Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study

Hadi Ghanbari
University of Jyvaskyla, Finland
hadi.ghanbari@jyu.fi

Abstract

In recent years, the metaphor of technical debt has received considerable attention, especially from the agile community. Still, despite the fact that agile practices are increasingly used in critical domains, to the best of our knowledge, there are no studies investigating the occurrence of technical debt in critical software development projects. The results of an exploratory field study conducted across several projects reveal that a variety of business and environmental factors cause the occurrence of technical debt in critical domains. Using Grounded Theory method, these factors are categorized as ambiguity of requirement, diversity of projects, inadequate knowledge management, and resource constraints to form a theoretical model. Following previous studies we suggest that integrating agile practices, such as iterative development, review meetings, and continuous testing, into common plan-driven processes enables development teams to better identify and manage technical debt.

1. Introduction

Despite all the financial and human resources that have been spent on large software and system development projects, the majority of these projects continue to fail or face severe challenges [1]. Although different technical and human problems might be the potential cause for software vulnerabilities and failure, previous research has shown that software defects are the main cause of most software vulnerabilities [2, 3]. Even though identifying and fixing software deficiencies such as bugs, missing requirements or flaws in software design [4] has major importance in increasing the quality and reliability of software products, some of these defects might stay hidden; even if they are identified, they may not be fixed rapidly [3]. In addition, fixing software deficiencies at the later stages of projects becomes more expensive and time consuming [5, 6]. Thus such deficiencies must be avoided in the first place, especially in critical systems where software failure might cause devastating financial and infrastructural consequences, or human life loss or injuries [3, 7].

In response to these problems, the software community has been mainly attempting to identify new software development tools and methods. Over time, thousands of software development methods were designed to manage complexity in software projects [8, 9]. However, it is widely reported in previous studies that these software development methods are rarely followed in their entirety but are customized [10-12]. A group of scholars explain this customization mainly in terms of quality compromising trade-offs for minimizing development costs and delivery times [13-16] . In such situations, developers are often forced or motivated to cut back on software development processes or to postpone certain activities [17, 18]. The metaphor of technical debt [19] has been increasingly used to point out such quality compromising shortcuts [15, 17, 20-22].

While minimizing development time and costs might play a key role in highly competitive markets [23, 24], quality of software often has a higher priority in developing critical systems [7]. Since technical debt has a negative impact on software quality, it must be avoided when developing critical systems. However, previous studies showed that technical debt does not always occur because of bad design and development decisions but also due to environmental factors that cannot be controlled by

development teams [17, 21]. Therefore, it is important for development teams to identify sources of technical debt in their context and to properly manage it in order to maintain software quality [20].

While in recent years technical debt has received considerable attention from the agile community [20, 21, 25, 26], to the best of our knowledge, there are no studies that explore this phenomenon in critical software projects. As it is suggested by [21], there is a need for empirical studies to investigate the potential sources of technical debt across development contexts. Therefore, we performed an exploratory field study to gain a better understanding of the nature of technical debt and its potential sources in critical domains.

The results of our study show that even in critical projects there are a set of common issues and challenges that might lead to the occurrence of technical debt. In particular our results provide an understanding of the circumstances under which software developers might make quality compromising trade-offs. These results can assist software development teams to better understand and consider the consequences of their decisions while making trade-offs between the productivity and quality of software processes. We propose that combining agile practices with plan-driven processes brings flexibility into critical software projects and, as a result, enables development teams to avoid or at least better manage technical debt in these projects.

The rest of this paper is structured as follows. In the next section, a brief overview of previous studies is provided. In the third section, the research method and research settings are explained. The paper continues in the fourth section with reporting research results and key findings. These findings are then discussed in the fifth section. Finally, the sixth section provides some concluding thoughts.

2. Related Work

In today's highly competitive business environment, development teams are under constant pressure to produce high-quality software in a shorter time and with minimum amount of costs [23, 24]. However, since producing high-quality software is usually associated with higher costs and delivery times [7], maximizing both software development productivity and software quality simultaneously becomes challenging. Software development productivity is often measured based on the number of lines of new code produced per person-day [27, 28] while software quality is measured based on the number and frequency of defects identified in the software products [27-29].

Sometimes firms have to make trade-offs between long-term software quality and short-term productivity [15, 22, 30]. In such situations often quality practices are neglected to deal with the urgent demands imposed by the business environment [13, 16]. For example, according to [16], planned tests are often neglected or postponed due to insufficient amount of time. A group of studies use the metaphor of technical debt [19] to explain such quality compromising trade-offs [15, 17, 21, 22].

The term technical debt has been originally introduced by Cunningham [19] to point out poorly written code. However, in last two decades the metaphor has been used in a variety of ways to explain flaws and imperfections in documentation, design, coding and testing activities [17, 20-22]. In this study, following [15, 17, 22] we use the definition of technical debt as conscious decisions to cut back on software development processes in order to minimize development costs and delivery times.

The most common demands reported in the literature to be the cause of technical debt are time pressure and insufficient amount of budget and human resources [15, 18, 21, 31]. However, some studies show that technical debt could be the result of environmental factors which are out of control of development teams [17, 21].

Even though it may sometimes be necessary for organizations to take on technical debt, such decisions lead to higher levels of software deficiency and complexity [21]. This is even more problematic in bigger projects where a larger number of developers simultaneously develop different parts of the system, and the increased complexity makes it difficult—and sometimes even impossible—for them to develop and maintain the software [5, 6].

While development costs and delivery time are important aspects of every software project, in developing critical systems factors such as software reliability and maintainability are of major importance. Due to high failure costs and consequences a set of expensive and trusted software development methods must be utilized for developing critical systems [7]. Often in such projects, developers follow plan-driven and document-centric software processes to show compliance with certain standards [4]. Despite using such strictly defined and heavily planned processes [4], development teams might sometimes ignore predefined processes and given standards due to the occurrence of unexpected business and organizational issues. As a result, the occurrence of technical debt becomes unavoidable even in critical software projects. Hence,

development teams must be aware of and manage technical debt to maintain software quality [20, 21].

3. Research Methodology

In this research, our aim is to gain an understanding about the nature of technical debt and its potential sources in critical domains. As it is suggested by [32], conducting exploratory studies is a suitable method for gaining such knowledge and generating insights about the phenomenon under study. Hence, we decided to conduct a two-stage exploratory field study to build an empirically grounded understanding about the nature of technical debt in critical software projects.

3.1. Data Collection

Following the qualitative interview guidelines suggested by [33], two rounds of face-to-face, semi-structured interviews with international software engineers were conducted. It must be mentioned that all of these interviewees hold a university degree in Software Engineering or relevant fields (see Table 1).

Table 1. Interview participants.

	ID	Role(s)	Experience in years	Domain
Stage 1	I 1	Software Engineer	8	Commerce
	I 2	Software Engineer	3	Healthcare
	I 3	Software Engineer/ Process Manager	9	Automotive
	I 4	Software Engineer	11	Automotive
	I 5	Software Engineer	14	Automotive
Stage 2	I 6	Software Engineer/ Business Manager	22	Aerospace
	I 7	Software Engineer/ Team Leader	7	Aerospace
	I 8	Software/System Engineer	6.5	Aerospace
	I 9	Software Engineer/ Project Manager	6	Aerospace
	I 10	Software Engineer/ Team Leader	8.5	Aerospace
	I 11	Software Engineer	20	Aerospace
	I 12	Software Engineer/ Team Leader	8	Aerospace

An interview protocol was prepared and continuously improved to guide all the interviews. Each of these interviews lasted from 60 to 120 minutes. After obtaining permission from each interviewee, the interviews were recorded and transcribed for further data analysis.

3.1.1. Preliminary interviews. In Stage 1, we interviewed five international software developers from the automotive, healthcare, and financial sectors. The aim of these interviews was to investigate whether technical debt might occur in such critical domains. Since the results from these interviews provided some initial evidence that technical debt occurs even in developing critical systems, we decided to further our investigation during the second stage.

3.1.2. Case study. In Stage 2, we conducted a single-case case study [32] in a company called AERO (pseudonym) that is active in the aerospace domain. During this stage we interviewed seven international software engineers from AERO. Due to the diversity of projects in the company, we decided to interview software experts with a variety of work experience from different teams.

In addition to the interview data, supplementary data sources such as firm's official procedures, project documents and public information available on their website were used to analyze different aspects of development processes. Since a large amount of data was collected, we used a tool called NVivo[1] for proper data analysis and management.

3.2. Data Analysis

To build an understanding that is empirically grounded in the experience of professionals involved in software processes, a systematic data analysis process was conducted following the techniques suggested by the Glaserian Grounded Theory method (as cited in [37]). Using NVivo, we first performed a line-by-line open coding to closely examine fractures of data and to form categories of codes [37]. During the next stage, selective coding, we grouped these categories of codes into four higher levels of abstraction (i.e. concepts) called *ambiguity of requirements, diversity of projects, inadequate knowledge management, and resource constraints*. These categories represent the challenging aspects of software processes from our interviewees' perspective, which might lead to the occurrence of technical debt in critical software projects. Finally, during the last stage of data analysis, theoretical coding, a theoretical model was formed by indicating the relations between the identified selective codes. The theoretical model is discussed in the next section.

It is worth noting that a constant comparison through iterative data collection and analysis enabled us to enrich the emerging theoretical concepts by

[1] http://www.qsrinternational.com/products_nvivo.aspx

identifying shortcomings in the collected data and to address them by collecting more data.

3.3. Case Description

AERO is a private international company that is active in the aerospace domain. The company consists of several sites and teams—each of them formed by highly educated international individuals with a wide range of technical skills and work experience. Each of these teams is focused on certain types of projects, including systems and software engineering and research and development (R&D) projects. These development teams are small and consist of a few engineers. Thus instead of assigning dedicated experts to each phase, often all of the team members are involved in every stage. Still some team members might have more responsibilities (e.g. the team leader) depending on their individual skills and experience.

Due to the criticality of the aerospace domain, the companies active in this field, including AERO, are expected to comply with certain standards and regulations such as ECSS-E-ST-40C [34] and ECSS-Q-ST-80C [35]. To comply with these standards a V-model [36] is followed in the case company (see Figure 1).

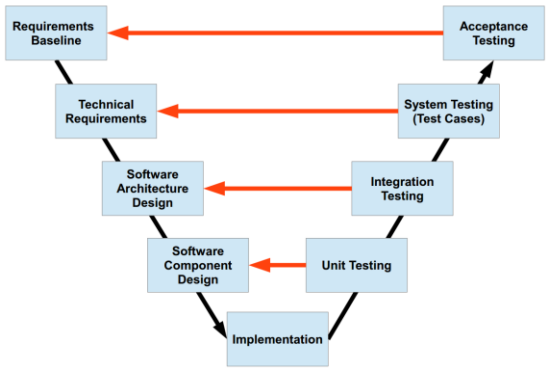

Figure 1. V-model used in the case company.

Depending on the nature of the projects and teams, the V-model is often customized in order to suit the development teams' requirements. As a result, the software development processes followed by some teams is closer to traditional linear models while in other teams, especially in R&D projects, more flexible approaches are followed.

At the moment, a wide variety of tools are utilized in the case company to support teams' day-to-day activities. However, the extent to which development teams utilize these tools depends on the type of their projects. Some teams mainly use simple tools and technologies, while in other teams more advanced toolchains and technologies are used in order to deal with the complexity and criticality of products under development. Additionally, when needed, some teams use configuration management systems, code repositories, and other tools for reporting and tracking bugs and source code documentation.

4. Study Results

Integrating dependability and safety requirements into software has a significant importance in the aerospace domain. Therefore, one of the key goals of AERO is to constantly enhance software development processes that enable developers to better comply with aerospace standards. However, during the data analysis phase, we identified four important categories of factors that make this challenging. In the following sections, we describe these four categories and explain how these factors might force developers to customize given software processes and, as a result, lead to the occurrence of technical debt.

4.1. Ambiguity of Requirements

The first issue is related to requirements analysis and specification. Requirements engineering and management activities in the case company highly depend on the development teams and the nature of their projects. In more operational projects, a statement of the work that consists of the main system requirements is usually provided by the customer. In such projects, extensive requirements and engineering approaches are followed by using different tools in order to identify, document and trace the customers' requirements during the project. On the other hand, in more flexible R&D projects, informal and iterative approaches are preferred. In such projects, the customers' requirements are not reflected in a clear and precise way but more in the form of a long-term vision for identifying new and innovative solutions.

Even though a set of high-level requirements are suggested by the customer in the statement of work, development teams need to break these requirements down into a set of more detailed and feasible technical requirements. Following this stage, they need to prepare a convincing requirements specification document that indicates all the suggested requirements comply with the statement of work provided by customer. However, it is almost impossible for developers to fulfill all the customers'

requirements within the fixed budget assigned to projects. For example, one of our interviewees said:

> *"It is difficult to keep the requirements feasible within the agreed budget since it is hard to anticipate the effort required to implement each of those requirements." – I 7*

This issue is more problematic in competitive projects where an official proposal must be prepared and sent to the customer. Usually in such projects the development team has no opportunity to have a direct discussion and negotiation with the customer before preparing the proposal. Depending on the novelty of the system under development, these requirements might be described precisely and in detail or in a high level and ambiguous way. Often in R&D projects, the concept of the system and its requirements evolve over time. Thus the identified requirements need to be changed and improved constantly during the project and, as a result, more iterative and flexible practices are needed to maintain and to keep track of these changes. Keeping in mind that in critical domains projects are often planned in advance and within a fixed budget, it becomes challenging for developers to follow predefined software processes entirely. One of the interviewees describes this problem as follows:

> *"What [customers] think they can do, their requirements, everything is very dynamic over three years of the project. As the project will be evolving, there is a very strong need for flexibility, which is one of the reasons why we are not applying a very formal process."– I 12*

Furthermore, our data show that the intangibility of software products makes it hard for software stakeholders to trace their requirements carefully during the development phase and to validate if all their needs and expectations are fulfilled properly by the final product. Often, only the final software solution is delivered to the customer; therefore, the customer might not be able to evaluate the quality of other artefacts (e.g. architectural design or requirements document) or processes. Since such aspects of software development are not visible to customers and authorities, in case certain requirements are dropped or practices ignored, it is almost impossible to identify them:

> *"We have standards for everything. We are supposed to comply with the coding standards that are there. Now I'm saying 'supposed to' because who really verifies them? From my own knowledge, nobody does." – I 6*

In addition to this and despite the fact that in each team there are several internal technical reviews to evaluate the quality of the products and processes, an official and precise quality review mechanism is missing in the company. As a result, it is very demanding for teams to indicate if the end results comply with the recommended guidelines and standards. As can be seen from the following quote mentioned by one of our interviewees, this becomes more problematic in more research-oriented projects where no official feedback is provided by customers and, as a result, developers do not have the opportunity to receive feedback regarding their performance or any potential defects.

> *"We haven't had much luck in convincing the project partners who are playing the role of the end-user to actually spend some time on using our deliveries and to provide valuable feedback to us." – I 9*

Therefore, it becomes almost impossible for teams to identify and fulfill a complete set of requirements based on customers' expectations. As a result, it is likely that some features or requirements are missing from the final product or have not been implemented according to the standards requested by the customers or authorities.

4.2. Diversity of Projects

In AERO, different teams are active in a variety of projects. The diversity of projects might reduce the quality of communication and information transmission between different teams and, consequently, the collaboration between these teams becomes problematic. For example, several interviewees mentioned that there might be similar activities and projects that are going on simultaneously in different teams but these overlapping efforts are not communicated properly.

Additionally, due to the diversity of the projects going on in the company, each team might follow certain kinds of development processes and practices.

> *"Since there are really different projects here, each team defines its own ways of doing their work and, because of that, everybody has a really different approach." – I 10*

While in more critical and operational projects, development teams utilize advanced tools for

preparing extensive design documents, in more flexible development projects the architectural design is usually prepared in a more informal and iterative manner to deal with the high rates of requirement change and the evolving nature of the product. Thus there are obvious differences between the software components and documents produced by different teams. This might be problematic because sometimes the software components produced by one team are needed to be used by other teams. Therefore, it might be challenging for developers from other teams to understand and make sense of that component.

> *"It [has] happened that I used code that is written by developers from other groups and I could completely see the difference […] For me, it was hard to understand some parts of the comments that are very important." – I 11*

This becomes even more problematic if the person who has originally developed the component is not working in the team or company anymore, which makes it impossible for other developers to easily solve potential ambiguities and misunderstandings. Additionally, due to diversity of projects, development teams need to use tools and technologies differently. While in large projects, there is an inevitable need for different kinds of advanced tools to assist developers in tracing a large number of requirements using such advanced tools in smaller projects might be seen unnecessary.

> *"In bigger projects, tools are for sure necessary, but the risk of using a tool in smaller projects, where things are done manually and quickly, is to spend more time using the tools than doing the technical work" – I 7*

Thus, using different kinds of tools with different functionalities seems to be unavoidable among these teams and, as a result, interoperability between different tools becomes difficult. The inconsistency between tools and technologies exchanging artefacts between teams might not be easy or straightforward, and developers need to spend extra time on making these products usable.

Our data show that in the case company developers often learn to use those practices and tools that are used within their teams. Therefore, in case they switch from one team to another, they need to spend some time to familiarize with the tools and practices utilized within the new team.

Additionally, we realized that if developers consider new software development processes or tools to be outdated or time-consuming, they might underestimate the value of these processes and tools. For example one of the interviewees mentioned:

> *"That was my best practice five years ago. I mean, this is an obsolete practice for me, it would be a regression to comply with certain rules of the company." – I 6*

Therefore, it seems that the diversity of projects makes it difficult for teams to use a consistent set of software development practices and toolsets, which makes it challenging for developers to follow planned software processes.

4.3. Inadequate Knowledge Management

Both technical and product knowledge are among the most essential resources for performing software development activities. It is suggested in previous studies that knowledge documented in a software company or held by its employees is one of the key competitive assets of that firm [4]. In the case company, extensive documentation is often required by regulations and, therefore, teams might spend a considerable amount of time and effort to fulfill this requirement. The following quotation indicates an example of such extensive documentation.

> *"In this company we follow a Waterfall approach because our projects are heavily document-centric. That is the reason why we often develop more documents than software." – I 6*

Using information from previous projects stored on company-wide data servers is one of the core knowledge management sources in the company. However, according to our data, it seems that this information is not stored in a structured way and is not maintained regularly. As a result, searching and finding the needed information might be problematic and time consuming for developers.

> *"We have most of the documentation from past projects in a server, which you can search to a certain degree. There isn't such a big history or database, but it is just a matter of trying and seeing what you can find." – I 10*

The majority of the employees in the company has same levels of education and basic knowledge of the aerospace industry. Still, if individuals move from one team to another, the archives from previous projects can be a key source of information for them to familiarize with the overall practices of the new team. Thus, the lack of a well-structured and updated

source of information forces them to spend extra time and effort to gain necessary knowledge. Due to such shortcomings, it is likely that technical debt occurs during knowledge-creation and management.

4.4. Resource Constraints

In the case company, the software processes are often extensively planned and a fixed budget is allocated to projects. However, during the data analysis phase, we realized that it is very common for development teams to run out of time and budget before completing the software processes; in many cases, they are forced to minimize the software development activities. Additionally, the lack of human resources is another issue that makes performing activities challenging. In some projects, especially if there are insufficient human resources, the same developers are responsible for performing all the software development activities—from requirements engineering to system testing. Thus, it might be impossible for them to perform every single step or activity as per the recommended standards.

Lack of human resources becomes even more challenging when projects are behind schedule and the deadline is closed. This is problematic especially with software evaluation and testing because software evaluation often is the last step in many projects. Therefore, testing and verification activities might be postponed to the delivery time or even pushed to the customer side. This issue was mentioned by one of the respondents as follows:

> *"Let's be clear or honest. If we have one guy, we have one guy, huh? We have a formal acceptance where the customer himself is supposed to be the independent tester at the end. It's a way of pushing the verification to the customer side somehow" – I 6*

This is problematic if stakeholders do not have enough resources to conduct proper software testing and verification, which eventually might lead to the delivery of defective software. As a result, the defects in the software products might not be identified at the time of delivery but only when the system is in use. Fixing these bugs not only requires extra time and effort but also becomes more challenging when the software is complete and operational.

When projects are delayed, additional human resources might be needed to accelerate the software processes and to follow the delivery schedule. In such situations, if the agreement is more flexible the development teams might be able to acquire additional resources (e.g. budget and time) from the customer. However, in fixed-bid contracts where the budget and deadline cannot be extended, development teams must decide either to assign additional resources themselves or to perform software development activities with the existing resources. The first option often has some cost overloads for the firm, which might lead to a reduction in turnover or even financial loss.

> *"We go to the customer, communicate the problem, and try to de-scope the things that are less prioritized. We can only internally decide that we'll accept less profit or no profit from the project, so that we can invest more time." – I 10*

The second option, on the other hand, might motivate or even force developers to simplify software development activities to keep costs and delivery times fixed. Thus, based on our data, it seems that resource constraints might lead to the occurrence of technical debt even in critical projects.

4.5. Theoretical Model

In previous sections, we discussed the four main categories that make software processes challenging in critical domains, particularly in the case company. As a result, development teams might not be able to perform planned development activities as recommended by given standards, which might eventually lead to the occurrence of technical debt. These four categories include ambiguity of requirements, diversity of projects, inadequate knowledge management, and resource constraints. Figure 2 shows our theoretical model, which is formed by indicating the relations between these identified selective codes.

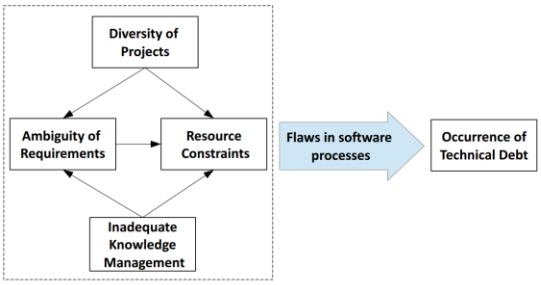

Figure 2. Technical debt in critical domains.

As shown in Figure 2, the diversity of projects influences both the ambiguity of requirements and resource constraints. Depending on the project, stakeholders' requirements might be vague or well defined. On the other hand, the amount of resources allocated to projects highly depends on the type of

the project. Inadequate knowledge management worsens the ambiguity of requirements due to the fact that performing proper requirements engineering and specification becomes challenging. Inadequate knowledge management alongside with ambiguity of requirements makes it difficult for development teams to perform a precise cost and effort estimation and, as a result, evaluation of necessary development resources becomes difficult. This might lead to a lack of necessary resources later in the project.

Under these conditions, it becomes very challenging for development teams to fully follow planned software processes and to comply with recommended standards. Therefore, certain practices or activities might be ignored or not performed properly, which leads to the occurrence of technical debt.

5. Discussion

Depending on the development context, there might be different constraints and regulations that force development teams to concentrate more on certain aspects of software development processes. Often in critical domains, quality of the systems and compliance with certain standards has a higher importance for stakeholders, due to which plan-driven processes and expensive techniques are followed by software development teams [4, 7]. However, our data collected from several projects indicate that even in critical domains, pressure caused by different business and organizational sources makes it challenging for developers to follow plan-driven processes. In order to deal with such challenges, development teams might decide to ignore or postpone certain software development activities [4, 16]. As a result of this minimization, technical debt might occur in such critical projects.

Ambiguity of requirements is one of the key factors that make software projects challenging in critical domains. Requirement change is reported by previous studies to be one of the biggest challenges of software projects [24]. Even in critical domains that are considered to be more stable, software requirements might change over the course of a project, especially if these requirements are not clearly defined and specified at the beginning of the project. Such deficiencies in requirements specification might lead to the occurrence of requirements debt [17, 21, 22].

Another issue identified in this study is the diversity of projects in the case company. Despite the fact that all of these projects are performed in the aerospace domain, a variety of processes and practices are followed in the company which makes collaboration between teams challenging. Therefore, following standards and procedures suggested by regulatory authorities becomes challenging.

The availability of necessary resources is another factor that directly affects the way software processes are followed. Often cost and effort estimation is challenging in software projects and it is almost impossible to clearly specify the necessary development resources at the beginning of projects. Our results show that due to a lack of necessary resources, development teams sometimes have to omit certain steps of software processes. Resource constraints are widely reported by previous studies to cause technical debt [15-17, 21, 22, 38].

Finally, inadequate knowledge management is another issue that makes software projects challenging. Technical and product knowledge is one of the key elements integrated into every software development process [4]. Thus any potential obstacle in proper knowledge creation and management might cause severe problems for individuals while performing their activities and, as a result, lead to the occurrence of technical debt. This kind of debt has been reported by previous studies as knowledge distribution and documentation debt [17, 21, 22].

Our analysis shows that the occurrence of technical debt becomes unavoidable even in critical projects. Therefore, it is critical for development teams to properly identify and effectively manage debt [20, 21, 22].

It is suggested by previous studies that using agile practices assist development teams to reduce and manage technical debt [14, 21, 26, 39]. On the other hand a group of studies [4, 39, 40], suggests that following a combination of plan-driven and agile methods in critical projects not only allows teams to perform their tasks in a cost-effective manner but also to comply with the different quality levels requested by customers or regulatory authorities. Following these studies, we suggest that integrating agile practices into common, plan-driven software processes used in critical domains enables development teams to tackle technical debt.

Following practices such as small releases, burndown charts, daily meetings, test-driven development, and continuous testing might assist development teams to avoid technical debt to accumulate in their projects. Using burndown charts and daily meetings help developers to monitor their progress and to identify potential obstacles in performing their tasks [14, 39, 40]. By this they will be able to better estimate the cost and effort necessary for performing their future tasks. In addition following test driven development and continuous testing [21, 26] enables developers to

identify defects and problems in small releases [14, 39]. As a result, teams are able to deal with their problems as soon as possible by renegotiating or even changing their initial plans as needed [40]. Especially in companies like AERO that have small, collocated teams, communication and collaboration between developers becomes easier and, as a result, teams are more flexible to follow iterative methods.

On the other hand, conducting review meetings and retrospectives and preparing technical debt backlogs [20] enables development teams to properly communicate, trace and manage their technical debt. In addition, organizing company-wide review meetings enables individuals from different teams and departments to communicate their problems and to identify possible solutions to deal with them [4, 14]. Using such meetings, as also suggested by a number of previous studies [14, 26, 39], enables teams to be engaged in more frequent information exchange and, as a result, better communicate any accrued technical debt. In addition, the general awareness of teams regarding the potential sources of technical debt increases. One of the most important benefits of such awareness is to avoid spending resources on overlapping attempts for identifying solutions that have already been identified by other teams [14].

It must be noted that this study has some limitations that might affect the validity of the results. First of all, our observations are based on a limited number of interviews. Even though we tried to compensate this threat by collecting data from several critical projects, our results cannot be fully generalized to other contexts. In addition, the data collected from interviewees were analyzed and interpreted by the researcher and, therefore, the findings might be biased by his personal perspectives. To address these limitations and to improve the generalizability of our results, further research is needed. In particular, there is a need for more empirical studies to further investigate the occurrence of technical debt and its underlying causes across critical domains and in different types of software projects.

6. Conclusions

We conducted an exploratory field study to gain an understanding about the nature of technical debt and its potential sources in critical domains. Upon collecting data from several projects, we discovered a set of challenges that software developers face in critical domains. Even though this study is a preliminary attempt at exploring the nature of technical debt in critical domains, it has some lessons for both scholars and practitioners. Our results reveal technical debt might occur even in critical software projects where certain standards and costly software engineering processes must be followed. Often due to requirement ambiguity, diversity of projects, inadequate knowledge management, and resource constraints software developers are forced to minimize software processes by ignoring certain practices or postponing certain activities.

According to our observations and following previous studies, we suggest that utilizing certain agile practices, such as conducting daily stand-up and regular review meetings, preparing burndown charts and technical debt backlogs, following iterative development and dividing projects into small releases alongside with continuous testing might assist developers to avoid, or at least identify and manage, accrued technical debt. However, further research is needed to support our suggestions and to investigate the effectiveness of agile practices to manage technical debt in critical software projects.

7. Acknowledgments

The author would like to thank the interviewees and also the anonymous reviewers for their valuable contributions and insights. This work was funded by Tekes (the Finnish Funding Agency for Technology and Innovation) and ITEA 2 (Information Technology for European Advancement) - program.

8. References

[1] The Standish Group International, Inc., “The Chaos Summary 2009", 2009.

[2] J. Fonseca, and M. Vieira, “Mapping Software Faults with Web Security Vulnerabilities", IFIP International Conference on Dependable Systems and Networks, IEEE, 2008, pp. 257-266.

[3] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, “Mining bug databases for unidentified software vulnerabilities", Fifth International Conference on Human System Interactions, IEEE, 2012, pp. 89-96.

[4] J. P. Notander, M. Höst, and P. Runeson, “Challenges in flexible safety-critical software development–an industrial qualitative survey”, In Product-Focused Software Process Improvement, Springer, Berlin Heidelberg, 2013, pp. 283-297.

[5] R. D. Banker, G. B. Davis, and S. A, Slaughter, “Software development practices, software complexity, and software maintenance performance: A field study”, Management Science, 44(4), 1998, pp. 433-450.

[6] E. Van Emden, and L. Moonen, “Java quality assurance by detecting code smells”, Ninth Working Conference on Reverse Engineering, IEEE, 2002, pp. 97-106.

[7] Sommervile I., Software engineering (10th ed), Pearson, 2015.

[8] D. Avison, and G. Fitzgerald, “Where Now for Development Methodologies?”, Communications of the ACM, 46(1), 2003, pp. 79-82.
[9] J. Iivari, and J. Maansaari, “The usage of systems development methods: are we stuck to old practices?”, Information and Software Technology, 1998, pp. 501-510.
[10] B. Boehm, and R. Turner, “Observations on balancing discipline and agility”, Proceedings of the Agile Development Conference, 2003, pp. 32-39.
[11] K. Conboy, and B. Fitzgerald, “Method and developer characteristics for effective agile method tailoring”, ACM Transactions on Software Engineering and Methodology, 20(1), 2010, pp. 1-30.
[12] R. Baskerville, B. Ramesh, L. Levine, J. Pries-Heje, and S. A. Slaughter, “Is Internet-speed Software Development Different”, IEEE Software, 2004, pp. 237-264.
[13] T. Vartiainen, and M. T. Siponen, “What Makes Information System Developers Produce Defective Information Systems For Their Clients?”, Proceedings of Pacific Asia Conference on Information Systems, 2012.
[14] Z. Codabux, and B. Williams, “Managing technical debt: An industrial case study”, Proceedings of the 4th International Workshop on Managing Technical Debt, IEEE Press, 2013, pp. 8-15.
[15] S. McConnell, “Technical Debt”, 2007, http://www.construx.com/10x_Software_Development/Technical_Debt/.
[16] J. J. Ahonen, and T. Junttila, “A case study on quality-affecting problems in software engineering projects”, IEEE International Conference on Software: Science, Technology and Engineering, 2003, pp. 145-153.
[17] E. Tom, A. Aurum and R. Vidgen, “An exploration of technical debt”, Journal of Systems and Software, 86(6), 2013, pp. 1498-1516.
[18] R. D. Austin, “The effects of time pressure on quality in software development: An agency model”, Information Systems Journal, 12(2), 2001, pp. 195-207.
[19] W. Cunningham, “The WyCash portfolio management system”, Addendum to the proceedings on Object-oriented programming systems, languages, and applications, British Columbia, Canada, 1992, pp. 29-30.
[20] P. Kruchten, R.L. Nord, and I. Ozkaya, “Technical debt: from metaphor to theory and practice”, IEEE Software, 6, 2012, pp. 18-21.
[21] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, and Others, “Managing technical debt in software-reliant systems”, In Proceedings of the FSE/SDP workshop on Future of Software Engineering Research, ACM, 2010, pp. 47-52.
[22] E. Lim, N. Taksande, and C. Seaman, “A balancing act: what software practitioners have to say about technical debt”, IEEE Software, 2012, pp. 22-27.
[23] A. Eberlein, and J. C. S. P. Leite, “Agile Requirements Definition: A View from Requirements Engineering”, International Workshop on Time-Constrained Requirements Engineering, 2002, pp. 4-8.
[24] I. Sommervile, “Integrated Requirements Engineering: A Tutorial”, IEEE Software, 2005, pp. 16-23.
[25] R. Bavani, “Distributed Agile, Agile Testing, and Technical Debt” IEEE Software, 2012, pp. 28-33.
[26] J. Holvitie, V. Leppanen, and S. Hyrynsalmi, “Technical Debt and the Effect of Agile Software Development Practices on It-An Industry Practitioner Survey”, In Sixth International Workshop on Managing Technical Debt (MTD), IEEE, 2014, pp. 35-42.
[27] M. Cusumano, A. MacCormack, C. F. Kemerer, and B. Crandall, “Software development worldwide: The state of the practice”, IEEE Software, 2003, pp. 28-34.
[28] A. MacCormack, C. F. Kemerer, M. Cusumano, and B. Crandall, “Trade-offs between productivity and quality in selecting software development practices”, IEEE Software, 2003, pp. 78-85.
[29] S. Kan H., Metrics and models in software quality engineering, Addison-Wesley Longman Publishing Co., Boston, USA, 2002.
[30] A. Potdar, and E. Shihab, “An Exploratory Study on Self-Admitted Technical Debt”, In International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 91-100.
[31] N. Nan, and D. E. Harter, “Impact of budget and schedule pressure on software development cycle time and effort”, IEEE Transactions on Software Engineering, 2009, pp. 624-637.
[32] P. Runeson, and M. Höst, “Guidelines for conducting and reporting case study research in software engineering”, Empirical Software Engineering, 2009, pp. 131-164.
[33] M.D. Myers, and M. Newman, “The qualitative interview in IS research: Examining the craft”, Information and Organization, 2007, pp. 2-26.
[34] ESA Requirements and Standards Division, “ECSS-Q-ST-80C: Space product assurance: Software product assurance”, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2009.
[35] ESA Requirements and Standards Division, “ECSS-E-ST-40C: Space engineering: Software”, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2013.
[36] B.W. Boehm, “Verifying and validating software requirements and design specifications”, IEEE Software, 1984, pp. 75.
[37] C. Urquhart, H. Lehmann, and M.D. Myers, “Putting the ‘theory’ back into grounded theory: guidelines for grounded theory studies in information systems”, Information Systems Journal, 2010, pp. 357-381.
[38] B. Yang, H. Hu, and L. Jia, “A study of uncertainty in software cost and its impact on optimal software release time”, IEEE Transactions on Software Engineering, 2008, pp. 813-825.
[39] F. McCaffery, M. Pikkarainen, and I. Richardson, “Ahaa--agile, hybrid assessment method for automotive, safety critical smes”, 30th International Conference On Software Engineering, ACM/IEEE, 2008, pp. 551-560.
[40] J. G. A. Silva, and P. R. D. Cunha, “Reconciling the irreconcilable? A software development approach that combines Agile with Formal”, In Proceedings of the 39th Annual Hawaii International Conference on System Sciences, IEEE, 2006, pp. 216b-216b.