conducted, whereas some examples are having track lists of architectural and design decisions in the backlog (Abrahamsson et al. 2010; Bellomo et al. 2013), a TD visualization board (Nord et al. 2012), pie and bar charts (Power 2013), or use TD visualization tools to detect code violations (Hanssen et al. 2010). These ways of visualizing TD will assist teams with identifying, tracking, and manage their TD in a more organized way.

It seems to be popular in TDM research and could be reasoned by TD monitoring working as motivational factors for teams (Santos et al. 2013, 10). Guo et al. (2011, 531) found that not carefully analyzing identified TD items could aggravate the harmful effects of TD. However, by monitoring identified TD-issues, findings seemed to lower their negative impact.

TD repayment is what kinds of techniques teams use in order to remove their TD. Research suggests that refactoring and rewriting code are the most common ways for TD repayment (Pérez et al. 2020; Codabux and Williams 2013). Despite these practices seeming like straightforward practices to conduct, it requires developers to be of a higher skill level, and teams can often not afford to use all their time focusing on refactoring code (Yli-Huumo et al. 2016). When it comes to the execution of TD repayment in development processes, research seems to show that it often is conducted by developers either (1) during TD's evolution or (2) by teams assigning weekly/monthly percentages of their time dedicated to repayment (Digkas et al. 2018; Martini et al. 2016). However, most teams seems to not follow assigned percentages for repaying their TD, and mostly conducts it as it is found in the code base. (Ernst et al. 2015).

TD documentation entails how teams represent their TD in a documented format. Teams often seem conduct this using a backlog management tool, such as Jira (Yli-Huumo et al. 2016, 211). However, teams can use such management tools to different extents for TDM. In some teams, it seems as if TD is documented just like regular features are documented in the backlog. Here, teams often does not practice any structured processes, and developers often only document TD-issues they perceive as important (Codabux and Williams 2013, 13). However, other teams also has seemed to use their backlog management tool more extensively, and follows defined protocols for documenting their TD in separate TD backlogs. Research seem to show clear advantages of having highly structured processes for documenting TD, caused by its ability to provide teams with clearer long-term understandings of their development processes, as well as avoidance of neglecting undocumented TD in which could later evolve into significant faults in the codebase (Codabux and Williams 2013, 13; Stettina et al. 2011, 164).

TD communication entails how TD is communicated and visible to internal and external stakeholders in a way for it to be further discussed and managed. This activity seems to be a popular topic within TDM research caused by its crucialness in development processes (Li et al. 2015). TDM communication is mostly conducted by teams through informal meetings or conversations, or by setting up TD-dedicated meetings or having it as a discussed topic in meetings (Yli-Huumo et al. 2016, 210). This is argued to provide teams with better control over their accumulated TD, and will also make sure that all team members and business stakeholders are a jour with their TDM (Klinger et al. 2011, 35). However, research has also highlighted a communication gap between developers and non-technical stakeholders as a significant challenge in TDM (Yli-Huumo et al. 2014; Klinger et al. 2011, 35). TD-related issues

has shown to not translate well to non-technical stakeholders, and consequently could result in teams not getting adequate time and resources for repaying their TD (Yli-Huumo et al. 2014). Therefore, arranging weekly meetings for communicating TD is a helpful means for product managers to receive clearer feedback from developers and give the team a clearer image of how they should prioritize TD (Martini et al. 2016, 165).

2.3.2 Challenges related to TDM

Despite literature mostly projecting TDM as beneficial for teams to practice, research has also identified several challenges with both practicing and implementing it. For example, Martini et al. (2016, 165) found that implementing a new TDM method requires substantial amounts of effort and resources in order to organize and collect all existing TD. Despite it being a one-time situation of collecting TD items, several of the researched teams did not have enough space in their budget to implement it. In addition to taking up substantial amounts of resources, some has also found that working with TDM can create more work top of the existing development work (Yli-Huumo et al. 2016, 213). Hence, many teams find it challenging to warrant the need for it and its beneficial purpose. Besides challenges related to cost and time-consumption, research has also shun light on developers experiencing that TD repayment and TD prioritization became more bothersome caused by product managers not having the same perspective of TD items' risks and benefits, and not being able to calculate risk/impact accurately on all items (165). Power (2013) has tried to summarize challenges related to TDM, and identified seven common challenges: (1) developing a common understanding of technical debt, (2) neglectation of technical debt over several releases, (3) understanding the cost of delay, (4) visualizing technical debt, (5) quantifying technical debt, (6) tracking technical debt, and (7) understanding technical debt as a root cause of challenges.

Current literature suggests some tools and processes that can be used in order to manage TD (Martini et al. 2014, 57). However, these have been shown to be challenging to implement, caused by product managers and developers not knowing how to estimate and identify accumulated TD, how it can change, and what consequences it will have in the future (Li et al. 2015). In addition, several studies seems to show that tools for practicing TDM are limited, and seems to be a common reason for several teams to not practice TDM (Yli-Huumo et al. 2016, 212; Ernst et al. 2015, 56).

In addition to challenges regarding the practice and implementation of TDM, research also seems to show indications on TDM being limited in practice. Martini et al. (2016, 163) found that teams averagely spend 25% of their development time on TDM. However, looking at a study on TDM conducted by Ernst et al. (2015, 56), they investigated 1,831 software ASD practitioners' relationship to TDM and found that 65% of the participants had no defined TDM practices - and of the remaining 35%, only 25% of them managed TD at team level.

2.4 TDM framework

Several researchers have created frameworks for investigating teams' performance in software development. However, empirically tested TDM frameworks severely lacks. However, Yli-Huumo et al. (2016) conducted a case study on autonomous teams and their approach to TDM, and found similarities in their findings compared to earlier maturity frameworks used in software engineering research. Paulk et al.'s (1993) Capability Maturity Model (CMM) has worked as a framework used by software engineering practitioners for decades to understand and improve software development in organizations through a systematic classification system with suggested sets of recommended practices in different process areas. Paulk et al. (1993, 19) point out that the CMM "guides developers to gain control of their development and maintenance processes, and how to evolve toward a culture of software engineering and management excellence." The model is designed to help practitioners select improvement strategies by investigating their team's maturity level. Paulk et al. (1993, 19) further point out that to use the framework, one must understand the difference between immature teams and mature teams. Immature teams refer to teams conducting software development activities being generally improvised, without any enforcement. Mature teams refer to teams having a stronger ability to manage development and maintenance, where managers accurately communicated the development process to different stakeholders, while development activities are carried out according to a planned process (19).

The implementation of CMM has gotten attention in research, and there exist several examples of CMM-implementation generating better team performance and higher time efficiency in software development (Astakhova et al. 2016; Osipov et al. 2015). In addition, other research seems to indicate that CMM implementation indeed improved the development process performance in terms of quality management (Titov et a. 2016, 4). Caused by the CMM's empirical validation from practitioners within software development and raised attention in research, the CMM has undergone several iterations of improvements, and several researchers have used it as a guiding star for creating new standardized frameworks towards maturity within different fields of software engineering, whereas the Test Maturity Model Integration (TMMI) for improvement of software testing practices (Garousi and Veenendaal 2021, 1), and the People Capability Maturity Model (PCMM) for successfully addressing critical people in an organization (Curtis et al. 2009, vi).

When Yli-Huumo et al. (2016) conducted an empirical study on teams' TDM and measured their degree of practice in Li et al.'s (2015) suggested TDM activities, they found that the teams practiced TDM at various levels and found similarities in their findings with the CMM's maturity levels. Some teams had no defined strategy for managing and reducing TD, while others had more organized processes for reducing, monitoring, measuring, and managing their TD. They, therefore, applied these findings to

create a similar maturity model addressing teams' TDM maturity. As a result, the TDM framework was created (see table 2). By categorizing the identified TDM activities into three different maturity levels, the framework shows a team's abilities in each TDM activity.

TDM activity / TDM levels	Repayment	Prevention	Documentation	Identification	Measurement	Monitoring	Communication	Prioritization
Level 3 Organized	Continuous repayment with monthly assigned percentage of development tasks.	Mandatory prevention practices used by the team. Continuous practice during development.	Documentation is a mandatory practice in development. Issues are documented in a separate TD backlog.	Continuous identification conducted manually and/or with tools during development.	Continuous measurement during development. Data analysis (various data used (e.g. quality, performance)). Assisted with tools.	Continuous monitoring during development with various data (e.g. quality, performance). Tools used to support.	Continuous discussions/meetings about TD issues with all the necessary stakeholders involved.	Prioritization conducted continuously during development. Prioritization follows a specific method or model.
Level 2 Received	Repayment during normal development tasks and previously identified repayment tasks. Repayment conducted based on current needs.	Optional prevention practices. Not mandatory to use, but recommended. Conducted based on current time constraints.	Documentation an optional practice, but recommended. Issues documented in a general development backlog without TD id.	Identification optional during normal development. Conducted based on current time constraints.	Measurement an optional practice. Measurement done with simple data (number of TD issues) from development, and the data not necessarily used for other activities.	Monitoring based on simple data (number of TD issues). Conducted occasionally.	Discussions/meetings organized only with some stakeholders.	Prioritization based on hunches and rough estimations based on previous experiences. Prioritization done in a simple way without any specific model.
Level 1 Unorganized	Repayment not conducted at all or only when it is not possible to avoid the issue any longer.	Prevention not assigned as part of the development practices. Conducted only occasionally.	Documentation not part of development. Issues are left in developers' own minds and notes.	Identification practices not assigned as part of development. Conducted only when issues occur. Development team, software architect(s)	Measurement not part of development practices.	Monitoring not part of development practices.	TD not a topic in discussions/meetings and often handled only in coffee table discussions.	Prioritization not conducted, and decisions done without reasoning or discussions.
Responsibility for activity	Development team, software architect(s)	Development team, software architect(s)	Development team, software architect(s)	Development team, software architect(s)	Software architect(s), team manager	Software architect(s), team manager	Development team, software architect(s), team manager	Software architect(s), team manager
Practices / Tools for activity	Refactoring, Redesigning, rewriting	Coding standards, code reviews, Definition of Done	Technical debt backlog/list, Documentation practice, project management tool (Jira, Wiki)	Time reservation for manual code inspection. Use of code analysis tools (SonarQube, CheckStyle, FindBugs).	Data from measurement tools (SonarQube) and data from project management tools (JIRA, Wiki).	Monitoring tools (SonarQube). Project management tools (JIRA, Wiki)	Specific TD meetings, TD included in discussion topics.	Cost/Benefit model, Issue rating

Table 2: TDM framework

Yli-Huumo et al. (2016, 196) explain that the TDM framework was created to create a generalized framework for companies to use to understand their current state of TDM practice and improve them. The TDM framework addresses five core elements: TDM activities, TDM levels, TDM stakeholders, TDM responsibilities, and TDM approaches (Yli-Huumo 2016, 209). Li et al.'s (2015) TDM activities are listed in the first row of the framework, creating eight different columns.

The TDM framework presents three maturity levels: unorganized, received, and organized. At level 1, the lowest-ranked maturity level is the unorganized maturity level. This refers to teams who are not putting effort into a TDM activity or when the focus of practicing it is minimal. This includes teams only conducting practices whenever they have to, if at all, in order to address the TDM activity. At level 2, the mid-ranked maturity level is the received maturity level. This refers to teams practicing the TDM-activity to a certain degree and when the team has acknowledged the value of practicing it. This could entail teams conducting TDM practices occasionally, often if it fits within their current time constraints. At level 3, the highest-ranked maturity level is the organized maturity level. This refers to teams who continuously practice the TDM activity and have recognized it as an essential part of their development process. These teams seem to have an active relationship towards the TDM activity and often dedicate parts of their development processes to conduct them.

The framework also presents TDM responsibilities and TDM practices/tools. In TDM responsibilities, Yli-Huumo et al. (2016) identified three primary responsibilities: most often seemed responsible for different TDM activities: The development team, software architects, and team managers. In addition, they also identified a responsibility closely related to TDM, which were business stakeholders. The business stakeholder is, however, mostly related to TD communication. Also, TDM practices/tools represent sets of identified approaches teams or individuals could use to practice specific TDM activities.

Yli-Huumo et al. (2016, 210) point out that the TDM framework can help companies evaluate and improve their internal and external TDM processes and improve these processes. However, the TDM framework lacks empirical validation. Yli-Huumo et al. (2016) emphasize that the framework has not been tested in a way that validates its use for improving TDM processes. This could therefore mean that teams practicing less TD monitoring and prioritization could still have less accumulated TD than teams spending more resources into doing so. Therefore, one could stress the need to empirically test the TDM framework to develop a better understanding of if the framework could be used the same way as CMMs for understanding and improving software development processes.

3.0 Research design

As mentioned in the introduction, this study aims to provide software development practitioners with insights into how technology companies work towards handling TD to develop a better understanding. In order to discuss the chosen research question, it thus seemed relevant to choose a qualitative study for answering the research question and conduct an exploratory multiple-case study. This chapter will further elaborate on the choice of research design, development of research instrument, description of the data collection and data analysis process, and present a case description of the case company.

3.1 Multiple-case study

This study intends to investigate teams and understand how they work towards TDM, which made the choice of using a case study an appropriate decision. Case studies are traditionally used in social science in order to study individuals or groups, focusing on factors, issues, politics, processes, and relationships generating “messiness in this world” (Oates 2006, 142). The research design lets the researcher test, validate, and develop theories (Anderson 1983; Kidder 1982; Eisenhardt and Graebner 2007), and are often used in order to understand the “what,” “why,” and “how” in research (Yin 2014, 3-11). Runeson and Höst (2009, 135) explain that case studies have a flexible design, meaning that parameters in the interview guide may be changed throughout the data collection process if needed, as opposed to quantitative research designs having a fixed research design. In addition, practitioners have emphasized that practicing case studies is a suitable research design for IS- and software engineering research caused by its ability to study a phenomenon in their natural contexts (Runeson and Höst 2009, 131).

There are two basic types of case study designs for explanatory purposes in research: single- and multiple-case study design (Yin 1981, 100). In research conducting a single-case design, one can investigate a phenomenon and test theories on individuals from a specific context. This could be to study team members of a team, working towards the same case. However, a multiple-case design is conducted by concluding a group of cases. This is argued to be appropriate when a phenomenon exists in several situations, and the researcher wants to find more generalized results rather than case-specific results (Tellis 1997). The results from each interviewed individual may differ, but the findings are supposed to provide a basis for validating results. Case studies also often get classified into three different case study types serving different purposes: exploratory, descriptive, and explanatory (Robson 2002). Table 3 summarizes the classifications with brief descriptions.

Exploratory	Finding out what is happening, seeking new insights, and generating ideas and hypotheses for new research.
Descriptive	Portraying a situation or phenomenon

Explanatory	Seeking explanations for certain situations or problems, mostly but not necessary in the form of a causal relationship.
-------------	-------------------------------------------------------------------------------------------------------------------------

Table 3. Overview of case study classifications

Withal, Runeson and Höst (2009, 135) explain that case studies were primarily found in the form of the exploratory classification, “focusing on finding out what is happening, seeking insightful data, and generating new hypotheses for future research.” The study intends to investigate how teams in a chosen tech company are handling TDM and figure out how this could be approved. However, research surrounding those topics and validating suggested frameworks and theories are inadequate. In addition, the literature emphasizes a need for additional empirical studies on TDM to increase the knowledge of TD (Nielsen et al. 2020, 12). Oates (2006) emphasizes that in research areas containing limited amounts of literature, explorative studies are recommended as a research strategy. Therefore, choosing to practice an exploratory approach towards researching individuals in their natural context could be argued to be done to obtain a better understanding and develop suggestions for future research. Appropriately, I chose an exploratory multiple-case study as the research design for this study.

When it comes to the practical features of the study, it is a distinct technical situation, whereas several variables are considered, and there are several sources of information the researchers can investigate (Yin 2014, 16). The data is usually collected through interviews and is often combined with supplementary data from archival documentation, observations, and physical artifacts (Oates 2006, 142). As this paper aims to answer a research question entailing a topic that the chosen company is strongly affected by, a case study approach will be beneficial as the company I am investigating is constantly dealing with TD in their development processes. In addition, the company contains multiple stakeholders of interest, with their own important contextual conditions. By, therefore, choosing a case study as the way of researching and answering the formulated research question, the researcher will gain a better illustration of the circumstances and context which teams in the company are a part of, as well as obtaining good and broad knowledge from the team members (Cousin 2005).

Choosing a case study may bring challenges. These types of studies are criticized by research for not generating results that can get generalized outside the specific researched context. This means that the findings from this study may not be relevant for other similar cases caused by its context. However, some literature seems to bring contradicting arguments. Walsham (1995, 79) suggests the following ways case studies can generate generalized outcomes. Case studies can generate (1) conceptualizations, where new ideas or notions emerge from the analysis, (2) theories, which can get translated into conceptual frameworks, (3) implications, in which in this case can be directly practiced by other companies, and (4) richer insights, where new understandings about situations are generated.

3.2 Construction of research instrument

Yin (2014) points out that a crucial strategy for conducting case studies is to follow the theoretical propositions that the hypotheses lean on. Since the study uses Yli-Huumo et al.'s (2016) TDM framework addressing Li et al.'s (2015) TDM activities as the theoretical framework, it thus seemed rational to use these theories as a point of departure when constructing the interview questions for this study. Therefore, the research instrument had a structure constructed out of the TDM framework to measure the teams' performance within TDM activities. In addition, I also wanted insights into the respondents' understanding of TD as a concept and their thoughts on their satisfaction with how they are handling it as of now, and their motivation towards TDM. The research instrument, therefore, had the following structure: (1) Introduction to research project, (2) TD understanding, (3) TDM activities, (4) TDM motivation.

The questions in the research instrument got constructed with efficiency in focus. In order to gain maximum data from each question, the research instrument for this study, therefore, took inspiration from McNamara's (2009) recommendations when designing efficient qualitative interview questions: (1) the questions should be open-ended, where the interviewees should be able to answer the questions however they want to, (2) the questions should come from a neutral point of view, and (3) avoid using the word "why" in the question formulation.

The research instrument is found in Appendix B.

3.3 Data collection

As earlier mentioned, the main data source in case studies comes from interviews and is often supplied with additional data from formal documentation, observations, and physical artifacts (Oates 2006, 142). In this study, the main data source was through semi-structured interviews. The reason for choosing semi-structured interviews as the interview method was because of limited knowledge about the case company and their teams beforehand, making it challenging to curate a fixed set of questions. I, therefore, found the need for the interview method to be flexible at the core. Runeson and Höst (2009, 145) explain that semi-structured interviews can be designed in a way where questions are planned but not necessarily asked in a fixed order. In addition, the method allows the researcher to improvise and explore the studied object without having to stay within fixed boundaries. Thus, I conducted semi-structured interviews in which fulfilled my wish to explore TDM more freely.

The data collection was performed over two months, and I interviewed eight individuals in total. Each interview lasted for approximately 30-50 minutes, which resulted in a substantial amount of material being transcribed and analyzed. After the interviews were conducted, the transcriptions were sent to the

interviewees in order for them to be able to provide feedback in case of being unclear with their answering, misunderstandings, or if they wanted to fill in with more input. An overview of all the interviewed participants is listed in table 4 below. Five different roles were interviewed, caused by my wish to collect data from different perspectives, representing teams as a whole and not from one specific work practice.

Team	Role	Duration
A	Product manager	40m 10s
B	Tech lead	53m 32s
C	Test automation engineer	52m 42s
D	Engineering manager	50m 12s
E	Developer	30m 32s
F	Tech lead	31m 09s
G	Product manager	33m 16s
H	Tech lead	30m 29s

Table 4. Overview of interviews.

3.3.1 Transcription of video recordings, coding, and data analysis

Researchers emphasize that case study interviews should be recorded in a suitable audio or video format (Runeson and Höst 2009, 146). In order for me to focus on the interview, I captured video recordings of the interviews. The video recordings were transcribed directly after the interviews were conducted in order for me to process the data that was collected and reflect upon if the questions gave me the data I desired. When the video recordings were completely transcribed, the data analysis took place in order for me to render the data and organize it respectively to my theoretical framework. The main objective of conducting the data analysis is to pull conclusions from data and derive clear chains of evidence (Runeson and Höst 2009, 150). This was done by practicing the “pen-and-paper” approach, where printing out the transcriptions and coding the data began. Coding gives certain parts of the texts a representation of themes, areas, constructs, etc. During the iterative process of coding data, smaller sets of generalizations get formulated and develop a knowledge map (Runeson and Höst 2009, 151).

After the data was coded, the data analysis was carried out as a parallel process. The data analysis took a point of departure from relying on theoretical propositions that led to the multiple-case study being conducted. Yin (2014, 134) explains that this data analysis strategy helps the researcher shape the data collection plan. The theoretical proposition organizes the entire analysis and helps the researcher to easier point out relevant contextual conditions to be described and explanations to be further examined. By therefore using the TDM framework and its TDM activities as the theoretical proposition, the

findings were categorized within its eight TDM activities to create a systematic mapping of each case. After that, the cases went through a cross-analysis, comparing the findings within each category to draw suggested conclusions based on these. The data collection process is illustrated in figure 1.

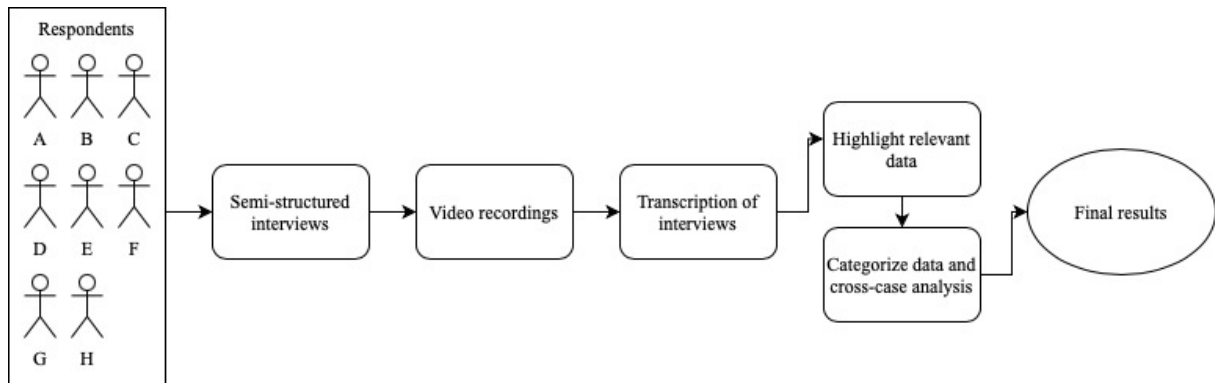


Figure 1. Data collection process

3.4 Validity and reliability

When conducting research, one must be aware of potential pitfalls to avoid and consider when designing a research strategy to ensure a good overall quality of the study (Yin 2014). It is therefore essential to ensure a degree of validity in the research design, which can warrant the researcher and reader with a warranty of trust in the research, as well as providing finding that is true and does not come from a place of the researcher’s subjectivity (Runeson and Höst 2008). There are several ways of addressing validity in research, but Runeson and Höst (2008) suggest four classifications one should consider in software engineering research. These are followingly briefly presented, together with how this study addresses each aspect of validity:

Construct validity entails whether the researcher and the research object have a common perception and understanding of what is being studied. For example, this could be if the interviewed individual did not know what type of TD the research focused on investigating. This was handled by (1) informing the purpose of the research project and relevant concepts in an information letter provided to the respondents beforehand and (2) clarifying the scope of TD and the research project at the start of each interview.

Internal validity is the concern of when causal relations are studied. When one factor affects another, the other factor is often affected by a third factor. Moreover, when the researcher is unaware of the third factor, this could threaten the project’s validity. Since this study bases its data collection on semi-structured interviews, it was easier for the researcher to ask additional questions whenever there was felt a need for better and more fulfilling answers from the respondents. In addition, the respondents did allow me to contact them in later instances if I had additional questions.

External validity entails to what extent the findings could be generalized and of interest by external individuals not involved in the research project. Case studies often try to generate analytical generalizations (Runeson and Höst 2008, 154). However, caused by their qualitative nature, the findings often provide in-depth data gathered from specific contexts, which could be a challenge to the external validity of the project. Therefore, the conclusion is written with a sense of humbleness and avoidance for generalizing the results. In addition, the findings from this study will be thoroughly discussed supported by previous generalized research.

Reliability involves to what extent the data and analysis are dependent on the researcher. If another researcher would sit with the same data material and information as me, they should be able to come to similar conclusions. One limitation here is that conducting semi-structured interviews generally can result in different data being collected caused by new and changing questions. However, this study addressed this through transparency in the method practiced in this study and by describing the data collection process and briefly describing how the data was analyzed. The used research instrument is also attached, so that other researchers can evaluate the questions asked in the interviews.

3.5 Case description

The case study was performed in one of Norway's largest tech companies within financial technology, housing hundreds of employees and delivering services in payment solutions. The company started its venture in the mid-2010s as a startup under another company and grew with tremendous speed. Their initial product reached its first million users a year later and a few years later became an independent company housing hundreds of employees and leading within the fintech industry. In recent years, the company has gotten significant attention both nationally and internationally, and caused by the success, has also expanded their services, and multiplied their product line into a library of products for both public and B2B.

Taking a point of departure from agile practices, the teams in the company work in iterative development cycles and has a Scrum-ish approach to their development model. Caused by their multitude of products and services, their company houses several autonomous teams who are the company's workforce and has sections they are responsible for developing and running. The teams are composed of 3-10 members, with a team composed of a product owner, tech lead, and developers. In addition, some of the teams also have test automation engineers, engineering managers, UX designers, and agile coaches based on their current needs for extra support.

As the company has worked heavily with software development for over five years with their extensive product library, they experience significant amounts of accumulated TD in their projects. With products

constantly evolving, the product complexity increases simultaneously. The products keep going through iterative development cycles, and with new code added, the older ones keep building up as TD. Furthermore, since the company's staff has been exchanged throughout the years, several codebases lack documentation. This has resulted in significant amounts of legacy code, which also was developed in a way that has now become TD. As a result, TD has affected several factors, such as speed of innovation, errors in the codebase, and lack of team motivation. Since their product is today used by millions of users, the case company must ensure constant run-time. In fear of the TD turning the company into a "house of cards," the company has therefore found a need to better understand their current ways of managing TD and a need to improve their processes towards TDM.

4.0 Results and analysis

In this section, the findings are presented. The findings were gathered from eight interviews with eight different respondents coming from their own individual team within the case company. Firstly, the chapter presents a summary of the findings. Secondly, the findings are more elaborately presented in the following subchapter, presenting relevant citations and findings from the cross-analysis conducted on each different TDM activity.

4.1 Summary of findings

In order to get a better overview of the findings gathered from the respondents, the findings are summarized in table 5 as well as in the following paragraphs within this subchapter.

Team / TDM activity	Team A	Team B	Team C	Team D	Team E	Team F	Team G	Team H
TD identification	Developers identifying manually throughout development process.	Developers identifying manually throughout development process.	Developers identifying manually through normal development . Also identified when errors occur in Splunk and Dynatrace dashboards.	Developers identifying manually throughout normal development based on current time constraints. Also through errors in Grafana Dashboards and Splunk.	Developers identifying manually throughout normal development . Also by weekly assigned developer, who's responsibility is to identify and handle TD.	Developers identifying manually throughout normal development .	Developers identifying manually throughout normal development . Also through automated tests for identifying bugs and TD, Also through errors in Splunk.	Developers identifying manually throughout normal development .
TD monitoring	No monitoring.	No monitoring.	Dynatrace, and other graphs/charts for continuous monitoring.	Jira-issues for monitoring occasionally .	No monitoring.	Jira-issues for monitoring occasionally .	No monitoring.	No monitoring.
TD measurement	No measuring.	Product manager occasionally measured using simple data based on hunch.	Developers measured occasionally using simple data based on hunch.	Team measured occasionally using simple data based on hunch.	No measuring.	Developers measured occasionally using simple data based on hunch.	Team measured occasionally using simple data based on hunch.	No measuring.
TD documentation / representation	Some documented in Jira. No separate backlog.	Some documented in Jira and Confluence. No separate backlog.	Some documented in Jira, Confluence, and GitHub. No separate backlog.	Documented in Jira and tagged as a TD-item for structured overview.	Some documented in Jira and Confluence. No separate backlog.	Some documented in Jira. No separate backlog.	Some documented in Jira. No separate backlog.	Some documented in Jira. No separate backlog.
TD prioritization	Team prioritized without defined process. Based on criticality.	Product manager mostly prioritizes based on hunch. Occasional workshops where the team prioritizes based on criticality.	Team prioritized without defined process, based on surface impact and security issues.	Product manager, tech lead and tech management prioritized without defined process. Usually based on hunch	Product manager and engineering manager prioritized usually based on urgency and security implications .	Team prioritized based on customer value and surface impact.	Product manager and tech lead prioritized based on customer value. Conducts impact mapping with the team.	Team prioritized without defined process. Based on hunch.
TD repayment	TD-items from backlog.	TD-items from backlog.	TD-items from backlog.	TD-items from backlog.	TD-items from backlog.	TD-items from backlog.	Mostly no repayment. Weekly	25% as monthly assigned

	Refactoring and rewriting of code based on current need.	Refactoring and rewriting of code based on current need.	Refactoring and rewriting of code based on current need.	Refactoring and rewriting of code based on current need.	Refactoring and rewriting of code based on current need. weekly assigned developer to identify and repay TD.	Refactoring and rewriting of code based on current need.	assigned developer to identify and manage TD.	percentage of repayment. Refactoring and repayment of code continuously through development .
TD prevention	Minimal coding guidelines.	Minimal programming practices. Ran tests and continuous integration. Not always practiced.	Programming practices. Ran tests and continuous integration continuously .	Code reviews, and programming practices. Ran tests and continuous integration continuously .	Code reviews, and programming practices.	Coding guidelines, and programming practices. Not always practiced.	Coding guidelines, code reviews, and programming practices.	Coding guidelines, code reviews, and programming practices.
TD communication	Team actively communicated TD within and outside team boundaries via informal and formal meetings.	Team actively communicated TD within and outside team boundaries via informal and formal meetings.	Team actively communicated TD within and outside team boundaries via informal and formal meetings.	Team actively communicated TD within and outside team boundaries via informal and formal meetings. Some challenges to communicate TD to company management .	Team actively communicated TD within and outside team boundaries via informal and formal meetings. Sometimes communicated outside with relevant stakeholders .	Team actively communicated TD within and outside team boundaries via informal and formal meetings. Sometimes communicated outside with relevant stakeholders .	Team actively communicated TD within and outside team boundaries via informal and formal meetings. Challenges with communicating TD to company management .	Team actively communicated TD within and outside team boundaries via informal and formal meetings.

Table 5. Summary of findings.

Starting with TD identification, one can see that the teams mainly conducted identification throughout the normal development process, by developers usually finding them along the way of developing features, analyzing, or maintaining their codebase. Half of the teams did not seem to use any tools to specifically identify TD (A, B, F, H), while the other half (B, C, D, G) actively used general monitoring software where they could get indications of TD occurring in code errors and as well through crash analyses.

For TD documentation, all teams documented their TD in the Jira to document to different extents. Many of the teams also used other services like Confluence and GitHub to document more comprehensive TD-issues. However, most of the teams did not have any mandatory processes they had to follow to document their TD. However, one team (D) had a system for categorizing TD-issues by tagging Jira-issues to filter out a separate backlog for tracking their TD.

TD monitoring mainly was not practiced in the case company, except for some teams (D, F) did have their way of monitoring TD from simple data, such as monitoring the number of TD-related Jira-issues, and monitored it occasionally based on their current needs. However, one team did seem to track their

TD continuously throughout the development process and had a dashboard, charts, and graphs visualizing TD with its various metrics.

TD measurement was mostly not highly practiced, and some teams did not conduct it (A, E, H). None of the teams had a defined process towards measuring their TD, but some of them did measure their TD individually or in discussions, mostly basing their decisions on a hunch. Some of the teams (B, D, F) occasionally would bring up TD to discussion and measure them using T-shirt sizes as labels in the backlog. One of the teams (C) also used simple data from the Jira-backlog as information for measuring TD, where the developers could use the number of TD-issues as data for measurement.

All teams conducted TD prioritization in one way or another. Most teams seemed to conduct prioritization as a normal part of their development process and did not use any specific models for prioritization (A, C, D, E, F, H). The teams based their decisions on a hunch and estimations. In addition, some of the teams would also prioritize TD based on different impact factors it could have, such as criticality, surface impact, or customer value. However, some teams also conducted prioritization through more structured processes and conducted workshops and impact mapping sessions with the team (B, G).

TD repayment was mostly conducted by refactoring or rewriting the code by the teams. Most teams (B, C, D, E, and F) practiced repayment occasionally as a normal part of their development process and without any strategies towards dedicating time to repayment. Also, some of the teams (E and G) practiced assigning one developer to repay TD in their codebase as their responsibility for a week. However, one team (H) did have a more structured approach towards repayment, had a certain percentage of their work assigned to repay TD.

Most teams actively conducted TD prevention. Most teams conducted code reviews and had coding guidelines the developers had to follow to prevent TD continuously throughout their development processes. In addition, it was identified that most other teams (C, D, E, F, G) prevented TD by practicing different programming practices, such as pair-programming and mob-programming, as a part of their normal development process, whereas the respondents seemed to show indications on this being a trusted and efficient practice for TD prevention.

TD communication was highly practiced in the case company. All respondents seemed to show clear indications on communication to be optimal within the team boundaries, and most teams seemed to have a good relationship towards communicating TD outside team boundaries to other stakeholders. Some teams (D and G) seemed to show signs of communicating TD outside the team as challenging and related this challenge to relate to the teams' motivation to work with TD.

4.2 TDM activities

In order to create a system for understanding the case company’s results within the TDM activities together with their rated TDM maturity level, table 6 provides an overview of the team evaluation in the TDM-framework. The grey colored boxes illustrate which TDM maturity level each TDM activity the findings most prominently projected. By looking at the table, one can see that the case company seems to mainly reflect a received level of maturity. However, the case company also has TDM activities placed within the other TDM maturity levels, organized and unorganized. The findings will be further explained in the rest of this chapter, highlighting remarkable findings and citations. The findings are presented activity by activity and are categorized according to their maturity levels in order to develop a generalized result from the case company’s performance in TDM activities.

TDM activity/ TDM maturity level	Repayment	Prevention	Documentation	Identification	Measurement	Monitoring	Communication	Prioritization
Organized	H	C, D, E, G, H	D	B, C, D, G		C	A, B, C, D, E, F, G, H	B, G
Received	A, B, C, D, E, F	A, B, F	A, B, C, E, F, G, H	A, E, F, H	B, C, D, F, G	D, F		A, C, D, E, F, H
Unorganized	G				A, E, H	A, B, E, G, H		

Table 6. TDM-framework and team evaluation.

4.2.1 Level 1 – Unorganized

TD monitoring

Most teams did not conduct TD monitoring in their development processes, and only three teams (C, D, F) seemed to conduct TD monitoring. Team C used tools, such as Dynatrace and Splunk, to track TD and charts and tables showing TD’s evolution over time and used these to monitor their TD continuously throughout their development processes. This would give insights to the application team and tell developers if it runs as it should. Two of the teams (D, F) also occasionally used Jira as a tool for monitoring.

“You could say that I’ve used Jira recently. If you’re disappointed about creating bugs, and like calling them TD, you could do that and say that’s a way of monitoring TD. Then we can keep track if its growing or not.”

Jira would provide the team with simple data for monitoring if their identified TD would grow or reduce over time. In addition, the Jira-backlog worked as a tool for both the team and tech management to maintain some monitoring of how much time the team/teams would spend on TD compared to time spent on developing new functionality.

However, the rest of the teams did not conduct any TD measurement. These teams mostly based their reasoning being no apparent need for measuring their TD, as well as too high maintenance costs of conducting it:

“We probably could create a system for it, but I don’t really feel the need of doing so, other than other than for reporting instances. Other than that, I feel like it would be to implement bureaucratic processes to things that maybe is unneeded. TD is always on top of our minds, and that creating a framework for presenting that we have solved a 10% of our technical debt would be unnecessary”

Monitoring was not a part of their development practices. Despite some teams emphasizing no need for conducting TD monitoring, some of the same teams did express that they saw the value of monitoring TD in Jira properly. The teams in the case company earlier had agile coaches coming to teams and helps the team to ensure velocity and good flows in their development process. The respondent from team G explained that creating a more defined system for labeling and categorizing Jira-issues with TD to keep track of how much they had of TD compared to other tasks was helpful. The respondent from Team H also expressed that he/she wished his team used Jira more actively and categorized each issue in order to easier create a system for TD monitoring based on simple data:

“I don’t think it would make a big difference to visualize it through monitoring software. I think it would only become lots of maintenance work. But if we could visualize it, not by creating charts, but by categorizing Jira-issues as TD, and had better control of how much we had of it. I think we would benefit from having a clearer strategy on that bit.”

4.2.2 Level 2 – Received

TD measurement

Most teams seemed to practice TD measurement, but more as an optional practice within the team based on their current needs. The measurement was mostly based on simple data, such as the number of TD-related backlog issues.

The teams also seemed to have different approaches for measuring TD. In teams B, D, and F, they would usually discuss identified TD over informal conversations or meetings and base their measurement on hunch and T-shirt sizes. Team C also conducted TD measurement based on which factors TD-items potentially could impact. The longer the list of things it could affect, the bigger the TD-item would believe to be.

“[...] it’s based on how many products we have. And how many of these different things in a checklist that ticks of in the security threats, or lack of documentation, or not being able to scale as much, or not being able to use it to build other products. And if that list is long, I guess that is TD measurement”

In team G, the product manager usually brings up their targeted customer segment’s pains, and the team tries to suggest solutions. Then, the team discusses their TD-issues as they are identified and measures them based on a hunch. However, the respondent further explained that measuring TD would be helpful in some cases, caused by stakeholders needing time estimates for when they would complete their projects, but that it sometimes can be impossible to measure the size of TD-issues they discover. Caused by the vast amount of TD coming from legacy code, the team would often find it challenging to measure TD caused by them never knowing what they would find:

“[...] as I experience it, the TD often encompasses the huge monolith of TD rooting from old codebases, and caused by its significance, it can be very challenging to estimate and measure because we don’t know what we will find.”

The respondent from team F also expressed that they did practice TD measurement but that they did not believe in conducting measurement and estimation as a mandatory practice:

“We try to avoid estimations. We could size the tasks with small, medium, large. But we don’t estimate on a detailed oriented level, or conduct planning poker or stuff like that. We don’t want to focus on the size of something that gets out, but instead the value it could create. [...] One can often get locked within time frames - and I’ve never been a fan of estimating anyway.

Some teams (A, E, H) did not practice TD measurement and did not view it as a necessary or valuable practice for them. This mostly seemed to be reasoned by the feeling of implementing time-consuming processes into something that should be naturally based on a hunch by the team. In addition, some of the respondent emphasized that it was more helpful for them to instead create short-term goals and divide bigger TD-items into smaller tasks in order for it to be more fathomable for developers to handle:

“The simple answer is that we don’t measure TD. But we could usually tell if a task would take one week, or a month. But we don’t have any way of measuring it. That’s why we want to focus on dividing it into smaller tasks, and repay it along the way”.

TD documentation

TD documentation was practiced among all teams. Seven out of eight teams seemed to document their TD and had some control over their identified TD in the product backlog. Most teams did not seem to have a structured way of documenting it, and some of the respondents further stated that they did not see a need for having highly structured documentation protocols:

"TD isn't something one always has to handle. [...] As long as it doesn't create problems now, it's an acceptable risk."

Most teams seemed to follow the same optional protocol of documenting TD. Whenever a developer would identify TD in their codebase, he/she would document it as a Jira-issue in the main product backlog and treat it like any other software feature. All teams seemed to use Jira as their primary TD documentation tool, and most teams would treat TD just like any other feature in the backlog-issue. The TD-issues were usually documented as Jira-issues, subtasks, or epics in the Jira-backlog based on the TD-items perceived size. Some teams would also use Confluence and GitHub to document TD more elaborately and supply the documentation with attachments, such as compliance- or risk matrixes. However, these teams did not have this process of documenting TD as a mandatory practice and only conducted it vocationally based on their current need.

Having undefined TD documentation practices did seem to bring its concerns. This was caused by respondents' wish for developers to document TD to not forget it and provide the whole team with a better overview of their identified and accumulated TD. The respondent from team H stated:

"I wish we had a better way of remembering TD. It's a bit too much in the developers' heads. So having a better structured way of documenting it would help us to get a better overview of TD in our backlog in order to understand how much TD we have, and would also help us to prioritize them and conduct it more methodically."

Only one out of the eight teams seemed to address this concern to some extent in their way of documenting TD. Their main difference was the teams' emphasis on mandatorily having to document TD whenever they would identify it and label them with "TD" to get an overview of the team's TD in a separate TD-backlog. However, most teams did not have mandatory practices when they documented their TD, and one could therefore argue that the case company reflects this TDM activity as strongly received.

TD prioritization

TD prioritization also was practiced by the teams, where all teams conducted it one way or another. The absolute majority of the teams seemed to conduct it based on a hunch, where different roles in each team

had the responsibility of conducting it. The product manager seemed to be mostly involved in the prioritization activities (B, D, E, G), and often would share the responsibility with a tech lead or engineering manager. Also, the tech leads and engineering managers seemed to have the responsibility of prioritizing in several of the teams (D, E, G), and in some teams (C, F, H) the whole team was included in the prioritization activities.

The teams mostly conducted their prioritization in a non-process-oriented manner and seemed to have the activity baked into their normal development processes. In some teams, the prioritization would happen over unformal conversations, and the decisions were often based on a hunch:

“What we often do, is that whenever we would identify TD, even though I now feel like we have good control over our TD, we would usually just immediately discuss how critical it is, and if we should handle it now or if we should wait.”

“We prioritize TD as a part of our regular development process. We don’t sit down and only discuss TD prioritization. It’s a part of the regular development routine.”

“I would say it would be the tech lead and the product manager discussing “These are the things we are dealing with now, and these have these consequences. How do we prioritize these?”. And it really just comes down to balancing time and resources in terms of people. But I think it’s more of a discussion with the team.”

Discussion would often take place in ad-hoc meetings between the responsible parts or in daily stand-ups or weekly team syncs where all team members discuss what they have been working on. Some teams would also conduct TD prioritization based on rough estimations surrounding specific impact factors:

“If the TD has security implications, it automatically has a higher priority. You need to look at what you’ve got time to do, what is most urgent, and what’s the risk of leaving it.”

“Whatever has the biggest surface area. That would be approached first. And of course there are factors of how bug the security issues are. These would be the main aspects.”

“We often prioritize based on insights. We are very keen about insights in this company. We have to talk with the ones who would get affected by TD. This could maybe be done by looking at measurements of how much something is used. If it’s a function that’s used daily that can be affected by TD, then we maybe have to look into it more urgently.”

The teams mostly based their decision on a hunch and considered the many impact factors, ranging from criticalness, surface impact, security threats, available resources, or customer value. However, two teams (B, G) shows signs of conducting it in a more process-oriented manner and conducted activities with the team in order to prioritize TD in which followed more specific models:

“Maybe once or twice a year, we conduct workshops where we try and write down every TD we could think of, and run a small risk analysis of it. Then we ask ourselves “Do we need to handle this TD now?”. And based on that, we prioritize accordingly.”

“It’s mainly me (product manager) who prioritize what tasks that will generate most end-user value. “Here we have the pain-points, we have to solve this, it will provide big end-user value”. Then the tech lead would sit and prioritize how much TD we have left. So it’s mostly me and the tech lead who agrees on what should be prioritized based on these findings. Usually end-user value is prioritized, but sometimes TD would come first. Thereafter, we present it to the rest of the team, and discuss whether these decisions further, and also conduct impact mapping, and look at how much work something gives, vs. how much end-user value it generates. And then we prioritize based on that. We usually base the decisions on hunch.”

However, most teams seemed to base their prioritizations on hunches and estimations. Caused by most teams practicing this occasionally and not using any specific models or approaches for TD prioritization, they were categorized as received.

TD repayment

In TD repayment, most of the teams conducted the TDM activity. In six out of eight instances (A, B, C, D, E, F), the teams seemed to repay their TD as a part of their normal development routines. All the teams who conducted the TDM activity repaid their TD by rewriting or refactoring code. Whenever a developer would identify TD in the code, it would either get repaid immediately or documented in the Jira-backlog for the developers to repay when needed and fit within their current time constraints.

“I don’t even know if we do it intentionally now, but if you’re in a bit of code that has TD, you fix it along the way. And you build it into the time estimates. Of course, if it’s a huge TD, you can’t do it along the way, but it is for smaller things, you definitely do it along the way.”

“It’s really just in the daily work routines. I’m a fan of refactoring as I go, instead of collecting bunches of TD and take it all in one go.”

Some team (G, E) also practiced more creative approaches and weekly assigned developers as a “janitor” or “hero of the week.” Their responsibility entailed investigating identified errors and repay TD as they found them, in addition to repay smaller TD-issues in the backlog that other developers already documented.

Most teams did not have a certain percentage of their development time assigned to repay TD. This further got reasoned by them wanting to eliminate TD “as they go” and whenever it was needed, rather than collecting bunches of it and then dedicate a certain amount of their planned work percentage to repay it. However, one team (H) did have a more structured approach to TD repayment and had as a general rule to assign 25% of their development work towards repaying TD:

“Overall we have a goal of using 25% of our time on TD or maintenance. It’s more of a statement towards the developers for them to know that they can use that much of their time on TD in their work, and not put all men on deck at the same time. We should use that much time on it.”

Some respondents (A, B) expressed a wish for their team to spend more time repaying TD caused by experiencing that developers would usually spend more time working with new features and not as much TD. However, most respondents explained it as unneeded in their TDM approach and experienced their flexible way of repaying TD throughout the process as a well-functioning approach as of now.

“I think it’s difficult to say that “now you’re going to use 25% of your development time on TD repayment”. Personally, I don’t believe in working with assigned percentages. I would rather make the team build a culture for repaying TD whenever it comes. If you identify refactoring needs, take it as you go and don’t neglect it.” (E)

One team (G) did not conduct TD repayment, caused by the team not owning any parts of the codebase they built upon:

“We don’t own any technical components in the application. And therefore we don’t own the company’s TD either. In our work, we pull out code, and build upon it.”

All in all, most teams seemed to practice TD repayment, however, mostly in an unstructured way and mostly based on their current needs. Some teams did conduct repayment more structured by assigning 25% of their development work to repay TD. Another team did not practice TD at all based on them not having a responsibility to repay the TD they identified. The teams, therefore, got categorized as having a received maturity level in the TDM-framework.

4.2.3 Level 3 – Organized

TD identification

Four teams got categorized as having an organized maturity level, while the other half got categorized as having a received maturity level. All teams conducted identification throughout their development processes and had this TDM activity baked into their development routines. The teams explained that the identification usually happened when developers would embark on new tasks from the product backlog or maintain their codebase and then discover errors in the code. The process got explained as a natural process for most teams, and many considered it as a normal part of their development work:

“Most often, it’s developers discovering that “Here’s something that should be here”. And then it’s often brought up for discussion in the morning stand-up.”

However, four of the teams (B, C, D, G) also expressed that they used tools to continuously identify TD, in addition to manually inspect the code. They continuously conducted crash- and security analyses of the code and others utilized dashboard-solutions, such as Grafana Dashboards or Splunk, that would indicate TD in their codebase by sending error messages whenever incidents happened, and the teams experienced these as helpful in their daily work:

“Either the developer just knows about it because he consciously made the decision and you know it lives in your soul and it eats away at you. In addition, quite a few times when you get a good dashboard for every application you have built, and you start to get obsessed over that, and you see the API-response time, and sometimes you see a blip, and you’re like “what’s that blip? I don’t like that.”, and you dig into it, and you realize that something bad is happening there. And that could very often be TD.”

However, the other half of the teams (A, E, F, H) did not use any tools for identifying TD, caused mainly by the respondents not being familiar with tools for conducting it. In addition, some of the developers explained that they usually would know where their TD would be located, and therefore did not use any tools for conducting it yet. A respondent from team B mentioned stated that:

“We don’t have tools for identifying TD. And we don’t run test coverages. But we have as a rule-of-thumb that all legacy code is TD.”

Caused by the case company’s history of starting as an internal project within a bigger company, most of the respondents seemed to experience that most TD came from legacy code, created by developers not working in the firm any longer. Some of the respondents, therefore, explained that they often would consider all legacy code as TD, and that the developers therefore would already be familiar with what

TD they had in the codebase. However, many respondents did perceive TD identification as a valuable practice in the teams, and some also expressed a wish for better ways of identifying TD and remembering it:

“Having a more defined and structured way of identifying it would be better. Because developers would know their own identified TD. But that wouldn’t be communicated to product managers, and the ones estimating stuff.”

TD prevention

In TD prevention, all teams conducted it in one way or another. Three out of eight teams (A, B, F) seemed to conduct it occasionally throughout their development processes and mostly based their practice on optional prevention practices. The mostly used prevention practice in the teams seemed to be coding guidelines they followed continuously through their development processes, which set standards towards how they wrote code in order to prevent TD accumulating. These could be different standards the developers were recommended to follow when they wrote code and often had in the back of their minds while working:

“An easy principle we follow is to leave the code in a better shape than what it was in. Another we try and follow is to avoid developing hacks in the code. I think that hacks in the code are one of the main reasons from TD accumulation.” (A)

Another practice they seemed to use to prevent TD was running occasional tests and practiced continuous integration throughout the development process to maintain a certain degree of control of their codebase and had this as a way of preventing TD from accumulating. In addition, different programming practices, such as pair-programming and mob-programming, was occasionally practiced by some teams, where several developers would collaborate on the same code development in order to (1) get different perspectives on the same work and (2) avoid development making short-cuts in the code to ensure quality. The teams seemed to show indications that they understand the value these practices give but also to express that it is experienced as cumbersome:

“We sometimes run tests and conduct continuous integration. We sometimes do pair-programming, especially nowadays with remote working. I do see its purpose, but I’m very bad at practicing it because it’s very time consuming. And when I work, I’m very often like “I know how to write this”, and won’t bother pair-programming it.” (B)

However, most teams (C, D, E, G, H) seemed to conduct TD prevention in a more well-structured mandatory manner, practicing it continuously throughout the development processes and using different

methodologies to conduct it. These teams seemed to practice the same kinds of practices as the teams practicing TD prevention, more based on current needs, but in a more structured, continuous, and mandatory manner. These teams also had a shared interest in practicing TD prevention through collaborative practices. In addition, some of these teams conducted agile practices, such as planning sessions, product demos, and post mortems. They emphasized the importance of knowledge sharing within the team in order to prevent TD and experienced these as “high leverage activities”:

“The best things to do is to just be on top of each other. Do code reviews, planning sessions, pair programming, mob programming. All these things contribute to preventing TD. The really interesting thing is that when doing these things, the entire team will probably not make that same mistake again. And that’s a high leverage activity!”

All in all, the teams seemed to mostly reflect a relatively high degree of practice in TD prevention, and most teams stressed the practices’ importance for quality insurance in software development. Despite not all teams actively performing the possible approaches to prevent TD, most teams had some mandatory practices to follow throughout the development process. The teams could therefore be argued to be categorized as organized in the TDM framework.

TD communication

All teams got categorized as organized in TD communication. This was caused by the teams’ common emphasis on communicating TD being essential continuously through their development processes and by the teams viewing this as the recipe for a well-working team dynamic. TD was usually brought up the topic during informal conversations, daily stand-ups/meet and greets, or by setting up ad-hoc meetings specifically for discussing TD, and the meetings would always include necessary stakeholders to discussions relevant for them to be included in:

“We usually communicate TD in Mondays and Fridays where we have available meeting spots in the calendar for people to bring up whatever’s on their minds. Otherwise, it’s often communicated externally through a mix of informal conversations and in different forums across work practices in the company. Product manager forums, tech lead forums, and stuff like that.”

“I think that good communication is just product and engineering having a good relationship. When I think I’m on a good team that I’m working in, I feel like it’s just an ongoing conversation rather than “we need once a week to talk about these things”.”

All teams also included necessary stakeholders in their communication continuously. It seemed as if the company management understood TD and teams having to dedicate time and effort to manage it and be engaged in the communication. However, the findings seemed to reveal that some teams (D, G) could find it challenging to communicate TD to higher stakeholders, such as the company management coming from a business perspective:

“Honestly, there’s always been an artform to communicate TD to extremely smart, but not technical people.”

One of the developers mentioned that they had experienced a notion of friction from the company management between new product development and feasibility. A respondent further expressed that communicating TD to external stakeholders could be challenging and unmotivating caused by the company management not having a proper focus on TD:

“I wish that there was a bigger focus on TD centrally in the company, and that TD had a more shared lift within tech in order to really communicate the importance of prioritizing TD in development work. [...] They want something new which no one has seen before. And that is a way for them to increase the company’s amount of TD. So there’s something weird about their focus. [...] If the company management would applaud us for our work with TD in another way than “Great work you guys” because they felt like they had to, and instead work have said more sincerely “This is really important for us, thank you!”. That would be much more motivating.”

The respondent did express a wish for a bigger focus on TD centrally in the company and to create a better understanding of the concept across the whole company. In addition, there seemed to be indications that this difficulty impacted the team members' motivation. It, therefore, seemed to be an apparent challenge towards communicating TD to stakeholders not coming from tech divisions within the company. However, this did not necessarily seem to influence the inclusion of business stakeholders in their communication threads.

5.0 Discussion

In this section, the findings will be discussed against the TDM framework and previous research on TDM to answer the research question this study has been based on: How do autonomous teams actively manage technical debt?

5.1 Discussion of TDM activities

TDM activity/ TDM maturity level	Repayment	Prevention	Documentation	Identification	Measurement	Monitoring	Communication	Prioritization
Organized	H	C, D, E, G, H	D	B, C, D, G		C	A, B, C, D, E, F, G, H	G, B
Received	A, B, C, D, E, F	A, B, F	A, B, C, E, F, G, H	A, E, F, H	B, C, D, F, G	D, F		A, C, D, E, F, H
Unorganized	G				A, E, H	A, B, E, G, H		

Duplication of Table 6. Team evaluation in TDM-framework

Looking back at Yli-Huumo et al.'s (2016, 196) explanation of the TDM framework, they created it as a tool for practitioners to use in order to better understand their current practice of TDM, as well as a tool in which could guide companies in how to improve them. The findings seemed to show clear indications on the TDM-framework working as a tool for understanding teams' level of maturity in TDM-activities in terms of their organization and management. When studying the teams, one could see that all teams seemed to reflect clear degrees of organization skills. Some teams had highly structured ways of conducting TDM continuously where teams followed defined protocols, while other teams conducted occasionally in more improvised ways, not following a planned process. This accurately correlates with Paulk et al.'s (1993, 19) explanation of the maturity spectrum teams often is located within.

Also, looking back at Paulk et al.'s (1993) CMM, this guiding star of a framework has provided practitioners with both achieving a better understanding and improving development processes in terms of management, as well as in success. Practitioners have used these frameworks to improve their current software engineering practices, whereas higher team performance, increased time efficiency, and higher process performance often have been achieved by increasing their maturity levels (Astakhova et al. 2016; Osipov et al. 2015; Titov et al. 2016, 4). It has seemed to be a relation between maturity levels and teams' success, where the two relational factors increase or decrease in accordance with each other. Based on the data shown in the filled out TDM framework as shown in table 6, one could therefore believe the TDM framework to act the same way, and argue this to be an easy point of departure for the companies to use in order to improve their current state of TDM practice.

In some TDM activities, the teams seemed to show clear signs of their ranked TDM maturity level being relatable to their perception of TDM successfulness. However, the collected data also seemed to provide interesting findings, where this relationship seemed to deviate within certain TDM activities. Caused by these deviations between the relation of TDM maturity levels and TDM successfulness, one could therefore wonder if the TDM framework actually could be used by practitioners in order to correctly understand their current state of TDM practice, as well as to improve and achieve TDM success. The following parts of this chapter will therefore bring up findings within each TDM activity and discuss the case company's execution of TDM activities in order to develop a better understanding of how they actively manage their TD, as well as to understand whether their identified maturity levels could be used as an accurate representation of their TDM successfulness.

5.1.1 Organized activities

TD prevention

Starting with TD prevention, most of the case company's teams got categorized as having an organized maturity level. All teams performed TD prevention in one way or another and had coding guidelines they followed continuously through their development processes, which set standards for writing code to prevent TD from accumulating. In addition, some teams ran automated tests of their code to have a continuous method of inspecting their code base for errors. This has been proven as an effective way of preventing TD in research, caused by its ability to let developers effectively fix the errors and stop the TD accumulation (Bavani 2013; Gat and Heintz 2011. Codabux and Williams 2013). Several teams also practiced different programming practices, such as pair-programming and mob-programming, to prevent TD. These activities have also been recognized as highly effective practices in TD prevention, caused by their knowledge-sharing leverage (Stolberg 2006; Nord et al. 2012, 98). The findings revealed that most teams argued that these activities got experiences as high-leverage activities, and they were highly satisfied with the value they created for the team, both for preventing TD and increasing the developers' knowledge. Since the teams got categorized as organized in the TDM framework and seemed to be satisfied with their TD prevention practices, one could argue that the TDM maturity level could relate to their TDM success. Therefore, this TD prevention in the TDM framework seemed to represent the team accurately, both in terms of its management and success.

TD identification

The case company classified as both received and organized in TD identification, where half of the teams conducted it occasionally throughout their development processes based on current needs, and the other half conducted it continuously and had automated tests and monitoring indications for TD. The organized teams experienced this continuous TD identification as helpful for them. However, the received teams did not practice it continuously, mostly caused by not knowing which tools one could

use for identifying TD. Therefore, they manually conducted TD identification whenever they would find it organically in their development work. This seems to correlate with previous research, finding that teams often find the lack of TDM tools as a challenge (Ernst et al. 2015, 56). Some of the teams categorized as received also expressed a wish for better ways of conducting identification. Based on this recognition of value and wish for improvement, one could argue that their maturity level seems to relate to TDM success. The TDM framework shows that using tools to conduct TDM identification would rate these teams as organized teams (Yli-Huumo et al. 2016). Also, previous research shows that using tools for conducting static code analyses is beneficial for detecting bug-related TD (Zazworka et al. 2013). Therefore, one could argue that the teams categorized as received would have improved their TDM success if they implemented such practices for TD identification (Zazworka et al. 2013).

TD communication

TD communication got categorized as having an organized maturity level in the case company, and all teams got categorized within the highest TDM maturity level. The case company seemed to have a common emphasis on communication being an essential part of their development process, and all teams had good internal structure TD communication through several sync meetings conducted weekly. Having a continuous and structured approach to TD communication provides the team with better control over their TD, and research seems to show consensus in this being a highly beneficial practice (Klinger et al. 2011). Also, all stakeholders, including business stakeholders, were included in meetings that were held regarding TD, whenever the team found the need of updating them. However, even though all teams highly practiced TD communication as organized in the TDM framework, the findings did reveal teams experiencing challenges regarding communicating TD to non-technical business stakeholders. This has been recognized as one of the most prominent challenges in TD communication in previous research (Klinger et al. 2011, 35). Two of the teams expressed that communicating TD to business stakeholders could be challenging caused by the company management not having a correct focus on TD and not appreciating the teams' effort to use time on TD enough. In addition, one of the respondents explained that the wrong focus had a direct impact on the teams' motivation caused by a lack of appreciation from the company management. Looking at previous research, TD-related issues in software development do often not translate well between technical and non-technical work practices caused by a communication gap of different perceptions of TD (Yli-Huumo et al. 2014). It was also expressed from one respondent that the company management often had different prioritizations in new feature development vs. improving current features. As a typical result, this gap could potentially aggravate TD accumulation by business stakeholders not providing teams with adequate time and resources to repay their TD (Klinger et al. 2011, 38). Therefore, despite the teams proving to practice a higher TDM maturity level caused by their constant communication of TD to all necessary stakeholders, it was difficult for the team to benefit from the organized communication. Therefore, one could argue

that the TDM aspect of communication seems to have several other factors than the organization of communication one should consider when trying to improve their perceived TDM success.

5.1.2 Received activities

TD prioritization

TD prioritization got categorized as received in the TDM framework. Some teams practiced it through dedicated prioritization meetings and impact mapping workshops. They viewed these as valuable for the team to better understand the severity of their TD and correctly prioritize TD-issues. Some teams also conducted an activity discussing their TD's different impact factors, whereas criticalness, surface impact, and customer value were often taken into significant consideration. Previous research seems to identify this as one of the most common ways of practicing TD prioritization, caused by many perceiving TD as having an equal or higher severity level than new features (Bavani 2012; Codabux and Williams 2013). However, most other teams seemed to conduct their TD prioritization through informal conversations, briefly in daily standup meetings, or through ad-hoc meetings whenever TD prioritization was needed. Their decisions were often based on a hunch, and they did not conduct any technical calculations to quantify their TD severity. Conducting TD prioritization activities in early iterations has shown its value in previous research and functions as a way of preventing TD from accumulating (Davis 2013). Previous research shows that good ways for teams to practice prioritization could happen through different technical calculations such as running static test coverages (Seaman et al. 2012). Also, conducting workshops and cost-benefit analyses of TD seems to be a practice teams have used for effective and accurate TD prioritization (Seaman et al. 2012; Zazworka et al. 2011). Thus, one could believe that the teams ranked as received in the TDM framework could benefit from increasing their maturity level and conduct TD prioritization through more structured methods to improve their TDM practice.

TD documentation

TD documentation was a relatively optional practice within most teams, and the TDM activity, therefore, got categorized as having a received maturity level. Most of the teams had no well-structured method of documenting their TD, and the developers usually treated identified TD-issues as regular product features. A few developers would also expressed that they did not always see the need of documenting all TD they would identify and only conducted it whenever they felt a need for it. One respondent explained that some TD could be perceived as acceptable risks, and therefore based their documentation decisions on their own interest. This correlates with previous research, proving that developers usually document the TD they perceive as relevant (Leithbridge et al. 2003, 38). However, these decisions could often result in developers neglecting TD, which is also identified as one of the most major challenges with TDM (Codabux and Williams 2013, 14; Power 2013). Many developers also expressed that they often would remember undocumented TD in the back of their minds. However, previous research also

has identified this approach as another way of neglecting TD, which could result in it growing into something potentially damaging to the codebase (Stettina et al. 2011, 164). Still, one team got categorized as organized and seemed to conduct TD documentation in a more defined manner. They had a system for filtering out TD-issues in the backlog by tagging the issues in Jira, which let them have a separate TD backlog whenever they would need to get a better overview of their identified TD. Previous research shows that having separate TD backlogs provides teams and the case company with a better long-term understanding of their development process and could help them significantly in practicing other TDM activities more maturely (Codabux and Williams 2013; 13; Stettina et al. 2011; 164). Also, some respondents expressed a wish for having a better structured way of documenting and remembering their TD. Therefore, one could argue that the received teams could increase their TDM maturity level by having a separate TD backlog as a simple step towards further improving their TDM success in TD documentation.

TD repayment

TD repayment also was rated as having a received maturity level. All teams who conducted repayment did it by rewriting or refactoring code, which are the most common ways of conducting it in research (Pérez et al. 2020; Codabux and Williams 2013). Repayment was mostly conducted occasionally and based on their current need. Whenever a developer would identify TD in the code, it would either get refactored/rewritten immediately or documented in the Jira-backlog for the developers to repay when needed and fit within their current time constraints. The respondents explained that they would often prefer repaying TD "as you go" rather than collecting them and repaying them in bunches. Most teams did not have a certain percentage of their development time assigned to repay TD. This correlates with previous literature, showing that most teams choose to repay TD during its evolution and focus their development work mostly on new features (Digkas et al. 2018, 153; Power 2013). This could be because developers often choose to prioritize new feature development instead of improving existing ones (Codabux and Williams 2013, 14). However, most teams expressed that they did not believe in assigning percentages as an effective TD repayment strategy for their instances. This was caused mainly by them viewing TD repayment as a regular part of their development routines and saw it as unnatural to have a specific percentage of their work assigned to repay TD. Overall, the teams seemed to project a satisfied expression of how they repaid their TD and did not express a need for implementing practices for improving their current TDM maturity level to achieve an improved success.

TD measurement

TD measurement got rated as having a received maturity level. Most teams seemed not to have a structured practice of the activity, and it was mostly conducted as an optional practice. Some teams conducted measurements based on simple data and T-shirt sizes as sizing metrics, while others would make a list of each TD-issue's potential impact factors and measure the TD-issue based on the length of

listed impact factors. However, none of the teams seemed to conduct continuous measurement during their development processes, such as the TDM framework suggests organized teams to do (Yli-Huumo et al. 2016). Several teams seemed to have received the value of practicing measurement to a certain extent. However, they did not see the value in having the activity conducted in any more structured way than they currently did. There seemed to be a common lack of belief in heavily measuring and estimating the size of TD. Some respondents explained that they avoided estimations caused by the fear of developers feeling locked within defined time frames of handling their TD. In addition, other respondents explained it also as impossible to measure their TD, caused by them not knowing how big the TD could be. In some teams, the TD they identified was mostly bug-related TD, while in other teams, the TD could often root from a "TD-monolith" of legacy code. As Kruchten et al. (2012) explained, TD can come from minor bugs as well as fundamental architectural and structural issues. Furthermore, caused by this measuring uncertainty, one could wonder if implementing TDM practices for improving their TDM maturity level to be beneficial for the team in terms of success or if it would have the opposite effect.

5.1.3 Unorganized activities

TD monitoring

TD monitoring was categorized as unorganized in the TDM framework, and previous research shows that this activity is rarely practiced caused of its time consumption and cost of implementation (Yli-Huumo et al. 2016). In this study, only three teams practiced TD monitoring to some extent by basing the practice on simple data from their backlog or using other monitoring software, which are proven ways to effectively identify, track, and manage TD (Power 2013; Hansen et al. 2010). Only one respondent acknowledged the activity's value and expressed a wish for better visualizing their TD for developers. However, most other respondents expressed it as uninteresting to implement, unnecessary to conduct, and highly time-consuming. Several respondents explained that this was of minimal interest when asked whether tools for visualizing TDs evolution were of interest. This came to a surprise, caused by previous research seemingly viewing TD monitoring as a crucial part of their development routines and viewing TD monitoring as having a direct influence on lowering the negative impact TD can create (Guo et al. 2011). Martini et al. (2016, 165) found that not carefully analyzing and tracking identified TD, adverse effects of TD could get aggravated. In addition, monitoring TD could work as a method to quantify TD in which works as a tool for better communication and understanding of identified TD (Klinger 2011, 36). Research seems to be unanimous in TD monitoring's positive impact on TDM. Then again, most teams expressed no need to conduct it in their processes, often caused by the fear of making complicated processes around TDM for the developers to follow. This argument could be supported by Martini et al.'s (2016, 165) findings of some TDM activities, creating more work on top for developers to follow to practice the activities. Therefore, one could wonder if achieving a higher TDM maturity level would be beneficial for the team and improve their processes.

5.2 Reliability of TDM framework

So far, the discussion has provided insights towards how the case company works within each different TDM activity, and one can see that the TDM framework could work as a framework for better understanding teams' current state of practice. The maturity levels accurately represented how much effort each team spent on each activity. An easy answer to the research question about how autonomous teams actively manage their TD could therefore be based on these findings. Teams seemed to work continuously with TD prevention, identification, and communication. They occasionally worked with TD repayment, documentation, identification, measurement, and prioritization. Also, they worked minimally with TD monitoring. However, when looking further into the TDM activities discussed above, the study identified deviations in several activities where the level of maturity did not apply to the teams' success. Therefore, one could question whether the TDM framework would provide practitioners with accurate representations of not only how autonomous teams manage their TD, but also if they are successful at it.

In TD prevention, identification, prioritization, and documentation, the findings identified that the teams' TDM maturity level accurately matched their perceived TDM success. For example, in TD prevention, most teams conducted mandatory practices to prevent TD from accumulating and performed these frequently and continuously throughout their development process. In addition, the teams seemed to have a common expression of these practices being essential and helpful in their work. Therefore, the team got categorized as organized. Also, looking into TD identification, one could see that the teams had their ways of conducting TD identification on occasional basis. The teams seemed to find the practice of identification as valuable in their work, and some respondents also found room for improvement within this activity in order to further improve their performance within TD identification. This activity, therefore, got categorized as received. What is present in both these activities is that both TDM activities seemed to reflect that a lower degree of TDM maturity levels projected lower TDM success. In contrast, higher TDM maturity levels projected higher TDM success. Therefore, this study could be argued to prove that Yli-Huumo et al.'s (2016) TDM framework could apply to these four TDM activities in order to accurately understand a company's current state of TDM practice, as well as to use these maturity level as a guiding framework for how to improve teams' TDM success.

However, in TD monitoring, measurement, repayment, and communication, one could begin the doubt the TDM framework's ability to "evolve toward a culture of software engineering and management excellence" in the same way as Paulk et al. (1993, 19) has described their maturity model to do. In these TDM activities, the case company's TDM maturity levels seemed to show signs of deviations in TDM success. For example, in TD repayment, the teams seemed to mostly conduct it based on their current

needs throughout the development process and did not follow any well-structured process. According to the TDM framework, these teams were categorized as having a received maturity level. However, most respondents expressed that they were satisfied with the way they practiced the activity and did not think of improving their current practice as necessary to accomplish a higher level of success. Unlike in TD prevention, identification, prioritization, and documentation, these TDM activities did not prove that a lower degree of TDM maturity level projected lower TDM success and vice versa. The findings seemed to project that implementing new TDM practices that would rate them higher in the TDM framework would not provide them with a higher degree of TDM success. Many respondents projected a common expression of implementing strict processes into their free-flowing development processes as time-consuming as well as unrealistic to be followed up by all team members in a way that would benefit them. Also, some respondents feared that implementing more structured processes in TDM practices would become more work in the developers' hands rather than generating value.

These deviations seem to correlate with previous findings of TDM practice and its challenges. Implementing new TDM practices has shown to cause a degrading effect on TDM in some cases, caused by (1) substantial amounts of effort and resources required in order to implement new tools and practices, and (2) the creating of more work on top of existing processes for managing TD (Martini et al. 2016, 165; Yli-Huumo et al. 2016, 213). Still, empirical studies on TDM seem to stress that implementing TDM practices and tools as highly beneficial. Looking at previous research, one could therefore question whether respondents acting rejecting towards implementing more structured processes could be caused by a lack of knowledge in TDM tools and practices to use (Ernst et al. 2015, 56; Martini et al. 2016, 168). However, the findings gathered from TD communication could then again counter these arguments. Despite the TDM activity getting categorized as highly organized in the TDM framework, some respondents did not project a higher degree of TDM success caused by challenges in their communication. Thus, one could question whether the TDM framework is reliable to use for all teams wanting to understand how teams are managing their TD, as well as whether the framework's maturity level is related to teams' success at them.

6.0 Implications

6.1 Limitations

Aristotle wisely said “The more you know, the more you know you don’t know.” This quote also applies well to research projects. The deeper you dig into a research process, the more limitations become apparent. This subchapter will therefore present the most significant limitations of this study.

Caused by the covid-19 pandemic, this case study has been affected and gotten limitations during conduction. Case studies are supposed to build their empirical saturation based on several sources, such as interviews, documentation, observations, and informal conversations. However, the pandemic has made this challenging and resulted in getting limited access to the additional sources. This study’s findings are therefore based on the conducted interviews, and not on other sources.

Another limitation of this study is that TDM research, and especially on TDM frameworks, is limited. I did not manage to find other studies in which empirically tested TDM maturity frameworks. In order to therefore provide the study with richer discussions towards how teams manage their technical debt, it would therefore have been beneficial to reference other literature than the paper written by the creators of the TDM framework, Yli-Huumo et al. (2016). This would further robustify the literature review, as well as increase the validity of this research project. Even though the TDM framework seemed to perform well to understand autonomous teams, it would not necessarily do the same in other companies.

In addition, the study also has some limitations in terms of data collection and research design. It would have been beneficial to interview autonomous teams outside the case company in order to gain richer insights and to gain further validity of findings from using the TDM framework. While the TDM framework did not apply well in translating maturity levels into TDM success in this study, this would might not apply for other companies or teams in the same way. Choosing to conduct a multiple-case study would normally address this limitation because the research design collects data from several contexts, but since all data is gathered from different contexts within the same case company, the project would benefit from more representative data coming from other teams and companies.

6.2 Practical implications

There are several practical implications that can be drawn from this study in terms of improving their current TDM processes, both in terms of management and success.

Starting with management, the study revealed the case company to mainly project a received level of maturity within TD repayment, documentation, measurement, prioritization, and partial identification.

This means that the company, as of now, generally practice these activities on occasional basis, but without any clear strategy in when they are conducted. In addition, the case company performed low in TD monitoring, where most teams ranked as unorganized. The teams invested minimal effort into conducting this, and had no ways of tracking their TDs evolution. Based on these findings, the case company could use the mentioned TDM activities as a point of departure, and implement the recommended practices from the TDM maturity model in order to increase their TDM maturity in terms of its management and organization. This would provide the teams with more control over their TD and increase their awareness as a result.

As for improving their current TDM practice in terms of success, the study found that the company could more safely look further into the TDM activities, proving a relation between TDM maturity level and TDM success to improve their current practices. In TD identification, prioritization, and documentation, there was identified room for improvement. Since these activities also got ranked as having a received maturity level, one could further improve these activities by implementing the suggested tools and practices as shown in the TDM framework and practice these more frequently rather than only based on their current need. By making these practices more mandatory and continuously conducting them, one could believe that their TDM success would improve accordingly.

Also, TD communication could be argued as an important TDM activity for the case company to look further into. Even though it got categorized as highly mature in the TDM framework, there was identified a communication gap between the team and case company management. Therefore, one could argue that the case company should look further into this and look for solutions in order to avoid the miscommunication to aggravate TD accumulation over time.

6.3 Implications for future research

The TDM framework shares similarities with other CMM frameworks. However, this study identified a need for the TDM framework to be further empirically tested and iterated on. The TDM framework worked well as a tool for understanding a case company's TDM processes, but did not translate too well for companies to use to improve TDM success. It would be interesting to further investigate the relation between TDM maturity levels and actual TDM success and dig deeper into each individual TDM activity in order to develop a better understanding of these challenges. One could, for example, investigate whether the TDM maturity levels' relation to success are stronger in some contexts compared to others.

Furthermore, it would also be interesting to address the identified limitations in further research on TDM and the TDM framework. Caused by the limited number of sources used for data collection in this study, it would be interesting to see whether the findings from this study would correlate with observations,

documentation, and other sources. Also, it would be interesting to test the TDM framework in autonomous teams coming from different companies and different industries in order to develop a better understanding for how the TDM framework functions in different settings. This would enrich the TDM framework's reliability and facilitate for other researchers to iterate on the framework's current state in order to possibly develop a reliable and generalized TDM framework. This all would also contribute to the scarce field of TDM research in order to develop a better understanding of how teams actively manages their TD.

7.0 Conclusion

This study has explored and answered the research question, how do autonomous teams actively manage technical debt? This was done through conducting a multiple-case study on one of Norway's biggest fintech companies, investigating how eight of their autonomous teams worked towards technical debt management (TDM) using Yli-Huumo et al.'s (2016) TDM framework. The TDM framework was created taking inspiration from Paulk et al.'s (1993) capability maturity model (CMM) and Li et al. (2015) TDM activities.

The findings revealed that autonomous teams practiced TDM activities to different extents. In some activities, the teams practiced these continuously throughout their development processes and often would use tools in order to address them. While in other activities, the teams did not seem to have any structured approach and often improvised practices based on their current need. However, when summarizing the findings, it was found that the autonomous teams got categorized as performing a generally received level of maturity in the TDM framework. Most teams practiced TDM activities based on their current needs and acknowledge the value they can create.

However, the study did identify challenges related to the TDM framework's relation between TDM maturity level and TDM success. When looking further into whether the TDM framework could work as a tool for improving current TDM practices, the study identified deviations. Teams seemed to show clear indications on implementing practices for increasing their TDM maturity level as highly unnecessary caused by high time-consumption on implementation and practice, as well as a fear of creating more work around TD than it already is. It can therefore be challenging to understand whether finding teams as received means that there is room for improvement in TDM or not.

Regardless, the TDM framework did provide valuable data in terms of better understanding how autonomous teams managed their TD in the context of a Norwegian fintech company. Based on these findings, the TDM framework has indeed proven to act like a well-functioning device for better understanding teams' TDM processes. The TDM framework could also be used to a certain extent in order to improve teams' TDM success, but practitioners could be recommended to consider the identified deviations between TDM maturity levels and TDM success in some of the activities. We therefore suggest the TDM framework to be further empirical tested and iterated on for it to work as an accurate tool for when trying to understand and improve teams' TDM processes.

References

- Abrahamsson, P., M. A. Babar, og P. Kruchten. 2010. «Agility and Architecture: Can They Coexist?» *IEEE Software* 27 (2): 16–22. <https://doi.org/10.1109/MS.2010.36>.
- Agile Alliance. 2020. «Manifesto for Agile Software Development». u.å. Opened 13th november 2020. <https://agilemanifesto.org/>.
- Anderson, P. A. (1983). Decision making by objection and the cuban missile crisis. *Administrative Science Quarterly*, 28(2), 201-222.
- Astakhova, Nadezhda, Liliya Demidova, and Evgeny Nikulchev. 2016. “Multiobjective Optimization for the Forecasting Models on the Base of the Strictly Binary Trees.” *International Journal of Advanced Computer Science and Applications* 7 (November). <https://doi.org/10.14569/IJACSA.2016.071122>.
- Bavani, R. 2012. «Distributed Agile, Agile Testing, and Technical Debt». *IEEE Software* 29 (6): 28–33. <https://doi.org/10.1109/MS.2012.155>.
- Beck, K. 1999. «Embracing change with extreme programming». *Computer* 32 (10): 70–77. <https://doi.org/10.1109/2.796139>.
- Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82, 139-158.
- Bellomo, S., R. L. Nord, og I. Ozkaya. 2013. «A study of enabling factors for rapid fielding combined practices to balance speed and stability». I *2013 35th International Conference on Software Engineering (ICSE)*, 982–91. <https://doi.org/10.1109/ICSE.2013.6606648>.
- Birkeland, J. O. (2010, June). From a timebox tangle to a more flexible flow. In *International conference on agile software development* (pp. 325-334). Springer, Berlin, Heidelberg.
- Blekinge Institute of Technology, Richard Torkar, Pau Minoves, i2cat Foundation, Janina Garrigós, og i2cat Foundation. 2011. «Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use». *Journal of the Association for Information Systems* 12 (1): 88–122. <https://doi.org/10.17705/1jais.00252>.

- Brown, Nanette, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, Nico Zazworka, Yuanfang Cai, mfl. 2010. «Managing Technical Debt in Software-Reliant Systems». I *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research - FoSER '10*, 47. Santa Fe, New Mexico, USA: ACM Press. <https://doi.org/10.1145/1882362.1882373>.
- Codabux, Z., & Williams, B. (2013, May). Managing technical debt: An industrial case study. In *2013 4th International Workshop on Managing Technical Debt (MTD)* (pp. 8-15). IEEE.
- Cousin, G. (2005). Case Study research. *Journal of Geography in Higher Education*, 29(3), 421-427.
- Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2), 29-30.
- Curtis, Bill, Bill Hefley, and Sally Miller. 2009. "People Capability Maturity Model (P-CMM) Version 2.0, Second Edition:" Fort Belvoir, VA: Defense Technical Information Center. <https://doi.org/10.21236/ADA512354>.
- Davis, N. 2013. «Driving Quality Improvement and Reducing Technical Debt with the Definition of Done». I *2013 Agile Conference*, 164–68. <https://doi.org/10.1109/AGILE.2013.21>.
- Digkas, Georgios, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. 2018. "How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem?" In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 153–63. <https://doi.org/10.1109/SANER.2018.8330205>.
- Dingsøy, T., Nerur, S., Balijepally, V., Moe, N.B., 2012. A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, Special Issue: Agile Development 85, 1213–1221. <https://doi.org/10.1016/j.jss.2012.02.033>
- dos Santos, P. S. M., Varella, A., Dantas, C. R., & Borges, D. B. (2013, June). Visualizing and managing technical debt in agile development: An experience report. In *International Conference on Agile Software Development* (pp. 121-134). Springer, Berlin, Heidelberg.
- Dybå, Tore, Torgeir Dingsøy, og Nils Brede Moe. 2014. «Agile Project Management». I *Software Project Management in a Changing World*, redigert av Günther Ruhe og Claes Wohlin, 277–

300. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-55035-5_11.

Eisenhardt, Kathleen M., and Melissa E. Graebner. "Theory Building from Cases: Opportunities and Challenges." *The Academy of Management Journal* 50, no. 1 (2007): 25-32. Accessed May 24, 2021. <https://doi.org/10.2307/20159839>

Ernst, Neil A., Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 50–60. Bergamo Italy: ACM. <https://doi.org/10.1145/2786805.2786848>.

Fowler, Martin, Kent Beck, og John Brant. u.å. «Refactoring - Improving the Design of Existing Code», 337. Frank, A., og C. Hartel. 2009. «Feature Teams Collaboratively Building Products from READY to DONE». I *2009 Agile Conference*, 320–25. <https://doi.org/10.1109/AGILE.2009.51>.

Garousi, Vahid, and Erik Veenendaal. 2021. "Test Maturity Model Integration (TMMi): Trends of Worldwide Test Maturity and Certifications." *IEEE Software* PP (February). <https://doi.org/10.1109/MS.2021.3061930>.

Gat, Israel, og John D. Heintz. 2011. «From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt». I *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 24. Waikiki, Honolulu, HI, USA: ACM Press. <https://doi.org/10.1145/1985362.1985368>.

Guo, Yuepu, og Carolyn Seaman. 2011. «A Portfolio Approach to Technical Debt Management». I *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 31. Waikiki, Honolulu, HI, USA: ACM Press. <https://doi.org/10.1145/1985362.1985370>.

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q., ... & Siebra, C. (2011, September). Tracking technical debt—An exploratory case study. In *2011 27th IEEE international conference on software maintenance (ICSM)* (pp. 528-531). IEEE.

Guzzo, Richard A., og Marcus W. Dickson. 1996. «TEAMS IN ORGANIZATIONS: Recent Research on Performance and Effectiveness». *Annual Review of Psychology* 47 (1): 307–38. <https://doi.org/10.1146/annurev.psych.47.1.307>.

- Hanssen, G., A. F. Yamashita, R. Conradi, og L. Moonen. 2010. «Software Entropy in Agile Product Evolution». I *2010 43rd Hawaii International Conference on System Sciences*, 1–10. <https://doi.org/10.1109/HICSS.2010.344>.
- Huang, Chun-Che, and Andrew Kusiak. 1996. “Overview of Kanban Systems.” *International Journal of Computer Integrated Manufacturing* 9 (3): 169–89. <https://doi.org/10.1080/095119296131643>.
- Humphrey, W. S. (1989). *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc..
- Kaiser, Michael, og Guy Royse. 2011. «Selling the Investment to Pay Down Technical Debt: The Code Christmas Tree». I *2011 Agile Conference*, 175–80. <https://doi.org/10.1109/AGILE.2011.50>.
- Kidder, T. (1982). *Soul of a new machine*. New York: Avon. Presented at the the 40th International Conference, ACM Press, Gothenburg, Sweden, pp. 75–84. <https://doi.org/10.1145/3183519.3183539>
- Klinger, Tim, Peri Tarr, Patrick Wagstrom, and Clay Williams. 2011. “An Enterprise Perspective on Technical Debt.” In *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 35. Waikiki, Honolulu, HI, USA: ACM Press. <https://doi.org/10.1145/1985362.1985371>.
- Klotins, E., Unterkalmsteiner, M., Chatzipetrou, P., Gorschek, T., Prikładnicki, R., Tripathi, N., Pompermaier, L.B., 2018. Exploration of technical debt in start-ups, in: *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE- SEIP '18*.
- Kniberg, Henrik. u.å. «Scrum and XP from the Trenches», 104.
- Kruchten, Philippe, Robert L. Nord, og Ipek Ozkaya. 2012. «Technical Debt: From Metaphor to Theory and Practice». *IEEE Software* 29 (6): 18–21. <https://doi.org/10.1109/MS.2012.167>.
- Lee, and Xia. 2010. «Toward Agile: An Integrated Analysis of Quantitative and Qualitative Field Data on Software Development Agility». *MIS Quarterly* 34 (1): 87. <https://doi.org/10.2307/20721416>.

- Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193-220.
- Lim, E., N. Taksande, og C. Seaman. 2012. «A Balancing Act: What Software Practitioners Have to Say about Technical Debt». *IEEE Software* 29 (6): 22–27. <https://doi.org/10.1109/MS.2012.130>.
- Agile Alliance. 2010. «Manifesto for Agile Software Development». u.å. Åpnet 13. november 2020. <https://agilemanifesto.org/>.
- Martini, Antonio, Jan Bosch, og Michel Chaudron. 2014. «Architecture Technical Debt: Understanding Causes and a Qualitative Model». I *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 85–92. Verona, Italy: IEEE. <https://doi.org/10.1109/SEAA.2014.65>.
- Martini, A., T. Besker, og J. Bosch. 2016. «The Introduction of Technical Debt Tracking in Large Companies». I *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 161–68. <https://doi.org/10.1109/APSEC.2016.032>.
- McConnel, S. (2007), 'Technical debt'. Accessed: 2013-04-20. URL: http://construx.com/10x_Software_Development/Technical_Debt
- McNamara, C. (2009). General guidelines for conducting interviews. Retrieved January 11, 2010, from <http://managementhelp.org/evaluatn/intrview.htm>
- Moe, N. B., T. Dingsøy, og T. Dybå. 2008. «Understanding Self-Organizing Teams in Agile Software Development». I *19th Australian Conference on Software Engineering (aswec 2008)*, 76–85. <https://doi.org/10.1109/ASWEC.2008.4483195>.
- Moe, Nils Brede, Torgeir Dingsøy, og Tore Dybå. 2010. «A Teamwork Model for Understanding an Agile Team: A Case Study of a Scrum Project». *Information and Software Technology* 52 (5): 480–91. <https://doi.org/10.1016/j.infsof.2009.11.004>.
- Nielsen, Mille Edith, Christian Østergaard Madsen, og Mircea Filip Lungu. 2020. «Technical Debt Management: A Systematic Literature Review and Research Agenda for Digital Government». I *Electronic Government*, redigert av Gabriela Viale Pereira, Marijn Janssen, Habin Lee, Ida Lindgren, Manuel Pedro Rodríguez Bolívar, Hans Jochen Scholl, og Anneke

Zuiderwijk, 12219:121–37. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-57599-1_10.

Nord, Robert L., Ipek Ozkaya, Philippe Kruchten, og Marco Gonzalez-Rojas. 2012. «In Search of a Metric for Managing Architectural Technical Debt». I *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 91–100. Helsinki, Finland: IEEE. <https://doi.org/10.1109/WICSA-ECSA.212.17>.

Oates, B.J., 2006. *Researching information systems and computing*. SAGE Publications, London ; Thousand Oaks, Calif.

Osipov, I. V., Nikulchev, E., Volinsky, A. A., & Prasikova, A. Y. (2015). Study of gamification effectiveness in online e-learning systems. *International Journal of advanced computer science and applications*, 6(2), 71-77.

Pan, Shan L., og Barney Tan. 2011. «Demystifying Case Research: A Structured–Pragmatic–Situational (SPS) Approach to Conducting Case Studies». *Information and Organization* 21 (3): 161–76. <https://doi.org/10.1016/j.infoandorg.2011.07.001>.

Patanakul, Peerasit, Jiyao Chen, and Gary S. Lynn. 2012. «Autonomous Teams and New Product Development: Autonomous Teams». *Journal of Product Innovation Management* 29 (5): 734–50. <https://doi.org/10.1111/j.1540-5885.2012.00934.x>.

Paulk, M.C., B. Curtis, M.B. Chrissis, and C.V. Weber. 1993. “Capability Maturity Model, Version 1.1.” *IEEE Software* 10 (4): 18–27. <https://doi.org/10.1109/52.219617>.

Pérez, Boris, Camilo Castellanos, Darío Correal, Nicolli Rios, Sávio Freire, Rodrigo Spínola, og Carolyn Seaman. 2020. «What Are the Practices Used by Software Practitioners on Technical Debt Payment: Results from an International Family of Surveys». I *Proceedings of the 3rd International Conference on Technical Debt*, 103–12. Seoul Republic of Korea: ACM. <https://doi.org/10.1145/3387906.3388632>.

Power, K. 2013. «Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options». I *2013 4th International Workshop on Managing Technical Debt (MTD)*, 28–31. <https://doi.org/10.1109/MTD.2013.6608675>.

- Ramasubbu, Narayan, A. Bharadwaj and G. Tayi. “Software Process Diversity: Conceptualization, Measurement, and Analysis of Impact on Project Performance.” *MIS Q.* 39 (2015): 787-807. <https://doi.org/10.25300/MISQ/2015/39.4.3>
- Runeson, Per, and Martin Höst. 2009. “Guidelines for Conducting and Reporting Case Study Research in Software Engineering.” *Empirical Software Engineering* 14 (2): 131–64. <https://doi.org/10.1007/s10664-008-9102-8>.
- Robson C (2002) Real World Research. Blackwell, (2nd edition)
- Rolland, Knut H, and Kalle Lyytinen. n.d. “Managing Tensions between Architectural Debt and Digital Innovation: The Case of a Financial Organization,” 11.
- Seaman, Carolyn, Yuepu Guo, Nico Zazworka, Forrest Shull, Clemente Izurieta, Yuanfang Cai, and Antonio Vetrò. 2012. “Using Technical Debt Data in Decision Making: Potential Decision Approaches.” In *2012 Third International Workshop on Managing Technical Debt (MTD)*, 45–48. <https://doi.org/10.1109/MTD.2012.6225999>.
- Sonar Source. 2021. «Enterprise Edition | SonarSource». u.å. Åpnet 16. mai 2021. https://www.sonarsource.com/plans-and-pricing/enterprise/?gclid=CjwKCAjwhYOFBhBkEiwASF3KGUbKAhFmOgP9FFjQVcm68ocS8iBsLs4Og8wFuUosKCxg9na-8X6RaxoCttYQAvD_BwE.
- Stettina, Christoph Johann, and Werner Heijstek. 2011. “Necessary and Neglected?: An Empirical Study of Internal Documentation in Agile Software Development Teams.” In *Proceedings of the 29th ACM International Conference on Design of Communication - SIGDOC '11*, 159. Pisa, Italy: ACM Press. <https://doi.org/10.1145/2038476.2038509>.
- Stolberg, Sean. u.å. «Enabling Agile Testing through Continuous Integration». I *Agile Conference, 2009. AGILE '09.*, 2009, 369–374.
- Stray, Viktoria, Nils Brede Moe, og Rashina Hoda. 2018. «Autonomous Agile Teams: Challenges and Future Directions for Research». I *Proceedings of the 19th International Conference on Agile Software Development Companion - XP '18*, 1–5. Porto, Portugal: ACM Press. <https://doi.org/10.1145/3234152.3234182>.
- Tellis, W. (1997). Introduction to case study. *The qualitative report*, 269.

- Titov, Sergei, Gregory Bubnov, Maria Guseva, Alexei Lyalin, and Irina Brikoshina. 2016. "Capability Maturity Models in Engineering Companies: Case Study Analysis." Edited by E.V. Nikulchev and E.I. Veremey. *ITM Web of Conferences* 6: 03002. <https://doi.org/10.1051/itmconf/20160603002>.
- Tom, Edith, Aybuke Aurum, og Richard Vidgen. 2013. «An Exploration of Technical Debt». *Journal of Systems and Software* 86 (6): 1498–1516. <https://doi.org/10.1016/j.jss.2012.12.052>.
- Trist, E. (1981). The evolution of socio-technical systems. *Occasional paper*, 2(1981), 1981.
- Verwijns, Christiaan. 2018. «How to deal with Technical Debt in Scrum». *Medium*, january. <https://medium.com/the-liberators/how-to-deal-with-technical-debt-in-scrum-f4ec3481eabb>.
- Walsham, G. (1995). Interpretive case studies in IS research: nature and method. *European Journal of information systems*, 4(2), 74-81.
- Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, xiii-xxiii.
- Wiklund, K., S. Eldh, D. Sundmark, og K. Lundqvist. 2012. «Technical Debt in Test Automation». I *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, 887–92. <https://doi.org/10.1109/ICST.2012.192>.
- Xebia. Agile Survey. (2013) <http://xebia.com/news/agile-survey2013/>
- Yli-Huumo, Jesse, Andrey Maglyas, and Kari Smolander. 2014. *The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company*. https://doi.org/10.1007/978-3-319-13835-0_7.
- Yli-Huumo, Jesse, Andrey Maglyas, og Kari Smolander. 2016. «How Do Software Development Teams Manage Technical Debt? – An Empirical Study». *Journal of Systems and Software* 120 (oktober): 195–218. <https://doi.org/10.1016/j.jss.2016.05.018>.
- Yin, Robert K. 1981. "The Case Study as a Serious Research Strategy." *Knowledge: Creation, Diffusion, Utilization*, 97–114.

Yin, R. K., 2014. *Case Study Research - Design and Methods*. 5th ed. London: SAGE Publications Inc..

Zazworka, Nico, Michele A. Shaw, Forrest Shull, og Carolyn Seaman. 2011. «Investigating the Impact of Design Debt on Software Quality». I *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 17. Waikiki, Honolulu, HI, USA: ACM Press.
<https://doi.org/10.1145/1985362.1985366>.

Appendix

Appendix A: Ethical approval and NSD approval



11th of May 2021

STATEMENT OF ETHICS APPROVAL

Proposer: Karl Omar Kareem Skeimo

The school's research ethics committee has considered your submitted proposal. Acting under delegated authority, the committee is satisfied that there is no objection on ethical grounds to the proposed study.

Approval is given on the understanding that you will adhere to the terms agreed with participants and to inform the committee of any change of plans in relation to the information provided in the application form.

Yours sincerely,



Asle Fagerstrøm
Professor

NSD NORSK SENTER FOR FORSKNINGSDATA

NSD sin vurdering

Prosjekttittel

How to get away with technical debt: A case study on agile autonomous teams and technical debt management

Referansenummer

766843

Registrert

12.02.2021 av Karl Omar Kareem Skeimo - skekar16@student.kristiania.no

Behandlingsansvarlig institusjon

Høgskolen Kristiania – Ernst G. Mortensens Stiftelse / School of Economics, Innovation, and Technology / institutt for teknologi

Prosjektansvarlig (vitenskapelig ansatt/veileder eller stipendiat)

Ranvir Rai, Ranvir.Rai@kristiania.no, tlf: 91586800

Type prosjekt

Studentprosjekt, masterstudium

Kontaktinformasjon, student

Karl Omar Skeimo, karlskeimo@icloud.com, tlf: 41341485

Prosjektperiode

15.03.2021 - 01.09.2021

Status

07.04.2021 - Vurdert

Vurdering (1)

07.04.2021 - Vurdert

Appendix B: Research instrument

Interview guide

1.0 Introduction

- a) My project
- b) Anonymity
- c) Use of data
- d) Tape recording

2.0 Project history

- a) What is the history of this team/project?
- b) What is your role?
- c) What is the size of your team?
- d) What software developments methods are you practicing?

3.0 Technical debt

- a) What is technical debt to you?
- b) Have you ever experienced having to take short cuts in the development process, writing low quality code, or skipping run tests in order to meet deadlines?
 - Examples?
- c) How did taking on technical debt help you?
- c) Why do you take on technical debt in your work?
 - Intentional?
 - Unintentional?
- d) How does technical debt usually affect you?
 - Speed of innovation?
 - Productivity?
 - Quality?
 - Bugs/errors?
 - Cost?
- e) How satisfied are you with how you handle technical debt today?

4.0 Technical Debt Management

4.1 Identification

- a) How do you usually identify technical debt?
- b) Do you use any tools?

4.2 Monitoring

- a) How is your team monitoring technical debt?
- b) What does this monitoring tell you?

4.3 Measurement

- a) How is technical debt measured?
- b) What tools are used?

4.4 Prioritization

- a) How is technical debt prioritized? (Backlog grooming? Or does it happen when its discovered?)

4.5 Prevention

- a) How are you preventing technical debt from occurring?

- Pair-programming?
- Continuous Integration?

4.6 Repayment

- a) How do you resolve technical debt?
 - Refactoring?
- b) How do you organize the repayment?
- c) How do you decide when to down pay it?

4.7 Documentation

- a) How is technical debt covered in a products documentation?

4.8 Communication

- a) How is technical debt communicated within the team?
- b) How is technical debt discussed with stakeholders in order to be managed further?

5.0 Motivation and improvement for TDM

- a) How is your motivation towards handling technical debt?
- b) How do you think technical debt could be better managed?
- c) How should this be done in order for you to actively take responsibility for it?