The Journal of Systems and Software 120 (2016) 195–218

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

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

Jesse Yli-Huumo[a,*], Andrey Maglyas[a], Kari Smolander[b]

[a] Lappeenranta University of Technology, School of Business and Management, Department of Innovation and Software, PO Box 20, Skinnarilankatu 34, Lappeenranta FI-53851, Finland

[b] Aalto University, School of Science, Department of Computer Science, P.O.Box 15400, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Received 14 May 2015
Revised 10 December 2015
Accepted 10 May 2016
Available online 11 May 2016

Keywords:
Technical debt
Technical debt management
Exploratory case study

ABSTRACT

Technical debt (TD) is a metaphor for taking shortcuts or workarounds in technical decisions to gain short-term benefit in time-to-market and earlier software release. In this study, one large software development organization is investigated to gather empirical evidence related to the concept of technical debt management (TDM). We used the exploratory case study method to collect and analyze empirical data in the case organization by interviewing a total of 25 persons in eight software development teams. We were able to identify teams where the current strategy for TDM was only to fix TD when necessary, when it started to cause too much trouble for development. We also identified teams where the management had a systematic strategy to identify, measure and monitor TD during the development process. It seems that TDM can be associated with a similar maturity concept as software development in general. Development teams may raise their maturity by increasing their awareness and applying more advanced processes, techniques and tools in TDM. TDM is an essential part of sustainable software development, and companies have to find right approaches to deal with TD to produce healthy software that can be developed and maintained in the future.

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 also 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 (Zazworka 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 (Falessi 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 TDM in a real-life software development environment (Li et al., 2015a). It is important to gather

* Corresponding author.
E-mail addresses: jesse.yli-huumo@lut.fi (J. Yli-Huumo), andrey.maglyas@lut.fi (A. Maglyas), kari.smolander@aalto.fi (K. Smolander).

http://dx.doi.org/10.1016/j.jss.2016.05.018

evidence about TD and TDM in real-life software development situations to understand how TDM is currently perceived by real software development teams, and to use that knowledge to improve the existing processes and tools.

In order to understand TDM in a real-life software development environment, we studied eight software development teams in a large organization that is a provider of multiple software solutions. For data collection and analysis, we used the eight TDM activities identified by Li et al. (2015a) in semi-structured interviews to gather empirical data about TDM in the selected software development teams. We used the exploratory case study method (Robson, 2002) to answer the following main research question:

RQ: "How do software development teams manage technical debt?"

Since the main research question can be considered quite a wide topic, including several other topics, we decided to create a set of sub questions to tackle specific topics of our interest.

RQ1.1: What TDM activities are used in the studied development teams?

Technical debt management can be separated into the following activities: *identification, measurement, prioritization, prevention, monitoring, repayment, representation/documentation, and communication* (Li et al., 2015a). However, it is not certain what activities are actually used and taken into consideration in real-life software development. Therefore, it is important to study and understand which TDM activities are currently applied/used and which are not. The results obtained from the studied development teams could reveal which activities will need more research in the future.

RQ1.2: What methods, models, practices or tools do the studied development teams use for each TDM activity?

There are a number of possible methods, models, practices or tools for every TDM activity (Li et al., 2015a). They have been developed and suggested in the literature, but they lack empirical evidence of their usability and functionality (ibid.). Therefore, it is essential to gather empirical evidence from real-life software development to understand what approaches different software development teams use for each TDM activity. Collecting such evidence could help to evaluate which TDM approaches should be categorized to each TDM activity.

RQ1.3: Are there any maturity differences in adopting TDM activities between development teams?

Every software development team is different, working with different products in different environments, and using different methods, models, practices, and tools in their unique way. It is highly possible that software development teams in general have different activities and approaches as regards TDM. Some software development teams may use more time on TDM, while some development teams may not pay much attention to it (Power, 2013). Therefore, it is important to understand if it is possible to distinguish between different maturities of TDM, similarly as in the capability maturity model (CMM)(Paulk et al., 1993). The results of this study can be used to develop a similar maturity model for TDM, which researchers and practitioners could use to conduct more research, or to improve companies' internal and external practices.

RQ1.4: What are the biggest challenges in TDM?

Software process improvement includes the challenge of adopting new practices and tools to development teams. Understanding this challenge in relation to TDM is beneficial for software development teams and researchers.

The rest of the paper is organized as follows: Section 2 introduces the theoretical background of TD and TDM in software development, Section 3 describes the research methodology used in this study, and Section 4 presents the results received from the empirical analysis of the studied software development teams. In Section 5 present the developed framework. In Section 6 we discuss the results and implications to future research. Section 7 concludes the paper.

2. Background

2.1. Technical debt

The metaphor technical debt (TD) has been introduced by Ward Cunningham (Cunningham, 1992). He describes the metaphor as *'Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt."* (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).

The TD metaphor was first associated with compromises on the code level of software (Cunningham, 1992). In addition, terms like *code smells* (Fowler 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 (Tom et al., 2013; Alves et al., 2014). The current literature identifies such terms as requirements (Brown et al., 2010), design (Zazworka et al., 2011b; Zazworka et al., 2011a), architectural (Nord et al., 2012), test (Brown et al., 2010), process (Lim et al., 2012), documentation (Kruchten et al., 2012a), and people debt (Kruchten 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 (Yli-Huumo 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 sometimes force the development team to create solutions with second-tier quality to meet the requirements within the deadlines set by the business stakeholders (Yli-Huumo 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 (Yli-Huumo 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 (Yli-Huumo 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 turn the effort of the development team to a new direction. This means that previously developed parts need to be changed to make the product more

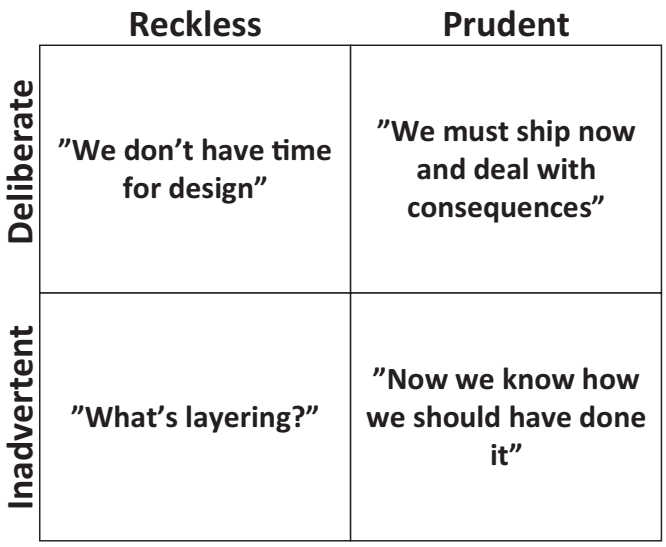

Fig. 1. Technical debt quadrant (Fowler, 2009).

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 (McConnell, 2007).

TD is often seen only as a negative concept in software development (Lim et al., 2012;Yli-Huumo et al., 2014). Software developers think that creating shortcuts and non-scalable solutions will increase the complexity within the code base (Yli-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 (Yli-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 (Yli-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 (Tamburri 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 (Codabux 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 behavior 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 (2011). 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 (Mantyla and Lassenius, 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 TDM. The study identifies eight activities for TDM: (1) identification detects TD caused by intentional or unintentional technical decisions in a software system through 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 repaid first and which TD items can be tolerated until later releases; (4) prevention aims to prevent potential TD from being incurred; (5) monitoring watches the changes of the cost and benefit of unresolved TD over time; (6) repayment resolves or mitigates TD in a software system by techniques such as reengineering and refactoring; (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 TDM, 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 few empirical studies on TDM. Guo 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 a 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 based on estimates. 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 (Zazworka et al., 2013).

Klinger et al. (2011) interviewed four experienced software architects to understand how decision-making 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 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 TDM. The SQALE method (Letouzey, 2012; Letouzey and Ilkiewicz, 2012) has been developed for the purposes of identifying, estimating, analyzing, measuring, and monitoring TD in a software. DebtFlag (Holvitie and Leppänen, 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 from software (Al Mamun et al., 2014; Griffith 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 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 TDM 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, 2008). There are multiple ways to study 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 possibilities in the code base. It is also possible to study TD from the organizational point of view. In an organizational TD study the research focuses on how various processes and practices are used in software-related TD. This type of a study can be performed with qualitative 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 TDM 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 TDM in real-life situations, to gather qualitative data, and to add to existing research related to TDM.

This study can be defined as an interpretive 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

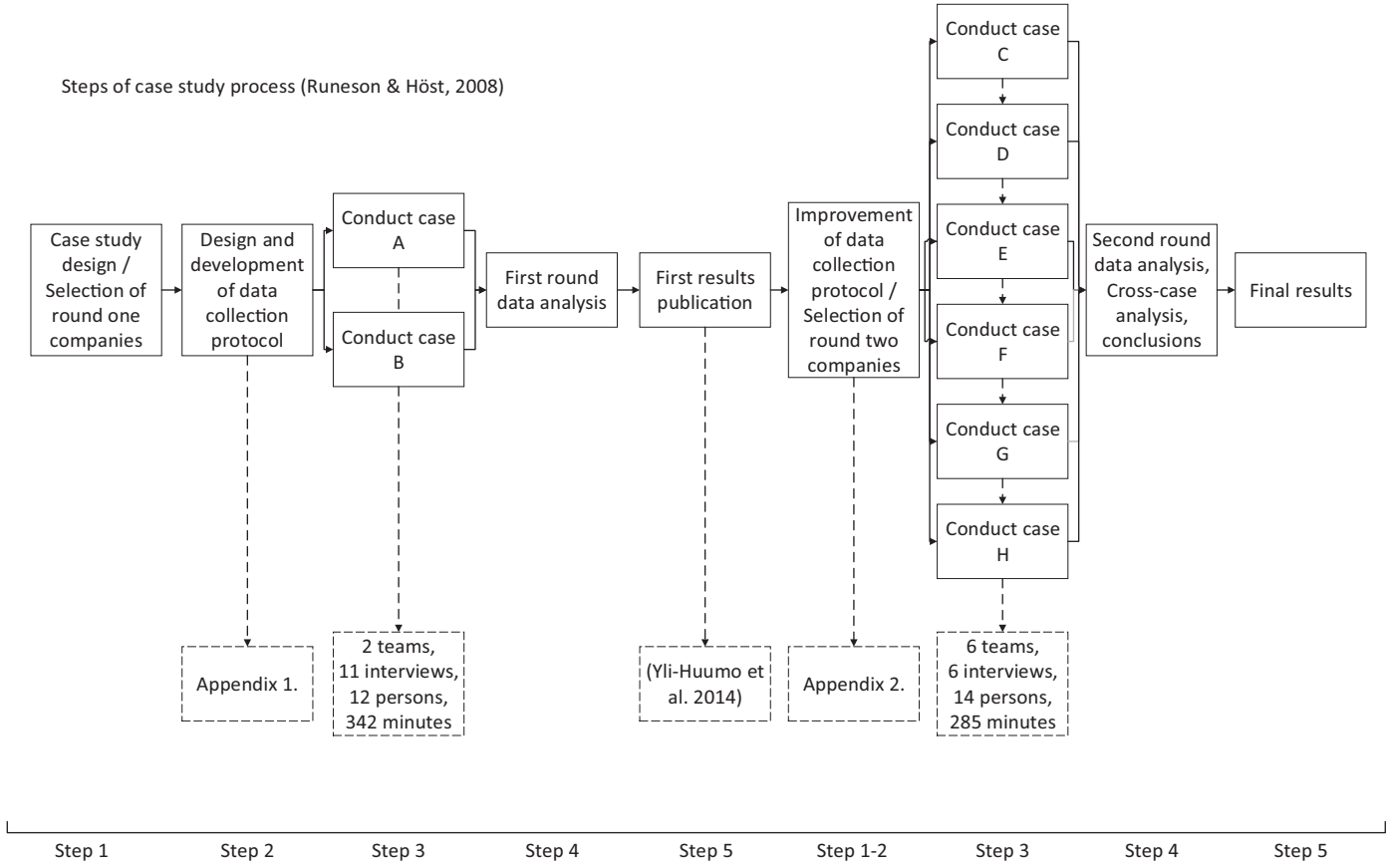

Fig. 2. Case study research process used in this study.

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; Robson, 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 interviewer and the interviewee. 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

Table 1
Summary of the case software development teams.

Team	Role of the team in the organization	Description
A	Development of a single product line.	The product provides a financial management solution as a cloud service. The current size of the team is 18. The whole development team is located in the same country. The development team uses a Scrum-like approach as the development methodology.
B	Development of a single product line.	The product is a SaaS-based project management solution for multi-organization projects. The current team size is 13, and the whole development team is located in the same country. The development team uses a Scrum-like approach as the development methodology.
C	Development of a platform infrastructure.	The platform infrastructure is used by the other development teams that develop the actual products for customers. The goal of the development team is to be a gateway between all the products and integrations within and outside the organization. The current team size is 12, and the development is distributed into several countries in Europe. As the development team works with other product lines in the organization, it uses Scrum and Kanban in parallel.
D	License integrations	The development team has two main responsibilities: (1) licensing and generating the invoices based on the usage, and granting the rights and accounting the usages for the services the customers have bought, and (2) integration of all the smaller systems that have been bought by the company. The current team size is 11, and the development is distributed into several countries in Europe. As the development team works with other product lines in the organization, it uses Scrum and Kanban in parallel. The development team also works closely with development team C.
E	Development of services and software for the organization.	The main responsibility of the development team is a platform from which all the services of macro segments are started and administrated. The current team size is 12. The development is also distributed into several countries in Europe. The development team uses a Scrum-like approach as the development methodology.
F	Development of a single product line in the organization.	The product is a web-based solution that allows approving invoices and expense claims online or mobile. The current team size is 8, and the whole team is located in the same country. The development team uses a Scrum-like approach as the development methodology.
G	Development of a single product line.	The product is a web-based solution for making budgeting and forecasting predictions for business monitoring and reporting. The software collects information on the company's financial records and other systems, as well as the form of real-time reports. The current team size is 8, and the development is distributed into several countries in Europe. The development team uses a Scrum-like approach as the development methodology.
H	Developing integration and security tasks for the organization.	The main goal of the development team is to handle integration and security tasks for the organization. The current team size is 17, and the development is distributed into several countries in Europe. The development team uses Scrum as the development methodology.

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 ones 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. In one of the interviews in team B (Interview ID B3), 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

Table 2
Roles of the interviewees.

Interview ID.	Round	Team	Role(s)	Experience in the organization
A1	1	A	Software architect	6 years
A2	1	A	Software designer	1 year
A3	1	A	Project manager	4 years
A4	1	A	Software test engineer	1 year
A5	1	A	Product line manager	14 years
B1	1	B	Software architect	6 years
B2	1	B	Software developer	6 years
B3a	1	B	Product line manager	2 years
B3b	1	B	Software test engineer	4 years
B4	1	B	Software architect	5 years
B5	1	B	Software developer	1 year
B6	1	B	User interface designer	1 year
C1a	2	C	Team manager	5 years
C1b	2	C	Software architect	17 years
C1c	2	C	Software architect	3 years
D1	2	D	Software architect	7 years
E1a	2	E	Team manager	15 years
E1b	2	E	Software architect	8 years
E1c	2	E	Software architect	5 years
F1a	2	F	Team manager	4 years
F1b	2	F	Software architect	9 years
G1a	2	G	Team manager	1 year
G1b	2	G	Software architect	4 years
H1a	2	H	Team manager	3 years
H2b	2	H	Software architect	3 years

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 interviewees 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 interviewee. 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 interviewees 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 priori 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 17 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 would be 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 in Word 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 repayment, TD representation/documentation, TD identification, TD prioritization, TD measurement, TD monitoring, TD communication, and TD prevention.* Table 3 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 case 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 repayment 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 repayment strategy for the 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 repayment 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

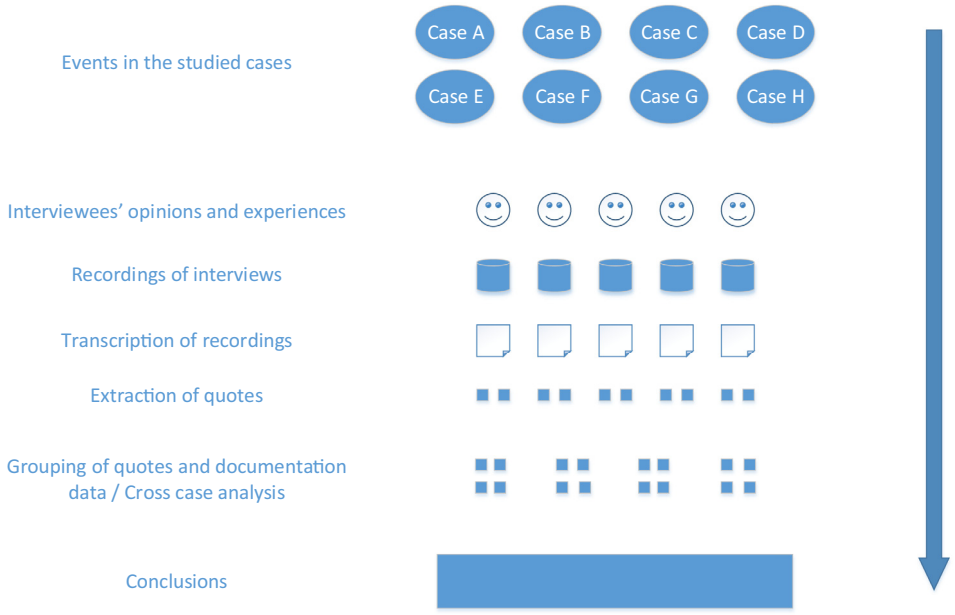

Fig. 3. The coding and analysis process.

Table 3
Example of the data coding process.

Interview transcripts	Categories
"We have Epics, and we have some kind of goal, so 20% of developer time to be in internal quality."	TD repayment
"We have quite **often security reviews**, and maybe some technical debt can come from security"	TD prevention
"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, think, **spend 218 h**, and we see people, some our system architects, some our developers, some our 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 monitoring, TD measurement, TD repayment
"Also as a team we have some **KPIs**. In the team, we are part of product unit, R&D department for ERPs and other product-related. And then we as a team define **KPIs to measure, and for example, we have one key control on technical debt**."	TD communication, TD measurement, TD monitoring
"And as a team, we have kind of demo, some kind of retrospective on monthly basis, then we do all the numbers and then discuss it: this strength is good, this number is too low, what to do, and then we put on backlog actions."	TD communication, TD monitoring, TD documentation

the JIRA 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 added 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 communication 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 issues 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 TDM 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 TDM, because the communication 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 out of 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 had 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 done during the development by the developers. When the code base was developed, the developers would identify the refactoring needs, when the currently developed part was extremely complex and hard to develop. Sometimes also 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 before them.

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 TDM 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 certain periods. Similar to Case A, this was the reason why it was extremely difficult for the team management to measure and monitor TD. However, the idea of TDM 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 to 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 bigger 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 and software architects. Smaller refactoring cases where just done during the development without mentioning them to the management.

TD representation/documentation was not always done systematically by the developers. The development team did not have any mandatory guidelines for the developers for 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 systematical 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 and 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 ones. 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 it 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 currently 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 hunch 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 sometimes they would take into consideration issues like 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 mentioned that he trusted people's opinions 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 chancing.

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 in communicating about issues related to internal 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 to produce 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 TDM 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 TDM in development team C were TD documentation and TD prevention. Even though the TDM structure was well-designed to repay TD systematically, the development team did not have a proper documentation 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 ran 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 estimation.

TD monitoring was conducted with the JIRA and SonarQube tools. The management was able to use JIRA as a tool to monitor the development time for various tasks on either new features or 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.

TD communication was performed between 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 to discuss and list things that 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 current team manager was not 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 whole team 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 TDM strategy in software development team D was constructed well. The management had a clear vision and understanding of the fact that 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 refactor 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 the other development teams with information about possible issues in the future where TD would be most disturbing.

TD identification 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 according to its type. 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 issues. The TD backlog in JIRA was used to gather some statistics and to estimate the current status regarding TD in the platform. The manager and software architects 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 currently highest 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 code guidelines and code reviews. The developed code was always checked with a tool called StyleCop 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 TDM 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 only when the issue started to be highly problematic for the development team to 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 repayment 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 shortcut or noticed 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 varied. 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 actively 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 management or 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 if 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 shortcut 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 memo 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 repay next, factors like time, functionality, further 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 active 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 owners 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 ýour 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.

TD representation/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 ticket to the system, where the manager and the team could follow the possible improvement needs.

TD identification was conducted during the continuous integration process. If the development team noticed that some part of the software needed technical improvement, it was something that should be focused on.

TD prioritization was based on story points that were assigned by the team management and software architect. If there was need for a big change in the code base, it would be prioritized higher. The prioritization was mostly based on a hunch and previous knowledge of the issue and that certain area of the code base. The team used figures from SonarQube and the opinions of project stakeholders to make the decision on what TD would be prioritized as the most important to repay.

TD measurement was done with the SonarQube tool. The team manager and software architect ran the code base with SonarQube and were able to measure the amount of TD every month. The results from the tool were compared to the standards and performance that the management had set up. This way the team manager was able to measure whether TD had increased or decreased during the previous month.

TD monitoring was based on JIRA and SonarQube information. The team manager felt that information from SonarQube was important information to monitor, to know how healthy the software was at any point of time. He also admitted that SonarQube did not necessarily offer information about big architectural issues, but still thought that it was valuable information to have.

TD communication was active between the management and the development team. If a developer identified a TD issue, it was discussed with the software architect, and the decision to allocate time for it was often granted.

TD prevention was conducted by creating a strict definition of the done process to every code that was developed. When a developer created or changed something in the code base, it was guided to be tested first locally in the developer's own machine. After this, the code went to the acceptance environment, where all the other components were connected to the system. When the code was tested and verified in the acceptance environment, it would go to the staging environment, where it would go through automated test cases. Finally, if no bugs or issues were found, the code would go to the production environment. In case an issue was found, the code would go back to the developer, who had to refactor or rewrite it.

Overall, the TDM strategy in development team H was organized systematically. The team manager described the process as continuous refactoring, where the goal was to keep the overall quality of the code base always on an acceptable level. This was conducted by having a continuous TD repayment and TD prevention strategy that was compounded with TD monitoring and TD measurement by the team management.

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 refactoring, rewriting or redesigning 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 teams (A, B, F, and G) that often started TD repayment when the TD 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 monitor 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 effect, 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 4
Summary of the cases.

Case/TDM activity	TD repayment	TD identification	TD measurement	TD monitoring	TD prioritization	TD representation/ communication	TD prevention	TD representation/ documentation
Case A	Issues from the general development backlog. Refactoring during normal development.	Team manager and software architects identifying TD manually. Mostly during normal development.	No measurement	No monitoring. Sometimes from JIRA issues.	Mostly based on a hunch. Business value and scalability taken into consideration.	Communication structure good with highly technical team manager.	Some coding standards and code reviews.	Some issues to JIRA. No separate backlog.
Case B	Issues from the general development backlog. Refactoring during normal development.	Team manager and software architects identifying TD manually. Mostly during normal development.	No measurement	No monitoring. Sometimes from JIRA issues.	Mostly based on a hunch. The number of places affected by the change taken into consideration	Communication structure good with highly technical team manager.	Some coding standards and code reviews.	Some issues to JIRA. No separate backlog.
Case C	20% of development assigned to improving the code base. Refactoring during normal development.	Software architects identifying TD manually. Identification with SonarQube tool.	Team manager measuring from JIRA, software architects from SonarQube statistics.	Monitoring with JIRA and SonarQube.	Mostly based on a hunch. Sometimes taken into consideration how much the feature was used now and would be used in the future.	Communication about TD active within the whole development team.	Some coding standards and code reviews. Not always conducted.	Internal quality backlog on JIRA. Not used systematically by the developers.
Case D	20% of development assigned to improving the code base. Refactoring during normal development.	Software architects identifying TD manually. Identification with the SonarQube tool.	Team manager measuring from JIRA, software architects from SonarQube statistics.	Monitoring with JIRA and SonarQube.	Mostly based on a hunch. The management using low/medium/ high/blocker - story points -Fibonacci scale	Communication about TD active within the whole development team.	Code reviews and coding standards. Definition of the done standard.	Internal quality backlog. Developers using systematically.
Case E	Decided once a month in a meeting. Refactoring during normal development.	Software architects identifying TD manually.	Measurement from the TD backlog.	Monitoring TD backlog.	Mostly based on a hunch. Taken in consideration how much it would affect the customer and future projects.	Communication in stakeholders' meeting monthly.	Some coding standards and code reviews. Not always conducted.	Separate backlog for TD items.
Case F	Refactoring only when TD became a huge problem.	Identification with tools (SonarQube, CheckStyle, FindBugs). Mostly during normal development.	Measurement from JIRA issues.	Monitoring with JIRA.	Mostly based on a hunch.	Communication currently challenging with business people.	Some minor coding standards and reviews of the used tools.	Some issues in JIRA. No separate backlog.
Case G	Refactoring only when TD became a huge problem.	Identification rarely done.	No measurement.	No monitoring.	Mostly based on a hunch. Time, functionality, further maintenance, scalability, business value and future plans taken into consideration.	Current communication of TD lacking.	No coding standards or code reviews.	Some issues in JIRA. No separate backlog.
Case H	Number of days in a month assigned for improvement.	Mostly during normal development. Identification with the SonarQube tool.	Team manager measuring from JIRA and SonarQube statistics.	Monitoring with JIRA and SonarQube.	Mostly based on a hunch. Story points based on how important the issue was.	Communication active within the development team and stakeholders.	Definition of Done to ensure code quality.	Issues reported to JIRA.

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
TDM framework.

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

therefore able to add also the responsible person to each TDM activity. When we had identified all the TDM approaches, practices and responsibilities, we started to compare the cases. During this comparison we realized that there was a lot of variation in the TDM approaches and practices.

The results indicated differences in the maturity of TDM. By the term maturity we mean the ability 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 acknowledgd 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 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 cross-case analysis. The first stakeholder 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 identification. 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 architects have responsibilities in all TDM activities. The cross-case analysis revealed that software architects often acted as a central mind in TDM. This was because software architects often have the best overall view 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 were directly related to technical development, and therefore his/her responsibility 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 to 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 efforts 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 need 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. RQ1.1-1.2: 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 TDM 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 Codabux 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 (Power, 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 included coding standards, code reviews, and the Definition of Done. 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, 1997; Kemerer 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 TDM 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 TDM 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 was 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/should 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 TDM strategy. When TD issues are not stored, it is highly possible that they will be forgotten at some point and will never be repaid. Without proper tracking and documentation of architectural changes and issues, it is also extremely challenging to quantify TD (Klinger et al., 2011). The inability to quantify TD also creates more challenges to other TDM 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 (Das et al., 2007; Forward and Lethbridge, 2002). Therefore, adopting even a simple documentation practice for TD representation/documentation improves other TDM activities and the overall TDM strategy. A systematic process to document and store all the TD issues can be used for creating a systematic TD repayment strategy (Lim 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, CheckStyles, 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 developed to scan the source code, such as SonarQube. Identifying issues found with these tools can mean for some development teams that TD has been identified, and they will use this TD data for other TDM activities.

However, the challenge in identification is that TD is not just related to simple errors, but especially to the architectural and 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., 2012a). 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 (Zazworka 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 (Zazworka 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; Seaman et al., 2012; Theodoropoulos et al., 2011; Zazworka et al., 2011a). 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 prioritizations were 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.

Ramasubbu 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 needs, but this can leave the most important TD from the technical perspective out of sight (Codabux 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 also 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 item: bad distribution of the 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 the development.

An estimate based on a tool should be more accurate, faster and reliable compared to an estimate based on simple data. However, TD measurement has the same problem as TD identification: "*what technical debt do you want to estimate*?" When a development team considers for example the criteria in SonarQube (2015) as TD, in 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 (Kruchten 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 larger 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 TDM 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 TDM activities.

TD monitoring and tracking is one of the most vital TDM activities (Ernst et al., 2015). Without monitoring, the development team is not able to have any reasoning for other TDM 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 repay or not.

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

In some development teams TDM focused on only two to three activities, while some development teams conducted all eight TDM 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 TDM activities that were conducted mostly by the development team (*repayment, 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 TDM?

6.3.1. Lack of tools

One of the main challenges in TDM is the lack of tools. TDM was mostly conducted as human activity, instead of using automated or enabling tools. Ernst et al. (2015, p.?) state that developers "*desire standard practices and tools to manage technical debt that do not currently exist*". If most of the current TDM 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 TDM activities without tools is time-consuming, and the addition of tools would provide faster TDM activities.

As Ernst et al. (2015, p.?) comment, "*tooling is a necessary component of any technical debt management strategy*", we also believe that an important research area currently in TDM is the research and development done for tools designed to tackle different TDM 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 TDM.

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 TDM activities to support the decision-making. However, one of the current main challenges of TDM 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; Zazworka et al., 2011a) 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 models and methods that take both the 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 model or method for doing it properly. Therefore, prioritization was mainly done just by the opinions of single persons, based on hunches and previous experiences, instead of estimations and measurements 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 customers, 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 TDM, 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).*

6.3.3. Having a proper mindset with TDM

One of the challenges is the mindset of the developers. The goal of TDM is to provide practices and tools to manage and reduce TD during software development (Li et al., 2015a). This obviously requires more effort on the already existing practices of tracking down and fixing issues to make technical improvements. Conducting TDM takes time, and it will have an effect on other software development activities. Instead of designing and developing a new feature, it could 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 TDM 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 TDM is to get the whole organization/team included in TDM 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 TDM. This way all the TDM activities will support each other successfully.

6.3.4. Time-consuming TDM

We also observed that TDM 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 TDM 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 TDM. 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 TDM, which will give confidence to the development teams to allocate more time and resources for TDM.

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 Baskerville, 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 TDM.

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, 2008). 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 questions' (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 interviewer 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 possible misunderstandings during the interviews. We also let the interviewees 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 causalities. 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 causalities 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 (chaotic), repeatable, defined, quantitatively managed, and optimizing* (Paulk et al., 1993). A similar maturity model to CMM is also adaptable in TDM, where development teams have different TDM maturities 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 basic 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 Need 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 Interviewee introduction

- Respondent's name:
- 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 debts?
  - Do you do it based on a hunch 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?
  - Coding 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 you current practices regarding technical debt management?
  - Is there anything else you would like to say?

References

Allman, E., 2012. Managing technical debt. Commun. ACM 55, 50–55. doi:10.1145/2160718.2160733.

Al Mamun, M.A., Berger, C., Hansson, J., 2014. Explicating, understanding, and managing technical debt from self-driving miniature car projects. In: 2014 Sixth International Workshop on Managing Technical Debt (MTD), pp. 11–18. doi:10.1109/MTD.2014.15.

Alves, N.S.R., Ribeiro, L.F., Caires, V., Mendes, T.S., Spinola, R.O., 2014. Towards an ontology of terms on technical debt. In: 2014 Sixth International Workshop on Managing Technical Debt (MTD), pp. 1–7. doi:10.1109/MTD.2014.9.

Baker Jr., R.A., 1997. Code reviews enhance software quality. In: Proceedings of the 19th International Conference on Software Engineering, ICSE '97. New York, NY, USA. ACM, pp. 570–571. doi:10.1145/253228.253461.

Barney, S., Aurum, A., Wohlin, C., 2008. A product management challenge: creating software product value through requirements selection. J. Syst. Archit. 54, 576–593. doi:10.1016/j.sysarc.2007.12.004.

Boehm, B.W., 2006. Value-based software engineering: seven key elements and ethical considerations. In: Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P. (Eds.), Value-Based Software Engineering. Springer, Berlin Heidelberg, pp. 109–132.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., 2010. Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10. New York, NY, USA. ACM, pp. 47–52. doi:10.1145/1882362.1882373.

Charmaz, K., 2014. Constructing Grounded Theory, second ed. SAGE Publications Ltd, Thousand Oaks, CA.

Codabux, Z., Williams, B., 2013. Managing technical debt: an industrial case study. In: 2013 4th International Workshop on Managing Technical Debt (MTD), pp. 8–15. doi:10.1109/MTD.2013.6608672.

Codabux, Z., Williams, B., Niu, N., 2014. A quality assurance approach to technical debt. In: Proc. Int. Conf. Softw. Eng. Res. Pract. SERP Steer. Comm. World Congr. Comput. Sci. Comput. Eng. Appl. Comput. WorldComp.

Cunningham, W., 1992. The wycash portfolio management system (experience report). OOPSLA.

Das, S., Lutters, W.G., Seaman, C.B., 2007. Understanding documentation value in software maintenance. In: Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology, CHIMIT '07. New York, NY, USA. ACM doi:10.1145/1234772.1234790.

Davis, N., 2013. Driving quality improvement and reducing technical debt with the definition of done. In: Agile Conference (AGILE), 2013, pp. 164–168. doi:10.1109/AGILE.2013.21.

De Bruin, T., Freeze, R., Kaulkarni, U., Rosemann, M., 2005. Understanding the main phases of developing a maturity assessment model. In: Campbell, B., Underwood, J., Bunker, D. (Eds.), Faculty of Science and Technology. Presented at the Australasian Conference on Information Systems (ACIS), Australasian Chapter of the Association for Information Systems, CD-ROM, pp. 8–19.

Eisenberg, R.J., 2012. A threshold based approach to technical debt. SIGSOFT Softw. Eng. Notes 37, 1–6. doi:10.1145/2108144.2108151.

Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I., 2015. Measure it? Manage it? ignore It? software practitioners and technical debt. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. New York, NY, USA. ACM, pp. 50–60. doi:10.1145/2786805.2786848.

Falessi, D., Kruchten, P., Nord, R.L., Ozkaya, I., 2014. Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt. SIGSOFT Softw. Eng. Notes 39, 31–33. doi:10.1145/2579281.2579311.

Forward, A., Lethbridge, T.C., 2002. The relevance of software documentation, tools and technologies: a survey. In: Proceedings of the 2002 ACM Symposium on Document Engineering, DocEng '02. New York, NY, USA. ACM, pp. 26–33. doi:10.1145/585058.585065.

Fowler, M., 2009. TechnicalDebtQuadrant [WWW Document]. URL http://martinfowler.com/bliki/TechnicalDebtQuadrant.html (accessed 7.7.14).

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code, first ed. Addison-Wesley Professional, Reading, MA.

Green, R., Ledgard, H., 2011. Coding guidelines: finding the art in the science. Commun. ACM 54, 57–63. doi:10.1145/2043174.2043191.

Griffith, I., Reimanis, D., Izurieta, C., Codabux, Z., Deo, A., Williams, B., 2014. The correspondence between software quality models and technical debt estimation approaches. In: 2014 Sixth International Workshop on Managing Technical Debt (MTD), pp. 19–26. doi:10.1109/MTD.2014.13.

Guo, Y., Seaman, C., 2011. A portfolio approach to technical debt management. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. New York, NY, USA. ACM, pp. 31–34. doi:10.1145/1985362.1985370.

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., da Silva, F.Q.B., Santos, A.L.M., Siebra, C., 2011. Tracking technical debt #x2014; an exploratory case study. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 528–531. doi:10.1109/ICSM.2011.6080824.

Holvitie, J., Leppänen, V., 2013. DebtFlag: technical debt management with a development environment integrated tool. In: Proceedings of the 4th International Workshop on Managing Technical Debt, MTD '13. Piscataway, NJ, USA. IEEE Press, pp. 20–27.

Kemerer, C.F., Paulk, M.C., 2009. The impact of design and code reviews on software quality: an empirical study based on PSP data. IEEE Trans. Softw. Eng. 35, 534–550. doi:10.1109/TSE.2009.27.

Klinger, T., Tarr, P., Wagstrom, P., Williams, C., 2011. An enterprise perspective on technical debt. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. New York, NY, USA. ACM, pp. 35–38. doi:10.1145/1985362.1985371.

Krishna, V., Basu, A., 2012. Minimizing Technical Debt: developer's viewpoint. In: International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), pp. 1–5. doi:10.1049/ic.2012.0147.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. IEEE Softw. 29, 18–21. doi:10.1109/MS.2012.167.

Kruchten, P., Nord, R.L., Ozkaya, I., Visser, J., 2012. Technical debt in software development: from metaphor to theory report on the third international workshop on managing technical debt. SIGSOFT Softw. Eng. Notes 37, 36–38. doi:10.1145/2347696.2347698.

Lee, A.S., Baskerville, R.L., 2003. Generalizing generalizability in information systems research. Inf. Syst. Res. 14, 221–243. doi:10.1287/isre.14.3.221.16560.

Lehtola, L., Kauppinen, M., 2006. Suitability of requirements prioritization methods for market-driven software product development. Softw. Process Improv. Pract. 11, 7–19. doi:10.1002/spip.249.

Lethbridge, T.C., Singer, J., Forward, A., 2003. How software engineers use documentation: the state of the practice. IEEE Softw. 20, 35–39. doi:10.1109/MS.2003.1241364.

Letouzey, J.-L., 2012. The SQALE method for evaluating technical debt. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD '12. Piscataway, NJ, USA. IEEE Press, pp. 31–36.

Letouzey, J., Ilkiewicz, M., 2012. Managing technical debt with the SQALE method. IEEE Softw. 29, 44–51. doi:10.1109/MS.2012.129.

Lim, E., Taksande, N., Seaman, C., 2012. A balancing act: what software practitioners have to say about technical debt. IEEE Softw. 29, 22–27. doi:10.1109/MS.2012.130.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220. doi:10.1016/j.jss.2014.12.027.

Li, Z., Liang, P., Avgeriou, P., 2015. Architectural technical debt identification based on architecture decisions and change scenarios. In: Proc. 12th Work. IEEEIFIP Conf. Softw. Archit. WICSA..

Mantyla, M.V., Lassenius, C., 2009. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. 35, 430–448. doi:10.1109/TSE.2008.71.
McConnell, S., 2007. Technical Debt-10x Software Development | Construx [WWW Document]. URL http://www.construx.com/10x_Software_Development/Technical_Debt/. (accessed 3.25.14)
Nord, R.L., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M., 2012. In search of a metric for managing architectural technical debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)., pp. 91–100. doi:10.1109/WICSA-ECSA.212.17.
Norton, M., 2009. Doc On Dev: Messy Code is not Technical Debt.
Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V., 1993. Capability maturity model, version 1.1. IEEE Softw. 10, 18–27. doi:10.1109/52.219617.
Power, K., 2013. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options. In: 2013 4th International Workshop on Managing Technical Debt (MTD), pp. 28–31. doi:10.1109/MTD.2013.6608675.
Ramasubbu, N., Kemerer, C.F., Woodard, C.J., 2015. Managing technical debt: insights from recent empirical evidence. IEEE Softw. 32, 22–25. doi:10.1109/MS.2015.45.
Robson, C., 2002. Real World Research, second ed. Wiley- Blackwell.
Runeson, P., Höst, M., 2008. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14, 131–164. doi:10.1007/s10664-008-9102-8.
Seaman, C.B., 1999. Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. 25, 557–572. doi:10.1109/32.799955.
Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., Vetro, A., 2012. Using technical debt data in decision making: potential decision approaches. In: 2012 Third International Workshop on Managing Technical Debt (MTD), pp. 45–48. doi:10.1109/MTD.2012.6225999.
Seaman, C., Nord, R.L., Kruchten, P., Ozkaya, I., 2015. Technical debt: beyond definition to understanding report on the sixth international workshop on managing technical debt. SIGSOFT Softw. Eng. Notes 40, 32–34. doi:10.1145/2735399.2735419.
SonarQube, 2015. http://www.sonarqube.org/evaluate-your-technical-debt-with-sonar/ Accessed: 2015-10-29.
Strauss, A., Corbin, J.M., 1998. Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory. SAGE Publications.
Tamburri, D.A., Kruchten, P., Lago, P., Van Vliet, H., 2013. What is social debt in software engineering? In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 93–96. doi:10.1109/CHASE.2013.6614739.
Theodoropoulos, T., Hofberg, M., Kern, D., 2011. Technical debt from the stakeholder perspective. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. New York, NY, USA. ACM, pp. 43–46. doi:10.1145/1985362.1985373.
Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86, 1498–1516. doi:10.1016/j.jss.2012.12.052.
Yin, R.K., 2003. Case Study Research: Design and Methods. Sage Publications, Thousand Oaks, Calif.
Yli-Huumo, J., Maglyas, A., Smolander, K., 2015. The benefits and consequences of workarounds in software development project. 6th International Conference on Software Business.
Yli-Huumo, J., Maglyas, A., Smolander, K., 2014. The sources and approaches to management of technical debt: a case study of two product lines in a middle-size finnish software company. In: Jedlitschka, A., Kuvaja, P., Kuhrmann, M., Männistö, T., Münch, J., Raatikainen, M. (Eds.), Product-Focused Software Process Improvement, Lecture Notes in Computer Science. Springer International Publishing, pp. 93–107.
Yli-Huumo, J., Rissanen, T., Maglyas, A., Smolander, K., Sainio, L.-M., 2015. The relationship between business model experimentation and technical debt. 6th International Conference on Software Business.
Zazworka, N., Seaman, C., Shull, F., 2011. Prioritizing design debt investment opportunities. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. New York, NY, USA. ACM, pp. 39–42. doi:10.1145/1985362.1985372.
Zazworka, N., Shaw, M.A., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. New York, NY, USA. ACM, pp. 17–23. doi:10.1145/1985362.1985366.
Zazworka, N., Spínola, R.O., Vetro', A., Shull, F., Seaman, C., 2013. A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13. New York, NY, USA. ACM, pp. 42–47. doi:10.1145/2460999.2461005.
Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., Shull, F., 2014. Comparing four approaches for technical debt identification. Softw. Qual. J. 22, 403–426. doi:10.1007/s11219-013-9200-8, n.d..

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.