THE ROLE OF TECHNICAL DEBT IN SOFTWARE DEVELOPMENT

Thesis for the degree of Doctor of Science (Technology) to be presented with due permission for public examination and criticism in the Auditorium 2310 at Lappeenranta University of Technology, Lappeenranta, Finland on the 15th of June, 2017, at noon.
Supervisor
Professor Kari Smolander
LUT School of Business and Management
Lappeenranta University of Technology
Finland

D.Sc. Andrey Maglyas
LUT School of Business and Management
Lappeenranta University of Technology
Finland

Reviewers
Professor Ville Leppänen
Department of Information Technology
University of Turku
Finland

Associate Professor Davide Falessi
Department of Computer Science and Software Engineering
California Polytechnic State University
USA

Opponent
Associate Professor Carolyn Seaman
Department of Information Systems
University of Maryland
USA
Abstract

**Jesse Yli-Huumo**
The role of technical debt in software development
Lappeenranta 2017
107 pages
Acta Universitatis Lappeenrantaensis 741
Diss. Lappeenranta University of Technology

Technical debt refers to a concept in software development where extra development work arises, through intentional decision or unintentional side-effect, when code that is easy to implement in short-run is used instead of applying the best overall solution. Technical debt is an essential part of software development, which has to be acknowledged by software companies. The goal of this is thesis is to address the role of technical debt in software and software development.

Empirical research methodologies are applied in the study. The data has been collected through case studies and semi-structured interviews with practitioners from software development teams. The research process consisted of one preliminary phase and three main phases. In the preliminary phase, the existing literature on technical debt and technical debt management was examined. In the first main phase, the causes and effects of technical debt were studied. The second main phase focused on technical debt management practices. The third main phase aimed at developing and testing technical debt management processes.

The results of the study revealed multiple examples indicating that technical debt is a relevant phenomenon in software engineering. Several reasons and causes for technical debt were identified, and the results indicated that there is not one specific reason for companies to have technical debt in software. In addition, several short-term and long-term effects of technical debt, both beneficial and detrimental, were identified. The results were used to develop a technical debt management framework, which describes the activities, practices, tools, stakeholders and responsibilities related to technical debt management.

The findings complement the current technical debt state-of-the-art by providing empirical evidence and knowledge on the causes and effects of technical debt from both the technical and the organizational perspective. The results can be used by software development companies to improve their knowledge on the role of technical debt in management, and the developed technical debt management framework and its processes can be used to increase the visibility and manageability of technical debt.

**Keywords:** Technical debt, technical debt management, software development, software process improvement, empirical study, case study
Acknowledgements

This work was carried out in the department of Innovation and Software at Lappeenranta University of Technology, Finland, between 2014 and 2017. During this time, there were several people who helped me during the process. Even though everyone always says these same words, "this thesis would have not been possible without the help of these people" and so on… I actually do believe I would not have been able to do this without everyone who has been somehow part of this process, and I really do think all of them deserve recognition and a big thank you.

First of all, I would like to thank my two supervisors, Professor Kari Smolander and D.Sc. Andrey Maglyas. The number of hours and the amount of work provided by them for guiding me and correcting my mistakes are uncountable. Thank you for helping me to finalize this work, and hopefully we can work together also in the future. I would like to also thank the reviewers of this thesis, Professor Ville Leppänen and Associate Professor Davide Falessi. The reviewers provided a valuable review by giving comments for improvements that really increased the quality of this research.

During my doctoral studies, I visited Sogang University in Seoul, South Korea for 6 months. The research visit gave me a lot of good memories and provided valuable experience and knowledge on working in a different country and environment. I wish to thank especially Professor Sooyong Park for inviting me, and the whole department of Software Engineering Laboratory for being friendly and teaching me about South Korean culture and work ethics. Hopefully I can someday return to South Korea to work with you.

This research was done as part of the Need 4 Speed –research project, which provided me financial support to visit companies, attend conferences and seminars, and to conduct data collection. Therefore, I would like to thank the people behind the N4S –project and also TEKES (The Finnish Agency for Technology and Innovation), for providing me an environment to work with other researchers and Finnish software companies. In addition, I would like to thank all the companies, development teams and interviewees who had time to participate in this research. Without their insightful knowledge and experiences, the results of this research would not have been possible.

I would also like to thank my colleagues in the department. Special thanks to Tarja Nikkinen, who helped me numerous times with daily issues when I had no clue what to do with reporting papers or travel applications and so on, and Ossi Taipale, who provided me guidance related to thesis writing through several insightful discussions. I would also like to thank all my friends. However, special thanks go to Oskar and Kimmo, who have kept me entertained during and after each work day, playing video games and eating fast food (maybe sometimes going to the gym).

However, the biggest and most special thanks go to my family, my mother Terhi, father Hannu, and little sister Noora, because without all the sacrifices and the life they have
provided me, I would not be here writing this acknowledgement. They have kept me on a right path in life and always guided me when I did not know what to do. Also, all my grandparents and other relatives, who have supported me and kept me motivated to complete my studies, deserve big thank.

My longtime girlfriend Amie, I am truly lucky to have you on my side, thank you for supporting me always.

Lappeenranta, 2017  
Jesse Yli-Huumo
"A delayed game is eventually good, a bad game is bad forever."

-Shigeru Miyamoto
Contents

Abstract
Acknowledgements
List of publications 11
Author’s contribution 12
Symbols and abbreviations 13
1 Introduction 15
2 Technical debt 19
  2.1 Definition of technical debt ................................................................. 19
  2.2 History of technical debt ................................................................. 22
  2.3 Main attributes of technical debt ......................................................... 25
  2.4 Technical debt management in software development .......................... 29
  2.5 The relationship of technical debt to the field of software engineering 33
  2.6 Summary ......................................................................................... 35
3 Research problem and methodology 38
  3.1 Viewpoints and research problems ...................................................... 38
  3.2 The research questions ................................................................. 41
  3.3 Research methodologies .................................................................. 43
  3.4 Philosophical perspectives on qualitative research ............................... 44
  3.5 Selection of research methodologies ................................................. 45
    3.5.1 Case study ............................................................................. 47
    3.5.2 Action research ........................................................................ 49
  3.6 Research process ............................................................................ 50
    3.6.1 Data collection ........................................................................ 52
    3.6.2 Data analysis ........................................................................... 55
    3.6.3 Finishing and reporting on the research ....................................... 56
  3.7 Summary ......................................................................................... 57
4 Overview of the publications 60
  4.1 Publication I: The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company ......................................................................................... 60
    4.1.1 Research objectives .................................................................. 60
    4.1.2 Results .................................................................................... 60
    4.1.3 Relation to the whole ............................................................... 61
  4.2 Publication II: The Benefits and Consequences of Workarounds in Software Development Projects ......................................................................................... 63
    4.2.1 Research objectives .................................................................. 63
4.2.2 Results .......................................................................................... 63
4.2.3 Relation to the whole.............................................................. 65
4.3 Publication III: The Effects of Software Process Evolution to Technical Debt – Perceptions from Three Large Software Projects .................. 65
  4.3.1 Research objectives .............................................................. 65
  4.3.2 Results ....................................................................................... 66
  4.3.3 Relation to the whole.............................................................. 68
  4.4.1 Research objectives .............................................................. 68
  4.4.2 Results ....................................................................................... 69
  4.4.3 Relation to the whole.............................................................. 70
4.5 Publication V: Developing Processes to Increase Technical Debt Visibility and Manageability – An Action Research Study in Industry ...................... 71
  4.5.1 Research objectives .............................................................. 71
  4.5.2 Results ....................................................................................... 71
  4.5.3 Relation to the whole.............................................................. 72
4.6 About the joint publications.................................................. 73

5 Research contributions, implications and limitations 75
  5.1 Reasons and causes for technical debt in software ...................... 75
  5.2 Effects of technical debt on software and software development ...... 78
  5.3 Technical debt management in software development .................. 80
  5.4 Summary of contributions to technical debt state-of-the-art ............ 83
  5.5 Implications for practice ............................................................ 84
  5.6 Implications for future research .................................................. 87
  5.7 Validity and limitations of the thesis .............................................. 88

6 Conclusions 94
  6.1 Contributions and summary .................................................. 94
  6.2 Future research topics ............................................................... 95

References 98

Appendix A: Publications 109
List of publications

This thesis is based on the following papers. The rights have been granted by the publishers to include the papers in the thesis.


In this thesis, these publications are referred to as Publication I, Publication II, Publication III, Publication IV, and Publication V.
Author’s contribution

Publication I “The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company” studied the role of technical debt by observing the causes and effects of it. The results included multiple examples of both intentional and unintentional technical debt with various short-term and long-term effects. The empirical research was conducted with semi-structured interviews, where Yli-Huumo designed, organized, carried out, transcribed and analysed all the interviews. Yli-Huumo was the main author of the conference paper.

Publication II “The Benefits and Consequences of Workarounds in Software Development Projects” is a study that was conducted to examine the benefits and consequences of workarounds during software development. The results provided several examples of technical debt, which included both benefits and consequences, in software and software development. The empirical research was conducted with semi-structured interviews in two case companies. The interviews were designed, executed, transcribed and analysed by Yli-Huumo. Yli-Huumo was the main author of the conference paper.

Publication III “The Effects of Software Process Evolution to Technical Debt – Perceptions from Three Large Software Projects” highlights the effects and relationships technical debt has to the evolution and improvement of the software process. The results indicated that there exists a relationship between technical debt and software process evolution. The results provided examples where companies faced both benefits and issues regarding technical debt during the evolution of software processes. The empirical research was conducted with several semi-structured interviews in three case companies. The interviews were organized, designed, executed, transcribed and analysed by Yli-Huumo. The study was published as a book chapter, where Yli-Huumo was the main author.

Publication IV “How Do Software Development Teams Manage Technical Debt? – An Empirical Study” developed a framework for technical debt management which can be used by software development companies to examine their current processes and activities related to technical debt. The empirical research was conducted in eight software development teams with semi-structured interviews. Yli-Huumo designed the interview structure, organized and executed the interviews, and performed data analysis. Yli-Huumo was the main author of the journal paper, which was published in Journal of Systems and Software special issue.

Publication V “Developing Processes to Increase Technical Debt Visibility and Manageability – An Action Research Study in Industry” presents the results of an action research study. The study developed and tested a set of templates and processes that can be used in technical debt management. Yli-Huumo designed and executed the action research process, which included interviews and a workshop. Yli-Huumo also analysed the data and developed the templates and processes with company representatives. The paper was published as a conference paper, where Yli-Huumo was the main author.
Symbols and abbreviations

ASA  Automatic Static Analysis
LOC  Lines of Code
MTD  Managing Technical Debt (workshop name)
MVP  Minimum Viable Product
N4S  Need 4 Speed (project name)
RQ   Research Question
SLR  Systematic Literature Review
SME  Small and Medium-sized Enterprises
TD   Technical Debt
TDM  Technical Debt Management
1 Introduction

Software industry is a growing market and business sector, where new software-intensive products, services and systems are developed and released to end-users (Shields, 2014). Therefore, the software industry today is highly competitive, and software companies are required to improve their products and features constantly to keep the end-users interested (Porter and Heppelmann, 2014). The high level of competition creates pressure for software companies to deliver high-quality products faster than their competitors (Vesey, 1992), which can be also called as time-to-market. To address these challenges, software companies are constantly searching for improvements to their software development processes to deliver high-quality products and features in lesser time with the same resources and time available (Fan et al., 2009; Garlan and Perry, 1995).

The highest quality product does not always necessarily acquire the most success (Cohen et al., 1996). Instead, a software product that was released to the market before other competition, can have a good chance to acquire early users (Bays, 1999). To ensure earlier release to market, the software company can make the strategic decision to release the product e.g. with lesser overall quality, functionality, or performance to make the product available to the end-users faster (Bays, 1999). Even though the software product might not be the most sound one from the technical perspective, it can still provide enough functionality to the end-user, which could be the deciding factor for purchase. Since the software development today has become more flexible (Neely and Stolt, 2013), it is possible to release new versions and patches on a daily basis. It is also possible for software companies to fix technical issues easier after the initial release, which has made earlier release cycles a popular strategy in software development (Greer and Ruhe, 2004; Humble and Farley, 2010).

As a successful example, this type of flexible software development was seen at the early stages of Dropbox, a file-hosting service company (Ries, 2011a). In the beginning of the product development, the company used lean software development methodologies to deliver a minimum viable product (MVP) (Ries, 2011b) that included major workarounds and shortcuts in the software (Ries, 2011a). By using this strategy, the company was able to test the markets, make the needed adjustments, and acquire early contact with the possible customers (ibid.). Even though the product was not in the best possible state from the technical perspective, the decision to enter the market early with an average product rather than polishing the product to technical perfection was successful when looking at the success Dropbox has today.

The benefit of fast software development and earlier release done with workarounds and shortcuts can be seen in the time-to-market (Kruchten et al., 2012a). Taking workarounds and creating shortcuts can speed up the software development and ensure that the product and features are released under the planned schedule, or even earlier (Yli-Huumo et al., 2015). In addition, a typical end-user or customer is not necessarily interested in the actual technical implementation in the source code and architecture, or there is not even permission for the software vendor to access it (Brown et al., 2010; Freeman, 2004).
Instead, the end-user can be interested in the functionality of the requested features and the promised release dates (Kitchenham and Pfleeger, 1996). This gives software companies more room to make strategic decisions to implement shortcuts and workarounds to the source code, as long as the product provides enough functionality to the end-user and it is done within the agreed deadlines and resources (Brown et al., 2010).

The phenomenon in software development described above is called ‘technical debt’ (TD) (Cunningham, 1992). Technical debt refers to a situation in software development where a technical decision is made to implement a non-optimal solution to the software product (Cunningham, 1992; Kruchten et al., 2012b). Non-optimal solution is a quick implementation that is easier to take in use compared to optimal solution that takes longer time to implement. Taking technical debt can speed up the development and provide time-to-market for software companies (Kruchten et al., 2012a). However, the drawback of leaving shortcuts and workarounds to software can be omitted quality (Zazworka et al., 2011b). Workaround or shortcut is a goal-driven adaptation and improvisation that aims to minimize negative consequences like anomalies or structural changes (Alter, 2014). When the number of non-optimal solutions in the source code starts to increase, it can create complexity, which can be seen as a technical debt in the software that has to be paid for eventually (Lim et al., 2012). Technical debt can be considered as a risk and challenge towards software and software development (Kruchten et al., 2012a; Tom et al., 2013). If technical debt is not controlled, there is the possibility that the sustainability and maintainability of the software product will become too difficult and expensive for the software company to deal with later (Guo et al., 2011; Zazworka et al., 2011b).

There is an urgent need to understand and address the role of technical debt in software development (Kruchten et al., 2012a). It is important to gain understanding through research on the reasons for technical debt and what causes it to occur in software. In addition, it is important to understand what effects technical debt has on both software and software development. By understanding the causes and effects of technical debt, an answer to what constitutes technical debt can be found (Kruchten et al., 2012a). In addition, by understanding the causes and effects of technical debt, it can be possible to develop activities for technical debt management that can help software companies to manage and repay technical debt more efficiently.

In this doctoral thesis, the main goal is to increase the empirical knowledge on the role of technical debt in software and software development. The research focuses on technical debt from the organizational perspective, with the aim to understand how technical debt occurs from the point of social, political and management-related aspects in software development, rather than the qualities of technical debt in the software source code and design. The focus can be further divided into two main topics. First, the focus is on understanding the causes and effects of technical debt. The objective is to observe the causes and reasons for technical debt to occur in software, and what are the effects of technical debt on software and software development. Second, the focus is on technical debt management, especially on the practices and processes that are conducted during software development to manage and reduce technical debt. The objective is to study how
software development teams manage, reduce and prevent technical debt during software development. To address these research objectives, multiple case studies and interviews with practitioners have been performed in the software development environment. The outcome focuses on increasing the overall empirical knowledge on technical debt in software and software development, and on providing various practices to manage and reduce technical debt efficiently during software development.

The thesis comprises two main parts, an introduction and an appendix. The introduction consists of six chapters. Chapter 2 introduces the background of the thesis by presenting the definitions, history and main attributes related to technical debt. In addition, Chapter 2 presents the background and main concepts that are related to technical debt management. Chapter 3 describes the research process, including the defined research questions and the used research methods. Chapter 4 gives an overview of the scientific publications included in the thesis. Chapter 5 discusses the contributions and implications of the research results. Chapter 6 summarizes the main research findings, and discusses possible future research topics. The appendix comprises five scientific publications, which present the research results in closer detail.
1 Introduction
2 Technical debt

The purpose of this chapter is to present the central themes, concepts and definitions which form the background of this thesis. The themes, concepts and definitions have been obtained from related scientific research and literature. First, this chapter presents various definitions of technical debt that include different viewpoints to the question of what the term technical debt means. Then, a brief history of technical debt is given from the time when technical debt was first discussed by using varying terminology, to the status technical debt has today. After this, the main attributes that are associated with technical debt in software development are presented in detail. The last part of this chapter presents the concept of technical debt management and discusses how technical debt is related to other aspects of software engineering.

2.1 Definition of technical debt

Technical debt (TD) is a metaphor used in software engineering to describe a situation where a non-optimal solution is used in a technical decision during software development. The metaphor was first coined in 1992 by Ward Cunningham, when he described technical debt as follows:

‘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... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.’ – Ward Cunningham (1992)

The original definition by Ward Cunningham (1992) shows that technical debt as a metaphor describes a situation in software development where implementing non-optimal code can speed up development in the short term, but in the long term it can incur a risk, if these non-optimal solutions are not fixed. Michael Lutz (1993) states that it is important for software companies to borrow time sometimes to get a product to market, but it will include an interest later on.

“Just as new firms borrow capital to get started, new software projects borrow ‘design capital’ (time) to get a product to market. Maintenance problems that ensue are the interest you pay for design errors introduced by schedule pressure” – Michael Lutz (1993)

Martin Fowler (2003) has simplified the metaphor later, as he describes technical debt ultimately as a decision between a messy solution done quick versus a clean solution done with time. Steve McConnell (2008) emphasizes in his own technical debt definition the perspective on cost, where implementing a non-optimal design or construct can have an
Technical debt can be a useful metaphor in communicating about software and its quality between technical staff; but interestingly it can also be used as an important technical concept for non-technical stakeholders (McConnell, 2007). Therefore, technical debt has been tried to be explained and defined through other domains, such as finance (Allman, 2012). Martin Fowler (2003) suggests that technical debt can be explained through finance debt because both share similar main properties.

‘Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future.’ – Martin Fowler (2003)

Technical debt has three main attributes that are similar to financial debt: Repayment, interest and high cost (Allman, 2012). These attributes are used to define technical debt in a non-technical manner (Allman, 2012). In a financial debt, the person who grants the loan has the expectation that the debt will get repaid as soon as possible or in a given deadline. With a debt, there is often an interest included, which is some percentage that has to be paid with the debt. A person often pays the debt back on a monthly basis, until the whole debt has been paid back. However, it is also possible that the person is not able to fulfil the repayment, which then results in e.g. bankruptcy. A similar situation can happen also in software development with technical debt. A developer implements a non-optimal code or design approach and decides to incur some technical debt. With this debt, the developer can usually finalize the current urgent development task. There is an expectation in the software project that the debt will be fixed as soon as possible to avoid later consequences. In software development, shortcuts and workarounds in the code can start to omit overall quality, because it generates complexity (Lim et al., 2012). The time required to improve this omitted quality can be seen as a principal. In software development, when too much technical debt has been taken and not repaid in time, the overall quality can decrease under a certain level, where it may not be possible to do any changes to the software without breaking it, due to complexity. In worst cases, this could mean the end of the software, similar to bankruptcy.
2.1 Definition of technical debt

In the past few years, the definition of technical debt has evolved from its origin (Tom et al., 2013). There exists a number of different definitions and opinions of technical debt in the academic literature at the moment (Tom et al., 2013). Tom et al. (ibid.) have conducted a systematic literature review (SLR) to explore the current understanding on technical debt. The results showed that technical debt as a definition can have multiple meanings in the literature, which can make it a challenging metaphor to define and discuss. Although most definitions of technical debt are still referred to as design and implementation related issues, similarly to the original definitions (Cunningham, 1992; Fowler, 2003; McConnell, 2008), there are some questions about the definition (Tom et al., 2013). Since the original definitions of technical debt did not take an accurate stand on what the actual technical debt is in software, it has generated a lot of confusion to the terminology (Tom et al., 2013) concerning technical debt. According to some definitions, technical debt should be everything ‘incomplete’ in software (ibid.). However, some definitions do not agree that e.g. defects and bugs should be included in technical debt (Li et al., 2015; McConnell, 2007; Tom et al., 2013). In addition, the original definition indicated that technical debt happens as part of software implementation (Cunningham, 1992). However, today several definitions also take in consideration other stages of software development (Tom et al., 2013). Therefore, the current definition of technical debt in the academic literature can be considered incoherent, requiring more research and discussion to reach a common consensus.

To address the issue of definition, the most recent notable definition of technical debt was developed in April 2016, when a technical debt seminar gathered technical debt researchers together to discuss the current status of technical debt. As a result, the definition of technical debt was refined, and it can be seen as the latest attempt to simplify the metaphor (Avgeriou et al., 2016):

“In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.”

Overall, the technical debt metaphor unites the technical and economic perspectives of software development. Even though technical debt can be considered as a technical approach in code design or implementation (Avgeriou et al., 2016; Cunningham, 1992; McConnell, 2008), it also includes economical aspects related to time and cost (McConnell, 2008). Some of the definitions of technical debt take account of a more technical perspective with technical attributes of software, while some discuss economic perspectives with attributes like people and social aspects of software development (Alves et al., 2016; Li et al., 2015; Tom et al., 2013). At the moment, there is not necessarily one accepted definition available for technical debt. This suggests that technical debt is a complicated phenomenon in software development that requires more research and understanding to reach a consensus on its definition in the academia and in practice.
2.2 History of technical debt

Even though the technical debt metaphor was first mentioned in 1992 (Cunningham, 1992), it is not the only metaphor that has been used to describe shortcuts, workarounds and non-optimal solutions resulting in complexity in software source code, design, or architecture. The same phenomenon has been discussed with varying terminology in the academia and industry both before and after the term technical debt was launched (Avgeriou et al., 2016). These metaphors include terms such as spaghetti code (Dijkstra, 1968), big ball of mud (Foote and Yoder, 1997), and code smells (Fowler et al., 1999), which are related to the sustainability, maintainability, and evolvability of a software. In addition, terms such as legacy code (Bennett, 1995), legacy system (Bennett, 1995), and software aging (Huang et al., 1995; Parnas, 1994) have been used to describe solutions developed years ago that could be replaceable with newer technology, and can be seen as technical debt.

Spaghetti code (Dijkstra, 1968) describes a situation in software development where the source code has a complex and tangled control structure (Cram and Hedley, 2005; Mikkonen and Taivalsaari, 2007). Examples of spaghetti code are e.g. the use of too many gotos, exceptions, threads, global variables, or other "unstructured" constructs (Mikkonen and Taivalsaari, 2007). The name spaghetti code is used to picture a situation in software development where the program flow looks like a bowl of twisted and tangled spaghetti, which is difficult to understand (ibid.).

Big ball of mud Foote and Yoder (1997) describe software or a system as a big ball of mud, when it is ‘haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle’. A big ball of mud is caused by issues in software design and architecture (Vainsencher, 2004). It emerges from throwaway code, which can be described as a quick and dirty code, intended to use once and then removed (Foote and Yoder, 1997). However, a piece of code that works does not often get removed from the code, even though it had been designed badly (ibid.), which can make it a challenge in the future.

Code smells Fowler (2006) describes code smell as an indication that usually corresponds to a deeper problem in the system or program, which can lead to less maintainable code. Code smells refers to poor solutions in recurring implementation problems that are found in both design and implementation (Khomh et al., 2009). Detecting bad smells usually helps to understand the need for code refactoring (Mäntylä et al., 2003). Fowler et al. (1999) present 22 examples of code smells, including examples such as duplicated code, long method, large class, long parameter list etc. Code smells as a term is a known metaphor in the literature and can often be associated with technical debt (Vale et al., 2014; Zhang et al., 2011).

Legacy code, Legacy system. Bennet (1995) defines a legacy system as a large software system that is hard to live with, but which is still essential and beneficial to the company. Legacy code can also be a software code that has been acquired from another person or
2.2 History of technical debt

company, which is tangled and difficult to make changes to, and does not have written tests in it (Feathers, 2004). A legacy system and legacy code were often developed with state-of-the-art techniques in the past, but successful software inevitably always evolves, which requires changes to be made (Bennett, 1995).

Software aging Software aging refers to a situation in software where accumulation of errors eventually results in a crash / failure (Garg et al., 1998). Software aging is like human aging, except that with software it is possible to slow the process and make changes that will reverse the situation (Parnas, 1994). Software aging also ties up a lot of economic issues, because changing and updating software is never cheap, and it’s also dangerous to change something old that is still working (Parnas, 1994).

The terminology and definitions related to technical debt combined with the original definitions of technical debt show that the phenomenon is often associated with situations in software design and implementation. However, some of the original definitions suggested that rather than just being a technical context, the metaphor also includes attributes related to time and cost (Fowler, 2003; Lutz, 1993; McConnell, 2008). Also, the original definitions did not describe accurately what technical debt is in software specifically (Tom et al., 2013). Therefore, the technical debt metaphor has evolved and got various new perspectives describing the same phenomenon of non-optimal solutions and work in other areas of software development (Alves et al., 2014; Tom et al., 2013). The metaphor has been expanded to other stages of software development, which has produced several other categories and subcategories for technical debt (Alves et al., 2014). Currently, the metaphor is no longer related only to software design and implementation as one single term technical debt, but also to shortcuts, workarounds, and non-optimal solutions done in other stages of software development (Alves et al., 2014, 2016; Li et al., 2015; Tom et al., 2013). There are systematic literature studies that have collected and identified the subcategories of technical debt (Alves et al., 2014, 2016; Li et al., 2015; Tom et al., 2013). The subcategories of technical debt are presented in Table 1.

<table>
<thead>
<tr>
<th>Subcategory</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Debt</strong> (Alves et al., 2014, 2016; Bohnet and Döllner, 2011; Li et al., 2015; Tom et al., 2013; Zazworka et al., 2013)</td>
<td>Poorly written code that can be found as a problem in the source code, requires refactoring and violates best programming practices or programming rules. Examples of code debt are code duplication, over-complex code, bad style that reduces readability, and poorly organized logic.</td>
</tr>
<tr>
<td><strong>Design Debt</strong> (Alves et al., 2014, 2016; Guo and Seaman, 2011; Izurieta et al., 2012; Li et al., 2015; Tom et al., 2013; Zazworka et al., 2013)</td>
<td>Violations and shortcuts in the principles of good design that underfocus on qualities such as maintainability and adaptability.</td>
</tr>
<tr>
<td><strong>Architecture Debt</strong> (Alves et al., 2014, 2016; Brown et al., 2010; Kruchten et al., 2012a; Li et al., 2015; Tom et al., 2013)</td>
<td>Sub-optimal solutions and bad decisions in software architecture, which compromise internal quality aspects, such as maintainability, performance, robustness, modularity etc. In addition, architecture debt can be a solution that becomes sub-optimal when technologies</td>
</tr>
</tbody>
</table>
and patterns come superseded. Architectural type of debt requires extensive development activities to fix.

<table>
<thead>
<tr>
<th>Debt Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environmental Debt (Tom et al., 2013)</td>
<td>Debt that happens in the environment of the application, which can include development, hardware, infrastructure, and supporting-related challenges; for example, manual processes that could be automated to increase productivity, or postponement of upgrades to infrastructures and components.</td>
</tr>
<tr>
<td>Knowledge Distribution Debt (Slinker, 2016; Tom et al., 2013)</td>
<td>The knowledge of software with long development history and millions of lines of code developed by original team members can suddenly change if the team is changed. This has a sudden effect, because the original knowledge is not transferred to new team members.</td>
</tr>
<tr>
<td>Documentation Debt (Alves et al., 2014, 2016; Guo and Seaman, 2011; Li et al., 2015; Tom et al., 2013)</td>
<td>Missing, inadequate, insufficient, incomplete, or outdated documentation found in any aspect of software development.</td>
</tr>
<tr>
<td>Test Debt (Alves et al., 2014, 2016; Guo and Seaman, 2011; Li et al., 2015; Tom et al., 2013)</td>
<td>Shortcuts in testing such as lack of unit tests, integration tests, acceptance tests, written test scripts, code coverage, or the number of test cases being completed.</td>
</tr>
<tr>
<td>Requirements Debt (Alves et al., 2014, 2016; Ernst, 2012; Kruchten et al., 2012a; Li et al., 2015)</td>
<td>The distance between optimal requirements specification and actual system implementation, for example, requirements that are only partially implemented.</td>
</tr>
<tr>
<td>Build Debt (Alves et al., 2014, 2016; Li et al., 2015; Morgenthaler et al., 2012)</td>
<td>Flaws in a software system built process, which makes it overly complex, time consuming, unnecessarily slow, and difficult.</td>
</tr>
<tr>
<td>Infrastructure Debt (Alves et al., 2014, 2016; Li et al., 2015; Seaman and Spinola, 2013)</td>
<td>Sub-optimal configuration of development-related processes, activities, technologies, supporting tools etc. Delays upgrades and infrastructure fixes, which have an effect on team productivity.</td>
</tr>
<tr>
<td>Versioning Debt (Alves et al., 2016; Greening, 2013; Li et al., 2015)</td>
<td>Problems in source code versioning, such as unnecessary code forks.</td>
</tr>
<tr>
<td>Defect Debt (Alves et al., 2014, 2016; Li et al., 2015; Snipes et al., 2012)</td>
<td>Defects, bugs, or failures found in software, which are often identified by testing activities or bug tracking systems.</td>
</tr>
<tr>
<td>People Debt (Alves et al., 2014, 2016; Seaman and Spinola, 2013; Tom et al., 2013)</td>
<td>Challenges with people in the software organization, which can have an effect on productivity cause delays in e.g. development activities.</td>
</tr>
<tr>
<td>Test Automation Debt (Alves et al., 2014, 2016; Codabux and Williams, 2013; Wiklund et al., 2012)</td>
<td>Lack of work involved in automating tests to support continuous integration and faster development cycles.</td>
</tr>
<tr>
<td>Process Debt (Alves et al., 2014, 2016; Codabux and Williams, 2013)</td>
<td>Inefficient processes designed to handle certain tasks may be no longer appropriate.</td>
</tr>
<tr>
<td>Service Debt (Alves et al., 2014, 2016; Alzaghoul and Balsoon, 2013)</td>
<td>In service-oriented architectures, inappropriate selection of web services that leads to issues and mismatch of service features and requirements.</td>
</tr>
<tr>
<td>Usability Debt (Alves et al., 2016; Potdar and Shihab, 2014; Zazworka et al., 2013)</td>
<td>Inappropriate usability decisions and standards that need to be changed in the future, for example, inconsistence in the navigational aspects of software.</td>
</tr>
</tbody>
</table>
2.3 Main attributes of technical debt

<table>
<thead>
<tr>
<th>Social Debt (Tamburri et al., 2013, 2015)</th>
<th>“Sub-optimal” development community, which affects both social and technical aspects of software development.</th>
</tr>
</thead>
</table>

Technical debt is not the first or the latest term to describe complexity in software design and implementation that results in issues in code sustainability and maintainability. The same phenomenon has been described with other terms for the past 35 years, but has received more attention in the past 10 years (Agerioui et al., 2016). Compared to other terminology, technical debt as a metaphor turns the attention to the economic side of having technical complexity, which is the reason for the phenomenon to be more interesting for both academia and industry (ibid.). Recent literature reviews have shown that the interest on technical debt has been increasing significantly lately (Alves et al., 2016; Li et al., 2015). The growing interest shows that technical debt is a recognized term today in both academia and industry to describe technical complexity in software.

2.3 Main attributes of technical debt

Even though there exist various definitions and subcategories of technical debt (Alves et al., 2016; Li et al., 2015), most of them share the same main attributes that explain how the overall phenomenon emerges and behaves. McConnell (2007) divides technical debt into two main types: *intentional technical debt* and *unintentional technical debt*. Intentional technical debt is often caused by a strategic decision to take technical debt knowingly during software development (Brown et al., 2010; Lim et al., 2012; McConnell, 2007). This type of technical debt can be associated with the original definitions of technical debt (Cunningham, 1992; Fowler, 2003). Intentional technical debt can happen in software development e.g. in a situation where the company and development team have a tight schedule in a new feature release to a customer (Kruchten et al., 2013). Tight schedules and deadlines can force the development team and management to make the decision to implement some shortcuts and workarounds during the development, to ensure that the product is released on time (Martini et al., 2014).

Even though technical debt was first described as an intentional and strategic decision between the optimal and a suboptimal solution (Fowler, 2003), there can also exist the dimension of unintentional technical debt (McConnell, 2007). This type can be described as technical debt that comes to software unknowingly (Klinger et al., 2011). Even though technical debt does not come in this situation through an intentional decision, it still results in the same end situation of having non-optimal solutions that should be repaid. For example, an unexperienced developer can write code that does not meet the standards and quality set by the company (Nord et al., 2012). Technical debt can also come to software unintentionally through legacy code and software aging, where the old code and design implemented several years ago do not match the current standards and technologies needed to ensure software quality and functionality (Brown et al., 2010). Therefore, unintentional technical debt can also be considered as technical debt.
In addition to intentional and unintentional technical debt (McConnell, 2007), Fowler (2009) adds that technical debt can be caused by different attitudes in software development as well. Figure 1 shows that reasons for technical debt can be divided into four dimensions. These dimensions are presented in a technical debt quadrant including the dimensions of reckless, prudent, deliberate, and inadvertent technical debt (Fowler, 2009).

![Figure 1. Technical debt quadrant (Fowler, 2009)](image)

Ramakrishan (2013) explains and discusses the dimensions of the technical debt quadrant. Technical debt can be deliberate and reckless when there is an urgent need to release something. This can often be a result of poor management, where the main goal is just to cut corners to ensure that the software will be released in time (Ramakrishnan, 2013). This type of technical debt can be seen as business-driven software development, where the goal is focused more on feature releases and deadlines than product quality and maintainability.

Technical debt can be deliberate and prudent when there is a business-driven need for release. Different from reckless, prudent tends to exploit a positive opportunity with return-on-investment, rather than just having the goal to release the product to meet a deadline. Prudent and deliberate technical debt is common in start-ups and companies that are entering new markets, which sometimes requires some shortcuts and workarounds. (Ramakrishan, 2013)

Technical debt can be inadvertent and reckless when the developer does not have the needed competence or is unaware of techniques to produce high quality code and
2.3 Main attributes of technical debt

solutions. Technical debt should be controlled with programming standards, processes and tools. (Ramakrishnan, 2013)

_Inadvertent_ and _prudent_ technical debt happens through natural occurrence (Ramakrishnan, 2013). Sometimes technical debt just occurs in software unknowingly without any specific reason. However, _inadvertent_ and _prudent_ technical debt helps developers to improve their performance through the experience and knowledge gained from this technical debt (Fowler, 2009).

Technical debt also includes the dimension of longevity. There exist both short-term and long-term technical debt (McConnell, 2007). Short-term technical debt is reactive and tactical, to ensure e.g. a release (Ramakrishnan, 2013). Long-term technical debt is proactive and strategic to disregard e.g. support for a certain feature, because it will not be used in the near future. Short-term technical debt is expected to be repaid quickly, while long-term technical debt can be carried in a software for years without trouble (Ramakrishnan, 2013).

Regardless the type of technical debt, there is always a reason for causing it and an effect as the outcome. To gain more understanding on the various attributes related to technical debt, Tom et al. (2013) have conducted a systematic literature review to identify the attributes, reasons, and outcomes of technical debt in software development. Table 2 summarizes the findings and presents the general attributes that can be associated with technical debt.

<table>
<thead>
<tr>
<th>Attributes of technical debt</th>
<th>Details</th>
</tr>
</thead>
</table>
| **Monetary cost**  
(Tom et al., 2013) | Technical debt has a negative impact on morale, productivity, quality, and risk, which shows that technical debt has also an effect on monetary cost. By having technical debt, the developer's velocity is slowed and the development time goes to fixing technical debt rather than for new features. Since developers' time and work cost, it is inefficiently used when working with technical debt. |
| **Amnesty**  
(McConnell, 2007; Ries, 2009; Tom et al., 2013) | Technical debt does not always have to be paid off. For example, a throwaway prototype can include technical debt while being developed, and in a situation where the feature or product is deemed to failure, the prototype is not needed and the technical debt does not need to be paid off. Technical debt can also retire with a system when it is at the end of its lifecycle. |
| **Bankruptcy**  
(Elm, 2009; Tom et al., 2013) | In a situation where technical debt has been accumulated for an overwhelming amount, bankruptcy can occur. In this situation, the software progress will stop and rewriting is required, because the cost for improvement can be greater than starting from the beginning. |
| **Interest and principal**  
(Tom et al., 2013) | A fundamental characteristic of technical debt is that interest of technical debt needs to be paid off in addition to principal payments. |
| **Leverage**  
(Tom et al., 2013) | Technical debt can be used intentionally as leverage, and it can show as increase in productivity in the short term. This allows software companies to use technical debt as a strategy to take shortcuts and use less work to achieve the goal. |
Repayment and withdrawal (McConnell, 2007; Tom et al., 2013) Repayments and withdrawals in technical debt can be characterized as small interactions done with a credit card. This type of technical debt is often a conscious decision, but it is also easy to accumulate unintentionally.

Reasons for organizations to take technical debt

<table>
<thead>
<tr>
<th>Pragmatism (Haack, 2005; Tom et al., 2013)</th>
<th>Sometimes the business needs of the software company have to be valued over design quality, which leads to the use of fast shortcuts and workarounds in the code. There exist numerous small refactoring instances, which is why pragmatism is a reason for sub-optimal implementation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prioritization (Tom et al., 2013)</td>
<td>In software development, there are situations where a decision has to be made between the prioritization of critical functions vs. overall quality. Often the critical functions are prioritized over quality in order to be able to deliver functionality within project constraints. Therefore, the development team has to make trade-offs in the overall quality to be able to deliver functionality, which is often a reason for technical debt.</td>
</tr>
<tr>
<td>Processes (Tom et al., 2013)</td>
<td>The processes used by a development team can also be a reason for technical debt. The visibility and manageability of technical debt decreases when there is poor communication and collaboration. This leads to the situation where technical debt accumulates slowly without knowledge.</td>
</tr>
<tr>
<td>Attitudes (Elm, 2009; Tom et al., 2013)</td>
<td>Technical debt can be caused by attitudes towards development. The developers and management can hesitate when it comes to code improvement. If the functionality and software is working and the customer is happy, it might be a challenge to start fixing something that is not necessarily broken. This can lead to more technical debt being taken and increase the level of carelessness.</td>
</tr>
<tr>
<td>Ignorance and oversight (Tom et al., 2013)</td>
<td>Ignorance refers to ignorance on how developers can avoid technical debt rather than ignorance of its presence. Oversight is a form of ignorance, where the developers can be unaware of issues and mistakes that create technical debt.</td>
</tr>
</tbody>
</table>

Outcomes of technical debt

| Impact on morale (Tom et al., 2013) | Developers have to work with some form of technical debt every day, and this may be the reason why the effects of technical debt are frustrating to deal with. |
| Impact on productivity (Tom et al., 2013) | If technical debt does not get repaid in time, it requires more interest payments to be dealt with (possibly even more technical debt). In the long term, development velocity and productivity can decrease in a number of ways. |
| Impact on quality (Tom et al., 2013) | Technical debt decreases the product quality, because it can create defects, bugs, and other structural quality issues. In the short term, technical debt creates various quality issues, depending on the type and size of technical debt. In the long term, when technical debt starts to accumulate, the impact starts also to show as further interest costs. |
| Impact on risk (Tom et al., 2013) | Technical debt can create uncertainty within development and management, because even for the best developers it is extremely difficult to make estimations around technical debt. This has an impact on the risks in the project, which is further challenging, if technical debt is not visible. |

Overall, there are both intentional and unintentional reasons for technical debt to occur in software. In addition, the effects of technical debt vary depending on its length (McConnell, 2007). Technical debt must often be repaid, but there are also situations where repayment is not necessarily needed (Buschmann, 2011). The variety of different
attributes associated to technical debt show that as a phenomenon, technical debt can include both technical and economic viewpoints, which make it an interesting research area (Ampatzoglou et al., 2015; Avgeriou et al., 2016). Even though technical debt is essentially a technical artefact in code or design construct (Cunningham, 1992), its attributes, reasons, and effects are not always related to code and programming activities, but can also include various aspects related to companies' social, political and organizational activities (Avgeriou et al., 2016; Tom et al., 2013). In addition, technical debt can also be associated with “non-code artefacts” (Brown et al., 2010), such as requirements and testing. Even though technical debt, as a shortcut or workaround, should never be the most optimal and best way to work with source code, taking technical debt can sometimes be the right decision to do for the company from the economic perspective (Buschmann, 2011; Lim et al., 2012). These aspects show that technical debt can vary a lot and it is a complex phenomenon, which requires research to gain more knowledge of its role in software development.

2.4 Technical debt management in software development

Technical debt management (TDM) includes activities that are used to manage and reduce technical debt in a software development project. Technical debt management can include different processes, models, frameworks, techniques, or tools that are used to achieve better manageability of technical debt. With technical debt management, the goal for a software development company is to manage, reduce and prevent shortcuts, workarounds, and non-optimal solutions successfully. (Li et al., 2015)

The attributes related to technical debt make the phenomenon complex in software development and are the reason why it poses challenges to management activities. Power (2013) presents seven challenges to technical debt and its management: (1) **Agreeing what technical debt is** – The definition of technical debt can be still considered unclear and it does not necessarily have a common terminology in both the academia and the industry. Therefore, a confusion during software development can be possible when the terminology has different meanings; (2) **Quantifying technical debt** – Whereas e.g. bugs and lines of code (LOC) are easy to quantify from software, a non-optimal code, shortcut or workaround can be much more difficult to understand and quantify, especially if it is unknown; (3) **Visualizing technical debt** – A non-optimal code, design or architecture is difficult to identify from the source code. Sometimes it may not be even possible to know whether some solution or design is good or bad; (4) **Tracking technical debt over time** – In a case where technical debt is taken intentionally, it is a challenge to track its evolution over time. A non-optimal solution taken in the past may have caused multiple smaller technical debt issues, which can be challenging to track down; (5) **Impact of neglecting technical debt over multiple releases** – Software development can often be time-consuming and include deadlines that the developers need to meet. This is why technical debt may sometimes be neglected during release, which will start to accumulate technical debt even more; (6) **Identifying technical debt as a root cause of defects** – Fixing a bug or small defect does not necessarily mean that the root cause is fixed. Small
problems can be caused by an underlying bigger technical debt problem, which can be challenging to point out; (7) **Understanding the cost of delay** – If the repayment of technical debt is delayed, it is difficult to understand and estimate what the real cost and effect on quality and productivity will be in the future.

Technical debt management has been studied in the current literature. Managing technical debt (MTD) workshops have provided several studies on technical debt and its management (Seaman et al., 2015). These studies provide several viewpoints on different activities in technical debt management. However, there is still need for standard practices and tools to manage technical debt (Ernst et al., 2015).

Technical debt management can be divided into eight main activities. The activities included in technical debt management are presented in Figure 2. The activities are technical debt **repayment, prevention, representation/documentation, identification, measurement, monitoring, communication, and prioritization**. (Li et al., 2015)

<table>
<thead>
<tr>
<th>Technical Debt Management Activities</th>
<th>Repayment</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation / Documentation</td>
<td>Identification</td>
<td>Measurement</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Communication</td>
<td>Prioritization</td>
</tr>
</tbody>
</table>

**Figure 2.** Technical debt management activities

**Repayment** resolves or mitigates technical debt in a software system by techniques such as reengineering and refactoring (Li et al., 2015, p. 205). In repayment, software developers and architects can utilize different approaches for technical debt, such as refactoring, reengineering, redesigning (Buschmann, 2011; Codabux and Williams, 2013). Refactoring is an approach to improve code quality and structure without changing the existing functional behaviour of a software (Fowler et al., 1999). Reengineering is used to switch an existing solution to an improved one (Chikofsky and Cross, 1990). Rewriting is used to reimplement a solution without using previous code (Chan et al.,
2.4 Technical debt management in software development

Repayment is also affected by the size of the required task (Zazworka et al., 2011b), which also shows in decision-making regarding technical debt (Zazworka et al., 2011a).

Prevention aims at preventing potential technical debt from being incurred (Li et al., 2015, p. 204). Technical debt prevention can be done with various approaches (Codabux et al., 2014; Krishna and Basu, 2012). For example, with code reviews developers can check each other's code to prevent possible bad design or code before software release (Baker, 1997; Kemerer and Paulk, 2009). The company can also set up some programming standards and guidelines for the development team to prevent technical debt and to improve code cohesion (Green and Ledgard, 2011). In addition, practices such as education and training, pair programming, test-driven development, refactoring, continuous integration, conformance to process and standards, tools, and customer feedback can help to prevent technical debt (Codabux et al., 2014).

Identification detects technical debt caused by intentional or unintentional technical decisions in a software system through specific techniques, such as static code analysis (Li et al., 2015, p. 204). Technical debt can be identified in the code and design through manual inspection or with specific tools. There are multiple tools developed that can be used for technical debt identification (Izurieta et al., 2012; Li et al., 2015), even though most of them were not necessarily developed for that purpose originally (Izurieta et al., 2012). Most of the code analysis tools can be used to identify minor technical debt issues, such as modularity violations, design patterns and grime build-up, code smells, and automatic static analysis (ASA) issues (ibid.). However, there are also technical debt types (e.g. large architectural and design issues (Kruchten et al., 2012a)) that cannot be identified with any specific tools at the moment (Izurieta et al., 2012). Therefore, larger technical debt issues have to be identified with human inspection and previous knowledge (Kruchten et al., 2012a).

Measurement quantifies the benefit and cost of known technical debt in a software system through estimation techniques, or estimates the level of the overall TD in a system (Li et al., 2015, p. 204). Technical debt measurement introduces a challenge to technical debt management (Curtis et al., 2012; Guo et al., 2011). Measuring the software code, design and architecture is not a simple process, and can consist of several variables that need to be taken in consideration (Curtis et al., 2012; Eisenberg, 2012). Without technical debt measurement, the decisions on technical debt are made on the basis of experience and hunch, which may cause misunderstanding on the real impact (Guo et al., 2011). This makes technical debt measurement an especially important research area.

Monitoring watches the changes of the cost and benefit of unresolved technical debt over time (Li et al., 2015, p. 204). Technical debt monitoring is one of the most vital activities in technical debt management (Ernst et al., 2015). Without monitoring activities, it is hard to understand and evaluate what effects taking and having technical debt has on the software. Technical debt identification and measurement enable on-going monitoring of technical debt over time (Seaman and Guo, 2011). Monitoring has the same challenges
as measurement, as the effects of changes in software are hard to follow and can include several variables that need to be taken into account.

**Representation/Documentation** provides a way to represent and codify technical debt in a uniform manner, addressing the concerns of particular stakeholders (Li et al., 2015, p. 205). Representation/documentation has been suggested for technical debt backlogs and lists (Guo and Seaman, 2011; Li et al., 2015; Seaman and Guo, 2011). The technical debt backlog/list could be used to collect all known technical debt issues to a common place. Collecting all the technical debt issues can increase the visibility of technical debt, which could be used to create an overall strategy for technical debt management (Lim et al., 2012).

**Communication** makes identified technical debt visible to stakeholders so that it can be discussed and further managed (Li et al., 2015, p. 205). The challenge with technical debt is the communication gap between technical and non-technical stakeholders (Klinger et al., 2011). Technical debt communication is the key for technical debt management. Bringing technical debt as part of companies’ iteration planning practices can help to narrow down the communication gap between stakeholders (Kruchten et al., 2013).

**Prioritization** ranks identified technical debt according to certain predefined rules to support deciding which technical debt items should be repaid first and which technical debt items can be tolerated until later releases (Li et al., 2015, p. 204). Technical debt prioritization has been suggested through several ideas, such as the finance domain (Seaman et al., 2012), formal approximations (Schmid, 2013), and quality models (Curtis et al., 2012; Letouzey, 2012; Theodoropoulos et al., 2011). However, prioritizing technical debt is a challenge in software development. In some cases, fixing technical debt can be positive from the economic perspective, e.g. increase in customer value, while in other cases it could be significantly important to fix technical debt from the purely technical perspective, e.g. future scalability and maintainability. Sometimes prioritization in the more business-driven development can consider non-technical debts to be more important to fix, rather than real technical issues (Codabux and Williams, 2013).

Overall, the attributes associated with technical debt set up several challenges to its management (Li et al., 2015; Power, 2013). It is difficult to identify and measure technical debt in software (Curtis et al., 2012; Guo et al., 2011). Minor technical debt issues in the source code can be found often with code analysis tools (Izurieta et al., 2012), but major technical debt issues in design and architecture require human effort to understand the underlying issues (Kruchten et al., 2012a), which can also have an impact from external changes (e.g. customer and market) (Zazworka et al., 2013). This raises the question of whether human-identified technical debt should be considered as “real technical debt”, rather than the smaller issues identified with tools (Zazworka et al., 2013). The challenges and difficulties in technical debt identification and measurement have an impact on its other management activities. Whereas working with technical debt in code and design requires technical knowledge from software developers to conduct e.g. refactoring, technical debt management also comprises organizational, political and social tasks, such
2.5 The relationship of technical debt to the field of software engineering

Software engineering as a research area is a multidisciplinary field that includes several technical, economic and social aspects required to create, sell and maintain a software product. There are some aspects in the field of software engineering that need to be examined in detail as to their relationship to technical debt. These aspects are software quality, software product management, software evolution, and software economics.

Software quality is a complex concept, because it can mean different things to different people, and it is highly context-dependent (Kitchenham and Pfleeger, 1996). Petrasch (1999) defines software quality as the existence of characteristics of a product that can be assigned to requirements. Kitchenham and Pfleeger (1996) present Garwin's (1984) description on five different quality perspectives. The transcendental view sees quality as something that can be recognized but not defined. The user view sees quality as fitness for purpose. The manufacturing view sees quality as conformance to specification. The product view sees quality as tied to the inherent characteristics of the product, and the value-based view sees quality as dependent on the sum a customer is willing to pay for it. (Kitchenham and Pfleeger, 1996)

The dimension of software quality and the attributes of technical debt have a relationship with each other. Essentially, technical debt can be seen as a sacrifice in software quality (Zazworka et al., 2011b), since shortcuts and workarounds increase the complexity of the code over time, which has an impact on software quality attributes, especially maintainability and evolvability (Avgeriou et al., 2016). Therefore, technical debt can be seen as the distance between the current solution and the optimal solution (Ernst, 2012; Letouzey and Ilkiewicz, 2012), which is why it can be seen also as the distance in software quality. On the other hand, “optimal solution” as a term is difficult and uncertain, because software evolves constantly (Ernst, 2012). It is difficult to argue what is the optimal “distance” between technical debt and software quality, especially in the constantly evolving software.

Technical debt as a phenomenon brings an interesting view on software quality, because it can variate a lot, depending on the viewer. For example, to a programmer, technical debt can be a quality issue and motivational concern (Tom et al., 2013) in software, because it can make the development more complex (Lim et al., 2012). People working with software development have competence and experience with quality in code and design, which shows in their view and opinion about technical debt (transcendental view, manufacturing view, and product view). On the other hand, for a person working with the
economics and business-related tasks of software, technical debt can be a more suitable and strategic way to ensure economic benefits and customer satisfaction (Lim et al., 2012). The quality in code and architecture in this case is not as great a concern and can be left unheeded (Brown et al., 2010; Ernst et al., 2015), as long as the product is functional and possible to sell to customers (value-based view). For the end-user, software quality is more the actual view on what is in the end-product and whether the initial requirements are fulfilled, which is the reason why the end-customer does not necessarily have an opinion on any solutions that are used in the software code (user view) (Brown et al., 2010).

**Software product management** is the discipline that governs a product from its inception to the customer delivery in order to generate biggest possible value to the business within a given timeframe and cost (Ebert, 2007). Technical debt occurs from the start of a software project (Avgeriou et al., 2016), and therefore, technical debt is also an aspect that has to be taken in consideration in software product management. Even though technical debt can be seen as a quality-related topic, it is not always about code and code quality (Kruchten et al., 2012a). Technical debt is a balance between software quality and “business reality” (Lim et al., 2012). From the software product management viewpoint, it is important to understand and acknowledge technical debt, because it includes the decision of whether to take or not to take, and to pay or not to pay for technical debt (Buschmann, 2011). The software product management has to decide if it is strategically wise to take technical debt and maximize the benefit of faster development and early release in the short term. For software product management, technical debt can be a solution for tight deadlines and market reality, but on the other hand, it is also important to know when there is too much technical debt, and decide how much and what to reduce.

**Software evolution** starts when the initial development of software has been successful (Bennett and Rajlich, 2000). Most software systems evolve during their lifetime, and sometimes the original architecture that was developed based on initial design decisions, might not be applicable in the future (Fitzgerald and Stol, 2015). Therefore, the goal of the software company is to adapt to these changes in the requirements and the environment (Bennett and Rajlich, 2000). Software evolution has an impact on software development, because when the current architecture is not suitable to fulfill the new requirements, changes have to made in the software (Fitzgerald and Stol, 2015). Software evolution has an obvious impact on technical debt accumulation. Even though the design and architecture that was used in the initial version of software did not necessarily contain as much technical debt, software evolution turns the initial solutions to technical debt when they do not fulfill the needs of the current requirements (Brown et al., 2010). When the design or architecture have to be changed, it requires a considerable amount of resources from the software development perspective to rewrite, refactor, and redesign the software, which can be seen as technical debt reduction. However, software evolution does not always force to make changes in the software, if the value of the current software outscales the required resources needed for change (Buschmann, 2011).
Software economics is a field that aims at improving software design and engineering through economic reasoning about product, process, program, and portfolio and policy issues (Boehm and Sullivan, 2000). Software development can be both engineering activity and business-driven investment activity (Avgeriou et al., 2016). The separation between these two perspectives exists often in how the company management sees the value of software and how software developers accelerate or hesitate those activities. Financial models for software economics and value do not easily translate to software development and software project management (Avgeriou et al., 2016). Models and decision-making regarding software can be heavily business-driven, and do not necessarily include technical debt and its impacts. Even though it is a challenge to estimate and measure technical debt from the software economics viewpoint, understanding the impact of debt to software economics and how the software should be built is important. Technical debt can be used as an investment activity to improve the time-to-market, but also as an investment activity to repay highly prioritized technical debt strategically from internal system qualities, such as maintainability and evolvability (Avgeriou et al., 2016).

2.6 Summary

This chapter introduced (i) definitions of technical debt, (ii) a short history of technical debt discussed with other terminology, (iii) the main attributes often associated with technical debt, (iv) the concept of technical debt management, and (v) the relationship of technical debt with the field of software engineering.

Technical debt as a metaphor has increased its popularity especially in recent years, and it can be considered as a recognized and impactful term in both the academia and the industry (Avgeriou et al., 2016; Li et al., 2015). Technical debt as a phenomenon is an essential part of software development, which should be acknowledged by software companies. There are various types of technical debt with different causes and effects (Fowler, 2009; McConnell, 2007). Taking technical debt can sometimes be considered a good strategic decision (Lim et al., 2012), but technical debt in code is often also a quality issue (Zazworka et al., 2011b). Therefore, it is important for software companies to understand the role of technical debt in software development in order to manage, prevent and reduce the negative consequences of technical debt.

The current state-of-the-art of technical debt and its management has some research gaps (Li et al., 2015). In the beginning, technical debt was referred to as a technical metaphor (Cunningham, 1992; Fowler, 2003), but since then it has been expanded to other parts of software development, which has resulted in a number of other debts (Alves et al., 2016; Tom et al., 2013). Therefore, there exists a lack of common understanding on what is meant when talking about technical debt. More cases of technical debt need to be studied to gain a more complete understanding on its role in software and software development. In addition, technical debt management requires more research, especially on processes
and tools. There is no overall view on technical debt and its management available, which is the reason why there is urgent need for more research (Li et al., 2015).
3 Research problem and methodology

This chapter outlines the research problem and defines the appropriate research questions. In addition, the chapter contains a description of the research methodologies and the research process used during this study.

3.1 Viewpoints and research problems

Technical debt is a recognized and accepted metaphor in software engineering for describing complexities in software (Avgeriou et al., 2016). The research and interest in technical debt have increased in recent years, and the metaphor has become a known concept in both academia and industry (Alves et al., 2016; Avgeriou et al., 2016; Li et al., 2015). Technical debt as a phenomenon can be considered as an essential part of software development, and if its role is not acknowledged by software companies, it can cause irreversible effects on future development (Seaman and Guo, 2011). Therefore, it is important to conduct more research on this topic to create an understanding on the role of technical debt in software and software development. The research on technical debt and its management can help software companies to understand (i) the reasons for technical debt in software, (ii) the effects of technical debt on software and software development, and (iii) to deal with technical debt, providing knowledge to prevent and manage it in order to forestall the issues turning up if technical debt accumulates.

The current state-of-the-art on technical debt shows that there exist various perspectives on technical debt as a metaphor, which has generated a great number of different definitions and subcategories to it (Alves et al., 2016; Li et al., 2015; Tom et al., 2013). Therefore, it is essential to define and explain properly, what technical debt each research have their focus on, to prevent any confusion with the overall metaphor. In this thesis, the following definition is used as the viewpoint to technical debt:

“A badly structured technical solution or architectural design in the system, incurred by either an intentional decision or an unintentional side effect, which causes omitted quality and productivity”

This definition is the main guideline used in this thesis as the viewpoint to technical debt. This study focuses on technical debt as a non-optimal technical decision in a code, design or architecture that is associated with software implementation. This study does not consider e.g. bugs and defects as technical debt, and therefore not everything that is ‘incomplete’ in software. However, the study includes both intentional technical debt and unintentional technical debt (McConnell, 2007) as part of the viewpoint. We believe that eventually both intentional and unintentional technical debt can be considered to have a similar end result in software, which has to be dealt with similar reduction strategies, even though the reasons and causes for them are different.

The two main reasons for narrowing down the definition of technical debt that happens only in software implementation with the code, design, or architecture, and not in other
3.1 Viewpoints and research problems

parts related to software development (e.g. requirements or testing) are the following: The research area would be too wide with all the identified technical debt subcategories included in the definitions of technical debt (Alves et al., 2016; Li et al., 2015; Tom et al., 2013). The goal of the thesis is to focus on technical debt in software code, design and architecture, as the original definitions implied (Cunningham, 1992; Fowler, 2003), and not on everything where there is a possibility for a non-optimal solution followed by some cost (Kruchten et al., 2012a). Even though we understand that it is important to have research on the subcategories related to technical debt (e.g. how shortcuts and workarounds done in requirements or testing phases affect software and software development), the viewpoint in this thesis is that most of them should not be categorized as technical debt. We believe that the subcategories of technical debt outside implementation (code, design, architecture) are more related to the reasons and effects for and of technical debt (e.g. social debt (Tamburri et al., 2013, 2015) can be something that causes technical debt to appear in software, while defect debt (Alves et al., 2014, 2016; Li et al., 2015; Snipes et al., 2012) is more the effect of having technical debt). Therefore, we do not consider shortcuts and workarounds in activities outside software implementation as technical debt, which is the reason they are left out of this viewpoint.

There exists some research problems and research gaps in the current technical debt state-of-the-art. These problems and challenges needs to be addressed and studied to get better understanding on the role of technical debt in software development. The current state-of-the-art in technical debt and its management has the following research problems and gaps that require more research:

**What is technical debt?** The current status of technical debt as a metaphor can be considered as incoherent (Avgeriou et al., 2016; Kruchten et al., 2013, 2012a). There exists a lack of consensus on the question “What is technical debt?” (Kruchten et al., 2013; Li et al., 2015). The metaphor started as a description of non-optimal code and design in software (Cunningham, 1992; Fowler, 2003), but it has been expanded and modified drastically since its origin (Kruchten et al., 2013; Li et al., 2015; Tom et al., 2013). At the moment there exist various definitions, which take into consideration multiple phases and perspectives in software development where a shortcut or workaround exists (Alves et al., 2014, 2016; Li et al., 2015; Tom et al., 2013). Therefore, one of the main research challenges and gaps in current technical debt state-of-the-art is studying the metaphor. It would be important to find a consensus between research communities and software industry to improve the overall landscape with technical debt. This could help to create better understanding on what should be considered as technical debt, rather than just being everything incomplete in software.

**The causes and effects of technical debt.** The current knowledge of technical debt in the academic literature is often based on theories and opinions and does not usually provide any real case examples of software development (Alves et al., 2016; Li et al., 2015). There is a need for case examples of real-life software environments to see the causes and effects of technical debt (Alves et al., 2016; Li et al., 2015). In addition, the examples of technical debt with empirical evidence found in the literature are often
focused purely on the qualities of technical debt in software. However, technical debt can be also affected by external factors of software development, such as organizational and social aspects, which are not related to the attributes of software (Brown et al., 2010; Klinger et al., 2011; Kruchten et al., 2012a). Conducting more research with technical debt and observing examples from empirical data is an essential part for creating new knowledge and understanding the role of technical debt. Understanding where technical debt comes from, what causes it, and what are its effects on software and software development is an important part of creating activities and processes for its management.

Technical debt management framework. In the current state-of-the-art, there exist some ideas and theories for technical debt management (Alves et al., 2016; Li et al., 2015). However, there is lack of studies that consider the entire landscape of technical debt management at the organizational level. Current studies focus often on a specific part of the technical debt management process, e.g. identification or prioritization of technical debt (Li et al., 2015). There is a need to create an overall understanding on what technical debt management is at both organizational and technical level, what activities are included, what processes and tools can be used in each activity, and who are responsible for those activities. Technical debt management as a concept is also relatively new and not yet necessarily fully understood and taken in consideration by software companies. Therefore, there is also a need for studies on adopting technical debt management activities to understand how they work in real software development environments.

Lack of empirical research on technical debt and its management. A major issue and research gap in the current research on technical debt and its management is the lack of empirical evidence. Even though there exist a number research papers related to technical debt and its management, the studies are not often validated by empirical data (Alves et al., 2016; Li et al., 2015). Therefore, there is a significant research gap on empirical evidence associated with technical debt, which requires more research and case studies conducted with real software companies.

The main focus in this thesis is to address the identified research problems and research gaps. The objective is to collect empirical evidence from software development companies to study technical debt and its management. The objective and focus in this thesis has been narrowed down to studying technical debt and its management from organizational, political and social perspectives of software development with qualitative data, rather than the quantifiable qualities of technical debt found in the code and design. The goal is to study how organizational and human aspects of software development cause technical debt to software, and what are the effects of having and taking technical debt in software and software development. In addition, the goal is to understand technical debt management activities, processes, models, frameworks, and tools that can be used to manage technical debt.

The thesis is divided into two main research objectives that address the found research problems and gaps. The first objective is to focus on the causes and effects of technical
The research questions

3.2 The research questions

debt by gathering empirical case examples of what has happened with technical debt in software companies. Gathering empirical evidence from real case examples can create new knowledge on why technical debt occurs in software. In addition, the examples can show the effects technical debt has on both software and software development. Creating theories of technical debt and understanding the causes and effects through case examples can help software companies to get a better perspective on their own technical debt to understand its role during software development. In addition, the knowledge on the causes and effects of technical debt can also help to create better management practices and to understand what processes and tools are needed for each technical debt case.

The second objective focuses on technical debt management. The goal is to gather empirical evidence from software companies to understand how technical debt is currently managed. Based on the results and other related literature, the objective is to create a theoretical framework that describes the role of technical debt management in software development. The developed framework can be applied and used in practice for improving companies’ internal and external processes, which will help them to manage technical debt better.

The case examples and empirical data have been observed and collected through a research project. This research is a part of a large Finnish software engineering project called Need 4 Speed (N4S). The N4S project (N4S-Program, 2016) is executed by forefront Finnish software companies. N4S consists of large industrial organisations, small and medium enterprises (SMEs), and research institutes and universities. The N4S project has had a significant impact on the data collection for this research. The project has provided an opportunity to access several software companies to acquire empirical evidence.

3.2 The research questions

The following research questions address the viewpoints, research problems and research gaps in this thesis. The main research question: What is the role of technical debt in software development?

- Research question 1 (RQ1): What are the reasons and causes for technical debt in software and software development?
- Research question 2 (RQ2): What are the effects of technical debt on software and software development?
- Research question 3 (RQ3): What are the current management strategies and practices used for managing technical debt in software development?
- Research question 4 (RQ4): How to adopt technical debt management in a software development company?
The main research question of this thesis is divided into two research topics. The first research topic focuses on the first two research questions. The goal of RQ1 is to understand the reasons and causes for technical debt to occur in software. This research question is the first essential part of understanding the role of technical debt in software development. It is important to understand the reasons and causes for both intentional and unintentional technical debt. By understanding the reasons and causes for technical debt, it can be possible for software companies to find ways to prevent and reduce technical debt in advance. The goal of RQ2 is to understand the effects on software development of taking and having technical debt in the software. This research question is another essential part of understanding the role of technical debt in software development. By understanding the effects of technical debt on software development, it can be possible for software companies to understand whether taking technical debt can sometimes be a positive thing, and what are the consequences the company has to face later when having technical debt. The goal is to collect data from the software development environment by observing and studying technical debt case examples, which can provide evidence to RQ1 and RQ2. The evidence acquired from the first research area can be used to understand the role of technical debt in software development and interpret what constitutes technical debt, based on the studied case examples.

The second research topic focuses on technical debt management. The goal of RQ3 is to understand how technical debt is currently managed in software companies. The empirical evidence gathered from case companies is a first step for developing a technical debt management framework. The developed framework can be used by software companies to improve the internal and external processes to make technical debt more manageable and visible to both developers and managers. The goal of RQ4 is to observe and test whether technical debt management can be adopted in a software company. The objective is to use the previously developed framework with a case company, to see if technical debt management can be improved with new activities and processes introduced in software development processes.

The results presented in this thesis have been reported in five separate publications. The first three publications (Publication I, II, and III) focus on RQ1 and RQ2, where the research topic is the causes and effects of technical debt. The last two publications (Publication IV and V) focus on research questions RQ3 and RQ4, where the research topic is technical debt management. The research questions and reporting are summarized in Table 3.
3.3 Research methodologies

<table>
<thead>
<tr>
<th>Publication</th>
<th>Objective</th>
<th>RQ1</th>
<th>RQ2</th>
<th>RQ3</th>
<th>RQ4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>To study causes and effects for technical debt in software and software development.</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>To understand the benefits and drawbacks of having and taking technical debt in software development.</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>III</td>
<td>To understand the relationship between organizational and human aspects of software development and technical debt.</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>To identify the management strategies and activities used in technical debt management.</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>V</td>
<td>To adopt activities and processes in technical debt management.</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

3.3 Research methodologies

There are multiple ways to conduct research, depending on the research context. Järvinen (2004) has created a taxonomy of research methodologies by the top-down principle, which describes the possible research methodologies available. All research methodologies can be first divided into two main classes, mathematical approaches and approaches studying reality. The research methodologies that concern reality can be divided into two classes, depending on the research question. The research questions can be shaped to ask about a part of reality, or the utility of innovation. Studies studying reality can have both conceptual analytical approaches and empirical approaches. Empirical research approaches can involve theory-creating and theory-testing approaches. When the research studies the past and present empirically, it can be done with theory-testing or theory-creating methods, depending on whether there already exists a theory, model or framework to guide the research or whether the goal is to develop a new theory based on the gathered raw data. The theory-testing approaches may involve laboratory experiments, surveys, field studies, and field tests, and the theory-creating approaches may involve case studies, ethnographies, and grounded theories.

When examining the taxonomy of the research method approaches presented by Järvinen (2004) and the research questions and objectives of this thesis, the research methodology in this thesis can be categorized as a research investigating reality by using empirical research methodologies to create and develop new theories, models and frameworks.
3.4 Philosophical perspectives on qualitative research

This section gives a brief description of views of philosophical perspectives that can be used in a study. Orlikowski and Baroudi (1991) divide philosophical perspectives to positivist, critical, and interpretive research.

**Positivist** perspective reflects the precepts informing the study of a natural phenomenon (Orlikowski and Baroudi, 1991). The phenomenon in positivist research is single, tangible and fragmentable. Research fields such as physics, chemistry, and biology are counted as true sciences in the positivist perspective. For example, in information systems, positivist research assumes that there exists an objective physical and social world independent of humans, whose nature can be apprehended, characterized, and measured without problems. The positivist perspective assumes that human action is intentional and rational, where human interaction is relatively stable and inconsistencies are not endemic to the organization (ibid.). Software engineering as a field cannot be easily considered as natural science. According to Sjoberg et al. (2007), a positivist approach would be a challenge in a field that is highly dependent on people and the environment, like software engineering is.

**Critical** perspective attempts to evaluate and transform the social reality under investigation critically (Orlikowski and Baroudi, 1991). The critical perspective is concerned with criticising existing social systems and finding conflicts within their structures. The critical research perspective incorporates the idea that the social reality is historically constructed and societies are not limited to the current state only (ibid.). Critical researchers often decide what research to conduct on the basis of whether it helps a certain society (Easterbrook et al., 2008). In software engineering, a research done with a critical perspective seeks to challenge existing perceptions about software practice actively. Most notable critical perspectives in software engineering field are the open source movement, the process improvement community and the agile community (ibid.).

**Interpretive** perspective beliefs that the reality is a social product and therefore impossible to understand without the related social actors (Orlikowski and Baroudi, 1991). The aim in interpretive research is to understand how the members in social groups and social processes set their beliefs, intentions and meanings to constitute their social actions. The interpretive perspective does not believe that organizations and social structures are objectively known and unproblematic, but attempts to understand how social world is produced by humans through their actions and interactions (ibid.). In software engineering, for example, one can study the culture of a software design team to understand how different members of the team think and what tools they have available, to gather empirical evidence for theory creation in that specific context (Easterbrook et al., 2008).

As the objective of this thesis is to study technical debt and its management in the software development environment from the organizational perspective rather than the measurable aspects of software, it includes a social aspect with humans involved in the
3.5 Selection of research methodologies

This section presents the selected research methodologies and how each method has been applied during the research process. Software engineering as a research area is a multidisciplinary field, including both social and technological perspectives. Software engineering research does not only concern the tools and processes used for development, but also the social processes, including the human activities and decisions surrounding them (Easterbrook et al., 2008). Software development is carried out by individuals, groups, and organizations, and therefore social and political questions are important during software development (Runeson and Höst, 2008). Therefore, the research methods studying human behaviour at both individual and organizational level can be used. This research was categorized above as having an interpretive philosophical view on investigating reality by using empirical research methodologies to create and develop new theories, models, and frameworks (Järvinen, 2004; Orlikowski and Baroudi, 1991). However, there still exist various possible stances to conduct the research, and it is important to select the right methodologies to achieve the best possible research process and results.

As stated above, technical debt as a phenomenon in software engineering can be studied with both qualitative and quantitative methods, because there are both organizational and technical perspectives included. For example, technical debt research with quantitative approaches can be done to study various technical qualities in the source code, to provide quantifiable numbers related to e.g., software code and design. Quantitative research could help, e.g., software development teams to measure technical debt issues in the software source code. Technical debt can also be studied qualitatively e.g., with the social and organizational aspects of software development. The objective of this thesis was to focus on the reasons and causes of technical debt through case examples from the organizational viewpoint, while trying to identify and develop management strategies. Therefore, this research can be categorized as a qualitative research. The selection of research methodologies was also significantly influenced by the N4S research program (N4S-
Program, 2016). Participation in the research program provided access to companies and their employees to collect data and make in-depth studies related to the research topic.

This thesis has been conducted with the following research methodologies. In Publications I, II, III, and IV, the selected research methodology was case study methodology (Yin, 2003), where the data collection was performed with semi-structured interviews (Robson, 2002). In Publication V, the research methodology was action research (Avison et al., 1999), where the data collection was performed with semi-structured interviews (Robson, 2002) and workshops (Stanfield, 2002). The reasons for choosing case study as the main research methodology were the following:

(i) The case study methodology is suitable for studying and collecting qualitative data in case organizations (Runeson and Höst, 2008). Studying technical debt from the organizational viewpoint, which involves social and political aspects of software development, with people involved in the development and decision-making, is optimal to conduct with case studies.

(ii) Participation in the N4S research project (N4S-Program, 2016) provided an opportunity to access and study multiple software companies and organize interviews with people related to software development. Therefore, the case study methodology was the optimal method to acquire qualitative data and understand how the technical debt phenomenon happens at the entire organizational level.

(iii) Interviews with people working at software companies can provide more in-depth information related to technical debt, which might not be possible to find in quantitative data (e.g. survey or repository) as easily.

Another popular research methodology for qualitative research is the grounded theory (Glaser and Strauss, 1967). Both case studies and grounded theory are popular qualitative research methodologies similar to each other that can be used in empirical research in software engineering. The grounded theory and case studies are both based on data that can be collected via multiple channels (interviews, observations, documents etc.). Both methods also follow a systematic procedure for data analysis, where the researcher seeks for codes and categories (Cho and Lee, 2014). However, there are some main differences between the methodologies:

(i) The philosophical view of the grounded theory lies in social interactionism and a reaction to a positivistic view of science, but case studies are a reaction to quantitative content analysis to understand the meaning of the context (Cho and Lee, 2014).

(ii) The grounded theory focuses more on creating a theory, while case study aims at a list of categories and themes. Therefore, the characteristics and data analysis procedures are different (Cho and Lee, 2014).

(iii) Case studies focus on a set-up of a priori categories that cover the data, while the grounded theory yields more data reduction, abstraction, and core
3.5 Selection of research methodologies

3.5.1 Case study

Case study is a research methodology that focuses on understanding the dynamics present within a single setting (Eisenhardt, 1989). Yin (2003, p.13) describes a case study as ‘an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident’. As a research strategy, case study is often used to contribute to our knowledge of individual, group, organizational, social, political, and related phenomena. Therefore,
case study is a commonly used research methodology especially in social sciences (ibid.). In addition, it has become more popular in software engineering as well, because software development involves individuals, groups and organizations. This makes software engineering a research area where case study is a relevant and usable approach (Runeson and Höst, 2008).

There can be both single case studies and multiple case studies to address research questions (Yin, 2003). Single case studies can be used to study e.g. the decision-making of a single person or a single group. Multiple case study can be used with several cases to understand e.g. the similarities and differences between the cases (Baxter and Jack, 2008). Case studies can be also looked at with a holistic or embedded viewpoint, depending on the research context (Yin, 2003). The holistic view refers to a single unit of analysis, whereas the embedded view concerns multiple units of analysis. A holistic case study takes e.g. a view of an organization as a single unit, while an embedded case study considers that there are multiple different units to study separately, such as management, development, maintenance, finance etc., within one organization (ibid.).

Case studies combine typically several methods for data collection (Eisenhardt, 1989). The data can be derived from e.g. archives, interviews, questionnaires, and observations. The evidence in case studies can be both qualitative (words from interviews) and quantitative (numbers from questionnaires) (ibid.). In this thesis, interviews and focus groups (workshop) were opted to use for data collection.

**Interviews** are the most commonly used method for collecting qualitative data (Cassell and Symon, 2004), and an important method especially in case studies (Runeson and Höst, 2008). Interviews are a process to see the research topic from the perspective of the interviewee to understand how and why a certain perspective exists (Cassell and Symon, 2004). Kvale (1983) defines the qualitative research interview as a purpose to gather descriptions of the real life world of the interviewee with respect to interpretation of the meaning of the described phenomenon. Interviews are often used to collect historical data, opinions and impressions, and to identify terminology in a particular settings (Seaman, 1999).

The types of interviews can be divided into unstructured, semi-structured and fully structured interviews (Robson, 2002). **Unstructured interviews** are generated from the general concerns and interests of the researcher (Runeson and Höst, 2008). In unstructured interviews, the questions are not prearranged and the discussion is non-directed. **Fully structured interviews** are planned in advance, and all questions to the interviewee are predesigned and always asked in the same order. Fully structured interviews have a lot of similarities to normal questionnaire-based surveys (ibid.). **Semi-structured interviews** can include both open-ended questions and more detailed questions. The objective of semi-structured interviews is to collect both foreseen and unexpected information (Seaman, 1999). Semi-structured interviews are commonly used in case studies (Runeson and Höst, 2008). In addition, interviews can be organized both locally (face-to-face interviews) or distantly (electronic interviews). Face-to-face interviews are
3.5 Selection of research methodologies

conducted in the same place with the interviewee, while the electronic interview is a method that uses electronic communication tools to access and communicate with the interviewees (Cassell and Symon, 2004).

**Focus group** can be defined as a group interview (Morgan, 1997). Focus group is a qualitative research method that is designed to gather descriptive data from population subgroups (Basch, 1987; Bender and Ewbank, 1994). Focus groups can be used to collect opinions on topics to develop an understanding of respondents' perspectives (Bender and Ewbank, 1994). Focus groups can also be called ‘workshops’. Workshops are discussion, brainstorming, or organizing sessions done in a group (Stanfield, 2002). Focus groups can also be used with the Delphi method, which is used to gather and refine ideas from a group of experts (Dalkey, 1969).

A focus group session often includes a small number of participants who are led by a moderator asking questions about the research topic (Berg et al., 2004). Focus groups should usually involve around 6-12 participants (Lengua and Others, 1992). It is also possible to use a variety of other methods in focus groups, such as brainstorming, scenario-based discussion and cognitive maps. Focus groups are not a common methodology in software engineering, but they can still provide valuable empirical experience quickly with low cost (Kontio et al., 2004).

In this research, the case study methodology was used in *Publications I-IV*. All the data collection in *Publications I-IV* was done with semi-structured interviews. *Publications I-IV* were done with multiple case studies with a holistic viewpoint (Yin, 2003). We decided to use the multiple case study methodology because we believed technical debt as a phenomenon can have various causes and effects. We believed that it was important to study and collect multiple examples of the phenomenon from a large number of software development teams to observe it in different environments and products. We decided to adopt the holistic viewpoint, because we believed that technical debt is caused by reasons appearing in various parts of software development organizations. In addition, the effects could be possibly seen not only in programming, but also other parts of the software organization. The decision to use semi-structured interviews was to have more open discussion with the interviewees, rather than having strictly structured questions that would be asked in every interview. We believed that with semi-structured interviews, we would be able to gather more insightful data in the interviews.

### 3.5.2 Action research

Action research is a research methodology which endorses the collaborative approach to provide people with the means of taking systematic actions to resolve certain problems (Berg et al., 2004). The action research process often combines researchers and practitioners to act together in a set of activities for problem solving (Avison et al., 1999). Usually the goal of the action research process is to provide and create value simultaneously to client organizations with new solutions and to the literature with theoretical knowledge (Sjoberg et al., 2007). Therefore, using action research
methodology requires a situation where participation and organizational change are necessary (Baskerville and Wood-Harper, 1996). Action research is not often associated with the software engineering field, but the interest in it has grown (Santos and Travassos, 2009).

The action research methodology was used in Publication V. The goal of the last phase of the research process was to develop and test whether technical debt management practices would be applicable in a real software development environment. The Need 4 Speed research program (N4S-Program, 2016) provided a case company that had a specific need to improve processes related to technical debt management. Therefore, it was possible to create a research environment where both the research group and the case company worked closely together, which allowed the use of action research methodology. In Publication V we also decided to use focus groups as the data collection method. The idea was to gather a few professionals from a software development company to discuss technical debt issues. This way it would be possible to have a discussion session with the participants, which could reveal different information compared to single interviews.

3.6 Research process

The research process was divided into a preliminary phase and three main phases (Figure 3). The preliminary phase started when the N4S research program (N4S-Program, 2016) was first initiated in January 2014. A brief literature review was performed to gain knowledge on the current state-of-the-art on technical debt and its management. The studies on technical debt showed that the research area was gaining an increasing interest in both academia and industry, which had generated a number of papers discussing the research topic (Tom et al., 2013). The knowledge acquired in the preliminary phase had a significant influence on the viewpoints, research problems, and research questions of this thesis, which were used as motivation and direction in the other main research phases.

Phase 1 was started in March 2014 when the preliminary phase had ended. Phase 1 included result reporting in three publications (Publication I, Publication II, and Publication III). The focus in Phase 1 was to understand the role of technical debt in software development through the identification of causes, effects, benefits, and drawbacks in the studied case companies. Phase 1 included five organizations, six development teams, and 29 interviews. The results of Phase 1 provided more insight into the role of technical debt in software development by providing empirical data from case examples that were identified in the studied companies and development teams. The results of Publication I, Publication II, and Publication III were used to continue the research further to Phase 2, where the main research objective was related to technical debt management.
Phase 2 was started in January 2015, after the data collection and analysis in Phase 1 was finalized. The focus in Phase 2 was purely directed on understanding the role of technical debt management in software development. The goal was to identify what activities were included in technical debt management, and how software development teams were managing technical debt. During Phase 2, a systematic mapping study on technical debt management was published by Li et al. (2015). The results of the mapping study were the
major inspiration in Publication IV. The data collection in Phase 2 included one organization, eight development teams, and 25 interviews (12 interviews from Phase 1). The results observed in Phase 2 led to the development of a framework for technical debt management. The developed framework can be considered as one of the main result of this thesis.

After the data collection for the technical debt management framework was finalized, Phase 3 was started in May 2015. Publication V was the outcome of Phase 3. The focus in this phase was to use the developed technical debt framework to improve a selected case company’s software processes related to technical debt management. Phase 3 included one organization, one development team, 7 interviews, and one workshop (with 12 persons). The results formed an example of maturity increase in technical debt management in the case company.

3.6.1 Data collection

The data collection was performed with semi-structured interviews (Seaman, 1999) and focus group techniques (Morgan, 1997). The participating companies included medium-to-large-size enterprises (European Commission, 2014) in computer software and telecommunications industry. Six organizations and 13 development teams were interviewed and included in the collected data. The sizes of the development teams can be considered small, varying from 6 to 25 people. Most of the data collection was performed in organization A with eight development teams, with one or two teams from the other organizations. A summary of the organizations in this research is given in Table 4. The interviewed people were working in technical and business-related positions in the organizations, as the goal of the research was to study technical debt from the organizational viewpoint. More detailed information about the interviewees with their roles, organizations, and teams is presented in Table 5.

During the data collection, a snowballing technique (Charmaz, 2014) was used to acquire new interviewees from the case organizations. When an interview was completed, we asked the interviewee for a referral to another person in the organization, who might be able to have knowledge related to the research topic. Using this technique, it was possible to arrange new interviews in the organizations and development teams.
### Table 4. Participating organizations in this research

<table>
<thead>
<tr>
<th>Organization #</th>
<th>Industry sector</th>
<th>Size (employees)</th>
<th>Number of people participating</th>
<th>Number of teams participating</th>
<th>Data used in publication #</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Computer software</td>
<td>6 700</td>
<td>25</td>
<td>8</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>B</td>
<td>Telecommunication</td>
<td>4 100</td>
<td>3</td>
<td>1</td>
<td>II and III</td>
</tr>
<tr>
<td>C</td>
<td>Computer software</td>
<td>340</td>
<td>3</td>
<td>1</td>
<td>II and III</td>
</tr>
<tr>
<td>D</td>
<td>Telecommunication, computer software</td>
<td>61 600</td>
<td>4</td>
<td>2</td>
<td>III</td>
</tr>
<tr>
<td>E</td>
<td>Telecommunication</td>
<td>118 000</td>
<td>7</td>
<td>1</td>
<td>III</td>
</tr>
<tr>
<td>F</td>
<td>IT, business consulting and outsourcing services</td>
<td>13 700</td>
<td>11</td>
<td>1</td>
<td>V</td>
</tr>
</tbody>
</table>

### Table 5. The roles of interviewees and workshop participants in this research

<table>
<thead>
<tr>
<th>#</th>
<th>Participation</th>
<th>Organizational</th>
<th>Team</th>
<th>Role</th>
<th>Data included in Publication #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interview</td>
<td>A</td>
<td>a</td>
<td>Software architect</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>2</td>
<td>Interview</td>
<td>A</td>
<td>a</td>
<td>Software designer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>3</td>
<td>Interview</td>
<td>A</td>
<td>a</td>
<td>Project manager</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>4</td>
<td>Interview</td>
<td>A</td>
<td>a</td>
<td>Software test engineer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>5</td>
<td>Interview</td>
<td>A</td>
<td>a</td>
<td>Product line manager</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>6</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Software architect</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>7</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Software developer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>8</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Product line manager</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>9</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Software test engineer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>10</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Software architect</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>11</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>Software developer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>12</td>
<td>Interview</td>
<td>A</td>
<td>b</td>
<td>User interface designer</td>
<td>I, II, and IV</td>
</tr>
<tr>
<td>13</td>
<td>Interview</td>
<td>A</td>
<td>c</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>14</td>
<td>Interview</td>
<td>A</td>
<td>c</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>15</td>
<td>Interview</td>
<td>A</td>
<td>c</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>16</td>
<td>Interview</td>
<td>A</td>
<td>d</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>17</td>
<td>Interview</td>
<td>A</td>
<td>e</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>18</td>
<td>Interview</td>
<td>A</td>
<td>e</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>19</td>
<td>Interview</td>
<td>A</td>
<td>e</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>20</td>
<td>Interview</td>
<td>A</td>
<td>f</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>21</td>
<td>Interview</td>
<td>A</td>
<td>f</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>22</td>
<td>Interview</td>
<td>A</td>
<td>g</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>23</td>
<td>Interview</td>
<td>A</td>
<td>g</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>24</td>
<td>Interview</td>
<td>A</td>
<td>h</td>
<td>Team manager</td>
<td>IV</td>
</tr>
<tr>
<td>25</td>
<td>Interview</td>
<td>A</td>
<td>h</td>
<td>Software architect</td>
<td>IV</td>
</tr>
<tr>
<td>26</td>
<td>Interview</td>
<td>B</td>
<td>i</td>
<td>Software architect</td>
<td>II and III</td>
</tr>
<tr>
<td>27</td>
<td>Interview</td>
<td>B</td>
<td>i</td>
<td>Project owner</td>
<td>II and III</td>
</tr>
<tr>
<td>28</td>
<td>Interview</td>
<td>B</td>
<td>i</td>
<td>Project owner</td>
<td>II and III</td>
</tr>
<tr>
<td>29</td>
<td>Interview</td>
<td>C</td>
<td>j</td>
<td>Senior consultant</td>
<td>II and III</td>
</tr>
<tr>
<td>30</td>
<td>Interview</td>
<td>C</td>
<td>j</td>
<td>Software architect</td>
<td>II and III</td>
</tr>
<tr>
<td>31</td>
<td>Interview</td>
<td>C</td>
<td>k</td>
<td>Technical project manager</td>
<td>III</td>
</tr>
<tr>
<td>32</td>
<td>Interview</td>
<td>D</td>
<td>k</td>
<td>Software architect</td>
<td>III</td>
</tr>
<tr>
<td>33</td>
<td>Interview</td>
<td>D</td>
<td>k</td>
<td>Software architect</td>
<td>III</td>
</tr>
<tr>
<td>34</td>
<td>Interview</td>
<td>D</td>
<td>k</td>
<td>Manager of R&amp;D department</td>
<td>III</td>
</tr>
<tr>
<td>35</td>
<td>Interview</td>
<td>D</td>
<td>k</td>
<td>Project manager</td>
<td>III</td>
</tr>
</tbody>
</table>
Before the interviews, the present author prepared semi-structured question sheets, which can be found as appendixes A and B in *Publication IV*. The present author designed and wrote the question sheets based on a priori categories identified in state-of-the-art literature, and modified them based on data acquired from the interviews. The thesis supervisors approved the question sheets and improved them in a situation where the goals were not clear. The first interview sheet (*Publication IV / Appendix A*) was prepared with detailed questions related to the causes and effects of technical debt, which answered **RQ1** and **RQ2**. The first interview sheet was used in *Publications I, II, III and partly in Publication IV*. The second interview sheet (*Publication IV / Appendix B*) was prepared with detailed questions related to technical debt management, which answered **RQ3** and **RQ4**. The second interview sheet was used in *Publication IV* and partly in *Publication V*. Even though the question sheets had some detailed questions, they were only used as a guideline during the interviews. If the researcher noticed something interesting during the interview, it was possible to direct the discussion more to the new interesting direction. Therefore, with the semi-structured interviews it was possible to gather more insightful data, which would have not been necessarily possible with more close-structured questionnaires (e.g. a survey).

Most of the interviews were performed locally in the case organizations' facilities. There were also interviews that had to be organized electronically with Skype or some other online discussion tool suggested by the company. The workshop in this study was also organized with an online tool. The reason for conducting some interviews and the workshop electronically was related to location or schedule problems. Most of the studied development teams were located in Finland, which made it possible to travel and organize
meetings face-to-face with the interviewees. However, there were also development teams or single persons who were not currently located in Finland, which required us to use an online tool for the interviews. All discussions and interviews were recorded by using a recording device or an online Skype recording tool. All the recording activities were always mentioned to the interviewee before the interview and a permission was asked. During the data collection, there was only one interview where the permission was not granted due to company regulations. The possibility to record the interviews helped the research group in data analysis, because it was possible to go back and re-listen to each interview if needed. The interviews lasted usually from 30 minutes to one hour. All the recorded interviews were transcribed after the interview session for further analysis by the present author or a hired professional.

3.6.2 Data analysis

The data analysis was performed with similar techniques as used in the grounded theory (Glaser and Strauss, 1967). The data analysis included categorizing, labelling and data coding. Considering the number of interviews and amount of qualitative data acquired during this research, it was not possible to complete the data analysis manually, because of the amount of categorizing and labelling. To handle a large amount of qualitative data, we used a tool specifically designed for qualitative data analysis, Atlas.ti (Atlas.ti, 2016). With Atlas.ti, it was possible to have coding that created several categories and labels while studying the interviews and to organize them more efficiently for further analysis.

Figure 4 shows an example of a data coding activity that was often performed during the analysis with Atlas.ti. While reading the transcribed interviews, we created codes, categories and labels based on what the interviewee had said during the interview. For example, in a situation where the interviewee was talking about the measurement activities for internal quality attributes, we used selective coding to assign that specific sentence to the a priori category for technical debt measurement. After the creation of codes, categories and labels, it was easier to observe and identify similar codes in various categories.

After the categories and sub-categories were identified and labelled, Atlas.ti provided a tool to create a map for the categories, where the causalities and connections of codes and categories could be linked to create a more complete view on the phenomenon. Figure 5 shows an example of a codes and categories map that was created and used in Publication V to identify the benefits of technical debt in evaluation activities related to technical debt management.
3.6.3 Finishing and reporting on the research

The finishing and reporting in this research was done by writing scientific publications, included as an appendix in this thesis. These publications are referred to as Publications I – V. The goal of Phase 1 (Publications I, II, and III) was to establish understanding of the causes and effects of technical debt through studying examples of a real-life software
development environment. Publication I presents findings from two software development teams, describing several identified causes and effects of technical debt. Publication I was published in a peer-reviewed international conference. Publication II reports on detailed examples of benefits and drawbacks of having technical debt through examples that had happened during software development in two case organizations. Publication II was also published in a peer-reviewed international conference. Publication III took a more organizational view on the causes and effects of technical debt by studying three large organizations, with the goal to understand how software process evolution and improvement affect technical debt. Publication III was published as a book chapter in a peer-reviewed scientific software process evolution-related book.

The study continued to Phase 2, where the goal was to study technical debt management. Publication IV examined the technical debt management activities used in eight software development teams. The study resulted in a technical debt management framework, which can be used by software development teams. Publication IV was published in a peer-reviewed international journal. Phase 3 focused on testing and adopting the developed framework in Phase 2. Publication V used the action research method in a selected case company to adopt technical debt management activities. The study resulted in a set of templates and processes that can be used for technical debt management. The results of Publication V were published in a peer-reviewed international conference.

3.7 Summary

This chapter explained the viewpoints, the research problem, the research methods, and the research process used in this thesis. The research process was divided into a preliminary phase and three main phases, which are summarized in Table 6.
### Table 6. Summarized research process

<table>
<thead>
<tr>
<th>Phase</th>
<th>Preliminary Phase</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Research questions</strong></td>
<td>Current state-of-the-art in research on technical debt and technical debt management</td>
<td>RQ1) What are the reasons and causes for technical debt in software and software development?</td>
<td>RQ3) What are the current management strategies and practices used for managing technical debt in software development?</td>
<td>RQ4) How to adopt technical debt management in a software development company?</td>
</tr>
<tr>
<td><strong>Data collection methods</strong></td>
<td>Literature review</td>
<td>Case studies, semi-structured interviews</td>
<td>Case studies, semi-structured interviews</td>
<td>Action research, semi-structured interviews, workshop</td>
</tr>
<tr>
<td><strong>Data analysis methods</strong></td>
<td>-</td>
<td>Qualitative analysis methods (Atlas.ti)</td>
<td>Qualitative analysis methods (Atlas.ti)</td>
<td>Qualitative analysis methods (Atlas.ti)</td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>-</td>
<td><em>Publication I, Publication II, Publication III</em></td>
<td><em>Publication IV</em></td>
<td><em>Publication V</em></td>
</tr>
</tbody>
</table>
4 Overview of the publications

This chapter presents an overview of the most important results in the publications included in this thesis. The results are based on the studies presented in the five publications attached as an appendix. Publications I, II, IV, and V have been published separately in peer-reviewed scientific conferences and journals. Publication III has been published as a book chapter in a peer-reviewed book. In this chapter, each of the publications mentioned are discussed briefly, including the research objectives, main results, and relation to the overall goal of the thesis.

4.1 Publication I: The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company

4.1.1 Research objectives

The objective of this publication was to study technical debt in a software development environment. The goal was to understand whether technical debt really exists in software development, and if so, where it comes from and why, and what are its effects on software and software development. In addition, the objective was to examine whether there exist any technical debt management strategies that are used systematically during software development. The research questions for this study were the following: (1) What are the causes and effects of technical debt? and (2) What management and reduction strategies/practices are being used for technical debt? The study was an exploratory case study with two independent software product lines in a middle-sized Finnish software company. 12 interviews were performed with industrial practitioners with both business and development background.

4.1.2 Results

The results showed various situations of technical debt in both of the studied product lines. The interviewees from the product lines described several examples of technical debt that had happened previously, and examples that were still happening and currently relevant. The interviews with the participants showed that the causes for technical debt could be divided into two a priori main categories: intentional and unintentional causes (McConnell, 2007). The primary cause for intentional technical debt in the studied product lines was the lack of time given for development. This resulted in other additional causes for intentional technical debt that were pressure to the development team, complexity of the source code, and business decisions, which were often caused by lack of technical knowledge and communication challenges. The observed causes for unintentional technical debt were lack of coding standards and guides, junior coders, lack of knowledge about future changes, and lack of documentation.
The identified effects of technical debt on software development could also be divided into two a priori main categories: short-term effect and long-term effect (Lim et al., 2012). In the short term, the decision to take technical debt seemed to have mostly positive effects. With the decision to take technical debt, the product lines were able to acquire time-to-market benefit, which sometimes resulted in increased customer satisfaction. However, in the long term, if these technical debts were not paid back, it caused multiple negative effects on software development. For example, product lines faced extra working hours, errors and bugs, customer unsatisfaction, and complexity of the source code from non-repaid technical debt.

The strategies and practices for technical debt management provided interesting results. Interestingly, neither of the product lines had any specific or systematic management strategies in place for technical debt management. Both collected technical debt issues sometimes to a backlog, but they were not systematically repaid afterwards. However, there were some practices that were used to reduce technical debt occasionally. These practices included refactoring, bug fixing days (even though bugs are not considered as technical debt in this thesis, bug fixing days can be used to track underlying technical debt), code reviews, and coding standards and guides. In addition, both product lines had a good communication structure between the business management and the development team, which helped to discuss and deal with technical debt more efficiently.

The results of this publication are summarized in Table 7. Overall, the results showed that the primary reasons and causes for technical debt were formed as a result of different management decisions that were made during the project to reach deadlines, or unknowingly due to lack of knowledge. In the long term, if technical debt is not paid back, it may generate quality issues to the software, which will later show as economic losses, such as extra work and decreased productivity. However, companies can use technical debt also to reach customers with faster releasing to gain an edge over the competition with time-to-market. To use technical debt correctly, companies need to create a management plan including practices that decrease and control technical debt systematically.

4.1.3 Relation to the whole

This study focused on understanding the reasons and causes for technical debt, and its effects on software and software development. The results of this study contribute to the overall knowledge on the role of technical debt in real software development by providing empirical data from case examples of two software product lines. This publication suggests that there is no single reason for technical debt. Technical debt can be caused by both technical and organizational reasons. The effects of having technical debt vary depending on how long technical debt lasts without being repaid. At first, the decision to take technical debt could be beneficial, but it can turn to negative consequences later.
This publication was the first step to creating overall understanding on the role of technical debt in software and software development. In addition, we were able to gain some understanding related to technical debt management. By understanding the reasons and causes for technical debt and observing how it will affect the software and software development, it is possible to understand what type of management activities are needed to manage and reduce technical debt efficiently. This study showed that the management of technical debt is not necessarily fully understood and taken in use yet. Researchers and practitioners need to find and develop activities to manage and reduce technical debt systematically from the software, to maintain a healthy and sustainable software product during its entire lifecycle. It is essential to understand first the reasons and effects of technical debt on software and software development, before developing any specific management activities.

<table>
<thead>
<tr>
<th>Table 7. Summary of the findings of Publication I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What are the causes and effects of technical debt?</strong></td>
</tr>
</tbody>
</table>
| **Intentional causes of technical debt** | • Lack of time given for development  
• Pressure to the development team  
• Complexity of the source code  
• Business decisions  
  - Lack of technical knowledge  
  - Communication challenges |
| **Unintentional causes of technical debt** | • Lack of coding standards and guides  
• Junior coders  
• Lack of knowledge about future changes  
• Lack of documentation |
| **Short-term effects of technical debt** | • Time-to-market benefit  
• Increased customer satisfaction |
| **Long-term effects of technical debt** | • Extra working hours  
• Errors and bugs  
• Customer unsatisfaction  
• Complexity of the source code |

| **What management and reduction strategies/practices are being used for technical debt?** |
| **Practices for reducing and preventing technical debt** | • Refactoring  
• Bug fixing days  
• Code reviews  
• Coding standards and guides  
• Communication structure between business management and development team |
4.2 Publication II: The Benefits and Consequences of Workarounds in Software Development Projects

4.2.1 Research objectives

The goal of this study was to understand the benefits and consequences of taking workarounds during software development. Workarounds and shortcuts have been recently become known as an example of technical debt during software development projects, and can be often categorized often as intentional technical debt. Whereas Publication I was more of a general study about the causes and effects of technical debt based on interviewees’ experiences, the objective in Publication II was to study technical debt through more in-depth scenarios. The objective was to understand what the reasons were for workarounds to be taken and what were their benefits and consequences in longer term.

The empirical data was collected in two case organizations. The first case was a middle-sized software development company with two separate product lines, and the second case was a large telecommunications company. The first case company was also a participant in the study presented in Publication I. We conducted 17 semi-structured interviews with managers and technical specialists to get a broader perspective to the workarounds and shortcuts taken in each of the cases.

4.2.2 Results

The results showed that taking workarounds was daily practice in software development. Seven unique scenarios were observed and examined in detail, where a workaround or shortcut was taken on purpose. The reasons, benefits and consequences in each of the identified scenarios were studied. A summary of the scenarios in the studied cases with a priori categories for the reasons for workarounds, benefits of workarounds and the consequences of workarounds is presented in Table 8.

The main reason for workarounds was the challenge of meeting a deadline, which can be seen as time pressure. Emerging business needs of the company had an effect on architectural decisions, because developers did not have time to change the selected software components, even if it would have been more beneficial for software development in the future. Prioritization of the features based on their business values also forced to implement workarounds to lower priority features. Another reason for workarounds was the complicated code base, as a workaround was easier to implement than refactoring the complex existing solution with longer time and higher risk.

The primary benefit of workarounds was the time-to-market. When taking workarounds, the companies were able to have increased speed of development, especially on high priority features. These factors then resulted in increasing customer satisfaction, in some cases saving company reputation, and sometimes even in significant financial savings.
due to early release. However, the workarounds were not always just about benefits, and they also had negative consequences. In all of the observed scenarios, the workarounds resulted in decreased code maintainability. The decreased code quality and maintainability then led to various outcomes, which resulted in extra working hours, extra costs, features of low scalability, major refactorings, increased time for newcomers to start working, and even sometimes lack of motivation to work with the code base.

Overall, the results of this study show that workarounds are often intentional decisions, which can be caused by both organizational and technical reasons. As for organizational reasons, the business people in companies often deal with the benefits of taking technical debt and can therefore underestimate the negative consequences of workarounds, like decreased maintainability of the code base. In contrast, engineering people have to deal with all consequences, and therefore they hesitate to take workarounds. However, they are often under the pressure from the business and have little power to make the final decision.

<table>
<thead>
<tr>
<th>Scenario #</th>
<th>Case</th>
<th>Reason for workaround</th>
<th>Benefits of workaround</th>
<th>Consequences of workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>Time pressure.</td>
<td>Time-to-market,</td>
<td>Decreased code maintainability, Extra working hours, Extra costs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Company reputation,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Increased customer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>satisfaction.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A (a)</td>
<td>Time pressure, Complicated code base.</td>
<td>Time-to-market, Increased customer satisfaction.</td>
<td>Decreased code maintainability, Extra working hours, Major refactoring.</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Time pressure, Complicated code base.</td>
<td>Time-to-market.</td>
<td>Decreased code maintainability, Lack of motivation to work with the code base.</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>Time pressure.</td>
<td>Time-to-market,</td>
<td>Decreased code maintainability, Extra working hours.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Significant financial savings due to early release.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A (a)</td>
<td>Time pressure.</td>
<td>Time-to-market,</td>
<td>Decreased code maintainability, Increased time for newcomers to start.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Increased speed of development.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A (b)</td>
<td>No time for changing the selected software components.</td>
<td>Time-to-market.</td>
<td>Outdated software components, Lack of new features available in newer versions of components, Decreased code maintainability</td>
</tr>
<tr>
<td>7</td>
<td>A (b)</td>
<td>Prioritization of features based on their business value.</td>
<td>Increased speed of developing high priority features.</td>
<td>Decreased code maintainability when scaling the feature.</td>
</tr>
</tbody>
</table>
4.3 Publication III: The Effects of Software Process Evolution to Technical Debt – Perceptions from Three Large Software Projects

4.2.3 Relation to the whole

This study focused on understanding the reasons, benefits and consequences of taking workarounds and shortcuts during software development. The study results provided seven observed scenarios of taking a workaround. The study results offer empirical evidence on the role of technical debt in software development, with real-life examples from software development companies. Similar to the results in Publication I, various reasons, benefits and consequences were identified for technical debt.

When looking at the combined results presented in Publication I and Publication II, it can be seen that the technical debt phenomenon is not only related to activities done with programming, but it is also strongly influenced by management and organizational activities, such as decision-making, which includes the economic perspective. Therefore, the causes and effects of technical debt should not be looked only from the technical viewpoint, but it also includes social and economic aspects of software development.

Publication I and Publication II focused mostly on examples of technical debt that can be categorized as intentional technical debt, where technical debt is taken strategically on purpose to e.g. speed up the development. The final publication in Phase 1 (Publication III) focused more on unintentional technical debt that could happen for more unknown reasons, such as the company's organizational structure or software processes, which are not necessarily very visible in a software company.

4.3 Publication III: The Effects of Software Process Evolution to Technical Debt – Perceptions from Three Large Software Projects

4.3.1 Research objectives

The goal of this study was to understand the relationship between software process evolution and technical debt. The empirical data was collected through 17 semi-structured interviews in three large software development projects with a long development history. All the case companies in this study were large, working in the telecommunications industry. The objective was to identify situations where software process evolution, including e.g. organizational structure changes or software process improvement, had happened during the project history, and if possible, to find out what its relationship to technical debt was.

The reason for this study was based on the results gathered during the research for Publication I and Publication II. The previous results showed that technical debt is not necessarily caused only by technical reasons and activities done by people in the software development team. Instead, technical debt can also be caused by organizational reasons, such as the company's business needs and management-level decisions, which are not always in the hand of software developers. In addition, the results of previous studies showed that technical debt was often an intentional decision to take during software
development. Therefore, there was a need to conduct more research on the causes and effects of unintentional technical debt, which can occur for more unknown reasons, such as software process evolution.

4.3.2 Results

The results of this study show that there exists a relationship between software process evolution and technical debt. Five scenarios of software process evolution had happened in the studied cases, which were examined in detail by studying their relationship to technical debt. The scenarios are summarized in Table 9. In the analysis, we used six a priori categories, which were used to describe each scenario. The a priori categories were developed on the basis of related literature and the research questions in this thesis. The categories were the type of process change, reasons for process change, challenges for process change, issues during process change, benefits of process change, and effect on technical debt.

Two types of process changes were identified that had happened in three studied software projects. Software development methodology included changes that were done to change processes, techniques, or tools. Organizational structure included changes in the structure of the organizational unit in the project or company. These changes were e.g. addition of a new team, changing the current structure of teams, and outsourcing/offshoring the software development to another company or country.

There were several reasons for these process changes. In some situations, lack of frequent releases and time-to-market forced companies to make changes to their software development methodologies to reach the customer faster. In another situation, the structure of the current project team or too high development costs created a need to switch organizational structure to increase productivity. Sometimes these changes were also a reason for technical debt, and the companies needed to increase the overall quality by making changes to their current software processes.

The software process changes also created a set of challenges. Lack of competence was one of the main reasons why software process change was seen as a difficult task to conduct. In addition, the challenges within the organizational side of the change were e.g. size of the organization, other companies and teams, and in some cases teams’ multi-location set difficulties to make the adaption. In some cases, it was also difficult for the teams in the project to switch to new ways of working and a cultural change, which led to team resistance.

The software process changes had clear issues and problems during the adaptation time. The main issue observed in the studied projects was decreased productivity and quality. When productivity decreased, it resulted in problems in architectural quality and scalability, because the development team did not have the same amount of time to use for development, which resulted in increased technical debt. However, the benefits of process change were the opposite after a certain period of adaptation time. There were
4.3 Publication III: The Effects of Software Process Evolution to Technical Debt

– Perceptions from Three Large Software Projects

situations where increased productivity and quality had started to show, and it was easier for the companies to develop new processes. Another benefit from process changes was increased competence, as now the developers had learned new ways of working, which made them more efficient.

Overall, the reasons for software process evolution can be business-related, where the company has to improve their current software processes in order to gain more advantage over competition and time-to-market. Software process evolution is often considered as a positive thing towards better development processes. However, companies rarely think of its negative consequences and the resistance to change. The inquiry into the practice of three large development organizations revealed that the evolution of software processes affects the technical debt accumulated in the code base and can decrease the quality of software in the short term. However, if the company takes no steps to manage technical debt, this may finally have a dramatic effect on software development and maintenance processes.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of process change</td>
<td>Software development methodology</td>
<td>Software development methodology</td>
<td>Organizational structure</td>
<td>Organizational structure</td>
<td>Software development methodology</td>
</tr>
<tr>
<td>Reason for process change</td>
<td>Lack of frequent releases, time-to-market</td>
<td>Lack of frequent releases, time-to-market</td>
<td>Project teams</td>
<td>Too high development costs</td>
<td>Project teams, technical debt</td>
</tr>
<tr>
<td>Challenges for process change</td>
<td>Lack of competence, team resistance</td>
<td>Lack of competence, needed time for change, size of organization, cultural change</td>
<td>Other companies and teams</td>
<td>Multi-location, other companies and teams</td>
<td>Lack of competence</td>
</tr>
<tr>
<td>Issues during process change</td>
<td>Architectural quality, scalability, decreased productivity</td>
<td>Decreased productivity</td>
<td>-</td>
<td>Decreased productivity and quality</td>
<td>-</td>
</tr>
<tr>
<td>Benefits of process change</td>
<td>Increased productivity, quality, competence</td>
<td>Increased productivity, quality</td>
<td>Increased productivity, quality, competence</td>
<td>-</td>
<td>Increased productivity, quality, competence</td>
</tr>
<tr>
<td>Effect to technical debt</td>
<td>Increased at the beginning. Decreased at the end.</td>
<td>Increased at the beginning. Decreased at the end.</td>
<td>At the beginning, technical debt was high. Decreased at the end.</td>
<td>At the beginning, technical debt was low. Increased at the end.</td>
<td>At the beginning, technical debt was high. Decreased at the end.</td>
</tr>
</tbody>
</table>
4.3.3 Relation to the whole

This study focused on understanding the relationship between software process evolution and technical debt. The results of the study provided five scenarios, where a relationship between software processes and technical debt was identified. The study contributes with empirical data to the understanding of the role of technical debt in software development by providing examples of unintentional technical debt in the studied cases. The results of this study emphasise especially the relationship of organizational and human aspects of software development to technical debt. The reason for technical debt in software is not always caused by programming and technical reasons, but the reason can also be related to higher-level organizational structures and processes used by the company, which are harder to identify and change.

These results had also a significant impact on the research presented in the two final publications of this thesis. The results gathered from Publication I, Publication II, and Publication III showed that the role of technical debt in software development has both technical and organizational attributes. Even though technical debt can be found in the source code or design of the software, organizational and human aspects such as the used processes, decision-making, communication etc. can also have a significant relationship to the causes and effects of technical debt. These results showed that when technical debt management frameworks and models are designed and developed, they should also include not only activities related to development, but also activities at the organizational level. The improved activities at the organizational level could help in the actual technical debt repayment activities. The combined results of Phase 1 (Publications I, II, and III) were used to create an overall view on the causes and effects of technical debt.


4.4.1 Research objectives

The objective of this study was to understand how software development teams manage technical debt. One large software development organization was investigated to gather evidence on technical debt management, especially on the strategies and practices that were currently used by the development teams. A total of 25 interviews with eight software development teams were performed during this study. There was one main research question “How do software development teams manage technical debt?” and four subquestions (1) “What activities are used in the studied development teams?”; (2) “What methods, models, practices, or tools do the studied development teams use for each technical debt management activity?”; (3) “Are there any maturity differences in adopting technical debt management activities between development teams?”; and (4) “What are the biggest challenges in technical debt management?”
The results of the studies in *Publication I*, *Publication II*, and *Publication III* showed that technical debt as a phenomenon is not related to technical aspects of software development only. Even though technical debt is a technical compromise in software, it still includes various causes and effects that are not necessarily related to the technical aspects of software development. Technical debt can be caused by various organizational reasons, and having technical debt can have an effect on both the software and software development. Therefore, technical debt management should also include not only technical activities done with software, but also organizational activities that include humans and social interactions in the development team.

The results of a previous technical debt management study were utilized in this research. An extensive mapping study by Li et al. (2015) collected several research papers related to technical debt management. The study identified eight technical debt management activities that were an inspiration for this study. The technical debt management activities were repayment, prevention, documentation/representation, identification, measurement, monitoring, communication, and prioritization. These categories were used especially during the data collection, when software development team members were interviewed.

### 4.4.2 Results

The results indicated differences in the maturity of technical debt management. The term maturity in this context means the ability of the development team in performing technical debt management activities. The results showed that technical debt management was conducted at various levels in the studied software development teams. There existed development teams that did not have any clear strategies or tools to manage and reduce technical debt. In this type of development teams, refactoring was often conducted just during normal development when there was time for it, and there was not anything preplanned in place. Some of the studied development teams had structured strategies and processes to reduce, monitor, measure, and manage their technical debt during software development. There were also development teams that could be seen to be in the middle of these two, where some technical debt management activities were suggested to be used, but they were used only occasionally. Also, various approaches were used in each technical debt management activity. The approaches varied from practices conducted by the whole development team to practices conducted by a single person. In some approaches tools were used to assist the activity.

The results also showed that there were several challenges in technical debt management. Firstly, technical debt management was mostly conducted as a human activity, rather than by using automated tools. This can create a situation where technical decisions are based on hunches rather than specific estimations done with tools. Secondly, it is challenging to evaluate and estimate why some technical debt issues should be prioritized higher than others. Thirdly, technical debt management requires the whole organization to work, which is why there needs to be a proper mindset to look at the issues and manage them constantly in all activities. Lastly, technical debt management requires time, and sometimes it is difficult to justify the benefits of conducting it.
Based on the results of this study, a technical debt management framework was developed. The technical debt management framework is one of the main results of this thesis. The framework describes the management activities, stakeholders, responsibilities and approaches/practices/tools that are associated with technical debt management. The framework can be used by software development companies to improve and evaluate both internal and external processes of technical debt management. It is important for software development teams to notice that working on the highest level of the framework does not always mean that it will reduce technical debt more efficiently and end up as healthier software. It is possible that a development team that does refactoring only when necessary has less technical debt in their software than a development team that conducts all types of technical debt management activities continuously. The reason for this is that technical debt as a phenomenon is complex and hard to identify and measure, which is why it is difficult to know the real amount of technical debt. In addition, technical debt has a tendency to be highly case-related, which affect the decision of whether to reduce or keep technical debt. However, using the framework and the activities in it can increase the visibility and knowledge regarding technical debt in software. With increased visibility and knowledge, software companies can make smarter and safer decisions in technical debt reduction and management. The framework has been presented in Publication IV.

Overall, the results of this study show that technical debt management is often an unfamiliar concept to software development teams. The results showed that there were maturity differences on the practices and tools used for technical debt management. Organized and systematic technical debt management through the developed technical debt management framework could help software companies to increase the visibility and manageability of technical debt issues. However, there is still an urgent need to develop more tools and practices for technical debt management, especially on measurement, monitoring, and prioritization activities. The developed framework can increase knowledge on technical debt management activities within software development teams.

4.4.3 Relation to the whole

The results contribute with empirical evidence on the state-of-the-art in technical debt management research by presenting how development teams manage technical debt in a real-life software development environment. The previous results presented in Publication I, Publication II, Publication III and a mapping study by Li et al. (2015) showed that technical debt management includes activities from both the programming and management parts of a software project. The results were used to create a framework that includes both technical and organizational activities in technical debt management. The framework for technical debt management developed in this study provides important knowledge on the role of technical debt management in software development. Other researchers can use the framework to conduct more research. Now, the final step in this thesis was to use and adopt some aspects of the developed technical debt management framework on a software development team. The objective was to see what practices and processes are needed to improve the visibility and manageability of
4.5 Publication V: Developing Processes to Increase Technical Debt Visibility and Manageability – An Action Research Study in Industry

4.5.1 Research objectives

The objective of this study was to test and adopt processes to increase the visibility and manageability of technical debt. The studied case company described the current situation in the development as a problem. The management of the case company mentioned that similar technical debt issues kept reoccurring, and it was a challenge for the management. The company was looking for new ways of identifying and fixing these problems. Therefore, the goal was to use some parts of the developed technical debt management framework presented in Publication IV. The company and research group decided to focus on developing new processes for technical debt identification, documentation, and prioritization. The research questions of this study were: “How to improve technical debt identification and documentation?”, “What factors should be taken into consideration when prioritizing technical debt?”

4.5.2 Results

The study resulted in a process that included basic activities and templates for technical debt identification, documentation, and prioritization. The action research process used in this study is summarized in Figure 6. This research can be summarized as five main activities and outcomes. The process does not consist all technical debt management activities that are presented in the framework in Publication IV, and it does not include management activities for new technical debt items that would be taken. Therefore, the process developed during this study is limited to only technical debt that has already occurred previously and is already known by the studied case company.

The first step of the process was interviews, where the researcher interviewed people related to the software development team, in order to understand the current issues related to technical debt and its management. The outcome of this step was problem identification, which helped to understand what technical debt management processes were needed in the development team. The second step was to develop and use a process for technical debt identification. In the case company, the identification was conducted by gathering the data from the members of the development team. The outcome of this step was an increase of the visibility of technical debt, which helped to gain an overall technical debt view. The third step was to develop a process for documentation. This research used a simple template, including ID, date/reporter, name, description,
alternatives, and rationale, to document each technical debt issue. The outcome of this step was getting technical debt stored, which will make it easier to manage it.

The fourth step was to organize a workshop. The objective of the workshop was to gather all the members of the development team from the identification and documentation step to review and discuss technical debt issues. During the workshop, the participants evaluated each technical debt issue based on a template for evaluation. The template included questions about the benefits, risks, reasons, solutions, and importance of each technical debt issue. The outcome of this step was getting technical debt evaluated. The benefit of evaluating technical debt was that now there existed some level of reasoning to decide on priorities and make decisions for future work. The last step of the research process was sorting. When there was evaluation for each technical debt item, it was easier to sort the issues out based on their importance. The last outcome of the process was a prioritized technical debt backlog. This backlog could be now used to add more development tasks related to technical debt reduction.

The outcome of the research was a set of templates and processes to identify, document, evaluate and prioritize technical debt. With the used process, the company and research group were able to collect 47 technical debt issues, which were later documented and prioritized based on their importance. These templates and processes were used successfully in the company to transfer from a situation where the knowledge of technical debt was not explicitly documented, to a situation where a specifically prioritized technical debt backlog was in place to reduce technical debt. The product unit is now using this new technical debt backlog to increase the visibility and manageability of technical debt. The technical debt reduction tasks are now part of normal development and the company is planning to expand the process to other development teams as well.

4.5.3 Relation to the whole

This study focused on developing and testing activities for technical debt identification, documentation and prioritization practices. The research results indicate that it is possible to start technical debt management with simple process improvements and additions to software development practices. The results contribute to the overall understanding of
technical debt management by providing an action research process conducted in a real software company. The developed process for technical debt management uses the technical debt management framework developed in the research presented in Publication IV. The results show a case example on how a software company can improve their maturity related to technical debt management.

When looking at the results presented in Publication IV and Publication V, it can be seen that technical debt management has its challenges. The concept of technical debt management is not yet fully understood by software development companies. It is important to understand that technical debt management does not only include a programmer and a source code, but also higher level organizational aspects, such as social and political activities. Even though activities such as technical debt measurement, monitoring and prioritization are extremely difficult and challenging to conduct, technical debt management can be slowly improved with other simple practices of the technical debt management framework, which were used in the study in Publication V. The results of this thesis regarding technical debt management can be used by software companies to improve their understanding on the issue.

4.6 About the joint publications

All the scientific publications included in this thesis were written and performed by a team of researchers that included Jesse Yli-Huumo as the doctoral student and the supervisors of the thesis, Professor Kari Smolander and D.Sc. Andrey Maglyas. In Publication V, two company representatives were included as co-writers. The study design and data collection protocols were designed by Yli-Huumo under the supervision of and collaboration with the supervisors. Yli-Huumo also conducted all the interviews and led the workshop, analysed all the collected data, and prepared the initial versions of the publications for supervisor reviews. Each phase of the research activities was evaluated and directed by the supervisors. The supervisors also provided insightful guidance into the procedures of collecting and analysing data, and how these should be reported in the publications. In Publications I, II, III, and IV Yli-Huumo was responsible for designing the research, implementation of the data collection protocols, the main data analysis, and reporting on the results in form of the publications included in this thesis. For Publication V, Yli-Huumo designed the action research process in collaboration with the research group and company representatives. In addition, Yli-Huumo organized the workshop and the interviews with the company representatives.
5 Research contributions, implications and limitations

This chapter discusses and summarizes the main results of the scientific publications presented in the previous chapter. This chapter also gives implications for practitioners and researchers, discusses possible future research directions, and evaluates the research while discussing the limitations and validity of the study.

5.1 Reasons and causes for technical debt in software

The first goal of the study was to establish an understanding for the reasons and causes for technical debt to occur in software. This part gives answers to the first research question (RQ1) “What are the reasons and causes for technical debt in software and software development?” This question (RQ1) was answered in Publication I, Publication II, Publication III, and partly in Publication IV and Publication V by observing a number of examples of technical debt that had taken place in the studied case companies.

The original definitions of technical debt divide the phenomenon into two main types: intentional technical debt and unintentional technical debt (McConnell, 2007). In addition, Fowler (2009) implies that the reason for technical debt can be different attitudes towards technical debt. The reasons and causes for technical debt, in addition to those given by McConnell (2007) and Fowler (2009), have been studied and described by other scholars as well. For example, Tom et al. (2013) conclude that the reasons for technical debt are pragmatism, prioritization, processes, attitudes, ignorance, and oversight. Lim et al. (2012) identified that technical debt is often caused by intentional decision-making rather than unintentional side-effects. Kruchten et al. (2012a) describe that pressure on the development team forces technical debt to be taken intentionally, but it can also be caused by unintentional reasons, such as lack of competence. Martini et al. (2014) mention that technical debt is caused by business pressure and business evolution, but can also be a result of legacy software and human factors. Klinger et al. (2011) state that bad organizational processes can also have an impact on technical debt.

The results presented in Publication I, Publication II, Publication III included similar findings as earlier studies and supported the fact that technical debt can occur in software for various reasons rather than for any one specific reason. However, our findings implied that technical debt should also include a separation to technical and organizational reasons, which were not explicitly included in the technical debt definitions by McConnell (2007) and Fowler (2009). In the current technical debt state-of-the-art, the research papers often discuss the reasons and causes for technical debt, but they can be limited to a specific context, e.g. a certain type of program code, and do not include a more broader view on the reasons and causes of technical debt at the entire organizational level. To our knowledge, there are no other studies that describe and categorize technical debt from both the technical and organizational perspective to provide a more broader view on its reasons and causes. Therefore, we have used our observations in the case companies to include a broader view to expand the technical debt landscape, which
Research contributions, implications and limitations

explains the causes of technical debt with both intentional/unintentional technical reasons in the software code, and intentional/unintentional organizational reasons that happen in software companies with processes and humans. In addition, there is an aspect of legacy reasons for technical debt to occur in software. The reasons and causes for technical debt to occur in software, based on our observations in the case companies, are presented in Figure 7.

![Figure 7. The reasons and causes for technical debt](image)

**Technical reasons for intentional technical debt** are related to existing complexity of the software code and design that force the developer or development team to implement shortcuts or workarounds intentionally to bypass the implementation of a long and complex solution, which often requires refactoring, redesigning, or rewriting to fix already existing technical debt. The reason to take technical debt in this category can be caused by motivational issues, where the developer does not have the will to work with the complex code, or by a decision not to break the software or feature, due to an overcomplex code or design. (Publications I and II)

**Organizational reasons for intentional technical debt** are common reasons for technical debt to occur during software development, related to the economic needs of the software company. When a software company has more business-driven development to improve e.g. time-to-market and customer satisfaction, this causes technical debt to be taken. Software developers often have to deal with lack of time and pressure given for
development, which forces them to implement shortcuts and workarounds to reach deadlines. (Publications I and II)

**Technical** reasons for unintentional technical debt are often related to lack of competence and knowledge. A software developer might unknowingly implement a bad code or use a wrong design because of lack of competence and knowledge of certain techniques available. It is a challenge to sometimes even know what is the best possible solution in software development. (Publications I and II)

**Organizational** reasons for unintentional technical debt are often caused by high-level aspects of a software project, such as software processes or organizational structures. Non-optimal processes or structures in software companies can unknowingly cause technical debt to occur in software. When development as a process is not the most optimal, it can cause decreased productivity and quality. These types of reasons for technical debt are difficult to identify by project management, which makes this technical debt type especially challenging for a software project. (Publication III)

The last categorization for the causes of technical debt in software is **legacy reasons**. Code, design, and architecture that become legacy and “old” after a long period of time, when technologies and tools are improved, can also be seen as a type of a reason for technical debt. Even though e.g. an architecture that was implemented several years ago and was not considered as technical debt back then, can be considered as technical debt in the future, when there are more efficient and improved technologies or solutions for that specific architecture available. This type of reason for technical debt is closest to the unintentional and technical category, but it should still be categorized on its own, because the reasons for this type of technical debt to occur are different. (Publication V)

As the contribution to the state-of-the-art of technical debt, Figure 7 offers knowledge and clarification to the overall landscape of the reasons and causes of technical debt. For example, Kruchten et al. (2012a) express that there is a need for organizing a technical debt landscape to define and identify properly what constitutes technical debt. The results of our research contribute to the technical debt state-of-the-art by organizing the reasons and causes for technical debt into a coherent view, indicating that there are various technical aspects with complex programming and design-related factors, as well as organizational reasons involving social and political aspects of managing software development project, that are the reasons and causes for technical debt. In addition, we believe that software legacy reasons for technical debt are a separate category that needs to be discussed outside of being just unintentional technical debt. These results were identified from empirical evidence, which has also been seen as a lacking aspect in current technical debt state-of-the-art literature (Li et al., 2015). As a contribution, Figure 7 can be used as a guide to understand the reasons and causes for technical debt to occur in software from a broader perspective compared to other existing literature.
5.2 Effects of technical debt on software and software development

The second goal of the study was to establish an understanding of the effects that technical debt has on software and software development. This part answers the second research question (RQ2) "What are the effects of technical debt on software and software development?" This question (RQ2) was answered in Publication I, Publication II, and Publication III, and partly in Publication IV and Publication V by identifying several examples of technical debt in the studied case companies.

The effects of technical debt were originally described as a cause of a quick and messy implementation used to reach a deadline that would have an effect as harder implementation in the future (Fowler, 2003). Lim et al. (2012) mention that taking technical debt does indeed have benefits in the short term, such as meeting deadlines, delivering software in time, reaching markets early, or the possibility to develop simple prototypes, but taking too much technical debt will turn to negative effects in the longer term, such as decreased customer satisfaction due to e.g. omitted product quality. Shortcuts taken in code and design can speed up the development, but will slow projects down eventually (Nord et al., 2012) because of the omitted quality and increased complexity (Izurieta et al., 2012; Tom et al., 2013). Tom et al. (2013) also add that technical debt will have an effect on the development team’s morality and productivity. Therefore, technical debt can be seen as both a possibility and a risk to software company (ibid.), because the effects can be either beneficial or detrimental.

Publication I, Publication II, and Publication III indicated that technical debt has various effects on both software and software development, similar to the ones that had been already discussed and identified in the related technical debt research. However, we believe that the current literature lacks a more broader view on the effects technical debt has on the technical aspects of software and the organizational aspects of a software development company. Therefore, a categorization of the technical effects taking place in different parts of a software development company would help to understand the overall picture that constitutes technical debt (Kruchten et al., 2012a). In addition, these perspectives should be contrasted to the length (short-term vs. long-term) (McConnell, 2007) and the type of technical debt (intentional vs. unintentional) (Fowler, 2003).

A view on the effects of technical debt based on our research is presented in Figure 8. The view includes a separation of technical effects (software-related), and organizational effects (software development team and software company -related). In addition, these dimensions are divided to short-term and long-term effects, because technical debt can have different effects based on its length (McConnell, 2007).

Technical effects of technical debt in the short term are often related to intentional technical debt (Publications I and II). The technical effects are often minimal and do not have much impact on software quality. When a developer makes an intentional decision to implement a shortcut during software development, it is often a small “duct tape” implementation, which can be easily changed or fixed in the future. The benefits in the
Research contributions, implications and limitations

Technical perspective on the implementation of shortcuts in the short term are that the required technical need is completed (e.g. a working feature). The effects of *unintentional technical debt* are often difficult to notice in the short term. A non-optimal code or design implemented unknowingly is a quality issue, but the effects are known in detail in the long term.

<table>
<thead>
<tr>
<th>Short-term effects</th>
<th>Long-term effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Technical effects</strong></td>
<td><em>Technical debt creates complexity to software, which decreases the overall quality, maintainability and scalability.</em> As a side-effect it creates bugs, errors and defects, which have to be dealt with by refactoring.</td>
</tr>
<tr>
<td><em>Simple technical solutions, which do not require refactoring of existing solutions and can fulfill the requirements of the task. Technical debt does not have a major effect on quality.</em></td>
<td></td>
</tr>
<tr>
<td><strong>Organizational effects</strong></td>
<td><em>Technical debt creates extra working hours and costs, because development productivity decreases and maintainability requires more time. Decreased productivity can be seen also in customer satisfaction and developers’ low motivation.</em></td>
</tr>
<tr>
<td><em>Increased development speed and time-to-market. Possible positive effects on increased customer satisfaction, company reputation and cost savings.</em></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. **Summary of the short-term and long-term effects of technical debt**

*Organizational effects* of technical debt in the short term are often the most beneficial to a software company (*Publications I and II*). These effects are often associated with *intentional technical debt*. Shortcuts increase development speed, software delivery and time-to-market. These benefits can also have a positive effect on e.g. customer satisfaction, company reputation, and project costs. The effects of *unintentional technical debt* are almost non-existent in this category and difficult to notice.

*Technical effects* of technical debt in the long term are often detrimental to software (*Publications I, II and III*). These effects can include both *intentional* and *unintentional technical debt*. Long-term unfixed technical debt starts to affect the overall software
quality, because it increases complexity in the code and design. Complexity in the code and design decreases e.g. quality, understandability, maintainability and scalability. These decreased aspects start to generate bugs, errors and defects as a side-effect. Existing technical debt can also sometimes be fixed with new technical debt, but eventually the complexity will increase to a certain level, where taking new technical debt is no longer possible.

**Organizational effects** of technical debt in the **long term** are often detrimental to software development (*Publications I, II and III*). These effects can include both **intentional** and **unintentional technical debt**. Long-term technical debt in software has negative consequences on software development and software projects. When complexity is increased in software, the time required for development tasks also increases. Therefore, in the long term technical debt causes extra working hours and costs, because productivity decreases and maintainability requires more time. Eventually, the lack of productivity can start to show in decreased customer satisfaction and developers' motivation.

It is important to notice that there can also be intentional or unintentional technical debt that are not considered relevant to the overall software quality by the software company (*Publications I and II*). Technical debt cases where e.g. the implemented code or design in some feature is average, and the effects show as complexity in programming, could be still considered as a low-value feature for the company, because it might not be a core functionality with economic value (*Publications I and II*). This type of technical debt cases is related to software economics and decision-making, but it introduces an interesting perspective to technical debt, where short-term benefits from taking technical debt will outscale the negative consequences in the long term (Lim et al., 2012). Therefore, Figure 8 can be used to categorize the effects of technical debt, but the impact of the effect can vary based on the company's needs in a certain feature or part of software.

As the contribution to technical debt state-of-the-art, Figure 8 organizes the view on the effects of technical debt on software and software development. The results contribute to the goal of understanding what constitutes technical debt and describe its role in software and software development (Kruchten et al., 2012a). In addition, the results have been observed and identified in software development case companies, which adds more empirical evidence to the current state-of-the-art (Li et al., 2015). Figure 8 can be used by researchers and practitioners to understand and increase knowledge on the effects of technical debt on software and software development.

### 5.3 Technical debt management in software development

The third goal was to study technical debt management. This part answers the third and fourth research questions (**RQ3** and **RQ4**): “What are the current management strategies and practices used for managing technical debt in software development?” and “How to adopt technical debt management in a software development company?” These questions (**RQ3** and **RQ4**) were answered in *Publication IV* and *Publication V*, and partly in *Publication I* by studying several software development teams' daily practices.
The main research gap in the current technical debt management state-of-the-art is the lack of research on the entire process of technical debt management. Most of the current studies in technical debt management are focused on the identification, measurement and repayment of technical debt (Li et al., 2015), which can be considered as technical activities, because they require software code and design-related knowledge. However, technical debt management also includes non-technical activities, such as communication, representation/documentation, and prioritization that require social and political perspectives of software development. There is a need to create a framework for technical debt management that includes and combines all the eight technical debt management activities and practices used in each activity, both technical and non-technical. The current state-of-the-art has studies that include one or two out of the eight identified technical debt management activities (ibid.). A more organized view on technical debt management and its various perspectives can help both researchers and practitioners to understand what is technical debt management in a broader perspective. In addition, most of the state-of-the-art research is not validated or conducted with empirical evidence in a software development environment (Alves et al., 2016; Li et al., 2015), and case examples from industry are not often provided.

As the main contribution to the technical debt management state-of-the-art, Publication IV presents a developed technical debt management framework that described the activities, approaches, practices, tools, stakeholders, and responsibilities in technical debt management. The technical debt management framework combines the previously identified technical debt management activities presented by Li et al. (2015) and empirical evidence observed in a software development environment. The framework is an effort to create a more complete view on technical debt management and its activities. The technical debt management framework can be found in Publication IV.

The results of Publication IV and Publication V indicate several interesting points related to technical debt management. The study revealed different maturity levels between software development teams on technical debt management (Publication IV and Publication V). This has been also identified in other related research. According to Lim et al. (2012), 75% of the participants in their research were not familiar with the term technical debt, which can indicate that technical debt management was also an unknown concept. Power (2013) states that there are differences in the efforts to technical debt management between software development teams. Our results showed that for some software development teams technical debt management and its activities were an unfamiliar concept, and it was considered as just an activity that software developers do to refactor and redesign the software code (Publication IV).

Technical debt management has a lot of similarities with the capability maturity model (CMM), which is used to measure the maturity of software processes in a company (Paulk et al., 1993). Similar to CMM, technical debt management can be conducted at different maturity levels (Publication IV). In CMM, there are five levels, initial (chaotic), repeatable, defined, quantitatively managed, and optimizing (Paulk et al., 1993). The technical debt management framework (Publication IV) is divided into three maturity
levels, *unorganized, received, and organized*. *Unorganized* is the level where a development team does not put any effort to the activity or when the focus is minimal. *Received* is the level when a development team has acknowledged the need for a certain technical debt management activity and when it already conducts it on some level. *Organized* is the level when the development team has recognized a technical debt management activity as an essential part of software development, and it is conducted continuously by the whole development team. It is also possible to increase the maturity of technical debt management activities. Publication V showed a case example where a software development team was able to increase the maturity of their technical debt management activities by improving the visibility and manageability of technical debt in *identification, representation/documentation, and prioritization* with simple templates and practices.

Regarding the maturities in the technical debt management framework, the results of Publication IV and Publication V also indicate that technical debt management has various challenges, which make it demanding to improve the maturity of the activities. Technical debt as a phenomenon in software is complex to understand. Since technical debt is a non-optimal solution in the code, design, or architecture, taken intentionally, or sometimes even occurring unknowingly, it is challenging to reason how a certain solution is not good, or how and why it should be improved (Lim et al., 2012). There are not yet tools available to specifically identify large technical debt issues (Izurieta et al., 2012; Kruchten et al., 2012a). There are some static code analysis tools, which can identify smaller technical debt issues taking place in some aspects of the code, such as modularity violations, design patterns, code smells etc. (Izurieta et al., 2012; Li et al., 2015). Therefore, the identification of larger technical debt issues has to be done with human knowledge (Kruchten et al., 2012a), which can be time-consuming. The challenges in technical debt identification (Curtis et al., 2012; Guo et al., 2011) make maturities in the technical debt management framework difficult in other technical activities as well (e.g. *measurement, monitoring, repayment*).

Publication IV showed that software development teams used non-technical activities (e.g. *communication, representation/documentation, and prioritization*) for technical debt management more often than technical activities. However, there are still challenges in them if there exists lack of communication between project stakeholders related to technical debt management (Publication IV and Publication V). Software developers often conduct refactoring and code improvement activities on their own, next to normal development, and this information is hidden from the management and the rest of the organization (Publication IV). Therefore, the knowledge and visibility of technical debt can be hidden from general knowledge, which makes technical debt management a challenge (Klinger et al., 2011). This was for example the case in the software development team in Publication V. However, with small additions to technical debt management practices, it is possible to improve the maturity related to non-technical activities (Publication V).
The results of the study contribute to research gaps identified in the mapping studies by Li et al. (2015) and Alves et al. (2016). As the contribution to technical debt state-of-the-art, the results of *Publication IV* and *Publication V* provide empirical evidence on how development teams manage technical debt in a software development environment. The results contribute to the state-of-the-art by identifying several activities and processes related to technical debt management, and showing that companies can have different maturities in technical debt management. The results were used to develop a technical debt management framework that describes the role of technical debt management in software development.

5.4 **Summary of contributions to technical debt state-of-the-art**

The contributions of this research to technical debt and its management state-of-art are summarized in Table 10.

<table>
<thead>
<tr>
<th>Research problem / Research gap in state-of-the-art (presented in chapter 3.1)</th>
<th>Contribution to technical debt and its management state-of-the-art</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition of technical debt</strong></td>
<td>Even though this research does not provide an explicit answer to this research problem, the results show various examples of technical debt, derived from empirical evidence in software development. These examples can be used by other researchers and practitioners to use more empirical data as evidence, when discussing what is technical debt in software engineering.</td>
</tr>
<tr>
<td><strong>Causes and effects of technical debt (RQ1 &amp; RQ2).</strong></td>
<td>The results contribute to the research gap by identifying and presenting several examples of technical debt in software development. These examples show several identified causes and effects technical debt has on software and software development. The identified causes and effects are summarized in Figure 7 and Figure 8, which present a more complete view on what constitutes technical debt in software engineering. Figure 7 and Figure 8 can be used by both practitioners and researchers to understand the role of technical debt from the technical and organizational perspectives of software development.</td>
</tr>
<tr>
<td><strong>Technical debt management framework (RQ3 &amp; RQ4).</strong></td>
<td>The results were used to develop a technical debt management framework. The developed framework contributes to the state-of-the-art by organizing several identified activities, approaches, practices, tools, stakeholders, and responsibilities to a high-level view on technical debt management, derived from examples of software development teams. The framework can be used in further research to organize and categorize aspects of technical debt management and to improve the maturity of technical debt management in practice.</td>
</tr>
<tr>
<td><strong>Lack of empirical research on technical debt and its management.</strong></td>
<td>The results of this research were acquired from empirical evidence through several interviews and case studies with real software companies. The results contribute to the state-</td>
</tr>
</tbody>
</table>
5.5 Implications for practice

The results have multiple implications to practice that need to be acknowledged and discussed further. These implications can be used by software practitioners to think of and discuss the various aspects and perspectives that technical debt as a phenomenon creates to software and software development.

The definition of technical debt is still a challenge. At the moment, there exist several definitions and opinions of technical debt (Avgeriou et al., 2016; Tom et al., 2013). This fact was also observed during the interviews (Publications I-V), where we were able to see that people can have differentiating views and opinions on technical debt. In addition, the fact that several subcategories of technical debt have been identified and created (Alves et al., 2016; Li et al., 2015; Tom et al., 2013) can cause confusion in the metaphor. This research focused on technical debt associated to intentional and unintentional shortcuts and workarounds in code and design during the implementation of software. Even though this definition was presented to the interviewees, it was still possible to see that some interviewees (e.g. Publication V) also considered e.g. process debt (Codabux and Williams, 2013), test debt (Guo and Seaman, 2011), etc. to be counted as technical debt.

We believe that technical debt should be only considered as non-optimal solutions in code, design and architecture that are constructed during implementation, and it should not include other phases and aspects of the software development process, similar to the definition presented by Kruchten (2016). Therefore, rather than including several subcategories to be counted as technical debt, they should be categorized as reasons and causes for technical debt. Subcategories, such as people debt, process debt, knowledge distribution debt, documentation debt etc. (Alves et al., 2016; Li et al., 2015; Tom et al., 2013) are more related to the reasons and causes of technical debt (which can be found in e.g. Figure 7) and not to the actual technical debt that can be found in the technical aspects of software. As an example, having “shortcuts” in communication with people during software development can be considered as debt, but it is still more a reason for technical debt, rather than actual technical debt.

On the other hand, it can be suggested to software practitioners that technical debt could be defined in a way that the entire software organization has the same view on technical debt. For example, in Publication V, a software development team considered some of the technical debt subcategories, such as process debt (Codabux and Williams, 2013) and test debt (Guo and Seaman, 2011) as technical debt. Even though this does not align e.g. with our view and definition on technical debt, it was still a view that was seen with the people in the development team. This way, the company can have a common view on
Research contributions, implications and limitations

Technical debt, which can help to understand and manage it better, rather than creating confusion with the overall metaphor. However, this might be problematic for the academia and scientific literature, as it makes the metaphor non-cohesive.

Technical debt is an essential part of software development that needs to be acknowledged and understood by software development companies. Taking and creating shortcuts and workarounds in code and design is a daily activity in software development (Publications I and II), which are caused for both intentional and unintentional reasons (McConnell, 2007). Technical debt can be a strategy to improve the time-to-market (Lim et al., 2012), but technical debt in software also causes complexity and omits the overall quality (Izurieta et al., 2012; Zazworka et al., 2011b), which have an effect on software development (Tom et al., 2013). It is essentially important for software development companies to understand where technical debt comes from, how technical debt affects software and software development, and when technical debt should be taken and when not. Therefore, the first step is to understand that technical debt is a real phenomenon in software development, and software companies have to acknowledge this fact.

Technical debt is not a single cause-effect relationship. Technical debt can be caused by multiple reasons, and the effects can vary depending on the purpose and length (Figure 7 and Figure 8). Technical debt in software can be intentional and unintentional, occurring for both technical or organizational reasons (Figure 7). The effects of having and taking technical debt show in the technical aspects of software and organizational aspects of software development (Figure 8). Therefore, having technical debt in software is not always a simple cause-effect relationship, which is easy to predict and manage. Instead, technical debt cases can vary from small strategic decisions to hit a deadline (Publication I and Publication II) to a higher-level organizational problem that causes technical debt to occur in software (Publication III).

Even though technical debt is a technical aspect in the program code, it is affected by social and political aspects in the software development organization. Technical debt is a non-optimal solution in a code or design (Avgeriou et al., 2016), but the reasons for it are not always necessary technical-related (Kruchten et al., 2012a; Lim et al., 2012) (Figure 7). The decision to take technical debt can be caused by social and political aspects of software development (Klinger et al., 2011; Kruchten et al., 2012a; Lim et al., 2012), where people in the organization create a reason to take strategic technical debt to avoid e.g. late releases or breaking down a software (Lim et al., 2012). Therefore, it is essential for software practitioners to understand that the role of technical debt in software development is also hugely affected by non-technical reasons. Therefore, it should not just be considered as an activity with developers and a refactoring task, but also as a decision-making activity at the organizational level, which can require both technical and software economics knowledge (Publication I and Publication II).

All technical debt is not bad. Even though having technical debt in software is never good and should not be an optimal way to build and design software, taking technical
Technical debt can provide benefits to a software company in some situations (Publication I and Publication II). Technical debt can be used as a tool to speed up the development strategically, which can appear as significant benefits in release cycles that can increase software economics through the time-to-market (Lim et al., 2012). In addition, there is necessarily no need to repay all technical debt (Buschmann, 2011). Sometimes “just an average” code and design can fulfill the needs of a customer (Publication II). The customer is not necessarily interested in technical implementation, and focuses on actual functionality. Also, lower priority features used by fewer customers do not necessarily require as much effort on the code and design by the software company, compared to higher priority features, which are a major factor in company economics (Publication I and Publication II). This sets up interesting and challenging questions to software development as to when the software code and design are good enough for the customer, how much technical debt the software can include, and when should technical debt be repaid.

Technical debt is different for software development and management. The view on technical debt is not always necessarily the same to the people in software development and those in software economics, such as business management and marketing (Lim et al., 2012). For software developers, technical debt is not often the wanted way of working, because low quality code will have an effect on development in the future, and a developer usually does not want to write bad code on purpose. On the other hand, people in software economics-related positions are often more open-minded for taking technical debt, because the goal of software economics is to create value for the company and customers through software, the deadlines of feature release are often a main priority, and technical debt is seen as a strategic investment (ibid.). It is important to acknowledge this aspect in practice, and it would be important to find a way to include both perspectives of technical debt to create high quality code in-time and in-budget.

There will always be technical debt in software and it is almost impossible to avoid it. Having and taking technical debt is a natural occurrence in software development (Publication I, Publication II and Publication III). The constantly changing markets and technologies force companies and development teams to adapt themselves into new situations (Kruchten et al., 2012a). In addition, software development itself is a complex process, which can include various views and starting points for code and design, producing different outcomes and complexities (Brooks, 1987). Therefore, having always some technical debt in software is not a surprise and it is almost impossible to avoid it. In practice, it is important to acknowledge that technical debt will always occur in some way, and it is important to have management activities to identify the issue and have a plan to reduce it in time.

Technical debt management should be acknowledged and adapted by software development companies. Technical debt is a complex phenomenon in software and software development and a challenge for decision-making. The goal of a software company should always be to keep technical debt visible and manageable. Even though all technical debt should not necessarily be ever repaid (Buschmann, 2011), there are also
research contributions, implications and limitations

87

technical debt cases that require more attention from software companies to be dealt with (Publications I, II, III). Therefore, it is important for software companies to have technical debt management activities in place (Publication IV and Publication V).

Even though several technical debt management activities have been identified, as well as practices, approaches and tools developed and presented, technical debt management can still be considered as a quite new aspect in software development that requires more research and empirical evidence to improve it (Publication IV) (Alves et al., 2016; Li et al., 2015). Software development companies are not even necessarily familiar with the concept of managing shortcuts and workarounds in code and design (Publications I, IV and V). It is often possible that in a software company these types of activities are left to software developers, who have to work with technical debt code and design within given deadlines. Therefore, technical debt is often invisible for people working in the management of a software company (Publications I-V) (Lim et al., 2012). This is the reason why the development team often has to work with strict deadlines on new features, rather than fixing and reducing technical debt that was taken a long time ago. Increasing the visibility of technical debt with different management activities, both technical and non-technical, will help the software company to understand the level of technical debt and the effort required for it.

There are still clear challenges and problems with technical debt and its management, which have to be assessed and evaluated by software practitioners and the research community. Even though a clearer vision on what is included in technical debt management has started to appear in practice and research, there are still multiple questions unanswered on how to do it in real-life software development. There are multiple ideas and suggestions on technical debt have been presented, but they are rarely tested and evaluated in a software development environment (Alves et al., 2016; Li et al., 2015). In addition, the identification, measurement, monitoring, and prioritization of technical debt seem to include several challenges, because software as an artefact is a complex phenomenon that is extremely difficult to analyse and draw conclusions on (Curtis et al., 2012; Eisenberg, 2012; Guo et al., 2011; Izurieta et al., 2012). It is a challenge to understand in depth how a non-optimal solution becomes optimal, what are the effects of a change to software and software development, and how to measure and monitor the change. The research on technical debt requires especially more studying and developed tools for analysing software to understand technical debt and its management better (Publication IV and Publication V).

5.6 Implications for future research

The first main implication for future research is that technical debt as a metaphor and its management still require further research. Current technical debt mapping studies (Alves et al., 2014, 2016; Li et al., 2015; Tom et al., 2013) show that as a research topic, technical debt is gaining increasing interest and has already generated a number of research papers in various research areas. However, the mapping studies also reveal that the definition of
technical debt still remains unclear and there is a need to reach consensus between researchers. In addition, the question on what constitutes technical debt (Kruchten et al., 2012a) requires more research to understand the causes and effects of technical debt through empirical evidence. Even though our research contributes to the state-of-the-art by providing evidence from several case studies and technical debt examples, there is still more room to conduct additional research in other software companies in different industries and environments to identify possible new cases of technical debt.

The second main implication for future research is technical debt management activities. The mapping studies by Li et al. (2015) and Alves et al. (2016) have collected and identified a large number of approaches and activities for technical debt management. However, the challenge is the lack of empirical evidence in these approaches and activities. Even though this research contributes to the state-of-the-art of technical debt management by providing a theoretical framework (Publication IV) that describes several attributes related to technical debt management, it still requires empirical evaluation. Therefore, as an implication to future research, it is important to study and test the framework in other companies, development teams, environments, and industries to evaluate it. Additional research can discover new maturity levels, activities, processes, practices, techniques, or tools, which can improve the applicability of the framework for software companies in the future.

The third main implication for future research direction is the tools needed for technical debt management (Ernst et al., 2015; Li et al., 2015). Currently, technical debt identification is often done with code analysis tools that can identify issues happening in the source code (Izurieta et al., 2012; Li et al., 2015). In future research, there is a need to study and develop tools for larger architectural and design-related issue management (Li et al., 2015). In addition, there is lack of tools for other technical debt management activities (Li et al., 2015), which require more ideas and additional research.

5.7 Validity and limitations of the thesis

This research was defined as a research with an interpretive view on investigating reality by using empirical research methodologies to create and develop new theories, models and frameworks. Publications I, II, III, and IV were performed with case study methodology (Yin, 2003), where the data collection was done with semi-structured interviews (Seaman, 1999). The validity in case studies can be distinguished between four aspects of validity: construct validity, internal validity, external validity, and reliability (Runeson et al., 2012).

Construct validity means to what extent the operational measures that are studied really represent what the researcher has in mind and what is investigated according to the research questions (Runeson et al., 2012, p.71). An example given by Runeson et al. as a threat for construct validity during an interview is a situation where the constructs discussed are not interpreted in the same way by the researcher and the interviewee. A
large majority of data collection in this research was performed with interviews. Therefore, construct validity may be a major threat to the validity of the study.

To improve the threat of construct validity, we performed some specific actions before and during the interviews. In Publications I-IV, the construct validity was improved by designing the data collection protocol carefully. Before the interviews, the data collection protocol was reviewed, discussed, and corrected if necessary based on suggestions by the research supervisors. During the interviews, we put emphasis on explaining the goals of the research, the research questions, and the research terminology carefully to the interviewees. This way it was possible to increase the validity, so that the interviewer and the interviewees had similar understanding on technical debt.

**Internal validity** is of concern when causal relations are examined (Runeson et al., 2012, p.71). Runeson et al. define that internal validity can become an issue when the researcher investigates whether one factor has effects on another examined factor, but there is a risk that it is also affected by a third factor. If this third factor is not identified, there is a threat to internal validity.

We improved the threat of internal validity with emphasis on interview structures. We used semi-structured interviews (Seaman, 1999) with open-ended questions, rather than having strict close-ended interviews with exactly the same questions. With the more open interview structure, it was possible to examine the phenomenon much more in depth, because there was always a possibility to ask more details of each answer. In addition, as the research was part of the N4S –research program with companies interested in the research topic, it was always possible to contact the companies after the interviews and ask additional questions related to found causalities, if there any new follow-up questions were found.

**External validity** is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case (Runeson et al., 2012, p.71). Generalizability has been seen as a challenge in case study (Flyvbjerg, 2006). Case studies do not provide statistical generalizability (Yin, 2003). However, the goal in this research was to create theories and generalize the technical debt phenomenon in specific case settings. We consider generalization as theoretical (Lee and Baskerville, 2003), i.e. abstraction from real-life events and actions to theoretical constructs. Case study can be used appropriately to develop theories that are generalizable within the case settings (Lee and Baskerville, 2003). Therefore, as a threat to the external validity of this research, we do not consider the developed theories and knowledge to be generalized in other settings, where they have not been empirically tested yet (Lee and Baskerville, 2003).

**Reliability** means to what extent the data and the analysis are dependent on the specific researchers (Runeson et al., 2012, p.72). In a situation where another researcher would replicate this study, the results should be hypothetically the same. The threat to reliability
comes when it is not clear how to e.g. code the collected data or understand the interview questions (ibid.).

The threats to reliability were improved by designing and reporting the data collection and data analysis protocols carefully for each publication. The research settings in Publications I-V are presented in each publication and should be repeatable by other researchers. However, the use of semi-structured interviews can bring some challenges and questions to reliability. As semi-structured interviews are mostly open-ended discussions (Hove and Anda, 2005), it may be possible that another researcher replicating this study would ask other follow-up questions compared to our questions. Replicating our research in other software development companies and environments can lead the discussions in interviews to other directions compared to our research.

In addition to Runeson and Höst's (2008) description of possible threats in research validity, this thesis is also evaluated based on Creswell's (2003) description on research validity. Easterbrook et al. (2008) have presented the eight strategies identified by Creswell (2003) for improving the validity of research, which are well suited for exploratory case studies in software engineering. These eight strategies are presented and reflected to this research in Table 11.

Table 11. Research validity evaluation in case study (Creswell, 2003; Easterbrook et al., 2008)

<table>
<thead>
<tr>
<th>Validity improvement</th>
<th>Addressed in this research</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Triangulation</strong> addresses the issue that</td>
<td>The data collection in this research was conducted with research partners in the N4S –research program. The empirical evidence included data from six case organizations, 14 development teams, and 55 people.</td>
</tr>
<tr>
<td>the research data should be shown to the</td>
<td></td>
</tr>
<tr>
<td>research participants to see if the</td>
<td>The scientific publications were sent to the participants for a review. In some cases, the present author received feedback on whether the results and analysis were done according to their statements.</td>
</tr>
<tr>
<td>interpretation of data makes sense to them</td>
<td></td>
</tr>
<tr>
<td>(Easterbrook et al., 2008).</td>
<td></td>
</tr>
<tr>
<td><strong>Member checking</strong> addresses the issue that</td>
<td>Each publication describes carefully the settings where the research was performed.</td>
</tr>
<tr>
<td>the research data should be shown to the</td>
<td></td>
</tr>
<tr>
<td>research participants to see if the</td>
<td></td>
</tr>
<tr>
<td>interpretation of data makes sense to them</td>
<td></td>
</tr>
<tr>
<td>(Easterbrook et al., 2008).</td>
<td></td>
</tr>
<tr>
<td><strong>Rich, thick descriptions</strong> addresses the</td>
<td>The research received feedback from several external sources in peer-reviews, which were sometimes used for improving the result presentation.</td>
</tr>
<tr>
<td>issue that the descriptions of settings and</td>
<td></td>
</tr>
<tr>
<td>results should be given with enough details</td>
<td></td>
</tr>
<tr>
<td>(Easterbrook et al., 2008).</td>
<td></td>
</tr>
<tr>
<td><strong>Clarify bias</strong> addresses the issue that</td>
<td>All the interview data was carefully coded, categorized and labelled, and also checked and confirmed by at least one supervisor, to prevent discrepant information.</td>
</tr>
<tr>
<td>the researcher has to respect the biases</td>
<td></td>
</tr>
<tr>
<td>brought by other researchers to the study,</td>
<td></td>
</tr>
<tr>
<td>and self-reflect them when reporting on the</td>
<td></td>
</tr>
<tr>
<td>findings (Easterbrook et al., 2008).</td>
<td></td>
</tr>
<tr>
<td><strong>Report discrepant information</strong> addresses</td>
<td></td>
</tr>
<tr>
<td>the issue that the researcher does not only</td>
<td></td>
</tr>
<tr>
<td>include the data to the findings that</td>
<td></td>
</tr>
<tr>
<td>confirms the theory, but also those which</td>
<td></td>
</tr>
<tr>
<td>appear from different perspectives to</td>
<td></td>
</tr>
<tr>
<td>findings (Easterbrook et al., 2008).</td>
<td></td>
</tr>
</tbody>
</table>
The data collection phase lasted usually approximately a month with one software development team. During this time, we were able to contact the interviewees with email after the interviews, to acquire additional data. We believe this time was enough to ensure reasonable understanding of the case.

Peer briefing addresses the issues that the researcher should find peer briefing to the research findings to discuss and question the ideas, to make the final assumptions as valid as possible (Easterbrook et al., 2008). The results and findings of this study have been discussed and reviewed several times by both supervisors. In addition, all the results and findings were published in peer-reviewed journals, conferences or books that generated a high quality inspection of results from other researchers in the same research area.

External auditor addresses the same issue as peer briefing, but the researcher should find an unknown external auditor to address the research (Easterbrook et al., 2008). The answer is the same as to peer briefing.

Publication V was performed with action research (Avison et al., 1999), where the data collection was performed with semi-structured interviews and a workshop. The challenge in action research is to maintain rigour (Baskerville and Wood-Harper, 1996). Baskerville and Wood-Harper (1996) present seven characteristic strategies to improve the challenge with rigour in action research study. These characteristic strategies are discussed and reflected to this research in Table 12.

Table 12. Seven characteristic strategies to improve rigour in action research (Baskerville and Wood-Harper, 1996)

<table>
<thead>
<tr>
<th>Rigour improvement strategy</th>
<th>Addressed in this research</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Consideration for the paradigm shift.</strong> Is action research appropriate for the research question? (Baskerville and Wood-Harper, 1996)</td>
<td>The aim of Phase 3 was to develop and test processes for technical debt management in the case company. We believe that close co-operation with the case company to develop and test these processes was suitable to acquire best results. The action research method provided us with the means to work together with people in the company to find out the best possible processes based on our existing knowledge on technical debt management and the company’s previous experience in software product and software development processes.</td>
</tr>
<tr>
<td><strong>Establishment of a formal research agreement.</strong> The researcher should prepare the research subjects for the “warrants” that will give authority to the research team to conduct research in the organization (Baskerville and Wood-Harper, 1996).</td>
<td>The research motives and process in Publication V were discussed and agreed on between the research group and the company representatives. We set up the rules with the company for data collection, which was told to the people participating in the action research.</td>
</tr>
<tr>
<td><strong>Provision of a theoretical problem statement.</strong> The theoretical foundation must be present as a premise if the experiment is to remain valid as research (Baskerville and Wood-Harper, 1996).</td>
<td>The objectives of action research were derived from research problems and gaps in the current literature on technical debt and its management. The developed and tested processes in Publication V were designed based on background literature on technical debt management and the experience of people participating in the action research.</td>
</tr>
</tbody>
</table>
Planned measurement methods. Collected data in action research can be very unstructured, so it is important for the researcher to plan methodical data collection methods carefully (Baskerville and Wood-Harper, 1996).

Maintain collaboration and subject learning. It is important to have good collaboration and relationship with the research subjects. It is also important to develop subject learning during the action research process (Baskerville and Wood-Harper, 1996).

Promote iterations. Action research is often a cyclical process where the data recording should be a repetitive plan to take and evaluate organizational actions (Baskerville and Wood-Harper, 1996).

Restrained generalization. For action research, generalization is a problem, because it is completed in unique settings and is difficult to repeat (Baskerville and Wood-Harper, 1996).

This is a limitation in Publication V. The data collection was performed with interviews and a workshop that were used to develop processes and templates in collaboration with the case company. However, the limitation is that we did not use any measurement methods to evaluate the results of the action research process.

During the action research process, we kept a good relationship with the case company by having discussions and result reporting multiple times. In addition, the interviews and the workshop organized with people in the company were used as a learning experience regarding technical debt management.

This is a limitation in Publication V. Due to scheduling and the time required for the design and execution of Phase 3, we were only able to perform one cycle of the action research process. This can be considered a major limitation, but we believe that the developed processes and templates would not change drastically with more iterations.

This is a limitation in Publication V. The results of this action research are not necessarily applicable to a larger population. However, our goal in this research was not to develop processes for a larger population. Our aim was to develop and test technical debt management processes that would be used in specific settings in the case company environment. We hope that the developed processes would be used also in other case study settings, which could then be used to improve the generalization.
Research contributions, implications and limitations
6 Conclusions

This chapter summarizes the research contributions and suggests possible directions for future research work.

6.1 Contributions and summary

The thesis consisted of one preliminary phase and three main phases. The preliminary phase examined the existing literature with the goal of understanding the current challenges and research gaps related to technical debt. The first main phase studied the causes and effects technical debt has on software and software development (RQ1 & RQ2). The second main phase studied technical debt management to observe and identify how software development teams use various activities and tools to manage technical debt (RQ3). The third phase tested the processes and practices for technical debt management (RQ4).

The results of this research provide high-abstraction concepts and theories, which can be used by other researchers and software practitioners to improve their knowledge on the role of technical debt in software development. The main results and findings of this research can be summarized as follows:

- Technical debt is an essential phenomenon that should be acknowledged by companies to understand its role in software development.
- The definition of technical debt is unclear, and it needs to be addressed more with research and discussion.
- Technical debt can be intentional, as a strategic decision to incur technical debt, and unintentional, as a negative side-effect of causing technical debt unintentionally.
- Technical debt is caused for both technical reasons in software and organizational reasons with people in software development.
- Technical debt can be also caused by legacy software over a long period of time.
- Taking technical debt can be beneficial in the short term with time-to-market and increased development speed, but having technical debt can be detrimental in the long term with omitted software quality and development productivity.
- All technical debt is not necessarily bad and can be sometimes left unpaid in software development.
- Technical debt management is a process including various technical activities done with a software and organizational activities done with social and political aspects in software development and economics.
- The developed technical debt management framework presents the maturity levels, stakeholders, responsibilities, activities and practices/tools that are used in technical debt management and introduces the idea that technical debt management can be described with various maturity levels similar to the capability maturity model (CMM).
Conclusions

- A software development team is able to increase the maturity of low technical debt management with simple additional practices to increase the visibility and manageability of technical debt, but it can evolve also to a continuous process with higher maturity.
- The current challenge in technical debt management is the lack of tools available, especially in identification, monitoring, measurement, and prioritization, which can make high maturity difficult to achieve.

In summary, technical debt is a relevant and essential aspect of software development, which needs to be addressed and acknowledged by software development companies. The research on technical debt and its management has increased in recent years, and produces currently high quality research on its role through different aspects in software development (Alves et al., 2016; Li et al., 2015). The results of this research provide better understanding on the role of technical debt in software engineering by giving examples and ideas on its causes and effects through real case examples. In addition, the results of this research were used to develop a technical debt management framework. Practitioners and researchers can use the results and the framework to improve their knowledge on technical debt and the current state-of-the-art in technical debt research.

6.2 Future research topics

Our research provided several real examples of technical debt in software development, especially bringing out more knowledge about the separation of the technical and organizational perspective on technical debt. We believe that even though technical debt should be considered as a technical artefact in software, it is highly related to the organizational perspective of a software development company, with social and political aspects of humans and human interactions. Therefore, as a suggestion, we believe that future research topics should include more studies on the causes and effects of technical depth from the organizational perspective, which should then be combined with more technical papers in current state-of-the-art studying technical debt in software. This way, it would be possible to improve current technical debt management by creating an extensive framework that would include both technical activities with software and organizational activities with social and political actions.

The developed technical debt framework is an effort to combine these two perspectives of technical debt management, bringing out a more extensive view involving technical and organizational perspectives. However, the framework requires more testing, improving and evaluation. Therefore, we believe that one of the future research topics would be to use the developed framework in different software environments and gather evidence on whether it can bring any value to software development teams when they try to increase their maturity with technical debt management.

In addition, there is urgent need to develop new tools for technical debt management, especially for more technical activities with identification, measurement, monitoring, and repayment. When it is possible to understand what and how much technical debt there is,
it will also help in other activities in technical debt management related to decision-making. Therefore, the final future research direction should focus on the development of tools for technical debt management.
References


References


References


References


Appendix A: Publications
Publication I


© 2014, Springer International Publishing Switzerland. Reprinted with permission
The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company

Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander
Lappeenranta University of Technology, Finland
(jesse.yli-huumo,andrey.maglyas,kari.smolander)@lut.fi

Abstract. Fierce competition in the software market forces companies to release their product under tough time constraints. The competition makes companies reactive and they need to release new versions often. To achieve this need for speed, companies take shortcuts to reach deadlines. These shortcuts and resulting omitted quality are called technical debt. We investigated one middle-size Finnish software company with two independent product lines and interviewed 12 persons in different positions to understand the causes and effects of technical debt. We were also interested in specific strategies and practices for managing technical debt. The results showed that technical debt is mostly formed as a result of intentional decisions made during the project to reach deadlines. Customer satisfaction was identified as the main reason for taking technical debt in short-term but it turned to economic consequences and quality issues in the longer perspective. Interestingly, neither of the product lines had any specific management plan for reducing technical debt but several practices have been identified.

Keywords: technical debt, software project management, case study, software company, software quality.

1 Introduction

The increased competition in the software market forces companies to think about their time-to-market strategy. Balancing the choice of releasing poor-quality software early or high-quality software late is challenging [1]. This leads companies to an awkward situation where they have to decide what quality is omitted and what shortcuts in the development process they have to take. These shortcuts and omitted quality are called “technical debt.”

The term technical debt was first introduced by Ward Cunningham [2]. According to Seaman et al. [3] compromises in software development are the reason for technical debt and they should be paid back to avoid decreasing maintainability and health of the system. Technical debt might also cause economic consequences in a software project [4]. McConnell divides technical debt into two different basic types [5]. Type I occurs unintentionally where a design approach turned out to be bad or a junior coder writes bad code. Type II is intentional where a company makes a strategic decision.
to incur technical debt. The technical debt metaphor appeals to both project managers and software development teams [6]. The goal of a software development team is to create quality software and embrace tools and techniques for it [6]. People responsible for the management of the project and the company also care about quality but are more focused on cost and schedule factors [6].

In this paper we define technical debt as a shortcut in a software development project to reach certain deadlines. We firmly believe that technical debt is not only related to technical decisions but also to management and business decisions. Moreover, the decision to have some technical debt can be conscious to deliver the product to the market faster.

This case study is conducted in one middle-size Finnish software company with two independent product lines. The objective of this case study is to understand the sources of technical debt in software projects. Our secondary objective is in identification of strategies and practices for the management of technical debt.

The rest of the paper is organized as follows. Chapter 2 shows the related work of this topic. Chapter 3 provides the research method used in this study. In Chapter 4 we introduce the results analyzed from the gathered data. Then, in Chapter 5 we discuss about the results and conclude the paper.

2 Related Research

In this chapter we identify the related research conducted on causes and effects of technical debt. We are also focusing on strategies and practices used for managing and reducing technical debt.

2.1 The Causes and Effects of Technical Debt

The study conducted by Lim et al. [7] shows that technical debt is not always a result of bad coding. It can also include intentional decisions to trade off competing concerns during business pressures. The study also identified that technical debt can be used in short-term to capture a market share and even to collect early customer feedback. In the long-term technical debt effects tended to be negative. These effects included increased complexity, poor performance, low maintainability and fragile code. This furthermore led to bad customer satisfaction and extra working hours. However, the authors also mentioned that there were cases where short-term benefits of technical debt clearly outweighed the future costs.

Klinger et al. [8] studied the causes of technical debt by interviewing four software architects at IBM. The study revealed causes for technical debt including pressure from stakeholders, technical communication gap between stakeholders and project team, decision making without quantification of possible impacts, and unintentional debt occurring from acquisitions, change of requirements, and changes in the market ecosystem.

Siebra et al. [9] analyzed an industrial project that lasted six years by analyzing emails, documents, CVS logs, code files and interviews with developers and project managers. The analysis revealed that technical debt was taken mainly with strategic decisions. They also found that the use of a unique specialist could lead the develop-
ment team to solutions that incur technical debt. The study also identified that the
effects of technical debt can both increase and decrease the amount of working hours.

Zazworka et al. [10] studied the effects of god classes and design debt on software
quality. God classes are examples of bad coding [11] and therefore include a possi-
bility for refactoring. The results showed that god classes require more maintenance
effort that include bug fixing and changes to software and are considered as a “cost”
to software project.

Buschmann [12] explained three different cases of technical debt effects. In the
first case technical debt in one platform started to grow so large that development,
testing and maintenance costs started to increase dramatically. In the second case
developers started to use technical debt to increase the development speed. This re-
sulted to significant performance issues that turned later on to economic conse-
quences. In the third case the software product had already incurred a huge amount of
technical debt that led to increasing maintenance costs. However, management ana-
alyzed that reengineering the whole software would cost more than doing nothing. This
resulted to situation where the management decided not to do anything for the tech-
nical debt because it was cheaper from the business point of view.

Guo et al. [13] studied the effects of technical debt by following one delayed main-
tenance task through a software project. The results showed that delaying the main-
tenance task would have almost tripled the costs if it had been done later.

Overall, the related research about the causes and effects show that technical debt
is not always caused because of technical reasons. Studies [7][8][9] showed that tech-
nical debt can also be caused by intentional decisions that were related to business
reasons.

The studies also show that taking technical debt may have short-term positive ef-
fects [7][9] such as the time-to-market benefit, but they will turn to economic conse-
quences and quality issues in a long run if not paid back [7][9][10][12][13]. However,
there are also situations where the short-term benefits outweigh the long-term costs
[7][9].

We also noticed from the related research that technical debt is not just related to
shortcuts in the coding phase. There are several different subcategories of technical
debt mentioned in literature. We have gathered all of these technical debt subcatego-
ries in Figure 1.

Overall, the existing literature reveals large set of causes and effects incurring from
technical debt, but lacks a clear mapping of relationships between different effects
and causes.

2.2 Current Strategies and Practices of Technical Debt Management and
Reduction

Lim et al. [7] found four different strategies for managing technical debt. The first
strategy is to do nothing because technical debt might not be ever visible to the cus-
tomer. The second strategy is the risk management approach to evaluate and prioritize
technical debt’s cost and value. The third strategy is to include different stakeholders
to technical debt decisions. The last strategy is to conduct audits with the development team to make technical debt visible.

Codabux & Williams [14] revealed practices such as refactoring, reengineering, and repackaging used for technical debt management.

The studies [15][16][17] propose the management of technical debt using the portfolio management. This approach is similar to the investment portfolio management. In the portfolio approach technical debt is collected to a “technical debt list” that is being used to pay the technical debt back based on its cost and value.

Krishna and Badu have also developed guides based on their own experience of technical debt management [18][19]. These guides are using different practices for minimizing technical debt. The practices include the basic steps that are focusing on improving refactoring, aspects of coding, continuous learning processes and teamwork. These practices were used in software projects and the results showed an improvement in the adaptation of new changes and better productivity in software project.

Overall, the existing literature is often concentrating on strategies and practices for reducing and preventing technical debt. As a result, the existing literature is lacking clear management approaches for controlling technical debt through the software development life cycle.

3 Research Methodology

Case study was selected as the research methodology for this study. Case studies have been around a long time and they account a large proportion of books and articles [20]. Case study is a method that involves an in-depth examination of a case [21]. Yin defines case study as an ‘empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident’ [22]. In this study, we focus on technical debt
in software project. We focus more on technical debt causes and effects, rather than on the qualities of technical debt in source code and how to measure them. Therefore, we decided to use exploratory case study methodology with semi-structured interviews for data collection. The purpose of this study is to increase our knowledge of the relationship between technical debt causes, effects, and management. This case study consists of the following five steps [22]:

1. Designing the case study
2. Conducting case studies (stage 1): Preparing for data collection
3. Conducting case studies (stage 2): Collecting the evidence
4. Analyzing the case study evidence
5. Reporting the case study

3.1 Designing the Case Study

The strategy for this research was to find a suitable company to study technical debt and to arrange multiple interviews with people in different organizational positions (e.g. development, management, quality assurance). The reason for interviewing different organizational positions was to acquire information about technical debt from different viewpoints in software development projects. The research questions this study addresses are:

1. What are the causes and effects of technical debt?
2. What management and reduction strategies/practices are being used for technical debt?

3.2 Preparing for Data Collection

The selected company is a Finnish software company that offers SaaS business solutions for professional services automation and accounting. The company has three product lines that are managed and developed independently. This study includes two of the product lines that are referred to as Product Line A and Product Line B.

Table 1. The roles of the interviewees

<table>
<thead>
<tr>
<th>ID</th>
<th>Product line</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A</td>
<td>Software architect</td>
</tr>
<tr>
<td>A2</td>
<td>A</td>
<td>Software designer</td>
</tr>
<tr>
<td>A3</td>
<td>A</td>
<td>Project manager</td>
</tr>
<tr>
<td>A4</td>
<td>A</td>
<td>Software test engineer</td>
</tr>
<tr>
<td>A5</td>
<td>A</td>
<td>Production director</td>
</tr>
<tr>
<td>B1</td>
<td>B</td>
<td>Software architect</td>
</tr>
<tr>
<td>B2</td>
<td>B</td>
<td>Software developer</td>
</tr>
<tr>
<td>B3</td>
<td>B</td>
<td>Product line manager/Software test engineer</td>
</tr>
<tr>
<td>B4</td>
<td>B</td>
<td>Software architect</td>
</tr>
<tr>
<td>B5</td>
<td>B</td>
<td>Software developer</td>
</tr>
<tr>
<td>B6</td>
<td>B</td>
<td>UI designer</td>
</tr>
</tbody>
</table>
Product line A provides a financial management solution as a cloud service. The solution has more than 10,000 customers and it is currently the biggest financial management system provider in Finland. The product has been developed since 2004. The size of the development team in Product Line A is 18 persons. The development team is using agile methods and more specifically a Scrum-like approach.

Product line B is a SaaS-based project management solution for multi-organization projects. The solution is used by 1000 companies worldwide. Product Line B was founded in 2004 and acquired by the parent company in 2010. The size of the development team in Product Line B is 13 persons. The development team is also using agile methodologies and a Scrum-like approach.

3.3 Collecting the Evidence

The interviews were conducted between February and March 2014. We had total of 11 interviews, of which five were from Product Line A and six from Product Line B. One of Product Line B interviews included an interview with two persons interviewed at the same time. The data collection started with a contact to the product line managers from both Product Line A & B. Both product managers agreed to have an interview and recommended persons from their own product line that might be suitable to interview about technical debt. We contacted all of the persons recommended and received positive response from each of them.

Table 1 presents the roles of the interviewees in this study. The interviewees were from different positions within the development team and we were able to discuss technical debt from various different viewpoints. We also tailored different questions depending on the role of an interviewee. For example if the interviewee was responsible for the management we focused our questions more on strategies and processes. All interviewees gave us a permission to record the session with a voice recorder. Seven interviews were conducted by one researcher in Finnish and the rest by two researchers in English. Later all interviews were transcribed and translated to English. Overall, the interviews lasted from 25 to 50 minutes, with the average of around 35 minutes.

3.4 Analyzing Case Study Evidence

The total amount of transcribed pages for analysis was 80. The collected data was analyzed with a tool designed for qualitative data analysis, Atlas.ti. The analysis of the transcribed data was performed with a similar method to open coding as in the grounded theory method [23].

4 Results

The main findings are presented in four subsections. The first subsection presents the causes related to technical debt. The second subsection examines strategies and practices for the management of technical debt. The third subsection focuses on the effects in short- and long term caused by technical debt. The final section presents improvements to product lines regarding technical debt.
4.1 Causes for Technical Debt

We asked from the interviewees what the causes for technical debt usually were. The most common cause mentioned was the lack of time given for the development. It was also added that the lack of time generates a lot of pressure to the development team, which ultimately leads to technical debt being taken.

“We have things that are almost well done but they are left undone properly because deadlines are coming and it is working like “ok” and we will just leave it like that.” – B6.

We also identified that when the source code is getting more complex and bigger, it becomes also harder to change. This makes it easier and cheaper in the short run to just take technical debt and use patch code, instead of fixing the bigger problem. A software architect from Product Line A explained the situation where patch code was used to deal with the problem.

“We were working with our CRM product and our job was to refactor the data structure to this new type of data structure. This current data structure was used by almost every customer and it would have been really hard to change it. So we just made some script that syncs the data between these data structures and brings data to our CRM same way. This has generated a lot of problems, like in CRM the data is showing little bit differently even though it is the same data.” – A1.

Another commonly mentioned reason for technical debt was the lack of knowledge. Sometimes it is just impossible to predict the future and some present decisions might incur technical debt later on. The interviewees also described different examples where lack of knowledge has caused technical debt to software product. One interviewee explained that lack of documentation in the source code causes technical debt because the code is then harder to understand and it takes more time to work with it. Also, if new coders do not have enough experience in coding with the company standards, they unintentionally incur technical debt because they produce different style of code compared to the company standards. The last example related to lack of knowledge described was about mistakes in specifications and requirements.

Another major aspect we were interested in at the causes of technical debt was the effect of the business decisions. We asked interviewees about the effects of business decisions to technical debt by causing pressures to the development team. A software architect in Product Line B thought that business decisions do have effect to the amount of technical debt.

“Previous years business decisions have affected us a lot. Reason for this that we have a release every two months and when we have a bigger release we just do not have enough time to do it. This means that we are in a hurry at the end of release and we have to just implement some faster solution. I think that problem is that management is not willing to see what people are actually doing during work day and how much time is doing something actually taking. This leads to problems in distribution of resources and I think that has been our problem these days.” – B4.
A project manager of Product Line A also thought that business decisions are generating pressure to the development team, but he also added that it is not completely the decision of business managers to make deadlines for features.

“Well of course there is pressure coming, but what is the weight of it is case-by-case. At some point if we have promised something to our customer, we try to give deadlines so that there should not be any pressure coming. Of course deadlines are changing and there are situations where we have to reach fast and everything else is postponed, but if we were in a perfect world, this is how we would deliver.” – A3.

We also asked about the communication between the development team and the business management related to decision making on technical debt. Interviewees from both product lines thought that they have a good communication structure where project managers act as good filters between business managers and development team for the decisions on technical debt. Some interviewees also added that they consider this communication structure to be much better compared to companies they have previously worked in. The reason for this was that the previous places were big companies, where getting an opinion about technical debt was more challenging due to more complex and larger communication structure.

We were also interested in whether business people usually listen to the technical opinions of the development team. Most of the interviewees thought that it is difficult to express technical opinions to business managers and to get more development time. However, a project manager of Product Line A expressed that the development team often mentions situations related to technical debt.

“Well we try internally say that if we don’t have capacity for some feature to make it 100% solution, it will be told usually at most of the cases. We will picture it that we don’t have time to make this and this feature completely. So they understand us quite well.” – A3.

We asked could one of the causes for technical debt be the lack of technical knowledge of business managers that might drive not to do the best possible solution. The interviewees thought that it might be the case but did not see it as a problem because project managers usually can express their opinions to these kinds of situations, which limits the amount of bad decisions.

Also, one interesting fact we noticed during interviews was that the examples given by interviewees were not just related to the implementation phase. Interviewees also described how shortcuts were taken for example in testing, architecture and requirements phases of the software life cycle.

### 4.2 Management and Reduction Strategies for Technical Debt

The goal was to find out if the product lines A & B have any clear strategies for managing and reducing technical debt. Neither of the product lines had any specific approach for managing technical debt. However, we were able to identify some practices that were used to report technical debt.

“Well for bigger things we do keep a backlog. If some feature is dropped out we do have a backlog for it. If I am thinking myself at coding and I take some shortcut
somewhere, so do I mark-up it somewhere, not really. But if we are dropping something or finding some things to take, we write them up.” – A1.

One interviewee from Product Line B mentioned that they are using a software called JIRA to store their actions during development.

“Well there is that if we decide that we are going to make it better later, we have this software called JIRA, where we have all the tickets about things we have done. So at least it will be stored in there, other thing is that when it will be fixed.” – B6.

Even though neither of the product lines used any specific approach to manage technical debt we managed to identify practices that were being used to reduce and prevent technical debt. Both of the product lines for example have a specific bug fixing day each week to reduce errors and bugs.

We identified that both product lines were using refactoring to reduce technical debt. However, they did not have any specific refactoring schedule. Refactoring was considered as a part of normal job and was done during the implementation.

Both product lines were using coding standards/guides to prevent and reduce technical debt coming from bad coding and to increase the consistency of the source code.

“We have guides, but they are still missing a lot of stuff. We are combining them all the time and adding stuff. But at the moment they are mostly describing naming the code, architectural stuff. We have different guides for different projects, like different guides for APIs compared to normal software. But at least we have something.” – B4.

For the situation where bad solutions or inconsistency still occurs even after coding standards/guides, both companies had code reviews to check all produced code. This served as the last checkpoint where technical debt can be seen before a release.

4.3 Short-Term and Long-Term Effects of Technical Debt

The analysis revealed several effects how technical debt can affect a software development project in a short-term. First, technical debt, or a shortcut, is used to save development time and deliver a solution faster to the customer. The production director of Product Line A explained a situation from the early stages of the product where taking technical debt saved the company.

“Well I think that the best example is when we started the company in 2001. We were quite out of money, so we were leveraged with debt quite heavily and we were in a hurry to get something done for the next presentation for pilot companies and investors. So we created a lot of stuff that looked it worked, but in reality it might not even work. Time-to-market was so important for the existence of a company, so we did everything we could just to get stuff out to convince that this is a viable solution.” – A5.

Second, the customer satisfaction can be increased by delivering the solution faster but it also increases the technical debt. As customers are more interested in getting the product on time rather than its technical details of implementation, they do not care
about technical debt as long as it does not directly affect the product quality. One interviewee also mentioned that taking technical debt doesn’t feel good to take from developer’s point of view because they know that the solution is not the best one and they might have to fix it later.

We were also able to identify long-term effects occurring from technical debt. The interviewees from both product lines explained that if technical debt is not being managed and reduced, it will have some serious effects in the long-term. A software architect from Product Line B explained a situation where technical debt effects have caused problems to the product line.

“We have things like calendar synchronization, which we have done years ago and there is a lot of bugs and errors in the functionality and problems in implementation. These kinds of things have generated us a lot of bugs in the long-term and lots of repairing. Repairing has been done by putting patches somewhere and not by creating a totally new base for it. For example we have had this calendar synchronization for two years and I believe we have used hundreds of hours for fixing bugs after its release. We should take plenty of time and look at the big picture.” – B4.

The common effect mentioned by interviewees was more working hours spent for recoding and fixing errors/bugs of solutions made with technical debt. The interviewees felt that the solutions built on top of the already bad solutions were basically already implemented wrong and fixing with patch code is just postponing the issue. They explained that technical debt lowers the quality and performance of the product in the long run and it ultimately leads to a decrease in customer satisfaction.

4.4 Future Improvements for Dealing with Technical Debt

We were able to some identify possible improvements related to dealing with technical debt in both product lines. Neither of the product lines had any specific approach for dealing with technical debt management and reduction. Interestingly, majority of the interviewees thought that they would need improvement for that. We asked the interviewees about the backlog type management where technical debt is being managed and reduced through listing all the shortcuts to a backlog and starting the reduction from there.

“Well some feature control would be good, where you can see that what has been done in a short way and other stuff. So some kind of feature management system. Usually companies have these, but smaller the company less systems.” – A2.

We were also able to identify practices related to refactoring, coding standards/guides and code reviews that were used to reduce technical debt. We noticed that the continuous delivery of new features is taking time away from refactoring. This might lead into a situation where technical debt stays in the software because the development team has to continuously implement new features. The risk is that if this debt is forgotten it might cause problems in the long-run. We think that having some refactoring time after every release to reduce the technical debt might be a good
solution, instead of just moving to the new features of next release. Also one possible solution could be to assign for example two developers to do only refactoring. Project manager of Product Line A mentioned that they are trying to improve the estimation of deadlines and include also technical debt in this.

“Lately we have been trying to include, if we have some gap in some feature and if we have to do something to that feature and we know that there is something existing in that feature, that we make some time to fix it correctly.” – A3.

We also identified practices in coding standards/guides and code reviews. We noticed that coding standards/guides are not used by everyone and they are also not updated. The lack of coding standards/guides has an effect on the consistency of the source code and especially junior coders are more exposed to write bad code that is considered as unintentional debt. Also, we identified that in the other product line the code reviews were not conducted on a regular basis, but only after a release. With code reviews it is possible to interrupt these bad solutions before they are included in the release of the software. However, we noticed that both product lines have acknowledged these problems and are currently improving them by updating the coding standards/guides and increasing code reviews. We think that improving these two aspects will have an impact on preventing unintentional technical debt.

4.5 Summary of the Findings

In Table 2 we summarize the results of this case study. We were able to identify several different causes for technical debt. These can be further divided into technical debt that is a caused with intentional decisions and technical debt is incurring unintentionally. We were also able to identify several short- and long-term effects of technical debt to a software project. As expected, the effects of technical debt seemed to be positive in a short-term, but turned negative in a long-term. Although we did not find any specific approach for managing technical debt, we were able to identify some practices for reducing technical debt.

5 Discussion and Conclusions

With this study we were able to identify empirical evidence from the relationship between technical debt causes, effects, and management. The results from both of the product lines are similar and clearly show that technical debt is appearing in both of them. McConnell [5] defined that technical debt can be divided into two different main types intentional and unintentional debt. The examples given by interviewees also show that technical debt is occurring in product either unintentionally or with intentional decisions. Based on these findings we agree with McConnell [5] that technical debt can be divided into these two main types. Moreover, based on our observations, technical debt does not seem to be only related to coding, where a coder takes a
Table 2. Summary of the findings

<table>
<thead>
<tr>
<th>RQ1: What are the causes and effects of technical debt?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intentional causes of technical debt</td>
</tr>
<tr>
<td>• Lack of time given for development</td>
</tr>
<tr>
<td>• Pressure to the development team</td>
</tr>
<tr>
<td>• Complexity of the source code</td>
</tr>
<tr>
<td>• Business decisions</td>
</tr>
<tr>
<td>- Lack of technical knowledge</td>
</tr>
<tr>
<td>- Communication challenges</td>
</tr>
<tr>
<td>Unintentional causes of technical debt</td>
</tr>
<tr>
<td>• Lack of coding standards and guides</td>
</tr>
<tr>
<td>• Junior coders</td>
</tr>
<tr>
<td>• Lack of knowledge about future changes</td>
</tr>
<tr>
<td>• Lack of documentation</td>
</tr>
<tr>
<td>Short-term effects of technical debt</td>
</tr>
<tr>
<td>• Time-to-market benefit</td>
</tr>
<tr>
<td>• Increased customer satisfaction</td>
</tr>
<tr>
<td>Long-term effects of technical debt</td>
</tr>
<tr>
<td>• Extra working hours</td>
</tr>
<tr>
<td>• Errors and bugs</td>
</tr>
<tr>
<td>• Customer unsatisfaction</td>
</tr>
<tr>
<td>• Complexity of the source code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RQ2: What management and reduction strategies/practices are being used for technical debt?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practices for reducing and preventing technical debt</td>
</tr>
<tr>
<td>• Refactoring</td>
</tr>
<tr>
<td>• Bug fixing days</td>
</tr>
<tr>
<td>• Code reviews</td>
</tr>
<tr>
<td>• Coding standards and guides</td>
</tr>
<tr>
<td>• Communication structure between business management and development team</td>
</tr>
</tbody>
</table>

shortcut in the source code. Instead, the results show that similar effects to take shortcuts were happening in different phases of the software development life cycle. Our observations suggest that similar phenomenon to take shortcuts can happen also in requirements, architecture and testing phases. We argue that technical debt should not be limited to shortcuts in source code only, but it should also include shortcuts in other phases of the software life cycle as well. Dividing technical debt into more specific subcategories may bring more clarity to the concept of technical debt over the whole software development life cycle.

The first research question was related to the causes of technical debt in the software development life cycle. The results suggest that technical debt is not necessarily caused by a single specific reason. The effect of the lack of time for the development was identified as the primary reason for technical debt in a software project. We also identified that common causes for the lack of time and pressure for the development team include business decisions. These findings are similar to other studies [7][8][9] where the researchers identified that taking technical debt is also caused by intentional decisions. We believe that the lack of time for development ultimately comes from business realities that set up the deadlines for project based on customer needs and current market situation. This makes the development team to take shortcuts to meet deadlines. However we also noticed that the both product lines had built a communication structure between the business and development departments that increased the
capability of the development team to express their opinions on technical debt decisions. Klinger et al. [8] found in their study at IBM that the cause for technical debt is the technical communication gap between business managers and the development team. Based on these observations we think that in small and middle sized companies technical debt decisions might be easier to deal with when compared to large companies where the communication structure is more complex. Another thing we also observed in these cases was that unintentional debt was mainly caused by the lack of knowledge about future and lack of coding standards/guides that will especially affect junior coders.

The second part of the first research question was the short- and long term effects of technical debt in the software development life cycle. Other studies [7][9] argue that the good thing about technical debt is the short-term effect of time-to-market. We were able to identify similar situations, where technical debt was used to get the solution out faster. In the long-term technical debt tends to have more negative effects [7][9][10][12][13]. Our observations also revealed situations where technical debt in the long-term started to generate extra working hours and errors/bugs. The effects of technical debt are mostly positive the moment you take them but might turn into problems later if they are not paid back. This could be the reason why business people think that technical debt is something that is really easy to take to meet the deadlines and just fix it later.

The second research question focused on strategies of managing and reducing technical debt. Neither of the product lines had any specific management strategy for technical debt. However both product lines were using some practices to collect the technical debt items to a backlog, but they did not have any reduction strategy for them. A portfolio management strategy proposed in other studies [15][16][17], where technical debt is stored to backlog and the development team can use that for management and reduction, would be a good option to technical debt management. This kind of a backlog strategy might be beneficial to product lines in a long-run when older technical debt is traceable, instead of forgotten. Even though neither of the product lines had any clear strategy for managing technical debt we identified several practices used for reducing it. The practices included refactoring, coding standards and guides, code reviews, and specific bug fixing days. Similar practices have been proposed also in other studies [14][18][19]. We believe that all these practices can reduce and prevent the amount of technical debt and also increase the overall quality of the product.

In conclusion, technical debt is something that companies are unable to avoid during their software development projects. In this case study technical debt was formed as a result of different management level decisions that were made during the project to reach deadlines or unknowingly due to the lack of knowledge. However, technical debt is not always a bad thing to take. Companies can use technical debt as a powerful tool to reach their customers faster to gain an edge over the competition in the market. Nevertheless, if technical debt is not paid back in time, it might generate economic consequences and quality issues to the software. To use technical debt correctly companies need to create a management plan including practices that decrease technical debt.
Acknowledgement. The authors would like to thank the company and their employees for participating to this research. This research has been carried out in Digile Need for Speed program, and it has been partially funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

References

Publication II


The Benefits and Consequences of Workarounds in Software Development Projects


© 2015, Springer International Publishing Switzerland. Reprinted with permission
The Benefits and Consequences of Workarounds in Software Development Projects

Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander

School of Business and Management, Innovation and Software, Lappeenranta University of Technology, Lappeenranta, Finland
{jesse.yli-huumo, andrey.maglyas, kari.smolander}@lut.fi

Abstract. Workarounds have existed in software from the very beginning. Being a formalized collection of knowledge rather than a physical artifact, software allows shortcuts in its development process. The shortcuts serve various purposes, like releasing a product to the market faster or postponing the solution of a problem. In this article, we present the findings of an investigation of workarounds in two software companies. Our analysis reveals that the decisions to take a workaround to resolve a technical issue are often intentional and forced by time-to-market requirements. However, the stakeholders are not always familiar with the negative consequences of taking workarounds, like additional hours, costs, and poor quality. We argue that the decision to take a workaround is often made by business managers who see short-term benefits only while developers have to deal with negative consequences in long-term.

Keywords: Workarounds · Technical debt · Case study · Software development project

1 Introduction

Starting from the release of the first software systems, the “software crisis” has been discussed widely by researchers and practitioners [1]. Many development and process management approaches have been developed to increase the productivity of programmers and to deliver products of higher quality to the market [2,3]. However, software companies are still constantly looking for new competitive advantages that would allow them to release new versions to the market faster than their competitors [4]. In the business of software, this can be achieved sometimes by taking workarounds in the development process. A workaround is a temporary solution that can be implemented in a shorter time than a proper solution, but it can also have a negative impact on the maintainability of the code base.

Workarounds are not specific for software business only, they have also been widely discussed in relation to management and public administration [5]. Although they have been applied in technological fields for a long time, workarounds have recently become known as technical debt. The concept of technical debt was introduced by Cunningham as “every minute spent on not-quite-right code counts as interest on debt. Entire engineering organizations can be brought to a stand-still under
the debt load of an unconsolidated implementation, object-oriented or otherwise” [6]. Initially defined as related to coding, technical debt has also been extended to other software development processes and artifacts. Therefore, it is common nowadays to talk about quality debt, testing debt, documentation debt and other debts [7]. To avoid misinterpretation of what technical debt is, we use the term workaround in this article and understand it similarly to [8] as “a plan or method to circumvent a problem without eliminating it.”

The aim of this study is to identify the benefits and consequences of taking workarounds, and to discuss the observations of how the decisions to take them are made in practice. The empirical data for the study has been collected from two case organizations. The first case is a middle-sized software development company with two separate product lines. The second case is a large telecommunication company currently conducting a software development project with subcontractors. To gain deeper understanding of the companies’ processes, we interviewed managers and technical specialists, which provided us with different perspectives to the studied phenomenon.

2 Background

Alter [9] defines a workaround as a goal-driven adaptation and improvisation aiming at minimizing negative consequences like anomalies or structural changes. Alter also proposes four preconditions for taking a workaround. These preconditions include (1) the existing workflow or work practice, (2) personal or organizational goal for taking a workaround, (3) an issue that requires taking a workaround to resolve or overcome it, and (4) skills to develop the workaround for this particular issue [9]. All these preconditions are not specific for software development processes only, but can be used to describe also structural changes in organizations. In software engineering, a more precise definition of a workaround is given by IBM as “some action that results in alleviating a computing or hardware problem, but which does not solve the problem” [10]. The term workaround has been widely used in information systems research when studying the use and development of various information systems like customer relationship management (CRM) and medical information systems (e.g. [11,12]). In software engineering, the term technical debt is also used to describe workarounds and other pitfalls of software development [7]. Guo and Seaman [13] discuss technical debt from the viewpoint of the portfolio theory, and conclude that up-to-date documentation is critical for the modules in which workarounds have been taken. Without documentation, changes in the module may lead to taking new workarounds. In addition, the authors confirm theoretically that several small workarounds are better to have than a big one, because it reduces the risk of breaking the system through diversification [13].

Since workarounds are often associated with making some changes in the code base, they can be described as “code smells” [14]. The perceived quality of code varies a lot among developers, and their evaluation of the code quality is subjective without uniform criteria [14]. This has a negative effect on code maintainability, as software developers’ viewpoints on what tricks are allowed to be used in the code differ.
Tom et al. [7] describe the two primary interrelated reasons for taking workarounds as pragmatism and prioritization. Pragmatism relates to setting goals, like releasing a minimal viable product (MVP) to the market quickly rather than developing it until its quality is high. Prioritization is about implementing the most critical tasks first, even if the overall product quality remains low. In this regard, prioritization involves pragmatism unconsciously in deciding the limitations and constraints of the project. Since prioritization is a collaborative process that involves technical and business people, the priorities of different parties may differ. Nagarjuna and Mamidenna [15] have studied engineering and business students. According to their results, engineering students have a tendency to perfectionism. Although the authors studied students only, this conclusion may also hold with real practitioners. For example, business-minded people, e.g. managers, aim at developing a minimum viable product in a shorter time with an appropriate level of quality, while engineers often aim at developing a cutting-edge solution [16,17].

When releasing a product to the market, a company aims at satisfying customers' needs [18]. Therefore, the perceived product quality is essential for product success. Dzida et al. [19] have identified seven dimensions of perceived quality as (1) self-descriptiveness, (2) user control, (3) ease of learning, (4) problem adequate usability, (5) correspondence with user expectations, (6) flexibility in task handling, and (7) fault tolerance. Out of these seven dimensions, only the last is directly related to the technical problems in the code base that could be raised because of the taken workarounds. The other six dimensions are primarily associated with usability and user experience. The perceived quality of web applications can be measured by four dimensions: technical adequacy, specific content, content quality, and appearance [20]. Only the first dimension of technical adequacy, which includes security, reliability and availability can be affected directly by taking workarounds in the code base. In both models of perceived quality, the quality of the code base itself plays only a partial role in how the user perceives the product. Therefore, technical excellence is only one dimension of how the product is perceived by customers.

Overall, the research on workarounds covers different aspects from organizational workarounds to shortcuts in the source code. In this study we focus on the latter case and contribute to the theory of workarounds, as e.g. Alter [9] states that workarounds are understudied and undertheorized. In this regard, we contribute to the theory of workarounds by providing empirical results of real practice of workarounds in the industry. Alter’s theory of workarounds is a model consisting of seven layers: (1) intentions, goals, interest; (2) structure; (3) perceived need for a workaround; (4) identification of possible workarounds; (5) selection of workarounds to pursue, if any; (6) development and execution of the workaround; and (7) consequences [9]. In this study, we focus particularly on the first and last layers in order to understand how the idea of taking a workaround is born and the workaround then taken, and what benefits and consequences workarounds bring to the organization.
3 Research Methodology

Interpretive case study was selected as the research method for the study. We followed the guidelines of Klein and Myers [21] for conducting interpretive case studies, because the selected topic focuses on understanding of social processes and interactions between development and management teams within a company that lead to making the decision about taking a workaround. To make a valuable contribution to theory and practice, interpretive case studies should be carried out and written up carefully, and therefore we adopted the principles for reporting interpretive case studies presented by Walsham [22]. These principles enumerate a minimum information required to be reported such as “details of the research sites chosen, the reasons for this choice, the number of people interviewed, their hierarchical positions, what other sources of data were used, and over what period the study was conducted” [22].

3.1 Case A

Company A is a middle-sized software company that offers SaaS business solutions. It has three product lines that are managed independently. We selected two of the product lines for this study. The first product line provides a financial management solution as a cloud service that has more than 10 000 customers. The second product line is a SaaS-based project management solution for multi-organization projects. The solution is used by around 1000 companies worldwide. Both development teams use agile methodologies and especially practices from Scrum. The development teams of the product lines are rather small and consist of 13 and 18 employees, respectively.

3.2 Case B

Company B is a telecommunications company offering services for communication and entertainment. The company employs around 4200 people and has about 2.3 million customers. We chose one of the projects conducted by Company B for this study. Five subcontractor companies have participated in this project, but the project has been mainly developed by Company C, which is a middle-sized development company. The project started in 2007 and is still running today. It has over one million lines of code and it has been integrated to over 70 background systems. The goal of the project is to create a self-service channel for customers and switch manual work to automated processes inside the system. The organization had used Scrum during the first years of the project, but has currently moved to the use of Kanban.

3.3 Selection of Companies, Data Collection, and Data Analysis

The selection of the companies for this study was primarily dictated by a list of partners (in total 30) in a research project. Out of several potential candidates for the study, the selected companies were chosen on the basis of various reasons. The first reason for the selection of Case A was related to the phase of the lifecycle of
the company. Company A has been growing through mergers and acquisitions of several smaller companies. In addition, the company itself is nowadays a part of a larger international enterprise. Due to these mergers and acquisitions, the company combines several product lines. We assumed that studying workarounds in the two product lines of this company should produce insightful details of how two product lines and teams coming from different backgrounds and cultures, but currently sharing the same environment, deal with workarounds. This also allowed us to constantly compare and cross check the information collected by interviews in two product lines. The second reason for the selection of the company was related to the type of products the company develops. Both product lines are SaaS products that share such characteristics as a common set of features for all users and short release cycles. These characteristics, together with the increasing number of SaaS products attracted our attention to their connection to taking workarounds in the development process. The selection of case B was primarily done based on the company size and the interesting nature of the project they were working with. We assumed that the development processes in large organizations are more mature than in smaller organizations, and therefore there should be less workarounds.

In both cases we focused on understanding why workarounds had been taken and what positive and negative effects they had. We conducted semi-structured interviews with 17 representatives related to the cases during February-June 2014. The positions of the interviewees are listed in Table 1. All interviews were sound-recorded and later transcribed. The interviews lasted from 25 to 105 minutes with an average of 50 minutes.

<table>
<thead>
<tr>
<th>ID</th>
<th>Company</th>
<th>Product line</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>A</td>
<td>a</td>
<td>Software architect</td>
</tr>
<tr>
<td>I2</td>
<td>A</td>
<td>a</td>
<td>Software designer</td>
</tr>
<tr>
<td>I3</td>
<td>A</td>
<td>a</td>
<td>Project manager</td>
</tr>
<tr>
<td>I4</td>
<td>A</td>
<td>a</td>
<td>Software test engineer</td>
</tr>
<tr>
<td>I5</td>
<td>A</td>
<td>a</td>
<td>Production director</td>
</tr>
<tr>
<td>I6</td>
<td>A</td>
<td>b</td>
<td>Software architect</td>
</tr>
<tr>
<td>I7</td>
<td>A</td>
<td>b</td>
<td>Software developer</td>
</tr>
<tr>
<td>I8</td>
<td>A</td>
<td>b</td>
<td>Product line manager</td>
</tr>
<tr>
<td>I9</td>
<td>A</td>
<td>b</td>
<td>Software test engineer</td>
</tr>
<tr>
<td>I10</td>
<td>A</td>
<td>b</td>
<td>Software architect</td>
</tr>
<tr>
<td>I11</td>
<td>A</td>
<td>b</td>
<td>Software developer</td>
</tr>
<tr>
<td>I12</td>
<td>A</td>
<td>b</td>
<td>UI designer</td>
</tr>
<tr>
<td>I13</td>
<td>B</td>
<td></td>
<td>Software architect</td>
</tr>
<tr>
<td>I14</td>
<td>B</td>
<td></td>
<td>Project owner</td>
</tr>
<tr>
<td>I15</td>
<td>B</td>
<td></td>
<td>Project owner</td>
</tr>
<tr>
<td>I16</td>
<td>C</td>
<td></td>
<td>Senior consultant</td>
</tr>
<tr>
<td>I17</td>
<td>C</td>
<td></td>
<td>Software architect</td>
</tr>
</tbody>
</table>

Table 1. Roles of the interviewees
The data analysis was done by identifying categories related to workarounds. We used an iterative approach of data collection and analysis, and coded the data using a procedure similar to open coding in the grounded theory [23]. The interview transcripts were read and their workaround-related parts categorized into labelled concepts. These initial concepts guided us to an explanation of how and why workarounds are taken in practice.

4 Findings and Results

We identified seven scenarios related to taking a workaround during software development in the studied cases. Below, we explain the context and environment in which the workarounds were taken with the reasons, benefits, and consequences.

4.1 Scenario 1: Upcoming Deadline

In Case B the company decided to develop a new feature to their system. The development task was given to a team that consisted of a few junior coders only. The management asked for a preliminary timetable from the development team in order to create a marketing campaign for the feature. The development team gave an estimate of the development time, and the marketing team started to plan the campaign. However, it was discovered later on that the development team had estimated the release date wrong, and they would not be able to deliver the feature before the deadline. At this point the company did not have the option to postpone the release date anymore, and they decided to implement heavy workarounds to the feature in order to get it released in time.

“A media campaign was designed, radio commercials were starting, and commercials for magazines were ordered. So at that point there were just no more options. There would have been so much business damage to us.” – I15.

In this scenario the feature itself was not very important for the company’s overall strategy. However, the company decided to use a workaround to reach the given deadline in order to meet the promised release date. With taking the workaround, the company was able to release the feature in time, and therefore damage to the company’s reputation by releasing late was prevented. If the company had announced that the feature release will be postponed, it could have had a significant effect on the company’s reputation and customer satisfaction.

The workaround for releasing the feature in time had some consequences. The released feature was taken in production unfinished and unstable. The feature itself looked the same in the user interface as it would have looked when done properly. However, the code base was unfinished and consisted of several critical components. This forced the company to fix and refactor the feature right after the release, which required extra working hours and costs for the project, in order to fix critical errors and to be able to develop the product further.
“Well, of course everything was working from the outside, but we knew that there are scary things inside. However, we appointed developers immediately to fix them after the release.” – I15.

4.2 Scenario 2: Complex Part of the Code Base

In case A (product line a), the software designer described a scenario that often required a workaround during the development. He explained a situation where fixing a feature demanded a lot of time due to the complexity of the code base. The complexity of the code base meant that some parts of the features were developed with bad solutions and architecture. Therefore, refactoring them was challenging and risky. Since the deadlines were strict, there was not enough time to analyze the whole code base. This was the reason why it was faster just to implement a simple workaround instead of fixing the bigger problem of the feature. Also the risk that the code base might not work anymore due to changes in a complex part of the code base was seen as a reason why it was safer to implement a workaround.

“Yeah, we often have to do some kind of a fix because it is complex and we can’t go any further. So we need to release a hotfix pretty soon and we don’t have enough time to make it work as it should be because of time.” – I2.

The benefit that the company gained from taking a workaround in this scenario was customer satisfaction, because the fix was released earlier. Also sometimes even the development team was happy for getting rid of the problem fast and being able to move on to other tasks.

“In the short term, the customer will be happy because the problem is fixed. Sometimes the development team is also happy because we can start different kind of work and tasks. So sometimes for us it is okay to have shortcuts.” – I2.

According to a company survey on user satisfaction, the users were happy with this workaround strategy and were not eager to switch to another solution by a competitor. Therefore, the company’s internal policy allowed taking workarounds and fixing them later.

However, these workarounds started to produce negative consequences later on. Sometimes the workarounds created temporary system breakdowns or slowness that needed to be fixed with other fast workarounds. This required many extra working hours for fixing and refactoring that were not planned in the beginning, and in some cases even a completed rewrite of the feature was required.

“For example let’s say we do one fix and take a shortcut and then after a while, like after a month or two months, something else comes up and we need to do another hotfix because of this previous fix. When time progresses, that feature needs to be redone because it is getting out of date or other fixes are getting slower or whatever.” – I2.
4.3 Scenario 3: Unpleasant Work

Case B has a long history, and the system has been developed for many years. During the project, the code base has grown to be large and complex, and there are features in the code base that have been implemented either properly or with workarounds. When the code base grows and becomes inconsistent, development becomes much harder. This was the situation in case B, and the senior consultant in the project felt that this generated serious effects on the developers’ mindset towards the code base.

“The pattern that the developers were talking about was this thing called princess-driven development. In other words, “This code is so ugly, I don’t want to do it like this. I will implement something else.”” – I16.

In this scenario some of the developers experienced that the old part of the code base that had been developed with complex solutions was not pleasant to work with. Instead of refactoring the entire code base, it was just easier to implement a workaround. The only benefit that this type of workarounds had was the faster release of the solution. The developer could move to other tasks sooner and start to implement features with more interest to him. However, when workarounds were implemented and not fixed afterwards, it turned the code base difficult to understand, especially for new developers.

“This led to a situation where you don’t clean up the old code and you implement something else next to it. When a new guy comes after a year and looks at this and sees “ok, well it has been done like this in here, and then this is implemented like this…” and it says nowhere how it should be done, and what our common way of implementing things is.” – I16.

This incomprehensible code base increased the lack of interest in developing solutions properly and therefore increased the amount of workarounds taken because it did not require much effort.

4.4 Scenario 4: Significant Economic Benefits

A large number of workarounds were taken at the early stages of case B to create significant internal financial saving for the company. The reason for this was that the company would be able to change their manual work to automated processes within the system. With the change, the company would be able to cut down personnel costs. A software consultant in company C estimated that by creating the feature, company B would save a significant amount of money.

“Roughly estimating, if the levels are correct, when we got a certain order type, like for example closing a subscription to self-service and automation, it started to save 30 000 euros a month to company B.” – I16.

When it was noticed that the savings were so significant, it was understood that the features had to be released as soon as possible. This created pressure for the development team to get the features released. The development team was able to release the
features really quickly, but it required some workarounds. However, with the workarounds the company started to make significant savings early. One of the product owners considered the decision to take workarounds as a smart one, even if it required extra effort from the company.

“If we had started to build this project really well and with a really fancy architecture, it might be that we might have been cut out of funding before it got to the production and we wouldn’t have gotten anything done. It has paid back a multiple amount of money, probably tens of times.” – I15.

The decision to take workarounds resulted later in extra working hours and slower development. Another negative consequence of these workarounds was difficulties in further development. Whenever the development team wanted to implement something to a certain part of the code base, it broke down something else.

4.5 Scenario 5: Unnecessary Work

In case A (product line a) we found that the company did not document the workarounds, as they were considered as temporary solutions. When a developer decided to take a workaround in a certain task, it was not documented. The developers relied on the information of workarounds as they remembered it and paid no attention to documentation. This way the developers could work faster, and large amount of documentation work would not slow them down.

The consequences of not documenting workarounds were especially well observed when new developers joined the team. When a new developer started to work with the code base that included workarounds, it was challenging because the code base was not self-documenting and actually no documentation existed.

“At least it affects situations when someone new comes to work for us. There has to be a place where people can get answers without asking, if they will have to work alone someday.” – I10.

The lack of documentation affected the future development a lot. When the developers had no documentation available about workarounds, they could just copy/paste the old code because it worked. In the situation where the old code was done with workarounds, the workaround code started to accumulate and show in the overall quality of the code base.

“They should be listed somewhere and we should be fixing them all the time, but we would need time for it. The reason is that every bad solution we implement intentionally, will be used also as copy/paste code and that is really bad for the future. I think that bad solutions will multiply in time and spread to other parts.” – I10.

4.6 Scenario 6: Outdated Version of Components

In case A (product line b), the workarounds had effects on compatibility with other systems. A software designer mentioned compatibility problems in integration with
different versions of the database. The current version of the database server used in the company was from 2012, but due to the workarounds taken previously, the database server worked in the compatibility mode with the 2008 version and in some cases even with the 2005 version.

“Well, when you think about it, there are compatibility problems between different versions. Now we have the 2012 version of the Microsoft SQL server, but it is up and running in compatibility mode with the version of 2005. So we can’t use new commands because of this.” – I2.

Due to this workaround, new commands were not available and the developers had to implement low-level features already implemented in the newer version of the database server. For example, some commands could be run in the default server mode but not in the compatibility mode, and the team had to implement the database features already available in the newer version by themselves. If this had been done properly from the beginning and kept up to date continuously, these problems could have been avoided. Now the team had to put additional efforts into implementing the middleware between the product and the database.

4.7 Scenario 7: Low Priority Features

In case A (product line b), the development team intentionally did not implement some of the features properly, as they were requested by a couple of customers only. The company put only minimal effort to this type of features because it was not planned to scale the feature to all customers.

“There might be some cases where there is a certain need coming from a customer that is really valuable to that customer, but it is not a scalable feature, so it is not valuable to any other customer. So in that kind of cases we can only put minimum effort to that feature, because it is not a scalable feature. Customers are happy if they just get what they want.” – I3.

A project manager in company A explained that the company evaluates the value of every feature and then decides what the planned effort for them is. If the business value of the feature is low for the company, it just simply gets done as easily as possible, with workarounds.

“Well, everything is based on the value of the feature. If the value of the feature is low from the business point of view, you always have to weigh the time used by the coder. Let’s say that implementing a feature takes one week, and we know that we can go with that solution like one or two years forward, it could be a good decision now, because we don’t have those two or three weeks to implement that.” – I3.

However, the company also faced situations where some of these low value features were so successful that the decision to release them to all customers was made later. Because these features were developed with a minimum effort, this led to a situation where the features had to be refactored and developed more as scalable when the number of users started to grow.
4.8 Summary of Scenarios

In Table 2 we summarize the scenarios observed in the studied cases with identified categories of reasons for workarounds, their benefits and consequences. The main reason for taking a workaround was related to the challenge of meeting a deadline (category: time pressure, Scenarios 1-5). The release planning of features defined by business managers was so important that the developers had to take workarounds in order to reach deadlines. Time pressure from business managers also affected architectural decisions because developers had no time to change the selected software components (Scenario 6) even if it would be beneficial for the development. The other reason was the complicated code base, which increased the number of workarounds to be taken, as the workarounds were easier to implement than refactoring the complex code. In Scenario 7 prioritization of the features based on their business values led to workarounds in implementing low priority features.

Table 2. Summary of scenarios

<table>
<thead>
<tr>
<th>Scenario #</th>
<th>Case</th>
<th>Reason for workaround</th>
<th>Benefits of workaround</th>
<th>Consequences of workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>Time pressure.</td>
<td>Time-to-market, Company reputation, Increased customer satisfaction.</td>
<td>Decreased code maintainability, Extra working hours, Extra costs.</td>
</tr>
<tr>
<td>2</td>
<td>A (a)</td>
<td>Time pressure, Complicated code base.</td>
<td>Time-to-market, Increased customer satisfaction.</td>
<td>Decreased code maintainability, Extra working hours, Major refactoring.</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Time pressure, Complicated code base.</td>
<td>Time-to-market.</td>
<td>Decreased code maintainability, Lack of motivation to work with the code base.</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>Time pressure.</td>
<td>Time-to-market, Significant financial savings due to early release.</td>
<td>Decreased code maintainability, Extra working hours.</td>
</tr>
<tr>
<td>5</td>
<td>A (a)</td>
<td>Time pressure.</td>
<td>Time-to-market, Increased speed of development.</td>
<td>Decreased code maintainability, Increased time for newcomers to start.</td>
</tr>
<tr>
<td>6</td>
<td>A (b)</td>
<td>No time for changing the selected software components.</td>
<td>Time-to-market.</td>
<td>Outdated software components, Lack of new features available in newer versions of components, Decreased code maintainability.</td>
</tr>
<tr>
<td>7</td>
<td>A (b)</td>
<td>Prioritization of features based on their business value.</td>
<td>Increased speed of developing high priority features.</td>
<td>Decreased code maintainability when scaling the feature.</td>
</tr>
</tbody>
</table>
The primary benefit of workarounds observed in the scenarios was *time-to-market* (Scenario 1-6). Taking workarounds was helpful for companies to deliver the needed features in time, which resulted in *increasing customer satisfaction* and *saving company reputation*. In Scenario 4 implementing the feature was critically important for the company because its implementation instantly allowed getting *significant financial savings* by using the feature in production. Workarounds in processes like documentation in Scenario 5 allow allocating more time for other processes, e.g., *increased speed of development* due to lack of documentation. Similarly, Scenario 7 is an example of *increased speed of developing high priority features* due to workarounds made in lower priority features.

However, when the workarounds gave the companies the ability to deliver the features in time and development seemed faster, they also had negative consequences. In all scenarios the workarounds resulted in *decreased code maintainability* and consequently *extra working hours* and *extra costs*. In Scenario 2 this required *major refactoring* of the code base. It seems that other scenarios will end up with major refactoring too because workarounds lead to *lack of motivation to work with the code base* (Scenario 3). In Scenario 5, *increased speed of development* led to *increased time for newcomers to start working on the project* due to lack of documentation. Being under pressure of *time-to-market* developers had to use *outdated software components* and introduce new workarounds due to *lack of new features available in newer versions of components*. The lack of time for architectural changes prevented the company from reducing the effects of accumulated workarounds and *decreased code maintainability* (Scenario 6). Focusing on high priority features (Scenario 7) led to the situation when workarounds in lower priority features were implemented on top of each other without any documentation. Finally it led to *decreased code maintainability when scaling the feature*, when the feature was so valued by customers that its priority had to be increased.

5 Discussion

The results of the study of the two cases show that taking workarounds is a daily practice in software development. This does not necessarily lead to business disruption, but it has negative consequences that a company should be aware of. Lim et al. [24] report that developers always try to make the best decisions based on the information, knowledge, and experience they have, but these decisions can lead to workarounds quickly and unintentionally. In the two companies in our study the decision to take a workaround was often intentional. This decision was often made by business managers who understood the negative consequences, but could underestimate their long-term impact. Regardless of their awareness of the consequences, they made the decision intentionally to benefit from releasing a product to the market faster. This was particularly done to satisfy customers, save company reputation, and gain an edge over competitors.

Kekre et al. [25] have developed a model of seven drivers of customer satisfaction for software products. The authors conclude that capability and usability are the main
drivers for customer satisfaction. Capability presents the product functionality in terms of its key features. Usability is a multidimensional driver itself, but it is not related to the product code base where workarounds have been taken. In this study, we identified decreased code maintainability as a major consequence of taking workarounds that has limited direct impact to seven drivers of customer satisfaction but has impact to the company ability to maintain releasing new versions of the product with increasing time and costs to maintain the code base in long term.

The negative consequences of workarounds have been already identified separately. For example, Li and Shatnawi [26] have studied the relationships between workarounds associated with “code smells” and class error probability. They revealed that refactoring a bad code is difficult after release, and associated with introducing new errors to the code. In addition, the authors argue that “code smells” should be constantly identified to find problematic pieces of code and refactored. In the studied companies, only one scenario went through major refactoring and we expect to other scenarios will also end up in refactoring the code base. Other consequences like a lack of documentation were already identified back in 1979 [27], but this problem still remains like in Scenario 5. In this regard, we do not consider the identified negative consequences of workarounds as a significantly new contribution. The value of this list is in the consideration of the impact of workarounds to the maintainability of the code base. Developers are aware of these consequences, and therefore our primary aim is to attract the attention of decision makers who see the immediate benefits of workarounds, but do not fully understand their drawbacks. It is important to release a product quickly to the market, but it is also important to understand the accumulation of workarounds and the related waste of time and resources in the future.

Alter [9] provides an integrative view on workarounds and states that the theory of workarounds will evolve over time. The study contributes to the theory by providing an explanation for the intentions of taking workarounds and their consequences. Alter believes that the theory of workarounds could be used in making “more realistic assumptions for systems analysis and design” [9]. Although the theory can be used this way, we see the underlying problem of workarounds is in misunderstanding and underestimation of their impact by decision makers. By pointing out that workarounds have consequences and these consequences impact on how further releases should be planned, we bridge the gap and provide a communication tool for developers and managers to find a balance between maintaining the code base and releasing a product to the market.

With this study we contribute to the theory of workarounds [9] in the context of software development organizations that take shortcuts in the code base. According to Gregor’s taxonomy of theories in information systems research [28], Alter’s theory of workarounds can be considered as an explanation theory that defines the phenomenon, describes and explains it, but does not make attempts to specify hypotheses for prediction. The present study contributes to the theory by providing an empirical investigation of the phenomenon in a real environment, and extends the scope of validity of the theory [29]. Rather than making a prediction on the long-term impact of workarounds to the business success, we explain the reasons behind taking workarounds and the ability of workarounds to accumulate over time.
The study has also limitations. The selection of the case organizations was partially limited to project partners. However, out of several organizations, the selected organizations could meet all our criteria. In addition, as we had two unrelated cases to study, we were not limited to one team only and could interview and compare the results from two cases to avoid the bias of interviewing only one team with its own experiences and culture. Both companies develop customer SaaS kind of software, therefore the transferability of the results is primary related to similar companies while it requires additional study to investigate workarounds in other types of products like embedded systems.

6 Conclusion

In this study, we explored how the decisions of taking workarounds are made in the organization, and used the qualitative case study approach as recommended by Klein and Myers [21]. The use of the interpretive case study allowed us to investigate social processes in the organizations in an attempt to understand how business and technical professionals communicate with each other regarding taking workarounds, and how they consider their benefits and consequences. We found that business people often deal with the benefit of time-to-market only and therefore can underestimate the negative consequences of workarounds, like decreased maintainability of the code base. In contrast, engineering people have to deal with all consequences and therefore they hesitate to take workarounds. However, they are often under the pressure from the business and have little power to make the final decision. We observed that the decisions to take workarounds are often made intentionally but the consequences of these decisions can be underestimated by the business people due to the lack of technical knowledge.

Acknowledgements. We would like to thank the companies for being a valuable source of information and all interviewees who found time to share their experiences. This research has been carried out in Digile Need for Speed program, and funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

References

Publication III


© 2016, Springer International Publishing Switzerland. Reprinted with permission
Chapter 15
The Effects of Software Process Evolution to Technical Debt—Perceptions from Three Large Software Projects

Jesse Yli-Huumo, Andrey Maglyas and Kari Smolander

Abstract This chapter describes a qualitative study with the goal to explore and understand how software process evolution affects technical debt. We investigated three large software development projects with a long development history with the aim to understand how software processes had evolved during the life cycle and how this evolution affected technical debt. We observed how companies had changed their software processes as well as the reasons, benefits, and consequences of these changes on technical debt. The main driving force for the software process evolution was business pressure from management to increase productivity and become cost-efficient. However, these changes were also the source of technical debt. The results show that software process evolution has a clear effect to technical debt. Software process evolution can be used to decrease technical debt by adopting new methods, tools, and techniques. However, software process evolution includes several challenges. These challenges have a possibility to decrease the productivity and quality of new software processes and technical debt might increase.

15.1 Introduction

The software industry struggles with increasing competition and time-to-market requirements in delivering new solutions to customers. Companies must be able to deliver their solutions faster than competitors to receive a share of the market [6]. In order to be fast, companies should enhance their software development processes and practices to achieve the best ways to produce quality software on time, within..
budget, and for the right market [27]. However, software development processes are not easy to change [4] or manage (see also Chap. 10 and Chap. 9 in this volume). If the new processes do not align to the organization and its way of working, serious consequences may follow. Decreasing quality and productivity can be the result of new software development processes if they do not align with the company’s way of working. When omitted quality and productivity issues start to have effect on a software development project, it can be a sign of “technical debt.”

The technical debt metaphor is related to shortcuts and workarounds in meeting urgent demands [35]. Implementing shortcuts to the system architecture incur “debt” that must be eventually paid back. If this debt is not properly managed, it might accumulate as “interest,” affecting the overall quality of the developed software systems [42, 43]. Although technical debt has negative consequences in a long term, it can be used as a competitive advantage in a short term [26]. Time-to-market and constant customer feedback through releasing software faster than competitors allow companies to gain a bigger market share [26].

This chapter describes a qualitative study that had a purpose to explore and understand how software process evolution affects technical debt. We investigated three large software development projects with a long development history with the aim to understand how software processes had evolved during the life cycle and how this evolution affected technical debt. We observed how companies had changed their software processes as well as the reasons, benefits, and consequences of these changes on the technical debt.

The rest of the chapter is organized as follows. Section 15.2 provides the background and the research process related to this research. Section 15.3 introduces the results analyzed from the gathered data. In Sect. 15.4, we discuss about the results. Section 15.5 concludes the paper, and Sect. 15.6 provides further reading related to the chapter’s topic.

### 15.2 Background and Context

The term “technical debt” was first introduced in 1992 by Ward Cunningham as a situation where a long-term code quality is traded for a short-term gain [13]. Technical debt can be compared to finance debt [1]. Similar to finance debt, technical debt incurs interest, which come in the form of the extra effort that we have to pay back in the future [1]. Often technical debt is related to the source code of the software, where a shortcut or a workaround is taken in order to save time. However, taking shortcuts and workarounds can happen in multiple stages of software development life cycle [38]. In the requirements phase, lack of documentation or lack of requirements can cause requirements debt [29, 44]. Architectural flaws in the design phase can also increase design debt [3, 38] and structural debt [44]. In a testing environment workarounds in running and writing test cases can also incur test debt [44] and automation debt [9].

In addition, technical debt is not always caused by intentional decisions to gain short-term advantages. Technical debt can be divided into two main categories:
intentional and unintentional technical debt [28]. Intentional technical debt incurs when company makes a strategic decision to cut down; for example, the feature quality in order to be able to release the product on time to customer. Unintentional technical debt forms unknowingly, when for example, a junior coder writes lower quality code that needs to be refactored later.

Companies change and evolve their software processes to improve software quality and reliability, employee and customer satisfaction, return on investment and time-to-market [12, 27]. Software process improvement (SPI) is used to improve productivity, quality, schedule, customer satisfaction, and return on investment [2, 17, 18, 23]. Overall, the current research on software process evolution consists of studies about the benefits and consequences of software process improvement. However, their relationship to technical debt has not been studied. In this study, we focus on contributing to the research of technical debt by studying the reasons, benefits, and consequences of software process evolution and their relationship on taking shortcuts and workarounds and provide empirical results by studying three real software projects.

Case study was selected as the research method for the study. Case study is a way to investigate an empirical topic by following a set of pre-specified procedures [41]. According to Verner et al. [40], case studies provide “a systematic way of looking at events, collecting data, analysing information, and reporting the results.” Case study method involves an in-depth examination of a single case or a multiple number of cases [40]. According to Yin, it “investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident [41].” We followed the guidelines of Yin [41] to conduct the case study process in this research. The process consists five different stages [41]:

1. Designing the case study
2. Preparing for data collection
3. Collecting the evidence
4. Analysing the case study evidence
5. Reporting the case study

The first stage of the case study process was to design and identify the research strategy for the case study. In this study, our focus was on the software process evolution and technical debt. The goal was to understand how software processes had evolved during the life cycle and how this affected to technical debt. We decided to use the exploratory case study method with semi-structured interviews [31] that are frequently used as a data collection technique in software engineering studies [20].

The second stage for the case study process was to prepare the data collection. In this stage, we designed the procedures to conduct interviews and contacted the key persons in the chosen cases. The interviews were designed to investigate the reasons, benefits, and consequences of software process evolution to technical debt, rather than what are the qualities of technical debt in the source code. Therefore, we arranged interviews with people from multiple different backgrounds (business and technical). Since this study was a part of a bigger research program, the selection of the companies was primarily dictated by the list of partners. The selected cases
for this study were three large companies in the data communications industry. The primary reason for the selection of these three specific cases was their long product history. We believed that a product with long development history would include more software process changes and it would provide us with more empirical data about the research topic.

**Case A** is a software development project that develops a self-service channel for customers and automated processes for the company employees. The company is a large telecommunications company and employs currently about 4,200 people and has about 2.3 million customers. The company expects to make significant economic savings with the project. The project was challenging because the company had multiple background systems in use and the goal was to combine these all in a single system. The project started in the beginning of 2007 with developers from a middle-sized software company and other external consultants related to the system integration. The system has been developed since then and the project is still running today. The system is being further developed with additional features and it currently has around 1 million lines of code and integration to over 70 background systems.

**Case B** is a software product developed for controlling and monitoring telecommunication systems. A large data networking and telecommunications equipment company have conducted the project. The company has around 58,000 employees and the organization size for the studied project is around 1,500 people. More than 320 customers are using the product. The project was started in 1992 and is still running today with multiple yearly releases. The product has currently around 50 million lines of code and it has faced several technology and operational system transitions during its life cycle.

**Case C** is a software development project conducted by a large company that provides communication technology and services. The company has currently around 1,15,000 employees and customers in over 180 countries. The goal of the project was to develop a product that connects different networks together. The development of the product started in the beginning of 2000 and since then it has been developed further with new features brought or requested by customers. However, at the moment the product is facing the end of its life cycle and includes currently mainly maintenance work.

Table 15.1 shows the overview of the cases. Even though all of the case companies are working in the same industry area, the Case A, as a smaller company, is somewhat different compared to the other two cases. Also the development of Case A started several years later compared to the Cases B and C.

The third stage of the case study process was to collect the evidence from the selected case companies. We conducted 17 semi-structured interviews with the snowballing technique [7] during March–October 2014. The interviews started from our key contacts from each of the selected cases and the next interviewees were referrals from the previous ones. We were able to interview people from various organizational positions and investigate the research topic from the viewpoint of software developers to managers. The interviews lasted from 31 to 105 min with an average of about 50 min. Table 15.2 presents the roles of the interviewees in this study.
Table 15.1 Overview of the selected cases

<table>
<thead>
<tr>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case A</td>
</tr>
<tr>
<td>Industry sector</td>
</tr>
<tr>
<td>Company profile</td>
</tr>
<tr>
<td>Company size</td>
</tr>
<tr>
<td>Employees</td>
</tr>
<tr>
<td>Case project</td>
</tr>
<tr>
<td>Project end</td>
</tr>
<tr>
<td>Lines of code</td>
</tr>
</tbody>
</table>

Table 15.2 The roles of interviewees

<table>
<thead>
<tr>
<th>ID</th>
<th>Case</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A</td>
<td>Software architect</td>
</tr>
<tr>
<td>A2</td>
<td>A</td>
<td>Project owner</td>
</tr>
<tr>
<td>A3</td>
<td>A</td>
<td>Project owner</td>
</tr>
<tr>
<td>A4</td>
<td>A</td>
<td>Senior software consultant</td>
</tr>
<tr>
<td>A5</td>
<td>A</td>
<td>Software architect</td>
</tr>
<tr>
<td>B1</td>
<td>B</td>
<td>Technical project manager</td>
</tr>
<tr>
<td>B2</td>
<td>B</td>
<td>Software architect</td>
</tr>
<tr>
<td>B3</td>
<td>B</td>
<td>Software architect</td>
</tr>
<tr>
<td>B4</td>
<td>B</td>
<td>Manager of R&amp;D department</td>
</tr>
<tr>
<td>B5</td>
<td>B</td>
<td>Project manager</td>
</tr>
<tr>
<td>C1</td>
<td>C</td>
<td>Line manager</td>
</tr>
<tr>
<td>C2</td>
<td>C</td>
<td>Manager of release verification department</td>
</tr>
<tr>
<td>C3</td>
<td>C</td>
<td>Software testing</td>
</tr>
<tr>
<td>C4</td>
<td>C</td>
<td>Manager of maintenance department</td>
</tr>
<tr>
<td>C5</td>
<td>C</td>
<td>Software developer/technical coach</td>
</tr>
<tr>
<td>C6</td>
<td>C</td>
<td>Innovation and business architect</td>
</tr>
<tr>
<td>C7</td>
<td>C</td>
<td>Software developer</td>
</tr>
</tbody>
</table>

The fourth stage of the case study process was to analyse the collected data. The total amount of transcribed qualitative data for analysis was over 150 pages. After the data collection, data was analysed with a special tool for qualitative data.
analysis (Atlas.ti\textsuperscript{1}). In the analysis process, we used similar procedure to open coding in grounded theory [11]. First, we read and examined through all of the transcribed interviews and related content. During the analysis, we categorized the parts related to software process evolution and technical debt into labelled concepts. Then, these data labels were grouped and linked together and we formed categories and subcategories. We used these categories to analyse the results.

15.3 Identified Scenarios of Software Process Evolution and Technical Debt

The last stage of the case study process was to report the gathered and analysed results. During the interviews our focus was to gather information about the history of the software process evolution in the case companies. Our goal was to identify situations during the interviews where companies had to change the software process for some specific reason and learn the reasons and effects behind it. We were able to identify scenarios of software process evolution in the studied cases. The interviewees described us real situations from the cases that had happened in the past or very recently. In this section, we present five scenarios and explain the context and environment in which the software process change happened and how it affected to the technical debt.

15.3.1 Scenario 1: A Need for More Frequent Releases

In Case C, the company made a decision to adopt agile methods after developing the product for several years with the waterfall model. The management of company felt that waterfall development model was not suitable to have frequent releases to customers. The customers required releases monthly, but the company was not able to accomplish that. The waterfall model was too rigid for inter-departmental cooperation and caused delays. The company could not organize the process of switching to agile methodologies on their own, so the decision was to hire a consultant team from another company to train people in agile software development processes and practices. The software process change was challenging during the first year and the project encountered several problems. The agile methodologies included a set of new practices and it was challenging for teams in the project to learn them. The change of methodology also encountered some resistance in the beginning and everyone in the project was not excited about the new ways of working. This resulted in a situation where some of the project members were using the waterfall model, while others were using agile methods.

\textsuperscript{1}Available from: http://atlasti.com/.
Statement (C4). People reacted differently, some people thought it is good and some were resisting. So at that point also you have to respect the people who are resisting. The most important thing is that there is need for everyone’s contribution. So the thing that you cannot force every person into to the same mold. Some people require more time for change. There have been cases where some people stayed in their offices and did systematic work even though other people were in different rooms working together.

In the beginning of the adoption, the productivity of development teams dropped. Previously, with the waterfall model people were assigned to certain jobs based on their competences. The change to agile methods created teams that were focusing on bigger components of the product and people needed to educate themselves to these new components they had never worked before.

Statement (C7). The productivity dropped for a while during the change. Before the change there was more focus on certain sections that there was competence gathered to certain section and it was really specific. So at the same time when this change was done to this agile way of working, there was a request or actually a demand that teams could focus on bigger sections instead of specific ones. So the effect here was that we needed to learn new things and sections we had not learned or touched before. So this took a lot of time.

The change also started to have an effect on the quality of the product. The line manager in the project explained a situation where the change of the development methodology started to have a negative effect to the architectural design of the product.

Statement (C1). We used Scrum discipline and moved to this sprints style, so the architecture went to totally wrong direction at the beginning. Consultants said that the architecture will create itself with the new methodology. So the architecture started to create itself with a method where teams are doing little pieces and it keeps evolving by itself. However, it did not become this sustainable architecture.

The scalability problems with the architecture resulted to the situation where further development was not anymore possible and the company had to refactor and redesign the whole architecture from the beginning after a year. This also resulted to significant extra costs.

Statement (C1). We had to take a timeout and stop for a while and start to do the architecture with small team from the beginning. This is how we created a new core for the architecture and we have continued to build around that. The technical debt that formed in the beginning was that we started to develop it wrong, or with wrong methodology. Everybody knew how it should be, but it never became like that. We lost almost a year because of technical debt. 20 people and one year, 60 Euros an hour, so you can calculate from that. I think that we also lost many years in other parts of the product because of this.
The reason for the bad architecture was that the teams formed at the beginning were not built according to project members’ competence and it started to show as low quality solutions. In addition, the new process was still challenging to the people in the project to use.

Statement (C1). It was that you could choose your own teams pretty much and friends were searching for their friends. There was not a knowledge that what a team should consist, but like I said that the architecture demands people with high knowledge about networks and everything. So they have to be strongly included to the architecture and after that it is possible to expand to other features. But this was not the case and there were like five teams doing pieces of the architecture and there is this collaboration of technical debt.

However, around one year after the change, the benefits of making the process change started to show. The project teams started to learn the use of new methodology, which resulted to increased productivity and quality.

Statement (C1). Now you can say that there are clear signs that defects have dropped and productivity has gone up. These are hard to measure, but still there are clear indications to it. We can make several releases in a year that was impossible in the past. However, you can’t deny that our productivity dropped during the change of the methodology at the beginning.

In this scenario, the need for software process change was caused by a need to answer to the requirements coming from the markets and customer. The company would not be able to keep the customer satisfaction and competitiveness in acceptable level with existing software processes. Therefore, the company was forced to change its software development process from waterfall to agile to increase the release cycles and provide faster product development. However, the adaptation to the new software development methodology was challenging and caused a large amount of technical debt to the product architecture during the first year, mainly because of lack of competence regarding the new processes. The company had to rewrite the whole architecture, which caused significant economic consequences. When the competence in agile methodologies increased after several months, the company was able to increase the number of yearly releases. The use of new software processes also started to show as increased productivity and decreased number of defects.

15.3.2 Scenario 2: A Problem with Installation Time

In Case B, the company also decided to switch their software development methodology from waterfall to agile. The reason for the switch was the growth of the product. Before the change, the company was able to do one release each year. It was not an option anymore in the current market. The company needed to be faster in their development in order to meet the business needs of the customers and therefore it decided to try another development methodology.
Statement (B4). When the product started to grow, like networks and systems to grow, so this meant that different data elements that product was supporting started to grow exponentially. So previously we worked like that we release once a year that includes every supporting items and then we just wait again a year. So the world changed and network technology changed, so instead of ten we could talk about fifty different versions. So this meant that releasing schedules for different versions to customers had to be changed.

The biggest problems with waterfall model were the slowness and unwieldiness, because the development was layered to so many different places. One of the problems mentioned by the interviewees was that with the waterfall model it took one month for teams to get newest releases in use.

Statement (B2). Whenever team B released a new version, it took one month for team A to take the new version to use. There were lots of manual things in the process on how you install the new versions. Things changed and then the team A guys complained that “you haven’t uploaded the documentation” and team B of course was like “What are you talking about, we have not changed anything.” So there was a lot of bad build communication between those sides. They were taken sides and defending their own positions. I guess they were not playing in to the same goal.

The company did not have any experience previously in using agile methodologies except some small web development projects in the past. The decision was therefore made to acquire a consultant company to train and conduct the needed software process change. The change was started slowly by organizing workshops with teams in multiple sites. The adoption of agile methodologies took a long time and it was difficult for the consultant company to cause a change the culture of the organization. However, after three years the company was able to use agile development methodology for the project without single help from the consultant company.

The process change had a significant effect to the installation time of the product. Before the adaptation of the agile methodology, the installation time of the product to the customer was estimated to take around ten hours. The reason for the installation time was that previously the processes for the installation required a lot of manual activities and the process was not automated that well. When the consultant company was able to add agile methodology practices and processes and co-operation between teams after three years, the installation time was dropped to around two hours, which started to benefit the company economically.

Statement (B2). So suddenly when we had this really fast cycle time, fast continuous integration, that was neutral, it was shared by every team on these levels. They could see logs on every system, automated test runs took from half hour to maybe four hours. Everyone could see the reports in the same way, so there was no one to blame. Actually it kind of switched the operation model from where people were in their own cycles to “now we have to fix these together.” It was fun to see that there was no coaching or like trying to push people that you should now talk to these guys.
But instead they were like calling each other’s instead of sending emails. So to get whole organization into continuous delivery and manage to get whole organization into sync, which was really nice to see.

In this scenario the company needed to change the software development methodology to answer to the growth of the product. The growth of the product meant that the company had to provide to customers more frequent releases instead of just one or two a year. The size of the organization was extremely large and the adoption of a new software development method was challenging. It took three years for the company to implement the processes and practices. When the company was able to implement the new software processes, the productivity and yearly releases increased and technical debt started to decrease.

15.3.3 Scenario 3: Problem with Organizational Architecture

In Case A the project encountered organizational architecture problems that started to have an effect on the project and product. The company that ordered the project did not have own software development unit and the product development was commissioned from two subcontractor companies. The first subcontractor company had participated in the previous projects with the company and had vital information about the background systems related to the product. The subcontractor company took care of the back-end coding and automation of business processes. The second subcontractor was a well-known software development company and it was acquired to take care of the front-end coding and the user interface. However, this separation of the teams started to generate problems for the project from the beginning.

Statement (A4). So there were people doing this web user interface and people doing this process automation. There were different teams working for these two, people from different companies with little bit different management processes. We had our own repository and they had own repository. So even if you have full access to other team repository, it is still different thing to code something there. So the main thing was that the architecture was really off at some points because of this.

The separation of the project teams started to affect the quality of the product. There were cases where the other team started to develop low-level features for their own use, instead of waiting for a better solution from the other team. Because of this, the company that ordered the project started to use these low-level features in their business processes. This resulted to the situation where same solutions had been made to different locations with multiple different styles.

Statement (A4). When they were doing the user interface for their own usage, it was taken in use everywhere. Because there was integration between these, it enabled features that clearly had business benefits. So the company started to use this, instead of waiting that it will be made version suitable for customers. So this was done really roughly, but it was enough for internal use. This resulted to situation where the other
team started to serve themselves. Well we were not really competitors between these teams, because they were not doing anything for this web user interface, but they were doing their own. Also if they made some API here, we started to use it also and that API did not fit as well as possible to here. But this is basically the Convey's law that results to situations where the best possible solution is not used.

The communication and processes between the two teams were unsynchronized and the separation of the two teams started to affect the quality of the product. The two teams were not using the same processes and the management of the teams did not work with each other. One team was, for example, focusing a lot on quality of the code, while other team was trying to implement solutions as fast as possible. This resulted in situations where one team had to wait for the releases of the other team for the use. After realizing the issues with the architecture of the product and project, the company that ordered the product decided to make changes to software processes. The decision was made to break down the separation of the teams by creating new teams that included people from both of the previous teams.

Statement (A5). In 2009 there was an attempt to get rid of this separation, so the answer for this was feature teams that still is a really good idea. So basically we had teams that included people from both teams. So the purpose was that when we have this line from back-end to front-end and teams should be able to everything related to that.

The process change had a positive effect on the productivity of the project. The removal of the separation of the development teams started to make the software processes between teams unified. The change also started to affect to the quality of the product, since now all teams started to be included to the development of both front-end and back-end, when developers from two companies started to work in same teams. This increased, for example, the knowledge of single developers who had worked previously only for the front-end, because now they had to also develop the back-end.

Statement (A5). The idea to remove this separation, I think that this was really good step that was taken. This helped a lot and after this we moved that every team was in their own Scrum cycles and we had long time that we had demos and this connected people and this was really good thing.

In this scenario the company started to incur technical debt at the beginning of the project by implementing challenging organizational architecture for the project teams. The built organizational architecture started to effect the project, which resulted to a lack of productivity that transferred to bad quality solutions in the code base. The reason was that the development teams were not synchronized under same software processes and it made the product architecture complex. However, when the companies noticed that the amount of technical debt started to increase, they decided to change the organizational architecture. Improvement in the organizational architecture started to show as increased productivity, when all the development teams used a same software development process. This software process change started to reduce the amount of technical debt.
15.3.4 Scenario 4: Addition of a New Development Team to Another Country

In Case A the company needed to cut down development costs of the project. In order to do that, the company decreased the number of teams and replaced them by recruiting one new team to another country, where the development costs were cheaper. Another reason for the change was that the top management of the company thought that the product had reached a certain level of completion during first four years of development and the project needed less resources to continue. The sub-contractor company that had developed the product from the beginning expressed their concerns to the top management of the company that the process change would bring challenges and problems with the addition of a new team without any experience with the project.

Statement (A4). The pattern here is that some people in the management think that the system is ready. Even though there is active coding being made all the time with same amount of people. Actually it is also challenging to do changes in the current system, because when you are writing new code, you cannot change the old that much. So even though the system is already completed, even though it is being improved, the management thinks that this can be done with less resources. So it is kind of sad that this thing has been going on few years, and we thought that this process change was ludicrous and it is not going to work at all.

Adding a new team to another location required many changes to the processes and existing teams needed to adapt to the new ways of working. The company improved their communication structure by adding video conferencing possibilities between the teams working in different locations. The company sent senior developers with a wide knowledge of the product to educate the new team about the development and software processes used previously. Also at the beginning, the company assigned the development of easier features to the new team instead of complex and challenging ones. These kind of activities helped the new team to be able to adapt to the existing processes that the project had been used before the addition of the new team.

However, the software process change also created many problems that had an effect to the project. When the new team was added to the project, it started to decrease the productivity.

Statement (A5). The project suffered a lot because suddenly there was this bigger team coming from another country and they joined the project without knowing anything and their competence was variable. So basically we had good thing going in the current agile method and suddenly how would this kind of a big change fit in to this process. Here our productivity went basically to zero. So basically from a technical debt point of view, it went to that there were components that did not know anything. So how to integrate this and there was a huge code base already consisting technical debt and just randomly take people to code this.
The reason for the lack of productivity was that the time of senior developers who went to educate the new team instead of developing or improving something in the product. The new team did not have previous knowledge about the project or the code base and it started to show as additional requests to senior developers.

**Statement (A2).** The problem has been that when there are those support request coming and it is showing here as a decrease in productivity. There were some estimation that in some point 25% of the senior coders’ time went to helping because there are lots of questions coming and there has been some single cases where the quality of the code has been terrible and we had to rewrite or revert.

Another effect of the process change was the decreasing quality of the code base. The quality started to go down since the people had limited knowledge about the code base and its history, which made the development harder. A consultant in the project felt that the overall quality of the code base started to go down.

**Statement (A4).** This leads to situation that the base is being destroyed little by little. The code will change to more complex and hard to understand and harder to maintain. Quality starts to go down slowly. So in that sense the technical debt starts to grow. There are these easy technical debt like for example that one thing is in wrong place or something is done with wrong framework and so on. But then there is this general level of code that goes down all the time.

In this scenario, the company had to add a new development team to another country to cut down the development costs of the project. When the older development team was replaced with a new development team, it started to increase the amount of technical debt. The reason was that the new development team did not have competences of working with the project. It started to show as lack of productivity and quality. This way the technical debt started to increase and older teams had to fix and educate the new development team that also decreased their own productivity.

### 15.3.5 Scenario 5: Switching from Scrum to Kanban

In Case A, the project had been using Scrum since the beginning of the project. However, the company needed a change and Scrum software development methodology was switched to another agile methodology called Kanban. The reason for the process change was that the company encountered problems with the project teams’ cooperation when using Scrum. The work in the project was divided into two teams with two backlogs that caused problems and confusion to the development in the project. Another reason for the change was that in Scrum the company was using, the development was based on 2-week sprints. The company felt that having deadlines every 2 weeks increased the amount of unfinished work and technical debt.

**Statement (A4).** Sprint is being planned and one of the driven forces is that team must engage to it. Sprints are 2-weeks long, so what do we do if things are not being
ready? There is something unexpected, something was planned wrong, and this thing
is not capable to be divided into 2 week job. So we just force it through to be ready.
So when there is this deadline every two weeks and the sprint model combined
with version management that includes unfinished work. These things might occur
technical debt, because there is no time. Also you can’t do any fixing stories to next
sprint, because you might look bad if you have to fix something.

The software development methodology change to Kanban had a positive effect to the
project and technical debt. The teams focusing only on certain parts of the product
were removed and the project became a common goal for everyone. The process
change also increased the knowledge of project members, since they were now able
to take part in multiple development tasks, instead than focusing only on some certain
area.

Statement (A3). We do not have separate teams anymore and everyone is doing
what just happens to be in work line. Basically like whoever just finishes work just
takes the next story from the backlog and it gets planned and groomed. So this thing
removed the fighting and teams were not blaming each other anymore and there are
no team silos. Everyone knows something about the project now even if it is different
side of the project you are not working.

The effect of the process changes was that the amount of unfinished work started
to decrease. The project teams were able to create better solutions and do more
refactoring when the sprints and deadlines used previously in Scrum were not that
tight anymore. Kanban also gave the project more flexibility to change the backlog
that was difficult previously.

Statement (A2). Also one good thing in Kanban is that when in Scrum there are
these sprints that are really closed and it is really hard to add stuff in there. In Kanban
it is possible to change after every story if that thing was not good. You might notice
that some feature is much more valuable to customer that something done at the
moment.

In this scenario, the company had a problem with the agile development methodology
they were using. The problem was that the frequency of deadlines was too high, which
started to increase the amount of not-so-good solutions, because there was not always
enough development time. The change of agile methodology removed the concept
of hard deadlines and the development time for features was increased. This started
to reduce the amount of technical debt and the company was able to pay back the
technical debt more efficiently.

15.3.6 Summary of Scenarios

In Table 15.3, we summarize the scenarios observed in the studied cases. We identi-
fied and developed categories for types, reasons, challenges, issuesk and benefits of
software process change and how they affected to technical debt. Types of software process changes in studied cases can be divided into two groups: software development methodology (Scenario 1, 2, and 5) and organizational structure (Scenario 3 and 4). Software development methodology changes included situations that were done to change the processes, techniques, and tools in the project. Organizational structure changes were situations done to change the structure of the organizational units in the project, such as addition of a new team, changing the structure of current teams, and outsourcing/offshoring the software development to another company or country.

The main reason for companies to conduct software process changes in Scenarios 1 and 2 was the lack of frequent releases and time-to-market that forced companies to change their software development methodologies to provide more releases to customers and adapt to changing markets and technologies. In Scenarios 3 and 5,
the issues with technical debt and project teams’ structure and co-operation made companies to change their organizational structure or software development methodology to increase the productivity and quality. In scenario 4, too high development cost forced the company to make changes to current organizational structure by replacing current development teams with new and cheaper teams.

The biggest challenge for conducting the process change was usually lack of competence (Scenario 1, 2, and 5). Companies did not have information or knowledge regarding the new process that made the adaptation difficult and often companies had to hire consultants from another company to educate and administrate the process change. The process changes also encountered sometimes team resistance (Scenario 1) and not everyone was willing to learn new ways of working. In addition, all case companies were large size, which meant that size of organization (Scenario 2), needed time for the change (Scenario 2), cultural change (Scenario 2), and other companies and teams (Scenario 3 and 4) created more challenges on adapting new software processes than in SMEs.

The adaptation time to the new software process created issues and problems to the project. The biggest issues during the adaptation time were decreased productivity and quality (Scenario 1, 2, and 4). The decreased productivity and quality then led to the architectural quality and scalability issues (Scenario 1), because it forced the development team to take technical debt to keep the release windows the same.

However, after the adaptation time the companies started to have benefits of the new processes. In Scenarios 1, 2, 3, and 5, the software process change started to show as increased productivity, quality, and competence. Companies were able to make releases more frequently and the amount of defects started to decrease. Also, the level of knowledge of project team started to increase regarding new processes and it increased the level of competence.

The software process change had also significant effect to the technical debt. At the beginning technical debt was low in Scenario 4, but the business reality forced company to change its current processes. The process change was not the most optimal and caused decreased productivity and quality. This was the reason why technical debt increased at the end. Sometimes (Scenarios 3 and 5) already at the beginning technical debt was high, because the current processes were lacking in productivity and quality and company was in a need for improvement. Companies conducted a successful process change and the level of productivity and quality increased and technical debt got decreased at the end.

Challenging period of the software process change regarding technical debt was the adaptation time. In this period technical debt was increased at the beginning (Scenarios 1 and 2). The more time adaptation took, more technical debt the project incurred. Low competence regarding the new software process during the adaptation time caused decreased productivity and quality. Omitted productivity and quality then transferred to the practices and methods. This led to situations where decreased productivity was compensated with workarounds and shortcuts to keep releases cycles same as before. However, when the competence level regarding the new processes started to increase, the technical debt decrease at the end.
15.4 The Relationship Between Software Process Evolution and Technical Debt

In this section, we discuss the results gathered from the three studied software projects. The discussion focuses on understanding the relationship between software process evolution and technical debt based on our findings.

15.4.1 Common Causes for Software Process Evolution and Technical Debt

Both software process evolution and technical debt were caused by business reasons and decreased productivity. The business needs often force companies to change their software development processes, even though the level of productivity and quality in the current processes might have been satisfying [10]. The quality of the software process is connected to the quality of the software process [8]. Because the technology, business environment, and company circumstances are changing all the time, there is a need for improving also software processes [8]. In our study, we observed situations where time-to-market, customer demand or technology change forced companies to make a change in the current software processes to be more efficient (Scenarios 1 and 2).

Business needs also increase technical debt. Multiple other studies [21, 26] have shown that when companies were acquiring time-to-market benefits by delivering faster the product to the customer, it required shortcuts and workarounds to the product. Shortcuts and workarounds were not necessarily dangerous to take in short-term, because they could advance the product release and therefore increase the time-to-market and customer satisfaction. However, if companies never fix these shortcuts and workarounds, it can lead to extra costs, productivity issues and omitted quality, because the code base turns overcomplicated [42].

Another reason for software process evolution and technical debt to happen was decreased productivity. Often when companies were experiencing that current software processes were not producing enough results and the quality of the product started to go down, there was a need to make a process change. Decreasing productivity forces companies to take shortcuts to keep up the release window that increases technical debt.

15.4.2 Relationship of Competence and Motivation to Software Process Evolution and Technical Debt

Changing and improving software processes requires resources, motivation and competence [3]. The top management of the company does not necessarily understand
challenges and resources required to conduct a successful software process change that would benefit the project and product [27]. Instead, they might just think that changing one experienced team in the project to a cheaper team from another country will lead to the same productivity and quality (Scenario 4) or that changing release cycles from once a year to once a month is easy to implement (Scenario 1). The study conducted by Morten Korsaa [22] shows that over 70% of the software process improvement projects fails because of poor understanding of the process. The reason is that companies have to educate employees to the new software processes and learning is an important prerequisite to improve software development practices [39].

Learning and education time for the new software processes takes time and the results are not showing instantly to the management. Changing the whole organizational structure of software delivery or adaptation to software development methodology with new techniques and practices is not easy to conduct and requires a lot of time to show the actual results [37]. The resistance of project members to change practices is also a challenge with software process improvement [25]. We identified these same issues, when changing from waterfall model to agile methodology or adding a new development teams from another country. It meant that suddenly project members had to start work with different methods and tools and learn new ways to communicate, which had a huge effect to motivation and productivity that in some cases led to technical debt (Scenarios 1, 2, and 4).

In Scenario 4, the lack of competence of new development caused motivation issues to existing development teams and led to significant decrease in productivity, when technical debt needed to be paid back constantly. It would have been interesting to know, if the company would had stayed with the old development teams, instead of recruiting new and unexperienced team, if the current extra costs coming from technical debt could compensate the costs of more expensive development teams that were removed to cut down the development costs.

15.4.3 Challenge of Adaptation Time in Software Process Evolution

The big source for technical debt during the software process evolution was the adaptation time to new processes. During the adaptation time, the productivity often went down, because company had to go through the learning and education period (Scenarios 1, 2, and 5). When the productivity dropped during the software process change it meant that project members had to compensate the decreased productivity with shortcuts and workarounds in their activities to meet the deadlines coming from business goals. Taking a shortcut in the code base of the product or leaving test cases untested to reach the deadline to customer was not dangerous in short-term. According to Eisenberg [15], the customer and business management is more interested on the delivery day of the feature than the quality of the code base. This is the reason why it was accepted sometimes to have lack of productivity in the processes, as long as the features were going to customers in time. However, when
these shortcuts and workarounds started to accumulate during the software process change, it started to hurt the overall quality of the product (Scenario 1 and 4). New shortcuts and workarounds had be taken on the top of solutions already consisting of technical debt, because the release cycle of the product remained the same. This way it was risky for companies to work with the same deadlines during the process change. This was the reason why the management of technical debt was important during and after the adaptation of software process change. This way companies had a possibility to reduce technical debt long-term effects to the product and create sustainable and healthy products.

15.4.4 Successful Software Process Improvement and Technical Debt

One study has shown that software process improvement can be used to improve time-to-market and advantage over competition, while increasing the productivity and quality [24]. We found similar results and the case companies used multiple different ways to improve their software processes to increase the efficiency. The companies acquired new teams, changed the existing teams, tried new software development methods, techniques, and tools to change the software delivery process and were able deliver the product quicker and with more quality to the customer.

The studied companies were able to use these new processes to increase the productivity and quality in the project. Increased productivity gives companies more time to focus on refactoring and improving the existing code base, because now the new software processes might take technical debt more in consideration (Scenario 5). Also, the increase in quality and the code base made development easier and the amount of defects will drop during the development. We observed situations (Scenario 5) where the successful change to new software development methodology made technical debt more visible to the company and they were able to reduce it more efficiently.

15.5 Conclusion

In this study, we explored how software process evolution affects technical debt. We used qualitative case study approach recommended by Yin [41] to understand how software processes have evolved in the selected three large case companies. We conducted 17 interviews with professionals from both technical and business background to see the reasons, benefits and consequences of software process evolution to technical debt. We found that often the reason for software process evolution can be business related, where the company has to improve their current software processes in order to gain more advantage over competition and time-to-market. Overall, software process evolution is often considered as a positive thing toward
better development processes. However, companies rarely think of its negative consequences and resistance to change. Our inquiry into the practice of three large development organizations revealed that the evolution of software processes affects technical debt accumulated in the code base and can decrease the software quality in short-term. However, if the company takes no steps to manage the technical debt, it may finally have a dramatic effect to the software development and maintenance processes.

15.6 Further Reading

Everett and McLeod [16] define software development life cycle as a “series of stages within the methodology that are followed in the process of developing and revising an information system.” The classical software development model is the waterfall [33]. The model typically consists of five stages: (1) requirements; (2) design; (3) implementation; (4) testing; and (5) maintenance [33]. Other software development models such as the spiral model, V-model, incremental model, prototyping model have been developed after waterfall model [30]. Also the use of agile development methodologies that emphasize iterative and incremental way of software development has spread throughout the software industry [32, 34]. In addition, using building blocks from both agile as well as other classical methods is popular, see also Chap. 9. Therefore, software development processes have been evolved for a long time but companies are still interested in finding better models, practices, techniques, and tools for their software development.

The benefits of software process improvement are the reason why companies change and evolve software processes. However, improving, changing and managing software processes include also a lot of challenges [36] (see also Chaps. 10 and 9 in this volume). Beecham et al. conducted an empirical study with twelve software companies that shows that companies aiming at improving their software processes are experience challenges especially in organizational, project and software development processes [4]. According to Dybå [14] successful software process improvement depends on (1) business orientation as the extent to which SPI goals and actions are aligned with explicit and implicit business goals and strategies, (2) involved leadership as the extent to which leaders at all levels in the organization are genuinely committed to and actively participate in SPI, (3) employee participation as the extent to which employees use their knowledge and experience to decide, act, and take responsibility for SPI, (4) concern for measurement as the extent to which the software organization collects and utilizes quality data to guide and assess the effects of SPI activities, and (5) learning strategy as the extent to which a software organization is engaged in the exploitation of existing knowledge and in the exploration of new knowledge [14]. An unsuccessful software process improvement starts to affect the quality and productivity of the software development project [19]. The lack of productivity and quality can be seen as a source for “technical debt” [42].
Acknowledgments  We would like to thank the companies that took part in this research and all the interviewees who shared valuable information related to the studied cases. This research has been carried out in Digile Need for Speed program, and funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

References


How do software development teams manage technical debt? – An empirical study

Jesse Yli-Huumo, Andrey Maglyas, Kari Smolander

1. Introduction

Technical debt (TD) is a metaphor used to describe a situation in software development, where a shortcut or workaround is used in a technical decision (Kruchten et al., 2012b). TD has similarities to three aspects of financial debt: repayment, interest, and in some cases high cost (Allman, 2012). In software development, a shortcut or workaround can give the company a benefit in the short term with quicker release to the customer and an advantage in time-to-market over the competition (Kruchten et al., 2012a; Yli-Huumo et al., 2015a). However, if these shortcuts and workarounds are not repaid, TD can accumulate and hurt the overall quality of the software and the productivity of the development team in the long term (Zawizorka et al., 2011b). Creating temporary solutions to the code base increases complexity, which makes further development hard and time-consuming (Yli-Huumo et al., 2015a; Yli-Huumo et al., 2014). A simple solution for the problem would be to repay the known TD before issues start to show. However, the highly competitive software market forces companies to work in tight schedules and deadlines to release software to customers in faster cycles. This creates constant pressure for the development teams to deliver working features to customers within the given deadlines. In addition, perfection as an objective is also a risk, because it may cause delays and that way frustration to the customers, who may then select other commercial alternatives. Therefore, it is important to identify and develop processes for companies to live with TD and to know how, what and when the TD should be repaid. Technical debt management (TDM) consists of activities, processes, techniques, and tools that can be used to identify, measure, prevent, and reduce TD in a software product.

TD and TDM receive attention currently both in the academia and the industry (Li et al., 2015a). Researchers and practitioners are becoming more interested in the concept of TD and the reasons why it should be an essential part of decision-making in software development (Falassi et al., 2014). The current literature has identified and developed some tools and practices to conduct TDM. However, according to a recent mapping study, the problem is the lack of empirical evidence about TD in a real-life software development environment (Li et al., 2015a). It is important to gather
The metaphor technical debt (TD) has been introduced by Ward Cunningham (Cunningham, 1992). He describes the metaphor as “Shrinking first-time code is like going into debt. A little debt pays the during such a long time 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.” (op. cit., p. 29-30). Even though the metaphor was first introduced over twenty years ago, a recent mapping study shows that it has received the attention of researchers and practitioners only in the past few years (Li et al., 2015a).

TD metaphor was first associated with compromises on the code level of software (Cunningham, 1992). In addition, terms like code smell (Fouater et al., 1999) have described situations where poor technical choices in software development have caused problems in code quality and architectural soundness. However, the TD metaphor has been rapidly expanded after the initial concept on the code level, and it has been associated with other stages of the software development lifecycle as well (Joum et al., 2011; Albers et al., 2014). The current literature identifies such terms as requirements (Brown et al., 2010), design (Zaworkska et al., 2011b), architectural (Noud et al., 2012), test (Brown et al., 2010), process (Lim et al., 2012), documentation (Kucheren et al., 2012a), and people debt (Kucheren et al., 2012b) to demonstrate the same effect of shortcuts or workarounds happening in the other stages of the software development lifecycle.

Shortcuts and workarounds in software development usually happen for intentional reasons, such as for business deadlines and development complexity (Yi-Huama et al., 2015a). Time-to-market and customer feedback are important factors for companies’ success, and it is essential to deliver solutions on time (Lim et al., 2012). This is the reason why business stakeholders are often more focused on deadlines and customers than the actual quality of the software, which is more in the developers’ interest area (Barney et al., 2008; Boehm, 2006). Therefore, strict deadlines may force the development team to create solutions with second-tier quality to meet the requirements within the deadlines set by the business stakeholders (Yi-Huama et al., 2014). When TD starts to accumulate, it is often a safer and faster choice to take more TD with a quick and dirty solution, because there is a risk of breaking the product even more by modifying a complex part of the code base (Yi-Huama et al., 2015a). Thus, code base complexity can force the company to take more TD intentionally, because the fixing of current TD would take too much time and money, while quick and dirty solutions are easier and faster to implement (Yi-Huama et al., 2014).

TD can also occur unintentionally (McConnell, 2007). The reason for unintentional TD can be lack of competence, a need to upgrade existing technologies, or a customer or market-induced need for change. A coder may lack competence to develop an optimal solution. A development team may not be able to provide adequate instructions and coding standards for development, which reduces the quality of the solution. In legacy software, the old technology that is still in use can also be seen as unintentional TD. In these situations, a company sometimes has to start upgrading the technology to a newer version. It is also possible that changes coming from the market or a customer can lead to the effort of a development team to a new direction. This means that previously developed parts need to be changed to make the product more
suitable for the changing business needs. Fig. 1 shows a TD quadrant ( Fowler, 2009) that identifies four categories of having TD for intentional and unintentional reasons (McCormell, 2007).

TD is often seen only as a negative concept in software development (Lim et al., 2012; Yi-Huumo et al., 2014). Software developers think that creating shortcuts and non-scalable solutions will increase the complexity within the code base (Yi-Huumo et al., 2014). When the code base starts to accumulate with too much TD that is not fixed afterwards, the development becomes more challenging, because the shortcuts are not designed to work well with other parts of the code base. Complexities in the code base start to reduce the overall quality and productivity goes down when new solutions and features must be implemented to the code base in debt (Yi-Huumo et al., 2015a).

Taking TD is never an optimal solution, and companies should avoid it when possible. However, actions that lead to TD can be beneficial to software companies, and in that sense TD can be seen only as a negative side effect. When taking TD, companies are able to speed up the release cycles to the customer, which can increase customer satisfaction and provide advantage in the market. Another benefit for companies is customer feedback (Yi-Huumo et al., 2015b). Companies are able to adjust the product and its business model based on faster customer feedback. This way the companies can identify and prevent both intentional and unintentional TD more efficiently, when customer feedback provides knowledge about the most important development needs in the software (ibid.). Therefore, while TD in the software is never a benefit, actions that incur TD into software can be beneficial to a software company in terms of acquiring business advantage and knowledge of customer and business needs.

Overall, the current conceptualizations of TD vary, and there is no clear, common definition. According to some scholars, TD should be associated only with intentional decisions happening in the code base, and messy code should not be counted as TD, while some think that old technologies in legacy software should also be counted as TD (Norton, 2009; Fowler, 2009). The addition of multiple terms related to shortcuts happening in other stages of the life cycle of software development also confuses the concept. In this study we focus on TD related to a badly structured architecture/code (“smelly code”) and a code that violates coding guidelines. Even though concepts such as social debt (Zamfir et al., 2013) and people debt (Alves et al., 2014) describe similar phenomena of having shortcuts and non-optimal solutions in software development and organization, we believe that they should be categorized as sources for TD rather than as actual TD. Therefore, the definition of TD we use in this study is the following:

“A badly structured technical solution or architectural design in the system, incurred by either an intentional decision or an unintentional side effect, which causes omitted quality and productivity”.

Our goal is to understand how software development teams manage intentional technical decisions when making compromises in software development. We also believe that unintentional TD is an essential part of software development. Our aim is also to identify how development teams try to prevent and reduce both intentional TD and unintentional TD.

2.2. Technical debt management

Technical debt management (TDM) is conducted to manage, prevent, measure and reduce technical debt (TD) during software development. TDM includes processes, techniques and tools that are used in software development. The current literature related to TDM has identified and developed some processes and tools (Li et al., 2015a). Managing technical debt (MTD) workshops have gathered multiple studies related to TD and TDM in the past years (Seaman et al., 2015). However, TDM is challenging to implement, and it is hard for managers and developers to estimate and identify what and how much TD the current system has, how it will change, and what effects it will have in the future (Li et al., 2015a).

Power (2013) identifies seven main challenges surrounding TDM: (1) agreeing what technical debt is; (2) quantifying technical debt; (3) visualizing technical debt; (4) tracking technical debt over time; (5) impact of neglecting technical debt over multiple releases; (6) identifying technical debt as a root cause of defects; and (7) understanding the cost of delay.

The reduction and repayment of TD are done by refactoring or rewriting the bad solutions (Coddax and Williams, 2013). Refactoring or rewriting can be seen as processes for “changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written” (Fowler et al., 1999, p. 9). However, changing old solutions in the code is not easy, because improving the code base requires a significantly competent developer, and the company cannot just use all development time on refactoring or rewriting the solutions. Therefore, having some assisting approaches to know when and what refactoring is needed can be useful for development teams.

A portfolio approach for TDM has been suggested by Guo and Seaman (2013). The approach is widely used in the finance domain as a risk reduction strategy for investors, to determine the types and amounts of assets to be invested or divested. The core component of the proposed approach is a “technical debt list” (ibid.). The list contains TD “items”, each of which represents an incomplete task that may cause problems in the future. Portfolio management could be adapted to manage TD, where the company would collect all the TD items to a list and use it to reduce TD and to conduct refactoring systematically. Li et al. (2015b) have also developed similar TD list management for architectural technical debt (ATD).

Unintentional TD caused by changes in the customer or market can be harder to manage and predict, because the development team cannot necessarily know these TDs in advance. However, the current literature has identified some practices to prevent unintentional TD. Implementing coding standards to the development process can prevent TD, when the developers have a cohesive way to
produce a similar style code, which makes it readable and modifiable (Green and Ledgard, 2011). Code reviews can be used to check other developers’ solutions before the release to catch possible TD issues in the design (Manthey and Lasenius, 2009). Also simple practices in agile methodologies, such as the Definition of Done practice can reduce TD in the early stages of development (Davis, 2013).

An extensive mapping study of 49 primary studies has been recently conducted by Li et al. (2015a) to understand the current state of the art on TD. The study identifies eight activities for TDM: (1) identification detects TD caused by intentional or unintentional defects; (2) technical documentation through a specific techniques, such as static code analysis; (2) measurement quantifies the benefit and cost of known TD in a software system through estimation techniques, or estimates the level of the overall TD in a system; (3) prioritization ranks identify TD according to certain predefined rules to support deciding which TD items should be repaired first and which TD items can be tolerated until later releases; (4) prevention aims to prevent potential TD from being incurred; (5) measuring watches the changes of the cost and benefit of unresolved TD over time; (6) replying resolves or mitigates TD in a software system by techniques such as refactoring and restructuring; (7) representation/documentation provides a way to represent and codify TD in a uniform manner, addressing the concerns of particular stakeholders; and (8) communication makes identified TD visible to stakeholders so that it can be discussed and managed further.

Overall, the current understanding of TDM includes some ideas for processes, techniques and tools to manage TD. Even though the current literature has started to tackle and identify the concept and solutions of TD, the problem is that there is a need for more empirical evidence from real-life software development (Li et al. 2015a).

2.3. Empirical studies on technical debt management in practice

There are a few empirical studies on TDM. Gao et al. (2011) use a specific TDM framework to track down one delayed maintenance task in a real software project. Their TDM framework starts from the identification of TD item, which then will be added to a TD list. After this, the TD item gets measured based on the principal and interest, which are used as an estimator. Then, the TD item is ready for prioritization based on cost and benefit. With this framework, the authors have been able to track down and quantify TD items, and see the costs of delaying maintenance tasks. A similar approach has also been used by other researchers to identify and document TD issues in order to make TD easier to manage (Zawoyska et al., 2013).

Klingner et al. (2011) interviewed four experienced software architected and understood how decisions regarding TD was conducted in an enterprise environment. The results showed that the decisions related to TD issues were often informal and ad hoc, which led to a lack of tracking and quantifying the decisions and TD issues. The study also identified that there was a large communication gap between technical and business people as regards discussion about TD.

Different tools have been developed for TD. The SQAIE method (Leturcq, 2012; Leturcq and Ikrortz, 2012) has been developed for the purposes of identifying, estimating, analyzing, measuring, and monitoring TD in a software. DebtFlag (Nolletta and Leggiones, 2013) has been developed to capture, track, and resolve TD in software projects. The SonarQube tool and its plugins have been applied in several studies to identify and measure TD in software (All estimates the level, Griffetti et al. 2014). A set of other tools to support TD management were identified in the mapping study by Li et al. (2015a).

Most of the empirical studies of TDM take in consideration only a few aspects of the eight TDM activities (Li et al., 2015a). A specific tool to identify and measure TD does not help in other activities, such as communication or prioritization. There is a clear need to know how TD should be managed from the organizational point of view. The mapping study by Li et al. (2015a) found a large number of different models, methods, practices, and tools in the literature for each separate TD activity. However, there is no single solution that takes the whole problem of TDM into account. Therefore, a framework or model for TDM that combines all TDM activities is needed, both by researchers and practitioners, to understand all the aspects of TDM.

3. Research process

3.1. Research methodology

This study is qualitative, and it uses case study as the research methodology. The definition by Yin describes a case study as an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident (Yin, 2003, p. 13). As a research strategy, case study is used to contribute to our knowledge of individual, group, organizational, social, political, and related phenomena (ibid.). Therefore, case study has been a common research methodology especially in social sciences. However, the case study methodology has also been used in economics (ibid.), and it has become more popular in software engineering as well (Runeson and Höst, 2008).

Software development is carried out by individuals, groups and organizations, and therefore social and political questions are of importance for software development, which makes software engineering a multidisciplinary area where case study is a relevant approach (Runeson and Höst, 2009). There are multiple ways to conduct TD in software engineering. The research can investigate the written code itself, where the focus is on understanding how a badly structured code affects the other parts of the software. This type of research can be done with quantitative research methods, where the results show measurements on how for example performance or other quality attributes change depending on different structural and other parts of the software. This type of research can be done with qualitative research methods, which can, for example, produce results that show how people and processes affect the existence of TD in the software.

Since the aim of this study is to understand how software development teams conduct TD to control and reduce TD, rather than what are the best possible code structures or architectural solutions, we believe that the case study methodology is an effective approach to understanding how people with different responsibilities working together in software development have organized TDM. The case study methodology makes it possible to examine the concept of TD in real-life situations, to gather qualitative data, and to add to existing research related to TDM.

This study can be defined as an interpretative exploratory case study (Robson, 2002), as the goal of the study is to discover how software development teams have organized TDM without a priori hypotheses. The purpose of an exploratory case study is to find out what is happening, seeking new insights and generating ideas and hypotheses for new research (ibid.). In addition, an interpretive case study aims at understanding phenomena through the participants’ interpretation of their context (Runeson and Höst, 2008).

We decided to use the guidelines provided by Runeson and Höst (2008) to conduct the case study process. The case study...
process is divided into five main steps: (1) case study design: objectives are defined and the case study is planned; (2) preparation for data collection: procedures and protocols for data collection are defined; (3) collecting evidence: execution with data collection on the studied case; (4) analysis of collected data; and (5) reporting (ibid.). Fig. 2 shows the research process used in the study, based on guidelines by Runeson and Höst (ibid.). The steps of the research process are discussed and explained in closer detail in the following subchapters.

3.2 Case study design and company selection

The design step of the case study process should contain the following elements: objective, the case, theory, research questions, methods, and selection strategy (Runeson and Höst, 2008; Robinson, 2002). The design and development of the data collection protocol was started by examining the current literature related to TD and TDM. On the basis of the literature, we designed and developed a set of questions and topics for discussion to understand the reasons, effects and management related to TD. We decided to organize the interviews in two separate rounds. The reason for conducting the interviews in two rounds was that we wanted to understand the concept of TDM first through a smaller number of development teams to be able to adjust the interviews for the second round cases. This approach is similar to the theoretical sampling used in the grounded theory method (Strauss and Corbin, 1998), where the next data sample is chosen on the basis of an analysis of a previous sample, creating an iterative process for theory construction. In this case, we wanted to be able to modify our questions and topics based on the data received from the first round of interviews.

We decided to use semi-structured interviews (Charmaz, 2014) for data collection, which makes this research a flexible study (Runeson and Höst, 2008). Semi-structured interviews include a mixture of open-ended and specific questions, designed to elicit not only the information foreseen, but also unexpected types of information (Seaman, 1999). We thought that semi-structured interviews would provide us with good results, since the term ‘technical debt’ might be unfamiliar to the interviewees, and also taking in consideration the complexity in its definition, it was important to explain it carefully to create similar understanding between the interviewee and the interviewer. In addition, it was important to introduce all the aspects of TDM activities in the interviews, as it was highly possible that the interviewees would not have knowledge on their definition. Thus, the use of semi-structured interviews would make it possible for us to talk with the interviewees face to face, and in a case of misunderstanding, we would be able to explain the questions more precisely and ask more specific follow-up questions to identify answers to the research questions. A potential drawback of using semi-structured interviews can be the trustworthiness of the answers. However, we believe that there was no issue with the trustworthiness of the answers, since all the development teams in this study had expressed their interest in developing TDM in their organizations.

As our goal was to study TDM activities and their maturities, and we also assumed that the TDM activities would be used differently in teams within an organization, we decided to use the multiple-case study approach. Yin (2003) separates case studies to holistic case studies and embedded case studies. In holistic case studies the case is studied as a whole, while in embedded case studies multiple units of analysis are studied within a single case. This research fulfills the characteristics of a holistic case study, as our goal was to study each development team as a whole to understand the
process related to TDM, instead of studying multiple units within one development team. The reason for studying multiple software development teams over one single team was gathering a broader amount of empirical data related to the research topic. We believed that studying several development teams would provide us with more information related to TDM, and comparing the results would help us understand what approaches were the most commonly used and why. The selected case company is a large software supplier with around 5600 employees, currently operating in multiple countries in Europe. The company is a supplier of business software and business process solutions, outsourcing services, commerce solutions, and IT consultancy. It has currently about 340,000 customers. We studied eight software development teams in the organization. A summary of the software development teams and their roles in the organization is presented in Table 1. We selected the case company because its size, number of teams, and industry area, which made it very suitable for studying TD and its management. In addition, even though all the development teams were from the same organization, most of them were not working on the same product. Instead, most of the teams had their own product in development and a separate management, originating from the company’s history of mergers and acquisitions. The company combines several product lines and includes teams coming from different backgrounds and cultures, but currently sharing the same organization. Therefore, we considered the company to be optimal for studying TDM activities in development teams.

3.3 Data collection

The semi-structured interviews were conducted in two rounds between February 2014 and April 2015. The first round with two development teams located in Finland (Cases A and B) was started in February 2014, and it lasted until April 2014. We started the interviews by contacting the manager of the team. The manager of team A gave us also a referral to the manager in development team B. We conducted the first two interviews with the product line managers of teams A and B. After that we used the snowballing technique (Charmaz, 2014) to get referrals to other persons in the teams. As both development teams were located in Finland, we were able to travel physically to the offices and conduct all the interviews face to face. The total number of interviews was five in development team A and seven in development team B. One of the interviews in team B (Interview ID B5), we interviewed two persons at the same time because of schedule constraints. The interview sheet for the first round interviews is enclosed in Appendix A.

The second round started in March 2015 and lasted until April 2015. Before starting the interviews, we decided to make changes to the interview structure for two reasons. The first reason was that the data gathered and analyzed in the previous round gave us new ideas for the interviews regarding TDM. In the first round interviews we identified a lack of TDM activities, which gave us an idea of focusing more on TDM. In the previous interviews, the focus was also on the effects and causes of TD. The analysis of causes and effects of TD is available as a separate publication by Yli-Huumo et al. (2014). The second reason for focusing more on TDM was the publication of the TDM mapping study by Li et al. (2015a). The mapping study identified eight TDM activities, which we considered as a good basic core for the inquiry on TDM. The results of the mapping study gave us new ideas for improving the interview structure more towards TDM activities. The updated structure for the second round interviews is shown in Appendix B.

The second round consisted of six software development teams located in various countries in Europe. The team manager of development team A gave us new referrals, which we used to
contact the other six development teams. Because the teams were not located in Finland, we had to change the interview method from face-to-face interviews to online video calls. The interviews also changed from single person interviews to two-three person interviews. This was required because the time allowed for us was limited. The interviewers were usually one team manager and one software architect discussing the approaches to TDM. The risk of interviewing two or more people at the same time is that the interviewees would not necessarily be able to speak openly because of the presence of another interviewer. However, we noticed during the interviews that this was not the case, and all the six development teams were eager to talk about the problems with TD and TDM, and wanted to find possible solutions for improvements. In addition, we noticed that all the software architects had multiple years of experience with the software product, which was appreciated by the project managers involved. Therefore, we believe that the interviews were not disturbed by having multiple people present at the same time. Instead, we believe that the quality of the interviews was improved, since there was a common goal from both business and technical perspective to understand and improve TDM. The roles of the interviewers are shown in Table 2. When the interview engaged more than one person, this is referred to in the interview ID as E1a, E1b etc.

3.4. Data coding and analysis

In exploratory case studies, the technique for the analysis of qualitative data is hypothesis generation (Seaman, 1999). As we did not have any prior hypotheses for this study, our goal was to use the techniques for data coding and analysis of qualitative data to find hypotheses from the collected data and interviews. The techniques for data analysis used in exploratory case studies are constant comparisons and cross-case analysis (Seaman, 1999).

Fig. 3 gives an overview of the data coding and analysis processes conducted in this study. The data coding and analysis were completed in various steps, guided by the work of Robson (2002). Overall, we conducted a total of 12 interviews with 25 persons related to eight studied cases, and had 627 minutes of audio-recorded data. When all the interviews were conducted, we began the data transcription phase. The first round interviews were transcribed by the authors, and the second round interviews by a hired person with English language proficiency. The reason for the authors to transcribe the first round interviews was that the interviews were conducted in the Finnish language. The authors transcribed and translated the first round interviews to the English language to make the coding and analysis stage easier, because there were only one main language in use in the study. All the second round interviews were conducted in English. During the interviews we were also able to gather some additional documentation data. In one of the interviews we received a PowerPoint presentation related to the TDM activity the team was currently conducting.

After all the data was transcribed, we started the data coding and analysis stage. The total word count of transcriptions read was 73,955. We used a tool specialized for qualitative data coding and analysis, Atlas.ti. In data coding, one code is usually assigned to many pieces of text, and one piece of text can be assigned more than one code. The codes can form a hierarchy of codes and sub-codes (Robson, 2002). Our data coding stage followed the top-down approach, because the categories were derived from the mapping study by Li et al. (2015a), which identifies eight activities for TDM. The categories used in the data coding were TD replication, TD representation/documentation, TD identification, TD prioritization, TD measurement, TD monitoring, TD communication, and TD prevention. Table 1 shows an example of the data coding process with Atlas.ti, where the interviews are used to extract quotations to the identified categories. We believe that using the top-down approach in the data coding was an effective way to understand how every TD activity was approached in every development team, which helped us to draw conclusions and understand the TDM process.

When all the quotations were extracted and identified to the specific categories, we analyzed every category independently and drew a conclusion on the process used for TDM in each case. When we had a complete view on every case, we started a cross-case analysis to find out the similarities and differences between the cases.

4. Results

4.1. Case A

TD replication with refactoring and rewriting was based on the general development backlog, where some of the code base improvement issues could be found. However, we were not able to find any replication strategy for TD that was incurred during the development. The developers in the team mentioned that it is sometimes impossible to get time to refactor the solutions that were developed previously with shortcuts. The reason was that new features were already waiting in the next sprint’s development backlog that were prioritized higher than technical improvements in the code base. Therefore, TD replication with refactoring or rewriting was mostly done unofficially during the actual development time that was reserved for new features. Sometimes this refactoring was not even mentioned to the management. The team management had adopted a practice where every Friday was dedicated to bug fixing. However, the developers felt that it was mostly dedicated to fixing only bugs, instead of conducting architectural-level refactoring or rewriting.

TD representation/documentation was not systematically conducted by the development team. The development team did not have a separate TD backlog to document TD items either. When a developer identified a possible TD in the code base, there was no clear process or guideline on how to document it to the management system. One of the developers mentioned that the team used
Table 1

<table>
<thead>
<tr>
<th>Interview transcript</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>“We have bugs, and we have some kind of goal, so 20% of developer time to be in internal quality.”</td>
<td></td>
</tr>
<tr>
<td>We have quite a lot of security concerns, and maybe some technical debt can come from security”</td>
<td></td>
</tr>
<tr>
<td>“We can measure also how much time we spend on this slice of the backlog. For example, on internal quality as the whole in the team, third, speed, 20% of, and we see people, some of our system architects, some of our developers, some QA testers and so on. So, we have this console and having this an objective 20% of developer time, we can check if we spend that time, that budget on not.”</td>
<td></td>
</tr>
<tr>
<td>“As a team we have some 20%. In the team, we are part of product unit, R&amp;D department will 20% and other product-related. And then we as a team define 20% to measure, and for example, we have one key instead on technical debt.”</td>
<td></td>
</tr>
<tr>
<td>“And at a team, we have kind of done, some kind of retrospective on monthly basis, then we do all the numbers and then discuss: is this standard is good, this number is too low, what to do, and then we put on backlog system.”</td>
<td></td>
</tr>
</tbody>
</table>

The JBK management system, where it is possible to create tickets for issues found during the development. However, this was not always done by the developers, which resulted in situations where some TD remained undocumented and was kept in the notes of the developers, and even sometimes forgotten.

TD identification was mainly conducted during the development, when a developer noticed a problematic area in the code base, which then sometimes resulted in fixing the case in the management system. TD identification was also conducted by the system architects and team managers, who sometimes analyzed the code base to find what should be changed to improve the quality and maintainability, and add some refactoring tasks to the development backlog.

TD prioritization was mostly done on a hunch. When a TD issue was raised in the management system, the development team would discuss the importance of that specific case and give it prioritization. The most important factors taken in consideration when deciding issues to be refactored were the scalability and business value of that specific feature.

TD measurement and TD monitoring were not conducted by the team manager or the software architects. The reason for not measuring or monitoring TD was the fact that the development team did not have any clear process for documenting TD items, which meant that the team management did not have the possibility to gather or analyze any clear data. Only estimations of TD were based on current knowledge about the code base and issues in it.

TD documentation was structured well and the development team members understood the concept of TD. The team manager had a good technical knowledge background, which helped the developers and software architects to communicate and discuss about the possible TD issues that had occurred during the development. This was the reason why the development team would sometimes get more time to fix and repay TD items that had been bothering them in the actual development. The team manager would also act as a filter between the business team and the development team.

When the business stakeholders gave tasks that were impossible to develop within the given deadlines, the team manager would explain the situation to the business managers, which sometimes gave more space to the developers.

TD prevention was done with coding standards and code review practices. The development team had taken in use some level of coding standards with coding books and instruction videos to show the developers what kind of coding was expected to be developed. Code reviews were sometimes conducted by two software architects, but it was not mandatory to check every newly developed code. As supervision of TD prevention was not always conducted, one of the developers mentioned that sometimes the developers would just use the old code and copy it to the other parts of the code base, which could be risky.

Overall, the TD strategy in development team A was not organized as a systematic process. The development team did not have any clear TD documentation or TD repayment process to gather TD
issues, and it was often organized unofficially. This was the reason why it was also impossible for the team management to monitor or measure TD. However, the development team had a good basis for starting TSM, because the code review was active regarding TD, and the team manager was considered to have good technical knowledge, which helped the development team to deal with TD. The development team also had good TD prevention practices in use, even though their actual use was not confirmed.

4.2 Case B

TD repayment in development team B was mostly considered as part of the normal work during the development. In a situation where a developer identified a small refactoring case, there was no need to create a separate issue for it. In a situation where the refactoring case was bigger, the development team would organize a discussion with the team manager and software architect to discuss the next steps and whether there was a need to conduct refactoring or rewriting of that specific solution. The team management also organized a practice where one day of the week was dedicated to fixing bugs and making small refactoring.

TD representation/documentation was not currently a systematic practice within the development team. When a developer decided to take a shortcut during the development, there was no mandatory process defined on how it would get stored and documented in the JIRA project management system that was used. We identified situations where the developers might have created JIRA tickets to the management system, but also situations where they were just left in the coder’s own notes. The developers also did not always inform the management about what shortcuts were made. The team manager and the software architect would sometimes add some TD issues to the development backlog, when the issues were raised during the development.

TD identification was mainly conducted during the development by the developers. When the code base was developed, the developers would identify the refactoring needs, when the currently developed software was extremely complex and hard to maintain. Sometimes the software architects would go through the code and try to identify possible places, especially in the architecture, where refactoring was needed.

TD prioritization was often based on a hunch and previous experience with the code base. However, prioritization would get done according to the location of the issue. If the issue was in the core of the code base, depending on several other places, it would get prioritized as highest. After this, issues in the business logic and user interface were prioritized under it.

TD measurement and TD monitoring were not currently done by the team management. The reason was that it was at the moment impossible for the team manager and software architects to know what TD the software currently had, because it was not properly documented anywhere. The team management did not have any specific tools in use to measure TD, either. This was the reason why there were no accurate measurements or monitoring to see the current status of TD.

TD communication was structured well in the development team. The team manager had wide technical knowledge, which helped TD communication between the management and the developers. This also helped the development team in situations where the business stakeholders gave impossible deadlines to work with, because the team manager would explain the issues of possible TD to the business management. However, the development team expressed a problem in communication, as the development of new features was always prioritized the highest, and the code base improvements were not done beforehand.

TD prevention was done with coding standards and code reviews. However, the team manager mentioned that they were not on a good level at the moment, and there was a need for improvement. The current coding standards did not fulfill the needed requirements, and the development team did not always follow them. Also, the code reviews were conducted by the software architects, but it was not always possible to go through all the developed code, as it was not prioritized enough.

Overall, the TSM strategy in development team B was similar to Case A. There was currently no mandatory process used to document TD issues in the JIRA system, and the development team did not have a special TD backlog in use. Refactoring was conducted mostly unofficially during the development, and there was no systematic process to repay TD in the TD strategy. This was the reason why it was extremely difficult for the team management to measure and monitor TD. However, the idea of TSM was understood by the management, and they were eager to find improvements. This is why TD communication was active within the development team, which gave more space for them to work on some TD issues.

4.3 Case C

TD repayment was identified as an essential part of software development by the software architect, and the management of the team had realized that it should be a part of the development process. Development team C used the Kanban methodology and JIRA tool to manage the software project. In the Kanban table, the team management had assigned 20% of the development time specifically to improving internal quality. Internal quality was divided into five main parts: refactoring, test automation, DevOps, platform security, and performance. The development team used these five internal quality factors to assign issues if something needed to be refactored, rewritten or redesigned. The development team identified TD as an important key performance indicator within internal quality. If a person in the development team saw a higher need for refactoring, he/she created an issue in JIRA under internal quality, which then was included in the actual development backlog after a discussion with the team management.

TD representation/documentation was not always done systematically by the developers. The development team did not have any mandatory guidelines for the development representation and documentation of TD. In a case where a developer created or founded a TD issue, there was no clear process of how to document it systematically afterwards. Instead, the developer may have sometimes created a JIRA issue ticket for refactoring, and it could be found in the internal quality section, or in some cases it would not get documented. The internal quality section in this case was used as a TD backlog. The management felt that this should be improved a lot in the future and there should be clearer guidelines for systematic documentation of TD.

TD identification was conducted mainly by the software architects. The two software architects were given responsibility by the management to identify TD in the code base. Therefore, the software architects usually went through the code base to understand and identify possible items to refactor or improve, which were then added to the internal quality issues. The identification was often done just by going through the code base manually, and trying to understand what parts of the code base were the most complex. Part of the identification was conducted with the SonarQube tool, but the architects mentioned that it was not necessarily the best way to identify all TD, because it does not take deep and complex architectural issues into consideration. For example, with SonarQube the software architects were able to find issues related to single line problems or code violations, but they could not detect some complicated business logic issues, which was considered as a
real technical problem. Therefore, the identification was seen more as the responsibility of the people and processes. The team manager and software architects also mentioned that the developers were not closely involved heavily in the TD identification process, and hoped that they would start to identify more TD issues in the future.

TD prioritization responsibility was given to the software architects. The prioritization of TD issues was usually done on a bunch and previous experience of the code base. The development team did not have any systematic way to give estimations or numbers to prioritize TD issues. One of the software architects mentioned that when teams would become involved with the TDoM process, they would need to consider additional time to estimate how heavily the feature was currently used or whether a lot of new features were expected to come to that area in the future. The team manager also mentioned that he had no opinion on more than numbers when making decisions about refactoring.

TD measurement was done by one of the software architects, who used SonarQube to measure TD. The SonarQube tool gave values of TD as automated test coverage and violations in the code. The software architect used these two measurements to estimate the current TD monthly. However, the software architect responsible for the measurement thought that using only SonarQube to have measures of automated test coverage and violations in the code base cannot be the only good way to measure the actual TD. The problem was that SonarQube only identifies minor TD issues, such as issues in the code, but not real problems in the architecture. This was the reason why it was hard to generate refactoring issues from the SonarQube tool to the internal quality backlog.

TD monitoring was done by using data gathered from the JIRA tool, which gave the management the possibility to estimate and follow how much time had been spent on internal quality compared to the overall development time in a certain period, and whether it was aligned with the agreed 20% rule. The software architect also used data from SonarQube to monitor the current status of TD monthly, and it was analyzed and reported to the management, to show whether TD was increased or decreased. The combined data from JIRA and SonarQube was used to monitor how TD was changing.

TD communication was an important area of discussions between the team management, software architects, and developers. The team manager worked closely with the software architects, which helped them to generate new issues from code and to inform the management about new issues. TD was discussed in team meetings, with the purpose of improving TD quality and TD. This way the development team was able to reduce issues related to internal quality and TD, instead of using the development time to create only new features with business value. The software architect also often discussed with the developers about issues related to TD.

TD prevention was conducted sometimes with coding standards and code reviews. The team used Java coding standards as a recommendation to developers in similar code. Both software architects also sometimes reviewed the code to catch bad designs. However, these were only used as recommendations, and it was revealed that in reality the coding standards were not always followed or code reviews conducted.

Overall, the strategy for conducting TDoM was structured well in development team C. The idea to use 20% of the development time to improve the code base and refactor architectural issues was a good strategy to reduce TD systematically in the software. Also, the measurement and monitoring with the JIRA and SonarQube tools gave the management some level of estimations about the current status of TD in the software. The issues with TDoM in development team C were TD documentation and TD prevention. Even though the TDoM structure was well-designed to repay TD systematically, the development team still did not have a proper estimation practice in use. When the developers took or found TD issues, they were not always reported or documented, which made it hard for the software architects to understand the status of the current TD issues. TD prevention with coding standards and code reviews was also lacking and considered a big problem by the management.

4.4. Case D

TD repayment was conducted, similarly to Case C, by assigning 20% of the total development time to reduce TD issues in the software. The time for improvements was mostly used for additions of automated tests and unit tests. The software architect of the team felt that they could reduce TD the most, because it prevents TD from occurring in the software. If a need for refactoring was found during the development, it was assigned to the 20% internal quality section in the JIRA management system that was used in the team. The 20% rule was also used for bigger refactoring and rewriting issues to remove bad designs from the code base. The development team had a two-month release cycle, where the last two weeks were dedicated to the stabilization of the code base. During the two weeks, the development team would discuss current TD issues and what should be refactored in the next two months’ iteration. The software architect also had the authority to use the JIRA system to see internal quality issues, and make decisions on what should be refactored in the next iteration. The goal was to fulfill the 20% rule in every iteration. Sometimes only for example 10% was required to be used on internal quality, because there may have been a need for new features with important business value. However, the team manager may have added 25% to the next iteration after that, to keep the average on the agreed 20%. The refactoring or rewriting of small TD issues was conducted during the development, and it was not necessary to mention them to the management or report to the system.

TD representation/documentation was done to the JIRA management system. When a member of the development team saw a possibility to have a refactoring case, the instruction was to create an issue to the system about it. In addition, if a developer needed to take an intentional shortcut during the development, it was also instructed to be reported in the system. In this case, the internal quality in the JIRA system worked as a TD backlog.

TD identification was done mostly by the software architect reviewing the code base manually or with a tool. The tool used for identification was SonarQube. The software architect used the SonarQube tool every night when the new version of the software was out and used it to gather statistics about TD. If the tool reported any major issues, it was the responsibility of the software architect to report and go through every critical issue and try to fix them before the end of the iteration.

TD prioritization was done by simple low, medium, high, and blocker scales. High and blocker TD issues were repaid immediately or in the next iteration. Medium TD issues were also repaid in the next iteration or the one after that. Low TD issues did not usually get repaid ever, because the backlog was usually flooded with them. The software architect felt that fixing low priority issues would not bring any value to the software. The management and software architect also assigned story points to TD issues, based on the Fibonacci scale. For example a medium case was usually assigned 5 or 8 points, while high or severe cases got 13 or 21 points. The management and software architect responsible for prioritization did not use any specific calculations to create these prioritizations. The decision about a single prioritization was mostly based on a hunch and the experience of the software architect with the code base.

TD measurement was conducted with the SonarQube tool by the software architect. The results of the SonarQube were used to have a measurement of the current TD. The team manager also used Fibonacci scale prioritization to measure the velocity in the
development in order to understand how much the development team could repay TD in the next iteration. The management and software architect felt that these two measurements for TD could be used to have good TD estimates.

**TD monitoring** was conducted with the JIRA and SonarQube tools. The management was able to use JIRA as a tool to monitor the development team's progress towards TD reduction. The management used this information to generate reports at the end of each iteration. The software architect also tried to use the data received from the SonarQube tool constantly as a way of monitoring TD.

The development team performed both the team manager and software architect, who discussed the importance of TD repayment. The team manager initiated the discussion at the beginning of each iteration with the team members. In an attempt to understand what would need to be done in the next iteration. The software architect mentioned that the development team was currently in a lucky situation, because the team manager understood the concept of TD and was eager to help the development team in dealing with it. Even though the team manager might have been described as someone who took part actively in technical decisions, she still understood the importance of TD and the fact that software quality would be an important factor in the long term, which gave increased visibility to the effort of reducing TD.

**TD prevention** was conducted with coding standards and code reviews. The development team had created a rule that nobody could not commit anything to the code base before another developer had reviewed it and it fulfilled the standards of the Definition of Done. Of course in reality this meant that if a developer changed a minimum amount of code, it would not be necessary to be reviewed, but in a case where there was a risk of breaking the software, a review was mandatory. In the most challenging cases, more than one review was needed. Also a discussion with the team manager was organized to understand, learn and find the best solution. The software architect mentioned that even though this rule was good to have, it was not always followed very strictly.

The overall TD strategy in software development team D was constructed well. The management had a clear vision and understanding of the use of 20% of the development would be used for TD repayment. This was compounded with the JIRA and SonarQube tools that were used to document, measure, monitor, and identify TD. The development team had also well-conducted rules in code reviews and standards to prevent TD.

### 4.5. Case E

**TD repayment and improvement decisions** of the code base were created on the basis of a stakeholders’ meeting once a month. The manager of the development team would make a suggestion in the stakeholders’ meeting on how much time would be needed to repay TD and improve the code base in the next month. The manager of the team mentioned that for example in the three previous months, the development team had made an agreement with the stakeholders that one third of development was assigned to repaying TD. However, the problem with the repayment of TD was that even if the development team got the time agreed to refactoring or rewrite issues, in reality it was not possible to do so, because the new features would always take more time to complete than estimated, which took away time that was reserved for TD repayment.

**TD representation/documentation** was done by creating a backlog approach for TD issues. When a development team member made a decision to create a shortcut to some solution, or identified a need to refactor old technology or bad design, it was documented in a separate technical debt backlog in the JIRA management system. This process was used by the development team to make TD more visible in the development process. As the nature of the development team was to act as a platform for other product lines in the organization, the backlog was also used to communicate about TD issues in the platform with other development teams. The development team was able to present the backlog to all other development teams with information about possible issues in the future where TD would be most disturbing.

The prioritization was mostly done by the software architects, who spent a lot of time with the code base, trying to identify possible improvements regarding TD. The development team did not have any special tool to identify TD in the code base, and it was mostly done by just “smelling the code”.

**TD prioritization** was done by the team manager and software architects. However, the prioritization process was described as not well constructed. The team manager and software architects mentioned that they would categorize the first TD type issue. Issue types were usually related to refactoring, security and performance. After this, the actual prioritization was mostly done at a hunch, based on opinions and experience with the issue. The most important factors taken in consideration when making a prioritization were often related to how TD would affect the customer and future projects. Also security and performance were mentioned to be important when deciding on the most important refactoring cases.

**TD measurement and TD monitoring** were mainly done with information that was available in the JIRA issue. The TD backlog in JIRA was used to gather some statistics and estimate the current status regarding TD in the platform. The manager and software architect used some basic information in the backlog to monitor and measure how much TD the platform had by calculating the number of issues and prioritizing them according to their importance, to make refactoring.

**TD communication** was done mainly in the stakeholders’ meeting. The management and software architects gave suggestions to the stakeholders in the meeting about the development team prioritized TD and why it should be important to repay and refactor as soon as possible to prevent future issues with it.

**TD prevention** was done with guidelines and code reviews. The developed code was always checked with a tool called Style-Cop to ensure that it was written according to the guidelines. It was also mentioned that the developers did sometimes do code reviews, but this was not described as mandatory. If a developer wanted someone to check the code, he/she could ask someone to do it. However, code reviews were instructed to be conducted if the developed code was done in a section of the code base that was known to be extremely complex. The management mentioned that the reason for not conducting code reviews in all developed codes was that most of the developers’ time was assigned to the development of new features, and there was no time to have all codes reviewed.

Overall, the TD strategy in development team E was mainly focused on the documentation of current TD issues, and trying to find time to refactor on the basis of stakeholders’ meeting once a month. The management did not currently have any systematic way to identify, measure or monitor TD. The management also mentioned that TD prevention was currently not the most effective, and more time should be reserved to it.

### 4.6. Case F

**TD repayment** was not done in any organized process. One of the software architects mentioned that in the current process, the repayment of a TD issue was usually started when the issue started to be highly problematic for the TD according to its handle and there was no other way than just to refactor, rewrite or redesign it. If a developer noticed a need for refactoring, it was
often taken care of by the developer, without any actual systematic refactoring process.

TD representation/documentation was not a part of the development process. Sometimes the software architects of the team would add some major issues identified in the code analysis tool which would need to be addressed, to the main backlog. In a case where a developer took a while to fix an issue, or when there was a need for refactoring, it was often just left in that developer’s memory, and may be documented somewhere.

TD identification was conducted by using SonarQube, CheckStyle and FindBugs as tools to analyze the code base to find possible TD. The software architect used the data gathered with the tools to understand the current status of TD in the software. However, similarly to Case C, opinions about the actual data acquired from the tool were not always used. The issue was that the tools did not necessarily give the needed information about TD in deep architectural structures of the code base. However, the software architects took the most critical issues identified by SonarQube and tried always to fix them. The common opinion was that the actual identification was done during the actual development, and the development team had some self-assessment cases to identify TD issues. SonarQube was not advised to be used by the developers, so identification with a tool was done mostly by the software architects.

TD prioritization was mostly done at a hunch and the software architects used their previous experience with the code base as the starting point when prioritizing TD issues.

TD measurement and TD monitoring were not conducted by the team manager and software architects, even though they used the SonarQube tool activity to identify TD. The reason was that SonarQube did not give valuable numbers for actual measuring of the real TD. The real metrics used were the actual global number of TD issues in JIRA, which was used to measure the current TD.

TD communication was seen as a problem during development. The development team felt that communication about TD to the business people in the organization was difficult. The software architects also felt that the development team did not currently discuss issues related to TD with the team manager or the software architects.

TD prevention was not conducted at a good level within the development team. The development team had set up some standards with the SonarQube, CheckStyle and FindBugs tools. The development team mentioned that they did not currently have much stuff related to coding standards or code reviews. However, the team manager mentioned that they were currently developing a definition of the done standard to improve TD prevention in the future. The future Definition of Done should comprise at least code reviews, unit tests, and errors found by SonarQube.

Overall, the TDM strategy in development team F was not an important issue within the development process. The development team did not currently have any systematic way to document, monitor or measure TD items. Repayment was often based on a hunch and was conducted when some TD issues started to grow too large, and the only way was to refactor or rewrite the solution. The reason why the management thought that implementing TD processes to development would be challenging was that the team conducted constant identification and repayment of TD, it would not be cost-beneficial to the organization.

4.7. Case G

The TD repayment process was often started on the basis of a feeling that something should be improved. When the software architect or a developer noticed that there was a need for bigger refactoring or rewriting in the code base, it was mentioned to the management. After this the management would organize a discussion about the issue, where the development team would estimate the effort to fix the issue. The team manager then used these estimations to insert TD issues into the development backlog in future sprints. However, sometimes these issues were forgotten in the JIRA system, and they were never repaid. The team manager mentioned that currently the development team did not spend very much time on TD repayment. Smaller TD issues were just fixed during the actual coding.

TD representation/documentation was not done in any separate backlog. When a developer took a hunch during the development, he/she would sometimes create a ticket to the JIRA management system, where information about quick solutions could be found. However, the software architect mentioned that the development team often took shortcuts that were not mentioned to the management, and this information was only stored in that developer’s own notes.

TD identification was not conducted by the development team. The manager mentioned that there was currently no systematic way to review the code and identify possible TD issues. Identification would only start when there was a clear issue and an urgent need for a fix. The software architect mentioned that he had created a menu in the development team’s WIKI page about TD issues he had identified and thought should be fixed.

TD prioritization was done by the manager and software architect. The prioritization process did not have any specific calculation to rate TD items, it was mainly done at a hunch. The team manager mentioned that when making the decision on what to replay next, factors like time, functionality, future maintenance, scalability, business value, and future plans with that feature were taken into consideration and used to give priority to various issues.

TD measurement and TD monitoring were not currently done by the manager or the software architect of the team.

TD communication was not described as good within the development team. The software architect mentioned that the development team was not currently talking about these kinds of issues during sprints. Also communication with the business architects was described as challenging, and usually the priorities they gave consisted only of development of new features, and not improvement of the code base.

TD prevention was not done by the development team. The software architect mentioned that the development process did not currently contain any coding standards or code reviews, and this was a huge problem. Everyone just used their own style of developing, and there was no consistency. The team had tried to use coding standards and code reviews before, but the usage was stopped because it was seen as time consuming. Another big problem mentioned by the software architect was that the development team did not have any proper Definition of Done to the development. The only Definition of Done was that when the solution was in production, it was considered to be ready. This was why a lot of bad solutions were created in the code base.

Overall, the TDM strategy in development team G was not organized systematically. It seemed that the management and developers did not have an explicit process of how to repay TD on a clear basis, and the development time was always put towards new feature development. The management did not have any way to measure or monitor TD, because the development team did not have any definite process to identify and document it. The development team was also lacking in the prevention of TD, by not having any coding standards or reviewing of the developed code.

4.8. Case H

TD repayment was organized systematically to conduct refactoring during the software project. The management had decided to
use a certain number of days each month for the improvement of code quality. In every month, two days were assigned for unit testing, where the developer unit tested every code of their own that had been created in the last month. Also, one more extra day of the month was dedicated to ‘your review day,’ where every developer’s code was reviewed by another developer. Also four to five days a month were dedicated for ‘your development day,’ where the goal was to improve the quality of the code base. In case a developer needed to take an intentional shortcut during the development, he/she was guided to create a JIRA issue, which would be fixed in the next sprint.

This documentation was done by using a backlog approach to document all possible TD items happening during the development. When a developer took a shortcut during the development, it was issued as a JIRA issue, which would be fixed in the next sprint.

4.9. Summary of the cases

A summary of the cases is shown in Table 4. In TD repayment, all the development teams used either retesting, refactoring or re-designing as the main process to repay TD issues. TD repayment was done during normal development and consisted of only small repayment cases or TD repayment that was done from issues assigned to the actual development backlog. Some of the development teams (Cases C, D, E, and H) had a systematic strategy to TD repayment, by assigning a certain amount of development time every month to improving code quality by refactoring or rewriting the solutions. We also identified cases (A, B, and G) that often started TD repayment when the issue started to become a problem and there was free time allocated to it.

TD identification was done during the development or with tools. In many cases, TD identification was done during the development, when a developer or software architect noticed a problem with a solution during normal development or analysis of the code base. Sometimes these identifications would happen accidentally during the development, or a software architect would spend some time with the code base to identify if there was anything important to refactor. In some cases (C, D, H) the SonarQube tool was used for TD identification.

Most of the development teams (A, B, E, F, and G) did not have or did not know a good way for TD measurement. Some of these teams (E, F) mentioned that the only TD measurement information they had was the JIRA management tool, where there was a possibility to measure and calculate how many TD issues had been assigned to the system. Some of the development teams (C, D, and G) used the SonarQube tool to measure TD in the software.

Some teams (A, B, E, F, and G) did not have systematic TD monitoring, because measuring and identifying TD was considered too difficult. Some of the teams (A, B, F, and G) used some basic information in the JIRA management tool to know how many issues had been assigned, and drew conclusions on the basis of that data. However, some teams (C, D, and F) used the SonarQube tool for TD monitoring.

We did not observe any specific calculations for TD prioritization, as prioritization was mostly based on hunches and previous experience and knowledge regarding the code base. The things that were taken into consideration when making a decision about TD prioritization were often based on scalability, business value, use of a feature, and customer effort, but they did not contain any exact numerical values.

TD communication was in a good shape in most of the cases (A, B, C, D, E, and H). The management and development team had a good TD communication structure, where the team manager had sufficient technical knowledge. However, we also saw cases (E, G), where the development team felt that the current communication about TD issues was lacking a lot. The developers felt that all the development time went to new features, and there was no time allotted by the business people to conduct refactoring of old solutions.

Almost every studied development team had set up coding standards to prevent TD. However, they were not always followed in reality, as they had been labeled as recommendations. Every development team also tried to catch bad design and solutions by implementing a code review practice to ensure the quality of the developed code before it would go to production. However, this was not always possible, because the review process was time-consuming, and effort had to be assigned to new features that were more important to the development team.

TD representation/documentation was done in three different ways: a development team with a unique TD backlog (Case E), development teams with quality/development backlogs consisting of
<table>
<thead>
<tr>
<th>Case/TDM activity</th>
<th>TD equipment</th>
<th>TD identification</th>
<th>TD measurement</th>
<th>TD monitoring</th>
<th>TD prioritization</th>
<th>TD representation/communication</th>
<th>TD prevention</th>
<th>TD representation/documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case A</td>
<td>Issues from the general development building, Refactoring during normal development.</td>
<td>Team manager and software architects identifying TD manually. Mostly during normal development.</td>
<td>No measurement</td>
<td>No monitoring.</td>
<td>Sometimes from JIRA issues.</td>
<td>Communication structure good with highly technical team managers.</td>
<td>Some coding standards and code reviews.</td>
<td>Some issues to JIRA. No separate backlogging.</td>
</tr>
<tr>
<td>Case B</td>
<td>Issues from the general development building, Refactoring during normal development.</td>
<td>Team manager and software architects identifying TD manually. Mostly during normal development.</td>
<td>No measurement</td>
<td>No monitoring.</td>
<td>Sometimes from JIRA issues.</td>
<td>Communication structure good with highly technical team managers.</td>
<td>Some coding standards and code reviews.</td>
<td>Some issues to JIRA. No separate backlogging.</td>
</tr>
<tr>
<td>Case C</td>
<td>Issues from the general development building, Refactoring during normal development. 2% of development assigned to improving the code base.</td>
<td>Software architects identifying TD manually.</td>
<td>Monitoring with JIRA.</td>
<td>Monitoring.</td>
<td>Monitoring about TD activity.</td>
<td>Communication about TD activity within the whole development team.</td>
<td>Some coding standards and code reviews.</td>
<td>Internal quality backlogging in JIRA. Not used systematically by the developers.</td>
</tr>
<tr>
<td>Case D</td>
<td>Issues from the general development building, Refactoring during normal development. 20% of development assigned to improving the code base.</td>
<td>Software architects identifying TD manually.</td>
<td>Monitoring with JIRA.</td>
<td>Monitoring.</td>
<td>Monitoring about TD activity.</td>
<td>Communication about TD activity within the whole development team.</td>
<td>Some coding standards and code reviews.</td>
<td>Internal quality backlogging in JIRA. Not used systematically by the developers.</td>
</tr>
<tr>
<td>Case G</td>
<td>Refactoring only when TD became a huge problem. Identified during normal development.</td>
<td>Identification rarely done.</td>
<td>No measurement</td>
<td>No monitoring.</td>
<td>Monitoring about TD activity.</td>
<td>Communication about TD activity within the whole development team.</td>
<td>Some coding standards and code reviews.</td>
<td>Some issues in JIRA. No separate backlogging.</td>
</tr>
<tr>
<td>Case H</td>
<td>Number of days assigned for improvement.</td>
<td>Mostly during normal development.</td>
<td>Monitoring with JIRA and SonarQube.</td>
<td>Monitoring.</td>
<td>Monitoring about TD activity.</td>
<td>Communication about TD activity within the whole development team.</td>
<td>Definition of Done to ensure code quality.</td>
<td>Issues reported to JIRA.</td>
</tr>
</tbody>
</table>

TD issues (Cases C, D, and H), and development teams not using any backlog for TD items (Cases A, B, F, and G).

5. Technical debt management framework

We developed a TDM framework based on the analysis of the eight studied development teams. The framework is presented in Table 5. The framework explains the activities, practices/tools, stakeholders, and responsibilities of TDM. After analyzing individual cases, we started to compare the cases to understand the similarities and differences of approaches and practices in TDM activities. We took all the approaches and practices found in the analysis and put them into the same table (Table 4) to understand how each activity was conducted in general, across the cases. We observed that all practices had a defined responsibility. We were
Table 5

<table>
<thead>
<tr>
<th>TDM activity/TDM levels</th>
<th>TD repayment</th>
<th>TD prevention</th>
<th>TD representation/ documentation</th>
<th>TD identification</th>
<th>TD measurement</th>
<th>TD monitoring</th>
<th>TD communication</th>
<th>TD prioritization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Organized (Level 1)</strong></td>
<td>Continuous repayment with monthly percentage of the development tasks.</td>
<td>Mandatory practices used by the team.</td>
<td>Documentation as a mandatory practice in development. Issues are documented in separate TD hacking.</td>
<td>Continuous identification during development.</td>
<td>Continuous measurement during development with various data (e.g., quality, performance).</td>
<td>Continuous monitoring and/or with tools during development.</td>
<td>Continuous communication about TD issues and in all the necessary stakeholders involved.</td>
<td>Prioritization conducted continuously during development.</td>
</tr>
<tr>
<td><strong>Received (Level 2)</strong></td>
<td>Repayment during normal development tasks and previously identified repayment tasks</td>
<td>Opitional prevention practices.</td>
<td>Documentation as optional practice in development. Issues are left to developers’ own minds and notes.</td>
<td>Identification optional during normal development. Conducted based on current time constraints.</td>
<td>Measurement of optional practices.</td>
<td>Measurement done with simple data (number of TD issues) from development, and the data not necessarily used for other activities.</td>
<td>Monitoring not part of development practices.</td>
<td>Discussions/meetings organized only with some stakeholders.</td>
</tr>
<tr>
<td><strong>Unorganized (Level 1)</strong></td>
<td>Repayment not conducted at all or only when it is not possible to avoid the issue anymore</td>
<td>Development team, software architect(s), development team, software architect(s)</td>
<td>Technical debt management. Use of meta tasarım tools (JIRA, Wiki).</td>
<td>Team manager, software architect(s), team manager</td>
<td>Data from measurement tools (SonarQube) and from project management tools (JIRA, Wiki)</td>
<td>Monitoring tools (SonarQube) and project management tools (JIRA, Wiki)</td>
<td>Monitoring not part of development practices.</td>
<td>TD not a topic in discussions/meetings and often handled only in coffee table discussions.</td>
</tr>
<tr>
<td><strong>Practices/tools for activity</strong></td>
<td>Refactoring, redesigning, rewiring</td>
<td>Coding standards, code reviews, Definition of Done.</td>
<td>Time reservation for manual code inspection. Use of code analysis tools (SonarQube, CheckStyle, FindBugs).</td>
<td>Software architect(s), team manager</td>
<td>Specific TD meetings. TD included in discussion topics.</td>
<td>Monitoring tools (SonarQube) and project management tools (JIRA, Wiki)</td>
<td>Monitoring not part of development practices.</td>
<td>TD not a topic in discussions/meetings and often handled only in coffee table discussions.</td>
</tr>
</tbody>
</table>

The results indicated differences in the maturity of TDM. By the term maturity we mean the capability of the development team in TDM activities. Firstly, we identified cases where a TDM activity was not at all conducted during the development. We defined this as the lowest level of maturity, where a development team does not conduct a particular TDM activity. Secondly, we identified development processes where TDM activities were organized and conducted continuously by the development teams as a part of their normal development process. We defined this as the highest level of maturity, where the TDM activity is an integral part of the continuous development process. These two extremes were identified as the lowest and highest levels of maturity. TDM activities that were conducted only sometimes and were not considered an important part of the continuous development process, were placed on a level between these two extremes. We used these maturities to assign every identified TDM activity with their own maturity levels. On the basis of the process described above, we developed a TDM framework divided into five sections: TDM activities, TDM levels, TDM stakeholders, TDM responsibilities, and TDM approaches. We use the eight activities identified by Li et al. (2015a) as TDM activities in the framework. The TDM activities are TD repayment, TD prevention, TD documentation, TD identification, TD measurement, TD monitoring, TD communication, and TD prioritization. Based on our findings in the studied development teams, we believe that the eight TDM activities are suitable for giving an overall view on TDM. During the analysis of the cases, we identified some level of approach in each TDM activity. In addition, we were not able to identify any new TDM activities during the analysis of the cases. The analysis revealed that the TDM activities were conducted at different maturity levels. For example, we observed that while one development team focused on and put effort to measurement and monitoring activities, another development team did not put any effort to them. We defined three TDM maturity levels: unorganized, received, and organized. A TDM activity can be considered unorganized when a software team does not put any effort to the activity or when the focus is minimal. A TDM activity can be considered received when the software team has acknowledged the need for a certain TDM activity and when it already conducts it on some level. However, the activity is not yet considered as a constant one and only a few people conduct it occasionally. A TDM activity can be considered organized when the team understands the need and importance of the activity and the team is continuously working on conducting it. This level requires more effort and resources than the previous levels. Moreover, it is important to note that the levels are not mutually exclusive and can be combined.
be considered as organized when the development team has recognized the TDM activity as an essential part of software development, and it is conducted continuously by the whole development team.

We identified three main stakeholders and one additional stakeholder related to TDM. These stakeholders came from the responsibilities found in the analysis. The first is the development team, which is responsible for software development. The development team is often responsible for the TDM activities that take place during the actual development of software. These activities are TD repayment, TD prevention, TD documentation, and TD measurement. The development team works with the code base, and is able to identify and refactor possible issues in the software. The development team is also responsible for TD prevention in terms of following coding standards and code review practices.

The second stakeholder is the software architect, who is responsible for the architecture of the software. The software architect has responsibilities in all TDM activities. The cross-case analysis revealed that software architects often act as a central mind in TDM. This is because software architects often have the best overview on the software and its design issues. Therefore, all TDM activities should be within the responsibility of software architects.

The third stakeholder is the team manager, who is responsible for managing the development. We observed that the team manager was mainly responsible for four TDM activities: TD prioritization, TD communication, TD monitoring, and TD measurement. The team manager did not often deal with activities that are directly related to technical development, and therefore his/her responsibilities was only on the management activities that required data collection as well. The team manager has a lot of communication with the business stakeholders to understand what direction the software is evolving. Therefore, the manager is highly involved in the communication about TD, when there is a need to change the software to a certain direction. This also has an effect on the prioritization of TD, because business changes need to be evaluated with TD issues, to understand what kind of development effort the software will need in the future.

An additional stakeholder is the business stakeholder who communicates about the software needs to the team manager. Business stakeholders are not necessarily directly related to TDM, but the needs coming from the business stakeholders do have an effect on the TDM activities. The business needs e.g. for a new feature may change the current TD repayment activity or TD prioritization.

We also identified various approaches for each TDM activity. The approaches varied from practices conducted by the whole development team to practices conducted by a single person. We also made observations about the tools used to support the TDM approaches. The practices, models, methods, and tools are presented in the framework in the section approaches to activity.

The framework can be used by software development companies to improve and evaluate internal and external processes regarding TDM. However, we cannot claim that working on the highest level of the TDM framework will reduce TD or produce healthier software. It is possible that a development team conducting refactoring only when necessary has less TD in their software than a development team that conducts all TDM activities continuously. Instead, we believe that using the framework will increase the visibility and knowledgeability regarding TD in the software, which can be used for smarter and safer decisions in TD reduction and management.

It is also important to mention that this framework is presented only at a high level, and it has been derived from the eight studied software development teams. Therefore, other researchers should improve this framework by adding approaches, responsibilities, levels, and activities that were not included in this study.

6. Discussion

6.1. RQ1.1-2: What TDM activities are used in the studied development teams? What methods, models, practices or tools do the studied development teams use for each TDM activity?

6.1.1. Commonly used activities

6.1.1.1. Communication. The most usual TDM activity in the studied development teams was communication. TD was an important discussion topic in most of the development teams. This is not a surprise, considering the popularity of TD research in the past few years (Li et al., 2015a). The biggest issue with TD communication has been the gap between technical and non-technical stakeholders (Klinger et al., 2011). Communication related to TD issues does not often transfer from the development team to the business stakeholders, which leads to TD issues not receiving the required time to get fixed (Yli-Huumo et al., 2014). Our observations also support the fact that the starting point for successful TD is good TD communication. If the development team does not have any communication of TD, it is difficult to gain any benefit from the other TDM activities. Most of the studied development teams had organized TD communication successfully, which also helped in the other TDM activities. Simply taking TD as a topic in various meetings and discussions between the stakeholders can already improve TD communication. Especially a product manager with high business and technical competence can work effectively as a middle-man between business stakeholders and development teams, and improve communication related to TD.

6.1.2. Occasionally used activities

6.1.2.1. Repayment. TD repayment was conducted with refactoring, rewriting, and redesigning practices in the studied development teams. Similar practices identified in a study by Cod hoch and Williams (2013) were reengineering and repackaging. Even though all the practices mentioned for TD repayment had a similar goal of improving the solutions in the code base, it is still important to understand that they were not the same practices. Refactoring, which is a known concept in the literature and probably most commonly used technique for code improvement, can be described as a practice to improve code structures without changing the existing functional behavior of a program (Fowler et al., 1999). Redesigning can be an act to change the solution for example with a better and faster algorithm, while rewriting is an act to reimplement a large portion of an existing solution without re-using the previous source code. It is important to understand the differences between the concepts. Using refactoring as a term to describe large-size rewriting of a software feature can be misleading for some stakeholders in the development. Understanding the differences between TD repayment practices can improve especially TD communication, when all stakeholders understand the nature of the required improvements and the resources needed.

There are many strategies for conducting TD repayment. A development team can either choose to repay TD continuously, occasionally, or not at all. The decision to choose the repayment strategy emerges from the question “do we have technical debt?” In a case where the development team is fighting with a large TD, it would be wise to have a systematic way to repay TD back continuously to avoid a crisis in the future. In a case where the development team has only little known TD, it is possible to repay TD occasionally e.g. during normal development. Development teams can also choose not to repay any known TD, if they do not see any good reason for it.
Companies react differently to TD repayment. Some teams opt to reduce TD by a certain percentage every month, while some teams opt to focus on new features, and leave TD reduction to minimum (Project, 2013). Our observations in TD repayment strategies suggest that there is not necessarily one right TD repayment strategy and practices. The decision for the strategy has to be made on the basis of the current needs and understanding of the significance of TD in the software product.

6.1.2.2. Prevention. TD prevention activities happened only occasionally during development. Practices used for TD prevention include checking code, setting coding standards, and conducting code reviews. A set of other practices for TD prevention have been identified in other studies (Codabux et al., 2014; Krishna and Basu, 2012). These practices include approaches such as education and training, pair programming, test-driven development, refactoring, continuous integration, conformance to process and standards, tools, and customer feedback (Codabux et al., 2014). Code reviews, where another developer checks your code can be used to prevent bad solutions from getting to the code base (Baker et al., 1997; Knüfer and Paulk, 2009), while setting up coding standards/guidelines for the development team to ensure as much cohesion as possible during the development (Green and Ledgard, 2011) can improve understandability and learnability.

TD prevention can be seen as one of the most influential activities of the eight TD activities that a development team can conduct. When the development team has set up mandatory coding standards, assisted with e.g. code reviews and Definition of Done practice, it is possible that the amount of TD that gets to the code base will decrease (Davis, 2013). When TD is prevented as much as possible, it also helps other TD activities. In addition, setting up TD prevention practices helps especially in catching unexperienced developers’ ‘not-so-good’ solutions.

Even though the benefits of TD prevention are quite clear and simple to implement in real-life software development, we observed that they are still not necessarily used. The biggest issue is that they were conducted only occasionally, because they are not mandatory. The software development teams in this study mentioned having coding guidelines and reviews set up, but they were not often used. There are possible reasons for the development teams not using TD prevention practices. First, working with strict standards and guidelines in software development can sometimes be exhausting and annoying for developers, when they are not allowed to use their own creativity in the development, but must follow strict guidelines instead. Second, adopting TD prevention practices requires resources. Using various TD prevention practices requires time and competence, which are always taken away from something else.

6.1.2.3. Representation/documentation. TD representation/documentation was conducted only occasionally. There can be several reasons for why developers do not conduct documentation. In tight schedules documentation is often not seen as a useful practice, and therefore writing TD documentation can be seen as waste of time. Developers may also value documentation differently, and they document only issues that they personally think are important (Lethbridge et al., 2003). The biggest reason why TD representation/documentation was lacking in our cases was that TD was not generally considered as something that could be documented.

The development teams had a variety of approaches for documenting TD. Some teams had a specific TD list, which consisted of TD issues only and nothing else. Some teams used a normal development backlog as the place to store TD issues. The tools used for these two approaches were JIRA and Wiki, which made the data available for everyone. There were also teams that did not use any documentation for TD issues, and just decided to leave them as common knowledge in the development team.

We believe that TD representation/documentation is essential for a successful TD management. When TD issues are not stored, it is highly possible that they will be forgotten at some point and will never be repaired. Without proper tracking and documentation of architectural change requests, it is also extremely challenging to quantify TD (Klinger et al., 2011). The inability to quantify TD also creates more challenges to other TD activities, such as communication, repayment, monitoring, and measurement due to the lack of TD data.

Documentation is a valuable practice that improves understandability and communication (Dai et al., 2007; Forward and Lethbridge, 2002). Therefore, adopting even a simple documentation practice for TD representation/documentation improves other TD activities and the overall TD strategy. A systematic process to document and store all the TD issues can be used for creating a systematic TD repayment strategy (Lin et al., 2012).

6.1.2.4. Identification. TD identification was conducted occasionally during the development. In manual identification a person tries to locate the sources of a TD problem. Also tools can be used to find bad code. Most of the identification in the studied cases was conducted manually because of lacking tools or knowledge about them. Some development teams used tools like SonarQube, Check-Styles, and FindBugs to scan the code base to find possible complexities and badly developed code.

TD can be completely different for different development teams. Some development teams consider smaller issues, such as bugs or single line errors, to be TD. These types of smaller errors are simpler and easier to fix and they can be found with tools like SonarQube, Check-Styles, and FindBugs or with manual inspection. However, the challenge in identification is that TD is not just related to simple errors, but especially to design issues of software. It is challenging to identify this type of TD with tools. The challenge is how the tools tackle architectural or structural issues and technology gaps (Kruchten et al., 2012). This was also mentioned by the architects and developers who did not have any tool available to find the types of issues that required manual identification. This issue has also been raised in a previous TD study (Zaworkska et al., 2014), questioning how TD issues could be identified from the code base. It seems that TD identification is often done during the actual development, where a developer notices that something bigger might be wrong in some part of the code base. An interesting question related to TD identification is whether developer-identified TD should be considered as “real TD”, while tool-identified TD should not, because it is not necessarily related to the effects of external (such as customer and market) changes in the software architecture (Zaworkska et al., 2013).

6.1.2.5. Prioritization. Another occasionally used activity for the development teams was TD prioritization. When TD issues were identified, there was no precise model or method used to calculate or estimate the effects or costs of the TD. The literature has suggested approaches for TD prioritization (Eisenberg, 2012; Scanlan et al., 2012; Theodoropoulou et al., 2011; Zaworkska et al., 2013). Some of the approaches are based on calculating technical values (e.g. duplicate code, test coverage, rules compliance, code comments etc.), some take aspects from the finance environment, such as cost-benefit analysis into consideration, while some use software quality attributes for the evaluation.

In our cases, the estimation and prioritization was just based on a hunch and previous knowledge of the person. The reason was
that calculating technical things like scalability and further maintenance is extremely difficult, as business items like plans and business value have to be considered as well. Therefore, the prioritization was often assessed on a low/medium/high scale or using story points to estimate the importance and effort of TD, based on hunches and rough estimations.

Kramarž et al. (2015) describe TD prioritization with three dimensions: customer satisfaction needs, reliability demanded by the business, and probability of technology disruption. These dimensions are essential for decisions, but quantifying these with exact numbers is extremely difficult. Prioritization can also be based on customer feedback from the technical perspective of sight (Codan and Williams, 2013). These prioritization issues exist also in requirements prioritization (Lehtola and Kauppinen, 2006).

6.1.3. Rarely used activities

6.1.3.1. Measurement. TD was measured rarely in the studied cases. The only identified measurement practices used either data available in project management tools (JIRA, Wiki), or a specific tool to measure TD (SonarQube). The data gathered from JIRA consisted usually of simple data only (reported TD issues, number of bugs etc.), which was used to get some level of understanding about the status of TD. The usefulness of this data could be questionable. For example, a decrease of TD issues from 50 to 48 in one month does not necessarily mean that TD has been reduced, because there may exist unidentified TD issues. Some development teams used tools, e.g. quality and productivity as a measurement to see in which direction the software was going.

The data gathered with tools (e.g. SonarQube) provides an estimate of TD based on calculations. This type of data could be easier to interpret in development and management. For example, SonarQube calculates TD from seven deadly sins (SonarQube, 2015), each one representing a major quality issue: bad distribution of complexity, duplications, lack of comments, coding rule violations, potential bugs, no unit tests or useless ones, and bad design (SonarQube, 2015). Some of the development teams in the studied cases used this value to get an estimate of TD, which was followed during development.

An estimate based on a tool should be more accurate, faster and reliable compared to an estimate based on simple data. However, TD monitoring tools require a clear vision of what technical debt do you want to estimate? When a development team considers for example the criteria in SonarQube (2015) as TD, it can guide TD management and other TD activities. However, TD can also be considered to consist of issues of a larger scale, such as architectural or structural issues and technology gaps (Knecht et al., 2012a). There are not necessarily any automatic tools available to measure these issues of a larger scale.

This can be currently seen as the biggest problem and challenge in TD measurement. There are no valid tools to measure large TD issues related to the deep architectural structures of software. Therefore, most TD measurement is done on the basis of human evaluation, which can be seen as a challenge especially in decision-making.

6.1.3.2. Monitoring. Similar to TD measurement, TD monitoring was also conducted rarely. The lack of TD monitoring is also related to the rare occurrences of TD measurement. Without any measurable TD data from the software, it is also almost impossible to monitor anything related to TD. Most of the TD monitoring was based on data derived from project management tools (JIRA, Wiki) or specific tools to measure TD (SonarQube). The team members responsible for monitoring TD used this data to monitor how TD was increasing or decreasing during the development, and used that information to assign work in other TD activities.

TD monitoring and tracking is one of the most vital TD activities (Ernst et al., 2015). Without monitoring, the development team is not able to have any reason for other TD activities. One of the questions related to TD monitoring that can be seen as major obstacle is “what should you monitor?” Tools may help in estimating technical aspects, such as bad distribution of the complexity, duplications, lack of comments, coding rule violations, potential bugs, and lack of unit tests. However, an essential part of TD monitoring is also monitoring the overall quality of the software and the productivity of the development team. Evaluating how a large-scale architectural change affects the developers’ productivity or the overall quality makes it possible to reason why some TD issues are important to repair or not.

6.2. RQ1.3: Are there any maturity differences on adopting TD activities between development teams?

In some development teams TD focused on only two to three activities, while some development teams conducted all eight TD activities. Some development teams opted to use tools for the activities, while some teams did not have knowledge of available tools. Some development teams opted to conduct activities continuously, while some teams did it just occasionally.

The biggest maturity differences were in TD activities that were conducted mostly by the development team (reorganization, prevention, representation/documentation, and identification), while the least differences were in activities done mostly by the software architects and the team manager (measurement, monitoring, prioritization). We suggest that this was because the activities conducted mostly by the software architects and team managers were considered the most challenging, and there was not necessarily many known tools or practices available, which resulted in the fact that activities were not often conducted.

6.3. RQ1.4: What are the biggest challenges in TD?

6.3.1. Lack of tools

One of the main challenges in TD is the lack of tools. TD was mostly conducted as human activity, instead of using automated or enabling tools. Ernst et al. (2015, p.7) state that developers “desire standard practices and tools to manage technical debt that do not currently exist.” If most of the current TD activities are done with rough estimations and are based on hunches, instead of tools and models based on precise data from specific tools, there is a risk that the choices made for TD reduction and management are not always the most optimal ones. In addition, conducting TD activities without tools is time-consuming, and the addition of tools would provide faster TD activities.

As Ernst et al. (2015, p.7) comment, “testing is a necessary component of any technical debt management strategy”, we also believe that an important research area currently in TD is the research and development done for tools targeted to tackle different TD activities. The development of new tools especially for identification, measurement and monitoring activities can and should be beneficial and should be in a high priority in future research related to TD.

6.3.2. Knowledge of TD priorities

Unlike the challenge with tools, TD prioritization is not necessarily as much dependent on tools, even though TD prioritization needs data input from other TD activities to support the decision-making. However, one of the current main challenges of TD is TD prioritization. The challenge is the lack of models and methods to prioritize TD issues successfully. There are no proper solutions to understand and explain why some TD items should be a priority to the development team over other TD items. Some
type of technical debt can be important for a development team to fix, while a similar type of technical debt is not seen as a problem for another team. Some papers (Eisenberg, 2012; Seaman et al., 2012; Theodoropoulos et al., 2011, Zacrowska et al., 2014) discuss how TD issues should be prioritized on various levels. They include ideas and suggestions of how to prioritize TD issues, but they have not been thoroughly tested empirically, or they do not take all the aspects related to TD prioritization into consideration, including both technical and business needs. In prioritization both technical and business needs need to be covered.

We believe that TD prioritization as an activity is currently lacking in development teams, and both technical and business needs of TD into consideration. The development teams in our study had a hard time prioritizing TD issues, because they had no clear and method for doing it properly. Prioritization was mainly done just by the opinions of single persons, based on hunches and previous experiences, instead of estimations and measures based on some precise data. There are cases where developers may have an idea of how to improve some part of the architecture to decrease complexity or increase velocity. However, if this improvement in architecture does not bring any value for the customer, it may not be prioritized as high as it should be from the technical perspective. On the other hand, a minor TD issue with lots of work and a high value to a customer could be prioritized high, since it has business value.

This is a current challenge in TD, because knowledge about the most important TD issues to fix may be missing, which may result in wrong decisions. The development of new models or methods for TD prioritization would help development teams to explain to the business people the real benefits of technical improvements more clearly, based on exact values (e.g. time, quality, maintenance, productivity, business value).

6.3.1 Having a proper mindset with TD

One of the challenges is the mindset of the developers. The goal of TD is to provide practices and tools to manage and reduce TD in software development (Ji et al., 2015a). This obviously requires more effort on the already existing practices of tracking down and fixing issues to make technical improvements. Conducting TD takes time, and it will have an effect on other software development activities. Instead of designing and developing a new feature, it is more useful to identify a badly designed code manually. There is a possibility that some stakeholders see this as a waste of time. Therefore, the mindset towards TD can sometimes be negative, and the developers or managers just want to focus on developing something new, which will lead to the use of hotfixes and quick solutions.

One of the challenges in TD is to get the whole organization/team included in TD with a proper mindset. Instead of only a few people documenting TD issues to the backlog or taking part in TD communication, it is important that every member of the team contributes to TD. This way all the TD activities will support each other successfully.

6.3.4 Time-consuming TD

We also observe that TD is time-consuming. Adopting new TD processes and tools can create more work on top of the existing development process. Therefore, it may difficult to justify the real need for TD and its benefits. For example, why should the development team have mandatory coding reviews or documentation practices, if they take time away from other important development practices, and there is no guarantee that they would provide immediate benefits? In addition, conducting e.g. manual code inspection takes a lot of time, and its benefits are uncertain. Therefore, adopting activities that require more time and resources to be successful, can be hard to justify.

This is the reason why there is an urgent need to provide more evidence of TD. Doing research on the benefits of conducting code reviews, on how documentation helps in TD visibility, or how manual code inspection can offer a possibility to detect serious architectural issues, can bring justification for the reasons to have TD, which will give confidence to the development teams to allocate more time and resources for TD.

6.4 Limitations of the study and threats to validity

6.4.1 Generalization of the results

A case study does not provide statistical generalizability (Yin, 2003), i.e. a case study with a limited number of cases cannot be generalized over a population. We, however, consider generalization as theoretical (Lee and Buskirk, 2003), i.e. abstraction from concrete events and actions to theoretical constructs. Case studies are generalizable to theoretical proportions, not populations or universes. We believe that the theoretical implications of this study are needed for creating a more focused approach to TD.

6.4.2 Construct validity

The threats to the validity of a case study can be divided to four aspects: construct validity, internal validity, external validity, and reliability (Runeson and Höst, 2009). Construct validity reflects to what extent the operational measures that are studied really represent what the researcher has in mind and what is investigated according to the research question (ibid., p. 153). To improve construct validity in this study, the data collection protocol was reviewed, discussed, and corrected if necessary by all the authors. During the interviews, we also put a lot of emphasis on the explanation of each research question, and tried to improve the fact that both the interviewers and interviewee had similar understanding of the research topic. In addition, most of the interviews were conducted by two authors. This increased the possibility for the other interviewer to correct misunderstandings during the interviews. We also let the interviewers review the first draft of the paper, in order to identify issues in construct validity.

One limitation of the study is the difference in the interview structure between the first and second round interviews. As the first round interviews were conducted roughly one year before the second round interviews, and the interview structure was changed between the rounds, the collected data was not congruent. The first round interviews were analyzed first with a different data coding protocol, but we reanalyzed them afterwards with the same data coding protocol as with the second round interviews, to ensure the same coding process.

6.4.3 Internal validity

Internal validity is a concern when causal relations are examined. The concern is being certain that when a causality between x and y is found, factor z is not included, which we did not identify during the interviews (Runeson and Höst, 2008). Improving internal validity in case studies is challenging, because it is sometimes hard to know if there is some underlying reason for the causality. We used semi-structured interviews to gain more in-depth knowledge related to the data in the studied cases. Therefore, when we were not completely satisfied with the gained data, we could ask more specific questions to understand the factors related to the causality better. In addition, we were also able to communicate with the interviewees after the interviews, if we had some smaller additional questions about issues related to the data analysis.

6.4.4 External validity

External validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case.
(Runeson and Höst, 2008, p. 154). One limitation of this study was the number of the studied software development teams and the fact that all of them were from the same organization. Obviously, adding more software development teams from several other organizations, the theory and framework could be possibly extended by adding new data. The goal of this study was not to create a complete and generalizable framework for TDM. Instead, the goal was to understand how the selected software development teams were managing TD in their current development environment. Therefore, the developed framework is not necessarily generalizable, because the data was derived only from one organization. However, the framework can be used for future research, and it can be improved and extended by adding new data from other empirical sources.

6.4.5. Reliability

Reliability is concerned with ‘to what extent the data and the analysis are dependent on the specific researchers’ (Runeson and Höst, 2008, p. 154). One limitation of this study is the semi-structured interview approach. In the semi-structured approach, the interview questions are often open-ended. Therefore, the answers from different interviewees can vary a lot, and the discussion during the interviews can be different in each interview session. In a situation where another researcher conducts the study, the data from the interviews will not necessarily be exactly the same. However, we improved the reliability of the study by designing and describing the data collection, data coding, and data analysis process carefully, which makes it more repeatable to other researchers.

6.5. Implications for future research

On the basis of our findings we believe that TDM in software development has similarities to the characteristics of the capability maturity model (CMM) (Paulk et al., 1993). There are similar differences in the maturity of TDM across projects and companies. The CMM was originally developed to present a set of recommended practices to enhance software development and maintenance capability. The fundamental concepts of CMM are capability, performance, and maturity. The five levels in CMM are initial (fuzzy), repeatable, defined, quantitatively managed, and optimizing (Paulk et al., 1993). A similar maturity model to CMM is also applicable in TDM, where development teams have different TDM maturity levels in activities and practices. This kind of maturity as a concept has been applied to other processes and domains as well (De Bruin et al., 2005).

It is important to point out that our results do not show if there are any advantages or disadvantages in using some specific approaches or their combination. The success of TDM is not necessarily related to the number of approaches that a development team uses. It is possible that development teams conducting refactoring only when it is necessary have a less TD than development teams that monitor and measure TD constantly. However, we believe that having defined and structured TDM activities and approaches can increase the visibility and knowledge regarding TD in software and projects. Therefore, we see the development of the TDM maturity model beneficial for both practice and research. Future research could focus on identifying TDM maturity levels and developing a practice-oriented maturity model, to improve the visibility and manageability of TD in software projects.

7. Conclusion

This study explored how software development teams manage technical debt in a real-life environment. We used the exploratory case study method suggested by Runeson and Höst (2008) to study eight software development teams in one large organization. For the analysis of technical debt management, we used the eight activities identified by Li et al. (2015a). We interviewed 25 persons to identify the processes, techniques and tools used for technical debt management.

We found that technical debt management was conducted at various levels. Some of the teams did not have any clear strategy or tools to manage and reduce technical debt, while some teams had defined structured processes to reduce, monitor, measure, and manage their technical debt. We also observed that there exist several challenges of technical debt management, which software development teams have to understand and acknowledge.

The study produced a technical debt management framework that describes the management activities, stakeholders and responsibilities on three levels and approaches/practices/tools used in them. The framework can be used for the definition of activities included in TDM, and how the activities are divided between the stakeholders.

Technical debt management has many similarities with the capability maturity model (CMM). We believe that the developed framework can serve as the basis element for researchers and practitioners in the development and improvement of technical debt activities.

Acknowledgments

The authors would like to thank the company and their employees for participating in this research. The research has been carried out in the Digile 4 Speed program, and it has been partially funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

Appendix A

1. General information
   1.1 Respondent’s name:
   1.2 Email:
   1.3 Role in company:
   1.4 Responsibilities:
   1.5 Company name:
   1.6 Organizational unit:
   1.7 Industry sector:
   1.8 Number of employees:

2. Technical debt
   2.1 Have you heard of the term technical debt before?
   2.2 Have you experienced situations where you had to take shortcuts in your projects, for example writing a code of lower quality or skipping a run of test cases to meet deadlines, and decided to fix them later?
   2.3 Describe examples of shortcuts (technical debt) in your projects.
   2.3.1 What kind of effect did they have right after?
   2.3.2 How did they evolve during the software life cycle?

   - Poor customer responsiveness?
   - Long delivery times?
   - Late deliveries?
   - Lots of defects?
   - Rising development costs?
   - Frustrated and poor performing teams (bad productivity)?

   2.3.2 What were the main reasons for you having to take these shortcuts?
   2.3.3 Did you ever fix or make better the shortcuts you took?
   2.3.4 Did you learn anything from these examples? Would you take the same shortcuts again? Why or why not?
2.4 Have you ever taken shortcuts in development because of pressure from business people or a customer due to deadlines?

2.5 Have you ever been “forced” to take shortcuts in a situation where business people did not necessarily understand the concept of technical debt and its effects on the project, and you thought it was a bad idea?

2.6 Are you willing to take shortcuts in development that will not cost much now but will cost more in the future, to meet the deadlines?

2.7 What is the business manager’s opinion usually about taking these kinds of shortcuts?

2.8 How do you communicate between different organizational units about taking shortcuts in a project? Do you communicate about it with the customer?

2.9 How do you make decisions regarding taking shortcuts on projects?

2.10 Do you have any strategies as regards managing or reducing these shortcuts?

2.11 Do you think business people should include these kinds of shortcuts in their business strategy and budget?

2.12 How do you ensure that the quality level of your code is high and easily changeable to maintain?

2.13 How often do you do refactoring? Do you inform about it to the business people? What is their reaction to it?

2.14 What do you think are the positives and negatives of taking shortcuts?

2.15 Do you have any ideas on how your company (or companies in general) should take care of managing, finding, reducing and paying shortcuts?

2.16 Do you think technical debt actually exists? Is it a serious threat to software companies and should they pay more attention to it?

2.17 Do you think all shortcuts are bad and must be paid for at some point? How would you describe the difference between a good and a bad shortcut?

2.18 What software development methods or models do you use?

2.19 Have you used any other methods?

2.20 Do you think that there are differences between methods as regards taking shortcuts? Is it easier to manage with one or the other?

2.21 Do you have any other thoughts, comments, suggestions of what you have learned about technical debt / taking shortcuts in development what you would like to share?

Appendix B Interview introduction

- Respondent’s name:
- Respondent’s name:
- Email:
- Role in company:
- Responsibilities:

Introduction to the case

- Case history
  - What is the history of this team/case?
- Product history
  - What is the history of the product?
  - What has changed during the history?

Stakeholders of the case

- Describe what teams are included in this case (development, management etc.)
- What are the sizes of the teams?
- Are there any problems with technical debt?
- Your team was interested in studying the topic of technical debt, so do you have some kind of a problem currently with technical debt?

Reasons for technical debt

- Intentional technical debt: Strategic decisions to incur technical debt during a project.
  - Do you have any examples of intentional technical debt in this case?
  - Why was the intentional technical debt taken?
- Unintentional technical debt: Lack of practices to retain the code quality level
  - Do you have any examples of unintentional technical debt in this case?

Software development methodologies/processes/tools

- What software development methodologies are you using (waterfall vs. agile)?
- Do you think that software development methodology has any effect on technical debt?

Effects of technical debt

- How does technical debt affect you?
  - Time-to-market?
  - Lack of productivity?
  - Lack of quality?
  - Extra work?
  - Bugs/Errors/Defects?

Management of technical debt

- TD repayment
  - How are you repaying technical debt back?
  - How has refactoring been organized in your team?
  - Do you refactor only when it is necessary or do you have a plan for it?

- TD identification
  - How do you identify technical debt?
  - Do you have any tool for it or do you do it manually?

- TD measurement
  - How do you measure technical debt?
  - Do you have any tool for it or do you calculate it manually from somewhere?

- TD monitoring
  - How do you monitor technical debt?
Do you have any tool for it?

• TD prioritization
  • How do you prioritize technical debt?
  • Do you prioritize on a branch and experience, or do you have a model/method for it?

• TD communication
  • How have you organized communication about technical debt?
  • Do you discuss technical debt often with the whole team?

• TD prevention
  • How do you prevent technical debt?
  • Code standards?
  • Code reviews?
  • Definition of Done?

• TD representation/documentation
  • Do you document technical debt issues in any way?
  • Do you have a separate technical debt backlog?

• Improvements for current technical debt
  • Possible suggestions for improvements

• How would you like to improve your current practices regarding technical debt management?
  • Is there anything else you would like to say?

References


Jesse Yli-Huumo is a Ph.D. student in the Department of Innovation and Software at Lappeenranta University of Technology, Finland. His research interests include technical debt, process improvements and software development methodologies. Yli-Huumo has a M.Sc. (Tech) in software engineering from Lappeenranta University of Technology.

Andrey Maglyas is a post-doctoral researcher in the Department of Innovation and Software at Lappeenranta University of Technology, Finland. His research interests include software product management, process improvements and management methodologies. Maglyas has a D.Sc. (Tech) in software engineering from Lappeenranta University of Technology and a M.Sc. (Tech) in management of information systems and resources from Saint-Petersburg State Electrotechnical University, Russia.

Kari Smolander is Professor of Software Engineering in Department of Computer Science, Aalto University, Finland. His current research interests are in the area of software development practices and includes especially the ongoing change in software and systems development practices and software development organizations related to digitalization. Smolander has a Ph.D. (2003) in Computer Science from Lappeenranta University of Technology, Finland.
Publication V


© 2016, Springer International Publishing Switzerland. Reprinted with permission
Developing Processes to Increase Technical Debt Visibility and Manageability – An Action Research Study in Industry

Jesse Yli-Huumo\(^1\), Andrey Maglyas\(^1\), Kari Smolander\(^2\), Johan Haller\(^3\), and Hannu Törnroos\(^4\)

\(^1\) Lappeenranta University of Technology, Lappeenranta, Finland
jesse.yli-huumo@aalto.fi, maglyas@gmail.com
\(^2\) Aalto University, Espoo, Finland
kari.smolander@aalto.fi
\(^3\) Tieto Sweden AB, Stockholm, Sweden
johan.haller@tieto.com
\(^4\) Tieto Oyj, Helsinki, Finland
hannu.tornroos@tieto.fi

Abstract. The knowledge about technical debt and its management has increased in recent years. The interest of academia and industry has generated many viewpoints on technical debt. Technical debt management consists of technical and organizational aspects, which make it a challenge in software development. To increase technical debt visibility and manageability, new processes must be developed and thoroughly empirically tested for their applicability. In this paper, we use the action research methodology to design processes for identification, documentation, and prioritization of technical debt. Our partner in this research is a large Nordic IT company Tieto, currently in a need for new ways to improve their technical debt management. The results include a set of processes and templates that were successfully used to identify and document technical debt. The identified technical debt items were later prioritized based on evaluation by Tieto employees. Tieto was able to create a prioritized technical debt backlog, which is now used for reduction activities to create a healthy and sustainable product for the future.

Keywords: Technical debt · Technical debt management · Software process improvement · Action research

1 Introduction

Technical debt refers to a situation in software development where shortcuts and/or workarounds are used in technical decisions to gain time-to-market [1]. The benefit of taking technical debt is an earlier and faster release, which can lead to customer satisfaction and other economic advantages [2]. However, the drawback is the ‘debt’ that is left in the system. In the long-term, shortcuts and workarounds will turn to unnecessary complexity (interest) in the source code and architecture. Complexities in software can become hard to fix and change, which may cause decrease in software…
quality and productivity of the development team [3]. Therefore, technical debt can be a major problem for a software development company.

While shortcuts and workarounds can be seen as intentional decisions to speed up release cycles, or to circumvent a complex part of the code, unintentional technical debt occurs without immediate awareness [4]. Unintentional technical debt is introduced to software, for example, by inexperienced developers or legacy software. An inexperienced developer can create technical debt unintentionally with non-optimal solution. Old legacy software can consist of obsolete or non-optimal technology and solutions from past decades, which may require a rewrite or replacement.

Technical debt management refers to activities that are used to manage and reduce both intentional and unintentional technical debt with various approaches, practices and tools [5]. Technical debt management not only includes technical development activities but also organizational ones, such as communication and decision-making.

This study is made in cooperation with one of the largest IT companies in Scandinavia, Tieto. Tieto’s Capital Market product unit is currently planning new processes for their technical debt management. The goal of the study is to develop new processes for technical debt identification, documentation, and prioritization. The outcome of this study includes new processes to increase the visibility and manageability of technical debt, which can be used in the future for better decision-making.

This paper is limited to studying technical debt that has already been acquired previously, and does not take in consideration the management activities related to decision-making process of acquiring new technical debt.

2 Background

Processes for technical debt management have been studied and suggested in the literature. Li et al. [5] gathered in a mapping study relevant research on technical debt management. The study showed that technical debt management can be divided into following activities: (1) identification, (2) measurement, (3) prioritization, (4) prevention, (5) monitoring, (6) repayment, (7) representation/documentation, and (8) communication [5]. Li et al. [5] also state that currently there is a lack of empirical evidence about technical debt management. In this study, we are mainly focusing on three out of the eight management activities. Our goal is to use processes for representation/documentation, identification, and prioritization of already incurred technical debt to provide empirical evidence with a real case company.

Technical debt representation/documentation has been studied and suggested in literature with specific lists and templates as an approach to store technical debt issues [6, 7]. A backlog or a list should increase technical debt visibility and manageability. When technical debt is properly documented, it is easier to start other technical debt management activities, because it is visible to the company.

Before a technical debt issue can be documented, it has to be identified. Identification of smaller technical debt issues from the source code is possible with specific tools [8]. However, technical debt is not always only related to issues in the source code [9]. Technical debt in software architecture and design is a larger challenge [5, 9].
The identification of architectural technical debt with tools is difficult and often the only solution is to use human knowledge and examination [9].

The prioritization of technical debt is difficult, because some technical debt might be important to fix for business reasons, while other for technical reasons. Some models and methods have been developed for prioritization. Seaman et al. [10] suggested four approaches for technical debt decision-making: simple cost-benefit analysis, analytic hierarchical process, portfolio management model, and options. These approaches have been used also in other domains, such as finance [10]. They support evaluating the tradeoffs between proposed enhancements, corrective maintenance, and the payment of technical debt items [10]. Schmid [11] developed a formal model based on providing several well-defined approximations, which can be used for technical debt prioritization. In addition, some papers have used quality attributes from ISO 9126 as an evaluation to technical debt [12–14].

Overall, there exists a variety of different ideas for technical debt documentation, identification and prioritization. However, most of them are focused on one specific activity only. Studies that approach the whole process from identification to repaying technical debt are rare. Therefore, we collaborate with a real software company to find and develop processes, including technical debt identification, documentation, and prioritization. We take inspiration from a study conducted by Li et al. [7] that had a similar goal. Their approach was to identify architectural technical debt based on architecture decisions and change scenarios [7]. Our approach extends this by expanding the technical debt evaluation and prioritization processes. Our goal is to create more reasoning possibilities in decision-making, which is required especially in organizational aspects of technical debt management.

3 Research Methodology

Action research was selected as a research methodology for this study. Action research combines theory and practice [15]. Action research is an iterative process involving researchers and practitioners acting together on a particular cycle of activities, including problem diagnosis, action intervention, and reflective learning [15]. Action research is especially relevant in situations where participation and organizational change processes are necessary [16]. It attempts to provide practical value to the client organization while simultaneously contributing to the acquisition of new theoretical knowledge [17]. The action research cycle [18] consists of three stages: (1) a pre-step - to understand context and purpose; (2) six main steps - to gather, feedback and analyze data, and to plan, implement and evaluate action; (3) a meta-step - to monitor.

The rationale for using action research as a research methodology is the nature of this study. The company in this study had a goal to improve their technical debt management. The research group in this study had previous experience on working with various companies and cases related to technical debt and its management. Therefore, action research, as an approach where both the company and the research group work together to understand the problem and develop a solution, was especially fit for the purpose.
The selected product line in this research is a financial system used in the capital market industry by multiple customers around Nordics. The product is one of the three main products provided by Tieto and it has a long development history including source code from over 20 years ago. The product and development team have faced both technological changes and organizational changes during their lifetime. Now the main objective of Tieto’s Capital Market product unit is to migrate to new technology with the aim to replace and rewrite old one, to improve quality and productivity, while still serving all of its customers.

The objective of the study was to increase technical debt visibility and manageability by improving processes related to identification, documentation and prioritization. Therefore, we set up the following research questions to address the problem:

**RQ1: How to improve technical debt identification and documentation?** The limitations of the tools currently available for technical debt identification can be seen as a big challenge. The identification of architectural technical debt with tools is very difficult. Therefore, most if not all technical debt identifications have to be done with manual code and architecture inspection, where developer or architect examines the system and the source code for possible issues. Our goal is to observe how technical debt is currently identified in practice and how it is documented afterwards. The objective is to identify possible improvements to these current processes, and test them in practice.

**RQ2: What factors should be taken in consideration when prioritizing technical debt?** The decisions related to technical debt can be sometimes made based on hunches without any specific model or method to follow. Business owners might prioritize issues that give direct value to customers, while technical people might put value more on software quality and sustainability. Understanding both business and technical effects of technical debt repayment can help technical debt evaluation and improve the prioritization process for safer decisions. We will observe the processes of technical debt evaluation and prioritization in practice with the aim to improve technical debt evaluation and prioritization.

### 4 Action Research Process

The action research process used in this study is presented in Fig. 1. This research can be divided into five main activities and outcomes.

**The first step** of the research process is interviews, where researcher interviews people related to the product line or company to understand the current issues related to technical debt and its management. We conducted seven semi-structured interviews with the average of 45 min. We recorded, transcribed and analyzed all the interviews. In the analysis of the interviews, we identified major issues. First, we did not find any systematic process for technical debt identification, evaluation or prioritization. This led to a technical debt communication gap between the development team and project managers. Knowledge of technical debt seemed to be tacit personal knowledge rather than explicitly stored in a common list. Secondly, we noticed that the developers and architects had much knowledge about the current issues regarding technical debt, but
there was not any systematic way to document it. Thirdly, when there were technical debt issues in discussion, the decision-making was mostly done based on hunches, rather than evaluating and prioritizing them first. The outcome of this step is a problem identification, which helps to understand the problem in current processes within a company.

The second step is to develop a process for identification. In our case, the identification was conducted by gathering the data from previous knowledge and history of people related to the product. The members of the product line used ten weeks to search and identify technical debt issues. The reason for manual inspection was that the company did not have any specific tools in use to identify technical debt. The outcome of this step is the increase of technical debt visibility, which helps to understand the overall technical debt view.

The third step is to develop a process for documentation. We decided to introduce a simple process to document all technical debt issues to a single technical debt backlog. The idea was to use backlog as an aid to make technical debt more visible to everyone in the product line. We used a similar template (Table 1) to Guo and Seaman [6] to collect all technical debt items. The template was sent to nine members of the product line that was later returned back to the managers. The managers then combined all the reported issues and created the technical debt backlog. The outcome of this step is getting technical debt stored.

The result of documentation process was technical debt backlog that consisted 47 identified technical debt issues. For categorization we used 15 different technical debt types identified by Alves et al. [19] in a mapping study. The majority of identified technical debt (33/47 issues) was related to issues in design, architecture, code, and a new category called legacy debt. Other types of technical debt (14/47) requirements,

<table>
<thead>
<tr>
<th>Table 1. Template for technical debt documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical Debt ID</td>
</tr>
<tr>
<td>Date/Reporter</td>
</tr>
<tr>
<td>Technical Debt Name</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Alternatives</td>
</tr>
<tr>
<td>Rationale</td>
</tr>
</tbody>
</table>
test, test automation, and process debt were associated more to activities outside product implementation.

The fourth step is to organize a workshop. We developed a process to prioritize technical debt issues with a simple technical debt evaluation and prioritization template (Table 2). This template was used when all the technical debt issues were collected to the backlog. In the workshop, the participants would evaluate each identified technical debt issue based on the five questions to create a prioritization. The outcome of this step is getting technical debt evaluated.

The research group also analyzed the returned evaluation templates based on each question to understand how technical debt was being evaluated.

We identified three different benefit categories: technical, economic, and organizational benefits. Technical benefits include improvements in software quality, software maintainability, software reusability, software performance, software testability, and software deployment. Organizational benefits include better software deployment, development team productivity, organizational communicability, and future adaptability. Economic benefits include economic value and customer satisfaction.

We identified three different risk categories: economic, technical, and organizational risk. Economic risks include cost, time effort, testing effort, and customer satisfaction. Organizational risks include management and competence. In addition, technical risks like system breakdown and instability are critical to companies.

For reasons, we identified three different categories: intentional decision, unintentional cause, and organizational cause. Intentional reasons were often related to time constraints, lack of resources, and business driven development. Unintentional causes were legacy product and lack of knowledge. For organizational causes, software processes and lack of management were the main reasons for technical debt.

We identified two types of solutions for technical debt: technical and organizational solutions. Technical solutions were refactoring, redesigning, rewriting, architectural analysis, and increased testing. Organizational solutions were new processes and new management plan/strategy.

The fifth step of the research process is sorting. When there is an evaluation for each technical debt item, it is easier to sort the issues out based on their importance. The last outcome of the process is prioritized technical debt backlog. The majority of the issues (27/47) were prioritized at the lowest priorities 5 or 4, which shows that most technical debt was not considered dangerous now. There were total of 14 issues rated as the highest priorities at level 1 or 2. There were three level 1 issues related to legacy

<table>
<thead>
<tr>
<th>Question</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Question</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>What are the benefits of fixing this issue? (Business value, quality, productivity, less bugs etc.)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Are there any risks in fixing this issue? (Expensive, breaks the system etc.)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Why was this issue done previously like that?</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>How to fix this issue and what resources the fix would require?</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>From scale 1 – 5, how important would you rank this issue to be dealt with? (1 – most important, 5 – not so important)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Technical debt evaluation template
debt, which can be explained by Tieto’s current goals to migrate to a new technology to replace and rewrite old technology. Interestingly, the priorities also show that most of the technical debt related to design, architecture, and code debt were prioritized as 4 or 5, while test, test automation, and process debt were rated higher. However, it is important to notice that number of technical debt issues in design, architecture, and code is much higher than other types of debts, which might explain the difference.

The outcome of this action research cycle was a prioritized technical debt backlog that can be now used to add more development tasks related to technical debt reduction. For example, Tieto managers expressed that the backlog would be used in the future by Tieto to reduce technical debt in small iterations. Tieto managers also mentioned that this same process would be applied in future to other product lines.

5 Discussion

**RQ1: How to improve technical debt identification and documentation?** Our study made technical debt identification and documentation possible with simple practices that make technical debt more visible and manageable. These similar practices have been already suggested in other literature [6, 7]. However, the problem is not the practices themselves, but the fact that changing or adding new practices in companies is always a challenge and takes time [20]. In our case, we had a company that was motivated to improve and change these practices. We started some new practices with templates and processes that gathered previously identified technical debt from the minds of architects and developers to a specific backlog designed only for technical debt.

Technical debt identification is a challenge in software development. Identification of smaller technical debt issues happening in single code lines can be done with static code analysis tools and often it can be fixed by single developers. However, larger issues in architecture and structure are often unreachable with tools [9, 21] and require technical knowledge and competence [22], and discussion on an organizational level. In our case, the people in the product line did not use any tools to find and identify technical debt. Instead, the technical debt was identified based on previous experience and history with the product. The experience with the product of workshop participants shows that the people responsible for identification had extensive knowledge of the product development history and high competence to build software. We used this fact to our advantage, since we did not have to guide developers and architects to investigate product history, because the knowledge was already acquired during the development years. This helped to identify existing technical debt and document it based on our recommendations.

An interesting perspective on identified and documented technical debt and is the variety of types of technical debts. The large variety of technical debt types shows that when talking about technical debt, it is not only related to issues in design or code. Instead, like in our case, the same phenomena of shortcuts and bad solutions happen in other parts of software product development as well. To some development teams, technical debt might include only issues happening in the source code and design,
while to some other teams, like in Tieto it might also include issues like those in testing and processes. We argue that technical debt management is successful when a company sets a clear standard to what is technical debt in their context and start to manage technical debt based on that standard. However, in academia, there is a need to create a common understanding for technical debt.

RQ2: What factors should be taken into consideration when prioritizing technical debt? The prioritization in this study was based on evaluation of benefits, risks, reasons, and solutions of technical debt. Using these factors to evaluate each technical debt issue could be a good beginning for companies that are trying to improve their technical debt management. However, these factors are not always measurable with a numeric value. The interest in technical debt that accumulates larger if not repaid, it is difficult to estimate [23]. Therefore, rather than trying to measure exact values, technical debt could be easier to understand from a management perspective, if evaluated based on factors related to it.

Companies should evaluate each technical debt issue on the basis of how fixing the issue can benefit both company and software, such as improved quality, and how does this quality improvement affect other factors such as maintainability, performance, or customer satisfaction. One challenge and risk of technical debt is that it often requires competence to fix or change existing solutions. When developers are changing very old parts of the code, it is not always certain that it will go as planned and it can be a huge risk that needs to be evaluated before.

Understanding the reasons behind technical debt can help to understand bigger underlying problems with technical debt. For example, a single technical debt issue in one smaller feature can be caused by some larger architectural issue. Instead of just fixing one single technical debt issue with the most economical value, it might be possible that another major technical debt item can be actually more beneficial to fix in a long term. Sometimes the solution might only require small refactoring, while sometimes it might need a full rewrite of that certain part of the code. Therefore, it is important to evaluate how much resources and effort does fixing technical debt require. Understanding the solution can enable a better evaluation, whether the time required for fixing is worth compared to its benefit.

We believe that technical debt prioritization should be done based on evaluation rather than measurement. The combination of the presented factors can be used as a simple way to create basic prioritizations, which can help companies to make decisions with more rationality. The decision-making may improve when development teams and management communicate and understand the benefits and risks in each technical debt issue, accompanied with knowledge on the reasons and solutions for technical debt.

Study limitations. The first limitation to this study is the generalization of the results. It is not certain that this process is usable in other companies. In our case, most of the involved people had many years of experience with the product. This helped the identification stage, since the people from the product line had already extensive knowledge about the issues in the product. The second limitation is that we conducted only one round of this action research. Conducting more rounds might change some results in the priorities and numbers of technical debt issues, but we believe it would
not have any changes to the actual processes that were used in this study. The third limitation is that the used process only takes into consideration already occurred technical debt, and does not include management processes for a situation, where a decision has to be made for a new technical debt case. This makes the developed management process limited to only already occurred technical debt.

6 Conclusions

We used the action research process [18] together with a large IT company Tieto to find and develop processes for technical debt identification, documentation, and prioritization to increase technical debt visibility and manageability. The action research process consisted of several interviews and meetings with the company representatives and an organized technical debt workshop to improve processes in the company. The outcome of the research was a set of templates and processes to identify, document, and prioritize technical debt. These templates and processes were used successfully at Tieto to transition from a situation where knowledge of technical debt was not explicitly documented, to a situation where a specifically prioritized technical debt backlog was available to reduce technical debt. Tieto’s Capital Market product unit is now using this new technical debt backlog to increase technical debt visibility and manageability. Since the results with the developed process were considered successful by both the research group and the company, the same process will be expanded to other product lines in Tieto. The main challenges and lessons learned can be summarized as following:

- Technical debt can be brought visible with simple practices and processes in a company that does not have a priori knowledge on technical debt management.
- Identification of larger scale technical debt, such as architecture and design, with tools is a challenge that needs to be addressed and improved in future research.
- Technical debt documentation can be done with simple templates, but requires motivation and resources from software organization.
- Technical debt prioritization based on measurements is difficult, and therefore rougher evaluations based on e.g. benefits and risks through opinions can be seen easier to start with.

References


703. RICHTER, CHRIS. Digital collaborations and entrepreneurship – the role of shareconomy and crowdsourcing in the era of smart city. 2016. Diss.


705. PATEL, YOGINI. Computational modelling of non-equilibrium condensing steam flows in low-pressure steam turbines. 2016. Diss.


707. AMOUR, IDRISSA. Variational ensemble kalman filtering applied to data assimilation problems in computational fluid dynamics. 2016. Diss.

708. SHESTAKOVA, MARINA. Ultrasound-assisted electrochemical treatment of wastewaters containing organic pollutants by using novel Ti/Ta2O5-SnO2 electrodes. 2016. Diss.

709. OLEKSIENKO, OLGA. Physico-chemical properties of sol-gel synthesized titanosilicates for the uptake of radionuclides from aqueous solutions. 2016. Diss.


712. LAHTELA, VILLE. Improving the properties of solid Scots pine (Pinus sylvestris) wood by using modification technology and agents. 2016. Diss.

713. NEVARANTA, NIKO. Online time and frequency domain identification of a resonating mechanical system in electric drives. 2016. Diss.

714. FANG, CHAO. Study on system design and key technologies of case closure welding for ITER correction coil. 2016. Diss.

715. GARCÍA PÉREZ, MANUEL. Modeling the effects of unsteady flow patterns on the fireside ash fouling in tube arrays of kraft and coal-fired boilers.


718. KNUTAS, ANTTI. Increasing beneficial interactions in a computer-supported collaborative environment. 2016. Diss.


MULTAHARJU, SIRPA. Managing sustainability-related risks in supply chains. 2016. Diss.

HANNONEN, JANNE. Application of an embedded control system for aging detection of power converter components. 2016. Diss.


RINKINEN, SATU. Clusters, innovation systems and ecosystems: Studies on innovation policy’s concept evolution and approaches for regional renewal. 2016. Diss.

VANADZINA, EVGENIA. Capacity market in Russia: addressing the energy trilemma. 2016. Diss.

KUOKKANEN, ANNA. Understanding complex system change for a sustainable food system. 2016. Diss.

SAVOLAINEN, JYRKI. Analyzing the profitability of metal mining investments with system dynamic modeling and real option analysis. 2016. Diss.

LAMPINEN, MATTI. Development of hydrometallurgical reactor leaching for recovery of zinc and gold. 2016. Diss.


ALMANASRAH, MOHAMMAD. Hot water extraction and membrane filtration processes in fractionation and recovery of value-added compounds from wood and plant residues. 2017. Diss.

TOIVANEN, JENNI. Systematic complaint data analysis in a supply chain network context to recognise the quality targets of welding production. 2017. Diss.

PATEL, GITESHKUMAR. Computational fluid dynamics analysis of steam condensation in nuclear power plant applications. 2017. Diss.

