Investigating the Causes of Software Technical Debt at the Architectural Level

by

Jen Tsu Hsu

B.A.Sc., The University of British Columbia, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

September 2018

© Jen Tsu Hsu, 2018
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, a thesis/dissertation entitled:

_Investigating the Causes of Software Technical Debt at the Architectural Level_

Submitted by Jen Tsu Hsu in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical and Computer Engineering

**Examin ing Committee:**

Dr. Philippe Kruchten, Electrical and Computer Engineering
Supervisor

Dr. Ali Mesbah, Electrical and Computer Engineering
Supervisory Committee Member

Dr. Sathish Gopalakrishnan, Electrical and Computer Engineering
Supervisory Committee Member

Additional Examiner

**Additional Supervisory Committee Members:**

Supervisory Committee Member

Supervisory Committee Member
Abstract

Architectural technical debt (ATD) describes the consequences of intentionally or unintentionally making architectural design decisions that compromise the internal quality of a software system. These consequences include long-term software evolution and maintenance issues that jeopardize software developer productivity. The architectural aspect of technical debt has been receiving significant interest from technical debt researchers, but many questions concerning architectural technical debt itself have yet to be answered. The main goal of this exploratory multiple-case study is to investigate architectural technical debt’s causes and accumulations at the design decision-level by examining software practitioners’ experiences with architectural technical debt and creating taxonomies that would aid software practitioners and technical debt researchers in the future. We conducted semi-structured interviews with 18 participants and an interview-style online questionnaire with 10 participants. From the data collected, we obtained 53 architectural technical debt issues in 15 non-trivial software intensive projects from eight organizations. The results of this study present two architectural technical debt taxonomies on the causes and accumulations of this issue. Then, we report our findings on current architectural technical debt management practices and practitioner experiences with architectural technical debt.
Lay Summary

In software development, developers often choose to use shortcuts in their code or designs that speed up the current development iteration so that they can better meet the delivery deadlines set by clients or management. These suboptimal decisions, although beneficial in the short-term, create a lower quality software that does not negatively impact on the functionalities of the system but does affect its future evolution and maintenance. This phenomenon is referred to as “technical debt”. In this study, we focus on the software architectural aspect of this phenomenon by trying to understand its causes and accumulation at the software design decision-level by examining industry software practitioners’ experiences with architectural technical debt. Our results will contribute towards the current technical debt research landscape and help future software practitioners and researchers better understand architectural technical debt better.
Preface

This dissertation is an original intellectual product of its the author, Jen Tsu Hsu, under the supervision of Dr. Philippe Kruchten. This work is covered by UBC Behavioural Research Ethics Board Number: H15-03436.
# Table of Contents

ABSTRACT .................................................................................................................................................. iii

L A Y S U M M A R Y ........................................................................................................................................ iv

P R E F A C E ................................................................................................................................................... v

T A B L E O F C O N T E N T S ................................................................................................................................ vi

L I S T O F T A B L E S ....................................................................................................................................... ix

L I S T O F F I G U R E S ....................................................................................................................................... x

L I S T O F A B B R E V I A T I O N S ...................................................................................................................... xi

D E D I C A T I O N .............................................................................................................................................. xii

C H A P T E R 1  I N T R O D U C T I O N ........................................................................................................... 1

1.1 S I G N I F I C A N C E A N D C O N T R I B U T I O N ....................................................................................... 5

1.2 R E S E A R C H O B J E C T I V E S .................................................................................................................. 6

1.3 S T R U C T U R E O F T H I S T H E S I S .......................................................................................................... 7

C H A P T E R 2  R E L A T E D W O R K ............................................................................................................ 8

2.1 T E C H N I C A L D E B T R E S E A R C H L A N D S C A P E ............................................................................. 8

2.2 T E C H N I C A L D E B T P H E N O M E N O N A N D I T S D E F I N I T I O N ......................................................... 9

2.3 T E C H N I C A L D E B T M A N A G E M E N T .................................................................................................. 10

2.3.1 Technical Debt Management: Management Strategy ........................................................................... 10

2.3.2 Technical Debt Management: Measurement ....................................................................................... 11

2.3.3 Technical Debt Management: Identification ....................................................................................... 11

2.4 A R C H I T E C T U R A L T E C H N I C A L D E B T ....................................................................................... 13

2.5 C O N C L U S I O N ..................................................................................................................................... 13

C H A P T E R 3  M E T H O D O L O G Y ........................................................................................................... 15

3.1 C A S E S T U D Y ....................................................................................................................................... 15

3.1.1 Exploratory Case Study .......................................................................................................................... 16

3.1.2 Cases and Units of Analyses – What Is It That We Studied? ................................................................. 16

3.1.3 Case Study Protocol .............................................................................................................................. 16

3.1.4 Triangulation ....................................................................................................................................... 17

3.1.5 Member Checking ................................................................................................................................. 17

3.1.6 Case Study Weaknesses ....................................................................................................................... 18

3.1.7 Ethical Considerations .......................................................................................................................... 18

3.2 D A T A C O L L E C T I O N ........................................................................................................................ 19

3.2.1 Participants ........................................................................................................................................ 19
List of Tables

Table 1. Interview Participant List .................................................................................. 20
Table 2. Questionnaire Participant List ........................................................................... 21
Table 3. Participants' Organizations ................................................................................. 22
Table 4. Distribution of ATD Items ................................................................................... 34
Table 5. Sources of ATD ................................................................................................... 52
Table 6. Why Was ATD Incurred? ..................................................................................... 57
Table 7. Comparison of Sources of ATD taxonomy to Ernst et al.'s ............................... 66
Table 8. Comparison of Driver of ATD to Categories from Martini et al.'s Study .......... 66
List of Figures

Figure 1. Technical Debt Landscape (Kruchten, Nord, & Ozkaya, 2012) .............................................. 6
Figure 2. Distribution of Participant Experiences ................................................................. 22
Figure 3. Technical Debt Item .............................................................................................. 26
Figure 4. Software Project Size Distribution ......................................................................... 32
Figure 5. Software Project Age Distribution .......................................................................... 33
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATD</td>
<td>Architectural Technical Debt</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>TD</td>
<td>Technical Debt</td>
</tr>
</tbody>
</table>
Acknowledgements

First, I would to express my gratitude to my supervisor, Dr. Philippe Kruchten, for patiently guiding me through my time as a graduate student. Without his support, expertise in software engineering, and experience as a researcher, this thesis would not have been possible.

I would also like to thank the software practitioners who have spent their valuable time in participating in this study and sharing their experiences with technical debt.

I would also like to thank Mitacs for providing me with the opportunity to collaborate with an organization and a fund to finish this work.

Most importantly, I am truly grateful to have my parents for all their supports and encouragements. None of this work would have been the same without them.
Dedication

To my parents.
Chapter 1

Introduction

The technical debt (TD) metaphor was introduced by Ward Cunningham in 1992 at the OOPSLA Conference held in Vancouver, British Columbia. In his presentation, Cunningham described the practice of writing immature code to speed up software development is like taking on debts. “A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt” (Cunningham, 1992). In his description, he pointed out that the presence of suboptimal implementation is not a serious issue as such trade-offs are made for the purpose of expediting software development and making on-time deliveries to the clients. However, maintainability and evolvability suffer when the existing technical debt is not managed and the debt interest accumulates. In the long run, if no debt “repayment” is made, the development team productivity will decrease.

The initial usage of the technical debt metaphor was to address code quality issues, but in recent years, its usage has been extended to the whole software lifecycle. In the 2016 Dagstuhl Seminar, “Managing Technical Debt in Software Engineering” (Avgeriou, Kruchten, Ozkaya, & Seaman, 2016), a newer definition of technical debt was agreed on by the attendees to reflect current usage of the metaphor:
“In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.”

In the last paragraph, technical debt is described as being intentionally created. However, there are instances where technical debt can be unintentional, or developers just simply run into debt without doing anything. An unintentional debt, for example, can be caused by low quality code in which the debt was not created through a conscious, decision, but a poorly done job. For debt that developers simply run into, it is usually caused by software systems not keeping up with the evolving operating environment. As a result, such software systems are no longer optimal even though the systems could have been designed and implemented correctly and without shortcuts some time previously.

Even though the term “technical debt” was coined in 1992, the reality it covers was being studied for decades prior to the introduction of the metaphor. Researchers have been looking at this problem under the topics of “software maintenance” and “software evolution”. In the early 1970s, Lehman and Belady (Meir M Lehman, Ramil, Wernick, Perry, & Turski, 1979) had observed the difficulties of the software evolution on their extended study on the IBM OS/360 operating system evolution. They suggested software systems must “be continually updated to maintain their validity and adapt to changed circumstances”, and introduced the Laws of Software Evolution to help developers improve the software development process to better
handle software’s evolution (Manny M Lehman, 1996). The laws that Lehman and Belady proposed still hold true today and the software development industry has become much more aware of such evolution issues as software systems become increasing more complex and difficult to maintain.

There are different types of technical debt such as test technical debt, architectural technical debt, code technical debt, and documentation technical debt (Li, Avgeriou, & Liang, 2015). Identifying and managing these diverse technical debts are essential parts of managing software evolvability and maintainability effort. However, more research is still required to better understand the technical debt phenomenon. In this research, we focus on the software architectural aspect of technical debt known as Architectural Technical Debt (ATD).

Architectural technical debt is a variety of technical debt that manifests at the software architectural level and compromises in some internal quality aspects of a software systems. Software architecture is an area in software design where developer make major choices on the structure of a software system and its required elements. As described in The Rational Unified Process: An Introduction, by Kruchten (2004), software architecture encompasses the following:

- The significant decisions about the organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed together with their behaviour as specified in the collaboration among those elements
The composition of these elements into progressively larger subsystems; the architectural style that guides this organization, these elements, and their interfaces, their collaborations, and their composition

These structural elements might be high-level, such as clients in a client-server architecture, or low-level such as a class in a UML class diagram.

To understand better, here are some examples of architectural technical debt is:

1. A development team did not spend sufficient time in making system components modular and easily expandable.
2. A development team has introduced architectural violations to save some development efforts to meet delivery deadlines.
3. A development team has used a noSQL database instead of a SQL database due to an incorrect requirement given to them.

The following examples are NOT architectural technical debt:

1. Code duplication in source code
2. A function produces incorrect outputs in some occasions. The cause of this issue is uncertain.

Researchers have been studying the ATD phenomenon at either the design- or code-level. At the code-level, the researchers have looked at software implementations for the ATD. At the design-level, the researchers attempted to locate ATDs in software architecture choices made by developers. The problem of searching for architectural technical debt at the code-level is that few real significant ATDs manifest there, and only a small subset of ATDs could be discovered. The
much more serious ATD issues are only visible at the design-level, and their identifications are much difficult to automate.

The goal of our research is to examine industry software practitioners’ experiences with architectural technical debt at the design-level and to create taxonomies to aid practitioners and future researchers in understanding architectural technical debt accumulations and causes.

1.1 Significance and Contribution

Research in technical debt metaphor did not gain significant interest from the academia field until the early 2010s. The first workshop on technical debt, the MTD 2010, was held in 2010 to discuss suggestions for future research on technical debt (Brown et al., 2010). While various research projects on technical debt have been conducted since then, there are still plenty of areas to be explored.

Architectural technical debt, as shown in Figure 1, is primarily an issue with software evolvability. In other words, software developers will experience difficulties in adding new features and updating old components if ATDs are overlooked and not actively managed. Research on this topic has received significant attention from technical debt researchers (Li et al., 2015). However, most research effort on architectural technical debt focuses on technical debt management strategies, code-level ATD identification, as well as ATD definition and understanding. Architectural technical debt’s causes and accumulations in software development industry setting still require more thorough exploration.
The outcome of our study will contribute to the existing research on architectural technical debt by providing taxonomies on the causes and origin of the design-level ATD experienced by software practitioners in the industry. This study will also provide insights gained from architectural technical debt discussions with practitioners and the data collected, and report practitioners’ current practices on managing this issue.

[Image: Technical Debt Landscape (Kruchten, Nord, & Ozkaya, 2012)]

1.2 Research Objectives

One objective of this research, as mentioned in the previous section, is to create taxonomies on the cause and origin of the design-level ATD experienced by software practitioners in the industry. Another objective is to understand the current practice of managing ATD in the industry and how practitioners perceive architectural technical debt. The research objectives will help us answer the following research questions:

- RQ1: How is architectural technical debt introduced in a software system?
- RQ2: How do software practitioners manage architectural technical debt?
1.3 Structure of This Thesis

This thesis is structured in the following way:

- **Chapter 2** reviews the related work on technical debt.

- **Chapter 3** describes our exploratory multiple-case study. This chapter also analyzes the validity of this thesis and introduces our subjects’ background.

- **Chapter 4** reports the findings of this study.

- **Chapter 5** discusses the findings and what we have learned.

- **Chapter 6** concludes this thesis by answering our research questions, discussing our contribution by comparing them to those of other studies, and recommending future work.
Chapter 2

Related Work

Technical debt has received research interests from both industry and academia (Li et al., 2015). Numerous studies have covered all aspects of technical debt, including its identification, management, definition, and phenomena. In this chapter, we shall review the literature relevant to technical debt and to our study.

Following the introduction of the technical debt metaphor in 1992 (Cunningham, 1992), this phenomenon’s definition and usage of has been revisited frequently. Notable early work includes McConnell’s taxonomy (2007) and the technical debt quadrants created by Fowler (2009). In MTD 2010, researchers have argued that the software engineering research community needs to focus more on technical debt to improve the concept and present open research questions for future inquiries (Brown et al., 2010). Kruchten et al. (2012) have also called for more exploration on the technical debt concept.

2.1 Technical Debt Research Landscape

Some studies have investigated the general landscape of technical debt research through literature reviews. Tom et al. have conducted an exploratory case study that involves a literature review and interviews with software practitioners to better understand the technical debt phenomenon (Tom, Aurum, & Vidgen, 2013). A systematic literature review was conducted to
understand research efforts specifically on the financial aspect of the technical debt metaphor (Ampatzoglou, Ampatzoglou, Chatzigeorgiou, & Avgeriou, 2015). In 2015, Li et al. (2015) performed a systematic mapping study to review technical debt and TD management research between 1992 and 2013. In the study, they discovered that there were 10 different types of technical debt, with code TD being the most studied and architectural TD being the second. Furthermore, they concluded that there was no universal usage and interpretation of the “debt” aspect of TD and more studies were needed on technical debt management process. Recently, Behutiye, Rodríguez, Oivo, and Tosun (2017) looked at the research in the context of agile software development (ASD) and made conclusions on its causes, consequences, and managing strategies.

2.2 Technical Debt Phenomenon and Its Definition

A number of studies have been examining the phenomenon, definition, and understanding of technical debt in the industry and collecting the opinions of software practitioners by conducting surveys and interviewers. Lim, Taksande, and Seaman (2012) interviewed 35 software practitioners to understand technical debt and their experiences with it in the industry context. Spinola, Zazworka, Vetrò, Seaman, and Shull (2013) identified common beliefs on TD in the existing literature and blogs and tested them against 37 practitioners. In a multiple case study, a group of researchers (Ampatzoglou et al., 2016) studied the perceptions of TD in seven different companies in the embedded systems industry. The IBM Watson Research Center also published a study (Klinger, Tarr, Wagstrom, & Williams, 2011) that interviewed a group of software architects to understand the decisions of acquiring technical debt in an enterprise context.
2.3 Technical Debt Management

Technical debt management research can be divided into the following areas: management strategy, measurement, and identification. In Li et al.’s work (2015), they further expand management strategy into six classifications.

2.3.1 Technical Debt Management: Management Strategy

Researchers have been proposing novel frameworks and strategies to help with the TD management strategy and process. Guo and Seaman (2011) proposed a portfolio approach to managing technical debt. A portfolio, in financial terms, is a collection of assets and asset classes possessing a certain degree of diversification to minimize risks. Guo and Seaman wished to leverage the diversification and risk management aspects of a portfolio management for their TD management strategy. Fernández-Sánchez, Garbajosa, and Yagüe (2015) created a technical debt management framework by identifying elements that were essential in managing technical debt first through a systematic literature review. Then, they mapped these elements onto stakeholders’ (engineering, engineering management, and business management) point of views. Ramasubbu and Kemerer (2017) proposed a three-step process framework that integrated both the management of technical debt and software quality. Also in this study, a field test was performed for the proposed framework; the researchers claimed they have helped firms to reduce their technical debt related defects by 19%.

Another area of research interest in management strategy involves examining the current technical debt management practices in search of the potential best practices in the industry. For example, Codabux and Williams (2013) interviewed software practitioners from an industry
partner to observe their technical debt management strategy. Yli-Huumo, Maglyas, and Smolander (2016) performed similar research by interviewing 25 software practitioners in their case study. Recently, Martini, Sikander, and Madlani (2018) performed a multiple case study to understand the current state of practices in technical debt management with 15 organizations.

2.3.2 Technical Debt Management: Measurement

Technical debt measurement research attempts to quantify the benefits and costs of known TDs by estimation (Li et al., 2015). The quantification of technical debt has been a difficult task, because each technical debt issue is different and the consequences (future costs) of failing to fix each issue are difficult to foresee. Several studies (e.g., Guo, Spínola, & Seaman, 2016; Kosti et al., 2017; Nugroho, Visser, & Kuipers, 2011) have been attempting to develop models for technical debt measurement. Other studies (e.g., Curtis, Sappidi, & Szynkarski, 2012; Guo et al., 2011) are interested in estimating technical debt using formulas and numbers such as development time, lines of code, and monetary value.

2.3.3 Technical Debt Management: Identification

The final area of research into technical debt management is TD identification, whereby researchers focus on detecting technical debt in development team decisions and source code these teams produce. Researchers have experimented with various novel identification techniques. Bellomo, Nord, Ozkaya, and Popeck’s work (2016) presented and tested an idea of identifying TD items in issue trackers where developers communicate and record their work and decisions. In two separate studies (Bavota & Russo, 2016; da Silva Maldonado, Shihab,
Tsantalis, 2017), researchers explored the potential of self-admitted technical debt: Detecting TD that is based hidden within source code comments by observing patterns and language usages.

Tool supported identification is an important research area under TD identification. Some of tools often discussed and tested in studies include CAST, Structure101, Titan, SonarGraph, and SonarQube. In an experience report, Fontana, Roveda, and Zanoni (2016b) described their experiences when using three different tools to evaluate technical debt and software quality. In a different study (Fontana, Roveda, & Zanoni, 2016a), they further discussed the quality metrics and technical debt indices provided by these tools. Kazman et al. (2015) put different tools into a large-scale industrial software project to determine their usefulness and observe how they compared.

As Li et al. (2015) pointed out, most identification efforts were on source code technical debt issues. Even the tool supported identifications are based on source code investigations (Kazman et al., 2015). However, code-level TD is only a sub set of all the technical debt issues within software systems and other areas still require further exploration. Furthermore, one of Macia et al. (2012)’s studies suggests that code anomaly issues detected by code analyzers are often irrelevant to architectural problems. In this study, we wish to contribute towards research on design-level technical debt, so we have purposely excluded technical debt that is observable at source code-level.
2.4 Architectural Technical Debt

Architectural technical debt has been found to have a significant role in the world of TD. For instance, Ernst, Bellomo, Ozkaya, Nord, and Gorton (2015) concluded that “architectural decisions are the most important source of technical debt.” Also, the participants of Codabux and Williams’ study (2013) pointed out that “the most difficult type of debt to reduce is architecture debt.” As discussed previously in this chapter, Li et al. (2015) discovered that ATD is the second most studied TD type.

Recently, researchers have been encouraged to focus solely on architectural technical debt due to its importance. In the past few years, a team of researchers, namely, Antonio Martini, Terese Besker, and Jan Bosch, have been publishing studies extensively on different aspects of architectural technical debt. These include the impacts of ATD on daily software development work (Besker, Martini, & Bosch, 2017), the causes of ATD (Martini, Bosch, & Chaudron, 2014), the effects and dangers of ATD in a software system (Martini & Bosch, 2015), and the creation of a model to aid with ATD management (Besker, Martini, & Bosch, 2018). For architectural technical debt identification, Verdecchia, Malavolta, and Lago (2018) conducted a review on the ATD identification research landscape and discovered that identification studies, similar to TD identification ones, are usually based on issues rooted in the source code. There is still an absence of studies extending beyond the code-level.

2.5 Conclusion

In this Chapter, we first listed literature that reviewed the technical debt research landscape and some pioneering studies on this topic. Then, notable studies on the technical debt phenomenon,
technical debt management: management strategy, measurement, and identification, and recent studies focusing on architectural technical debt were discussed. In our research, we hope to contribute to architectural technical debt knowledge by providing taxonomies on ATD and reporting our findings by looking into the current practice of managing technical debt in some non-trivial software systems. We hope in this way that future researchers will be able to use our results to better understand the ATD phenomenon and create improved ATD management processes.
Chapter 3

Methodology

In this chapter, we will describe the methodology selected to explore design-level ATD and how this study was designed. Our research is an exploratory, multiple-case study using interview and questionnaire methods as our data collection strategy. We have obtained most of our knowledge on case studies from the classical text: *Case Study Research: Design and Methods* by Yin (1994) and a much more recent case-study guidebook specific to software engineering discipline, titled: *Case Study Research in Software Engineering: Guidelines and Examples* by Runeson, Host, Rainer, and Regnell (2012).

3.1 Case Study

To explore design-level ATD in the software industry, we have chosen to perform a case study, an empirical inquiry that investigates a contemporary phenomenon within its real-life context (Yin, 1994). The goal of this type of strategy is to answer “how” and “why” research questions to attain a clearer understanding of the phenomena. In social science research, case studies are one of the research methods used extensively to increase the knowledge of phenomena in question and to bring about change (Runeson et al., 2012). Similarly, technical debt research has been focusing on the “how” and “why” of technical debt and looking for ways to improve the software development process to mitigate the occurrence and impact of this problem. In this study, we seek to understand how design-level architectural technical debt is introduced into
non-trivial software systems. We believe that performing a case study is the best approach to answering our research questions.

3.1.1 Exploratory Case Study
In social science research, a research strategy can serve one of following three purposes: exploratory, explanatory, and descriptive (Yin, 1994). While a case study can use either one of these three strategies, exploration remains the primary purpose for using a case study strategy. Thus, the purpose of our case study is exploratory, meaning that we seek to find out what is happening, while seeking new insights and generating ideas and hypotheses for further research (Runeson et al., 2012) on ATD.

3.1.2 Cases and Units of Analyses – What Is It That We Studied?
In the case study context, a case is a contemporary phenomenon in its real-life setting. A case can be an individual, a group, a product, an event, or a policy. In software engineering, a case can be a team of developers, a software project, or a development process (Runeson et al., 2012). In our study, we define a case as a software project that is affected by ATD, and our unit of analysis is each design-level ATD incident that research participants reported in this study.

3.1.3 Case Study Protocol
A study protocol is a set of procedures for data collection and analysis. In our study, we have a protocol for interview data collection. This protocol was first written during the design phase of the study to ensure that the plan was concrete and decide on the data to collect and the questions to ask. Then, we constantly updated this protocol after each interview session to reflect what we
had learned and refined our interview questions and procedures. Since our questionnaire has a “mini-interview” style, the interview study protocol has also been used to help construct the questionnaire’s questions and procedures.

### 3.1.4 Triangulation

To improve a study’s the strength and credibility, triangulation is employed to collect broader and richer data. Four types of triangulation that may be applied (Runeson et al., 2012) in an empirical study: data triangulation, observer triangulation, methodological triangulation, and theory triangulation. In our research design, we have applied data triangulation by collecting experiences from different software practitioners from a variety of software projects and from different organizations which were geographically dispersed. The results of this study would not be convincing if only a single company’s projects were studied and no data triangulation was applied. Furthermore, we have used methodological triangulation by employing both interview and questionnaire methods to collect data. The interview method provides us with access to rich and detailed participant responses while the questionnaire method gives us the ability to easily access more individuals for data collection, providing us with a greater number of cases and more units of analysis.

### 3.1.5 Member Checking

To improve the accuracy of the collected data and interpretations, member checking is performed during the data collection process. Member checking is a process in which the researchers share data and interpretations with participants to determine if researchers understand them correctly (Marshall & Rossman, 2014). When conducting our interview sessions, the interviewer would
provide a short summary of what the interviewer had heard and understood and ask for corrections or further insights from the interviewee if necessary. The interviewer, following each interview session, would listen to the interview recordings while typing out the transcripts and raise follow-up questions to interviewees by email. Regarding the questionnaires we sent out, if any response did not show a clear picture of the technical debt event described, we would email the participants to clarify this as well.

3.1.6 Case Study Weaknesses

Attention must be given to case study weaknesses. The case study approach is appropriate when realism is required and real-world issues can be closely observed. However, as Runeson et al. (2012) have described, there are trade-offs between the level of control and the degree of realism when conducting research on real world issues. Realistic settings are often complex, with noises that can hinder the understanding of what is happening. On the other hand, increasing control of the study reduces the degree of realism. Another weakness in case studies is that the data collected and analyzed is more open to interpretation and researcher bias (Easterbrook, Singer, Storey, & Damian, 2008) depending on the researcher’s experience with software engineering and technical debt.

3.1.7 Ethical Considerations

Software engineering is a professional field involving organizations, software practitioners, and their clients. Thus, “it is essential that the case study researcher takes account of legal, professional and ethical issues in the design, conduct and reporting of his or her research” (Runeson et al., 2012). Prior to the start of our data collection, we obtained an ethics approval
from the University of British Columbia (UBC)’s Research Ethics Board (REB). UBC requires that research involving human subjects must be reviewed and approved by the UBC-affiliated REB prior to data collection. In the application to the REB, we stated the intended purpose of our study, potential participants and recruitment strategy, plan of informed consent, possible conflicts of interests, data collection strategies, handling of confidential data and their results, and source of research funding. For each interview session to proceed, the participant would be required to review and sign our REB approved consent form to give us an informed consent. For our questionnaires that were sent out electronically, the ethics description was displayed on the very first page of our online questionnaire to inform our participants on how the data would be handled and how their confidentiality will be maintained.

3.2 Data Collection
Data was collected between September 2016 and October 2017. During this period, a two-phase collection strategy was performed. In the first phase, a series of face-to-face, semi-structured interviews was conducted. In the second phase, an online questionnaire was developed and sent out by email to reach more participants.

3.2.1 Participants
In total, we interviewed 18 software practitioners from three different organizations and 10 software practitioners submitted the questionnaire. The interview and questionnaire participants were recruited through my thesis supervisor, Dr. Philippe Kruchten’s industry contacts and the contacts’ work colleagues. Each selected participant has met our criteria of currently practicing in the software development industry and having been involved in a non-trivial software system.
development in recent years. Tables 1 and 2 present the roles and experiences of each interview and questionnaire participant. Table 3 provides an overview of the kinds of organizations for which the participants were working for.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Role</th>
<th>Years of Experience</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI01</td>
<td>Data Scientist</td>
<td>4.5 years</td>
<td>A</td>
</tr>
<tr>
<td>SI02</td>
<td>Development Team Lead</td>
<td>4 years</td>
<td>A</td>
</tr>
<tr>
<td>SI03</td>
<td>Development Team Lead</td>
<td>17 years</td>
<td>A</td>
</tr>
<tr>
<td>SI04</td>
<td>Development Team Lead</td>
<td>15 years</td>
<td>A</td>
</tr>
<tr>
<td>SI05</td>
<td>Software Developer</td>
<td>3.5 years</td>
<td>A</td>
</tr>
<tr>
<td>SI06</td>
<td>Software Developer</td>
<td>18 years</td>
<td>A</td>
</tr>
<tr>
<td>SI07</td>
<td>Software Developer</td>
<td>3 years</td>
<td>A</td>
</tr>
<tr>
<td>SI08</td>
<td>Development Team Lead</td>
<td>15 years</td>
<td>A</td>
</tr>
<tr>
<td>SI09</td>
<td>Software Developer</td>
<td>10 years</td>
<td>A</td>
</tr>
<tr>
<td>SI10</td>
<td>Software Developer</td>
<td>3 years</td>
<td>A</td>
</tr>
<tr>
<td>SI11</td>
<td>Data Scientist</td>
<td>1.5 years</td>
<td>A</td>
</tr>
<tr>
<td>SI12</td>
<td>Data Scientist</td>
<td>3 years</td>
<td>A</td>
</tr>
<tr>
<td>SI13</td>
<td>Research Scientist</td>
<td>4 years</td>
<td>A</td>
</tr>
<tr>
<td>SI14</td>
<td>Executive Director of Research</td>
<td>26 years</td>
<td>C</td>
</tr>
<tr>
<td>SI15</td>
<td>Software Architect</td>
<td>20 years</td>
<td>C</td>
</tr>
<tr>
<td>SI16</td>
<td>Project Engineer</td>
<td>22 years</td>
<td>B</td>
</tr>
<tr>
<td>SI17</td>
<td>Software Developer</td>
<td>14 years</td>
<td>B</td>
</tr>
<tr>
<td>SI18</td>
<td>Software Architect</td>
<td>25 years</td>
<td>B</td>
</tr>
</tbody>
</table>

Table 1. Interview Participant List

These interview participants had software development experiences ranging from 1.5 year to 26 years. The distribution chart in Figure 2 demonstrates that most of the interview participants had either fewer than five years of experience (eight participants) or greater than 15 years of
experience (eight participants). Only two participants are outside the range of these two groups, possessing 10 to 15 years of experiences. Note that SI13’s role is that of a research scientist, which is not a development position but a software researching position. Thus, SI13’s responses are excluded from the data analysis.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Role</th>
<th>Years of Experience</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ01</td>
<td>Product Owner</td>
<td>5 – 10 years</td>
<td>C</td>
</tr>
<tr>
<td>SQ02</td>
<td>Technology Manager</td>
<td>15 – 20 years</td>
<td>D</td>
</tr>
<tr>
<td>SQ03</td>
<td>Software Architect</td>
<td>15 – 20 years</td>
<td>G</td>
</tr>
<tr>
<td>SQ04</td>
<td>Software Architect</td>
<td>15 – 20 years</td>
<td>G</td>
</tr>
<tr>
<td>SQ05</td>
<td>Software Architect</td>
<td>10 – 15 years</td>
<td>G</td>
</tr>
<tr>
<td>SQ06</td>
<td>Principal Engineer</td>
<td>20+ years</td>
<td>G</td>
</tr>
<tr>
<td>SQ07</td>
<td>Principal Engineer</td>
<td>15 – 20 years</td>
<td>F</td>
</tr>
<tr>
<td>SQ08</td>
<td>Technology Manager</td>
<td>15 – 20 years</td>
<td>F</td>
</tr>
<tr>
<td>SQ09</td>
<td>Head of Architecture Team</td>
<td>5 – 10 years</td>
<td>F</td>
</tr>
<tr>
<td>SQ10</td>
<td>Senior Director Project Management</td>
<td>15 – 20 years</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 2. Questionnaire Participant List

We sent out questionnaires to 16 participants who had agreed to participate, and 10 responses were received. Of the 10 participants who had submitted questionnaire responses, seven had had more than 15 years of software development experiences, while three participants had between five and 15 years of experience (Figure 2). All interview participants were employed in Canada, while the questionnaire’s participants were employed either in Asia, North America, or Europe. Some of the recruited participants have responsibilities for both the business and product
development aspects of their organizations, which provided both development and business views on technical debt for our data analysis.

Figure 2. Distribution of Participant Experiences

The organizations that the participants were employed in ranged from a small private company (D) to large, multinational, publicly traded organizations (B, F, and G). Organization H is reported as confidential as requested by SQ10. All organizations are for-profits.

<table>
<thead>
<tr>
<th>Organization</th>
<th># of Employees</th>
<th>Industry</th>
<th>Operating Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,000+</td>
<td>Information Technology</td>
<td>Multinational</td>
</tr>
<tr>
<td>B</td>
<td>10,000+</td>
<td>Health Care</td>
<td>Multinational</td>
</tr>
<tr>
<td>C</td>
<td>1,000+</td>
<td>Information Technology</td>
<td>North America</td>
</tr>
<tr>
<td>D</td>
<td>11 – 50</td>
<td>Information Technology</td>
<td>North America</td>
</tr>
<tr>
<td>E</td>
<td>10,000+</td>
<td>Defense</td>
<td>North America</td>
</tr>
<tr>
<td>F</td>
<td>10,000+</td>
<td>Manufacturing</td>
<td>Multinational</td>
</tr>
<tr>
<td>G</td>
<td>10,000+</td>
<td>Manufacturing</td>
<td>Multinational</td>
</tr>
<tr>
<td>H</td>
<td>(Confidential)</td>
<td>(Confidential)</td>
<td>(Confidential)</td>
</tr>
</tbody>
</table>

Table 3. Participants' Organizations
3.2.2 Interview

All interviews were conducted on-site at the participants’ workplaces. In total, there were 13 interviews took place, consisting of 10 one-on-one interviews and three group interviews. One-on-one interview sessions were 20 to 40 minutes in length, while group interview sessions were of 30 to 70 minutes long.

Each interview was guided by a predefined interview protocol, consisting of a set of open-ended questions relating to each participant’s background and the objectives of this study. The protocol was updated and the questions were refined constantly throughout the data collection process as we gained more knowledge on the nature of ATD and greater experiences in conducting interviews. Before each interview, the participants were asked if they agree to be audio recorded. Several participants chose not to be recorded, so I accordingly took extended field notes for those sessions. To ensure the accuracy of the notes and my understanding of the information obtained, I reviewed my notes with the participants during the sessions. The recorded interviews were transcribed into text for data analysis purposes. Follow-up interviews were requested if the results of the initial interview analysis revealed the necessity for further exploration or extended clarifications with the participants. Otherwise, minor clarifications were done by email exchange.

In general, the interview questions can be broken down into following four areas:

1. What is participant’s background?
2. What is participant’s understanding of technical debt?
3. During development, what decision you or your team has made that you regretted the most? What is the nature of this project, and what happened?
4. What is the current technical debt management in participant’s organization?

Notice that Question 3 does not focus on technical debt itself but on the decisions that led to its developing, since we were interested in the cause rather than the phenomenon. Initially, the participants were allowed to respond to these questions with experiences from different projects. The focus was later altered to only on one specific project, as more detailed responses could be collected within the limited interview time.

3.2.3 Questionnaire

The questionnaire was designed to mimic the interview by using a series of open-ended questions derived from the interview protocol. The purpose of using the questionnaire in the second phase of data collection was to reach out to participants who resided in other continents such as Asia and Europe without needing to deal with time-consuming video conference interviews and while taking time zone differences into consideration. The questionnaire was hosted online by UBC Survey Tool, a Canadian-based survey solution through FluidSurveys.

The selected participants were invited by email with a short summary of this study and the URL to the questionnaire. The participants were required to each discuss one non-trivial software intensive project and one to three different decisions that led to the development of technical debt for that specific software project. Since the questionnaire consisted of mostly open-ended questions where text responses were expected, the participants were warned that completing the questionnaire might take more than 30 minutes, and they were allowed to save their progress before submission.
The participation for the questionnaire was voluntary and each participant was given a unique identifier for tracking purpose. The first version of the questionnaire was tested with SQ01 and SQ02 to ensure that there was no issue with their understanding of the questions and data collection’s effectiveness. Both participants’ completion times were likewise monitored to predict the required completion time for future submissions. Both participants provided valuable feedback and suggestions, some of which were incorporated into the questionnaires.

### 3.3 Data Analysis

For a qualitative research, an analysis should be conducted in parallel with data collection. This process is necessary because qualitative research is flexible, and the analysis might reveal the need for collecting additional data or updating the research techniques as a result of unexpected insights. Because of this, our data collection and data analysis were performed iteratively.

#### 3.3.1 Coding

The first step of our data analysis involves coding, by which data sources such as texts or quotes are given labels to assign meanings, summaries, or classifications to them. These codes can also be used to organize data and group texts with similar ideas together. It is possible that the same text could have more than one code. For our coding strategy, we employed the *editing approach* which we created *a priori* codes that were “defined based on findings of the researcher during the analysis”(Runeson et al., 2012). A set of codes including “Decision”, “Payment”, “Interest”, and “Cause” was formulated based on the research questions and objectives of this study. Other codes, such as “Not meeting requirements”, “Uncertainties”, “Business decision”, and “Time-to-market” were created later on, during open coding stage. Some of the codes emerging from open
coding were also used in questions in the subsequent interviews. Axial coding was performed following open coding to explore the connections and relationships between the codes and categories. Coding for interview transcripts was performed at least twice for the same transcript. The goal of conducting coding multiple times was to make sure that all the interesting data were captured and to accommodate any new ideas that might come up between different iterations.

3.3.2 Uncovering ATD Items

While the coding was performed, we simultaneously extracted technical debt items; these, by definition, represent delayed maintenance tasks that may cause future problems (Guo et al., 2011). To extract TD items from the collected data, each issue reported by the participants was reviewed and analyzed to see if it were an ATD issue. If so, a technical debt item was created, and its properties were populated (Figure 3).

<table>
<thead>
<tr>
<th>Technical Debt Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short description of this TD:</td>
</tr>
<tr>
<td>Cause of TD:</td>
</tr>
<tr>
<td>Principal:</td>
</tr>
<tr>
<td>Interest:</td>
</tr>
<tr>
<td>Payment:</td>
</tr>
<tr>
<td>Current Status:</td>
</tr>
</tbody>
</table>

Figure 3. Technical Debt Item

3.3.3 Analytic Memo

An analytic memo is used to document the current progress of research with the data already collected and it is written during and after each coding session. Each analytic memo that I have
written included the key findings from the coding, events or issues that seemed important, possible patterns and explanations, my hypothesis on what was going on, and update on interview or questionnaire techniques for the subsequent iterations.

3.4 Validity
To address the validity and trustworthiness of this study, we have chosen to follow the validity classification scheme described by Runeson et al. (2012). This scheme was originally suggested by (Yin, 1994) who proposes four aspects of validity concerns which are construct validity, internal validity, external validity, and reliability.

3.4.1 Construct Validity
Construct validity concerns whether a study has established the correct operational measures for the concepts of being studied (Yin, 1994). Since the concept of technical debt is still evolving and interpretations might vary between individuals (Fontana, Trumler, Izurieta, & Nord, 2017), this could pose a threat to the validity of the results if participants were to interpret the concepts differently from that which the researcher’s had done during data collection. Therefore, we have prepared a short definition of technical debt given to the participants during interview sessions or on the first few pages of questionnaire to let them know our definition of technical debt. Another potential threat to construct validity is the research participants. The participants were required to discuss software projects they have worked on recently and were heavily involved in, so it would be their own experiences and understanding of the issues.
3.4.2 Internal Validity

Internal validity deals with causal relation concerns, including, for instance, whether one event truly influences another event. In Yin’s (1994) text, it is stated that internal validity is not applicable to exploratory studies as this kind of study does not seek to make casual statements in the study result. Thus, internal validity will not be discussed here.

3.4.3 External Validity

External validity is concerned with the extent to which the findings are generalizable beyond the investigated case. In our case study, the threat would be the way in which software practitioners of various roles from different organizations, organization sizes, industries, and regions would experience and react to technical debt in different ways due to reasons associated with cultures and contexts. To mitigate this threat, I have recruited participants from different countries and regions with diverse roles and experience in software development (Tables 1 and 2). Efforts were likewise made to collect data on technical debt from both the business and development sides of views. Also, knowing that the cultural and background differences between the organizations could pose a threat, I have selected organizations of varying sizes in different industries in Asia, Europe, and North America (Table 3). However, the organizations selected were mostly giant corporations which could potentially impact the generalization, so more investigations are needed. Thus, it is important to note the potential limitations to how generalizable the results of this study are.
3.4.4 Reliability

The objective of reliability is to ensure that if another investigator who follows the same procedures and conducts a study on the same cases, the investigator would arrive at the identical findings and conclusions. Yin (1994) describes that, for this to take place, one prerequisite would be documenting the procedure in details. In this study, I have documented the interview protocol and questions that were used in the interviews and questionnaire that was distributed to the participants. The main threat to the reliability of this study is the coding in data analysis, as this is prone to different interpretations of texts and events such that the readers must be aware of this potential limitation.
Chapter 4

Findings

In this chapter, the findings of this study are presented and summarized. First, we present the finding of how the technical debt definition is understood among the participants. Then, the categorizations of both the ATD sources and causes are detailed, based on the analysis results from last chapter, are discussed.

4.1 Technical Debt Familiarity and Understanding

As discussed in the previous chapter, the definition of technical debt could vary between individuals. It was interesting to learn how participants perceived the metaphor and whether an understanding was shared among the studied organizations, participants, and researchers.

Many of the participants stated that they were already familiar with the term, technical debt, before we reached out to them. Their technical debt definitions have described the trade-offs between short-term gains and additional long-term development efforts. This was similar to the definition concluded in 2016 Dagstuhl technical debt seminar (Avgeriou et al., 2016).

- “I think both of us are pretty familiar with what it [technical debt] means… We sacrifice design for certain reasons.” (SI14)
- “Technical debt is just like a debt out of a bank. It is you make some decisions. Hopefully fully aware that they won’t be the best decisions at the time... So, the decision
was not the best technical decision to make but it was understood that we were making a choice based on timeline. Therefore, the technical solution might not be the best but it was good enough at the time.” (SI18)


From their technical debt definitions, some have pointed out that trade-offs were frequently made due to business reasons such as “budgets” (SI18), “time-pressure” (SQ5), “delivery to market” (SQ6), “financial and business consequences”, and “time to market” (SQ7). SQ4 further suggested technical debt “is a metaphor to communicate to management”. This implies that technical debt not only involves trade-offs between speed and quality, but also between short-term business gains and long-term development costs.

From among the participants, eight claimed that they had never previously heard of the term before. Interestingly, these were all employees of Organization A and three of them had more than 10 years of software development experience. However, after explaining our definition of technical debt to them, they quickly grasped the concept quickly and were easily able to relate the concept to their experiences at Organization A.

4.2 Types of Projects Reported

This study asked participants to discuss non-trivial software intensive projects they had been involved in with their current employers. In some cases, an interview participant would discuss issues from a sub-project of a current project or a preceding project that had led to the current
project. In such cases, we have treated them as a part of the current project. Since the selected interview participants were often colleagues, the projects discussed were repeated several times.

In total, we analyzed data from 15 non-trivial software intensive projects developed for very different purposes, such as highway tolling, smart industry, power grid management, and medical system. From the 15 software projects analyzed, there were software systems with sizes less than 500,000 lines of code (LOC) to more than 10 million LOC (Figure 4) and software ages from less than 1 year to 26 years (Figure 5). It is worth mentioning that LOC measurement can be inaccurate as some participants reported it as their best guesses, providing a range rather than the actual numbers retrieved from a static code analyzer. Nevertheless, LOC can still provide good rough estimates on the sizes of the software systems.

![Figure 4. Software Project Size Distribution](image-url)
As shown in Figures 4 and 5, 67% of the projects had more than 500k LOC and 67% also had been in development for more than five years. This infers that most of the data collected originates from projects that were very mature and had invested in significant development time. On the other hand, 33% of the projects are still young, with less than two years of existence. This increased the external validity of this study as it covers both young and mature software systems.

### 4.3 Architectural Technical Debt Items

In this study, we identified 53 architectural technical debt items, with 36 items from interview participants and 17 items from questionnaire participants. On average, each interview participant has discussed two items, while each questionnaire participants discussed 1.7 items. Also, about 50% of the questionnaire participants discussed only one item.
4.4 Sources of ATD

We created an ATD taxonomy by using the sources of these debt items. The categories were developed based on the codes emerged and themes developed during the coding process. After merging the codes of similar properties and giving them meaningful names, we have developed eight different categories: technology choice, obsolete system, knowledge loss, recycling & old code integration, suboptimal architecture, poor build & deployment methods, and lack of testing plan. Some categories had subcategories to better describe the sources of debt. In the follow sections, we will describe these categories and provide some examples of ATD cases.

It is possible for one ATD item to have multiple source and fit into multiple categories. In this study, we used the best fit strategy to categorize the ATD items and the distribution of the cases can be found in Table 4.

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th># of ATD Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology Choice</td>
<td>Technology Trap</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Early Adopter</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Suboptimal Technology Choice</td>
<td>7</td>
</tr>
<tr>
<td>Requirement</td>
<td>Incorrect &amp; Unclear Requirements</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>New Purpose</td>
<td>3</td>
</tr>
<tr>
<td>Obsolete System</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Knowledge Loss</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Recycling &amp; Old Code Integration</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Suboptimal Architecture</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Poor Build &amp; Deployment Method</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Lack of Testing Plan</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4. Distribution of ATD Items
4.4.1 Technology Choice

Technology choice is one of the most common sources of ATD that participants encountered. Out of 53 architecture debt items, 18 originated from the technologies that the software systems use. This category is further divided into three subcategories for a better classification.

4.4.1.1 Technology Trap

Technology trap describes ATDs where the source is associated with the reliance of external technologies and the developers are “trapped” with them. Some software development teams employ external technologies to be functional, and their reliance on them causes the system to be vulnerable to the changes or lack of changes from those technologies.

SQ18 commented that after integrating the technology they wanted, “they [the vendor of the technology] [have] forced us to buy license[s] for all the developers who are working on the project irrespective of whether they use it or not.” As a result, the cost of development went up significantly and the “management want[ed] to remove it”. SQ7 described an issue where, “The 3rd party component changed and this affected our design”, but the developers were able to take care of it before its release to customers. SI14 and SI15 pointed out that a key component of their software system was very dependent on their vendor’s support of the outdated 32-bit APIs. Moving away from these APIs have been discussed but it was too costly to be done.

4.4.1.2 Early Adopter

Early adopter is a term generally used to describe those who begin to use a product as soon as it becomes available. Here we use the term to categorize the an ATD source in which the
The development team uses the latest and the greatest technologies available, but receives development penalties due to code or product immaturity. From our analysis, there are two areas in which being an early adopter could cause architecture technology debt issues. The first area is where the development team does not fully understand the capabilities and limitations of the technology but proceeds to use it because it is the latest and greatest. The second area is where the development team believes in the new technology and deliberately takes on extra risks such as immature features and software-breaking bugs.

SQ03 reported that their team “[introduced] new fancy technical stuff without the knowledge/experience if that is a best fit for the problem.” Also, they created a “slow resulting software, mainly because [the fancy new technology] was not [built] for bulk inserts”. SQ03 later commented that “the fancy stuff is not [necessarily] the best. Sometimes old and boring solutions to a better job [sic].” SI07 discussed an issue where the development team skipped 1.0 version of a popular framework and used the 2.0 version which was still in Alpha. The team was correct that 2.0 will eventually replace the 1.0 version, but the team faced enormous issues involving unfinished features and code-breaking framework updates.

4.4.1.3 Suboptimal Technology Choice

When the technology choices do not fulfill the requirements or achieve the expected behaviours of a software system, the source for the technical debt is categorized as a suboptimal technology choice.
SI06 had an issue where the team used a server framework suitable for small server application to develop a large-scale server application due to inexperience. The team was forced to give up all of its development progress and rewrite the system with another server framework that met the requirement. SQ09’s team chose to use a “third-party service bus, with proprietary message format and automatic discovery of services.” He related that “the team spent a big amount of time working with the chosen solution, but couldn’t manage to adapt the solution to our system, and meet the necessary performance requirement with it.” SI02 had an issue in which the protocol framework that the team decided to use was heavy and the throughput failed to meet the requirements. SI04 mentioned that their team intentionally picked a framework that was suboptimal but fluent for most of the team members to quickly create a prototype as a minimum viable product. The downside of such a choice was the guaranteed time-consuming framework changes later in the development.

4.4.2 Requirement
Both the interview and questionnaire participants mentioned that ambiguous early requirements as well as major requirement changes in the later stage of the projects can easily introduce technical debt issues within a software system. This category can be further divided into two subcategories.

4.4.2.1 Incorrect & Unclear Requirement
Incorrect and unclear requirements can quickly introduce ATD issues and stay with a system for the remainder of the software life. These requirement issues usually occur in the early stage of a
software system when the management, development team, and clients do not know exactly what they want yet.

SQ08 described the management was unsure if a software system was required to have cross-platform support but still proceeded with the cross-platform route. This requirement was later dropped, but the cross-platform technology had already been invested and the team had had to work with this technology without reaping the benefits from it. SQ03 pointed out a software project whose architecture was designed to support a specific workflow for a client. However, it then became apparent that the client wished to have this software to support additional workflows for which the architecture had not designed to perform. SI04 provided an example where the development team used one programming language to develop a sub-project of a software project. The team was later forced to rewrite the entire sub-project with another programming language due to changes in the project’s scope and requirements.

4.4.2.2 New Purpose

It is often required for a component or an entire software system to receive major requirement updates late in its life to make it more useful or adapt to operating environment changes. But, the existing software system might see ATD issues because its architecture was not built to support the new requirements.

SQ07 pointed out that a logging component in their software project needed to be updated to log other system events due to a new client requirement. This new capability was beyond design of the component, forcing the team had to perform a rewrite. SI14 and SI15 had similar issues on
their logging system like SQ07’s but instead of rewriting the component, they decided to ignore it, since it was not a priority. SI18 commented that an one-off software system needed to be converted to become general purposed which the system was never designed for.

### 4.4.3 Obsolete System

A software system, following years of development cycles, contains obsolete technologies and designs unless those components are updated and revisited. Years of development could also cause drifts and erosions in the architecture. Even if no changes are made, a software system can become suboptimal because of advancing technologies.

SI14 and SI15 mentioned they had had to develop a custom memory management code because the one provided by the operating system had become insufficient years ago. Now, the operating system’s memory management worked better than the custom one, but changing to the operating system’s one was too costly to be done. SI14 and SI15 also said that they had identified various of obsolete technologies in their 20-year-old software system, but they would not replace any of them as they are still working correctly and replacing them would require too much time. SI09 commented on their custom Linux core, which had been used for years but did not work well with the latest hardware and firmware available on the market.

### 4.4.4 Knowledge Loss

Knowledge loss describes events when developers do not document their knowledge properly and new developers could not understand why certain designs and decisions were made.
SI09 observed an issue where their software project used two dependency managers, which was an uncommon practice for their projects. The details of this decision were no longer available because the designer left the organization and the reasons for the choice was not documented. SI18 likewise faced a knowledge loss issue because their developers had moved between different teams during less busy times and those who had left had not left much documentation behind.

4.4.5 Recycling & Old Code Integration

Reusing code from another project or component is a common practice in software development to save time, but the software system can inherit unwanted behaviours and aspects of the recycled code if not careful. Integrating old code has its own issue too as the old code might have been developed long ago with a different design and making it fit into the current system needs some shortcuts.

SI16 and SI17 pointed out that they used a database design and implementation from the previous software project. Even though development time was saved in the short-term, the inherited issue from the old database caused more maintenance effort. SQ05 related that an old component that needed to be integrated into the new code had been developed following a different standard. The development team found the integration process difficult and suboptimal design choices had to be made. SI02 faced an issue where the team had to migrate an old core component from a previous project to the new project. Since the old core component had not been designed to be reusable, a lot of bridging code and modifications had to be made.
4.4.6 Suboptimal Architecture

Suboptimal architecture is a generalization for the source of architecture debt. Here, we use this term to categorize suboptimal architectural design decisions made either intentionally or unintentionally that are the main sources of ATD.

Participant SI05, SQ02, SQ06, and SQ10 all mentioned that their software projects faced tight coupling and lack of modularity problems and were causing maintenance and evolvability difficulties. SQ01 and SQ04 both have pointed out architecture drift which was supposed to save development time was now causing them additional development efforts in the long run. S14 and S15 commented that one of their developers, who was under significant time pressure, had had to implement a critical component without a clean design. The deadline for the product was met, but future productivity had been heavily impacted.

4.4.7 Poor Build & Deployment Method

This category looks at ATD sources that come from quick and dirty strategies for handling system build, packaging, and parameter settings.

SI04 identified that their deployment method was overriding all existing libraries on the client side to avoid version mismatch. The participant argued that although this method had never failed before, it made deployment inflexible. SQ07 said that instead of correcting an installation script for a software system, the team created another executable to make the system work with the installation script. The workaround worked but the executable “[had] become a maintenance headache.”
4.4.8 Lack of Testing Plan

In software architecture, testability must be considered to allow easy access to the internal components during testing. This category describes events where developers do not consider testability or testing strategy during early design phase and the software system is hardly testable because of the architecture design.

SI08 and SQ04 both regretted that their team did not consider testability in the early stage of their software projects. SI08 said they had to go back and refactored the code to increase testability. SQ04’s team had not attempted to mitigate their issue, so the debt still manifest.

4.5 Why Was ATD Incurred?

In Chapter 4.4, we listed out the sources for ATDs. Here, we take a further look at the drivers of incurring ATDs from the coding results. From the responses, we can see that business plans, lack of experiences & awareness, development resources, attitudes & practices, and software age are most common reasons to incur technical debt.

4.5.1 Business Plans

Most participants have expressed, directly or indirectly, that business choices are the main reason that incur technical debt is incurred. As S14 have pointed out, “design decisions [are] greatly influenced by the business… [if] the business doesn’t survive, there is no point to make a right decision”. S14 also commented that, for them, incurring an ATD is not a development choice,
but a business choice. Some business causes have been discussed very frequently by participants: Time, and Changes & Uncertainties.

4.5.1.1 Time

In both questionnaires and interview sessions, time pressure and time-to-market issues were consistently referred to by the participants as a main reason or a catalyst for incurring ATDs.

S14 and S15 both revealed that their products were serving a very competitive industry, and late product releases would cost the company significant amounts of profit. As a result, many trade-offs were made to meet the release timeline. A similar issue was discussed by SQ06 where the development team had to modify an existing software solution for another use, but without a proper architecture update due to time pressure. SQ07 pointed out that their team did not have the time to design a component with evolvability because of an imposed time constraint. SQ07 believed if they were given more time for elaborated discussions on the design choice, it would have helped them to avoid this issue.

4.5.1.2 Changes & Uncertainties

Participants have pointed out that many of their requirement issues, as discussed in 4.4.2, had been caused by uncertainties and changes in the objectives and specifications of their software systems.

Participants from Organization A believed that they faced requirement and ATD issues because Organization A used a “develop product first, find buyers later” approach in its business strategy.
The problem with this approach was that the product, in this case a software system, usually had to be modified later to meet the buyers’ needs which usually included features and specifications that the original architecture had not been designed for. Also, SI18 talked about how their business goal had changed and the management wanted to modify an one-off software system to suit other potential clients, but the system was not originally designed to handle this new requirement.

4.5.2 Lack of Experience or Awareness

A number of participants have felt that inexperience in technology and lack of awareness in potential issues have caused their ATD.

SI01 pointed out that their development team was using a framework that they had had no prior experience with, and only minimal research was done on it. The team later realized that this framework did not fit the project requirements and needed to spend extra time to find replacement. SQ04 said the team was “not aware of the potential issue” of only relying on UI tests, and SQ05 claimed their testability had suffered because they had not considered having an interface to access old components. SQ07 felt that their framework choice had been a “pretty straightforward choice at that time” so they had not integrated that component with decoupling in mind, something the team lived to regret.

4.5.3 Development Resources

Development resource availability drives some participants to incur technical debt.
SI18 commented that their team had no developers with web app skills and no additional human resource was available. As a result, they had decided to employ a third-party code generation tool which had led to unexpected extra maintenance efforts. Other participants also have discussed similar events where the development teams had decided to use third-party components because their team did not have the resources to develop their own components and faced issues later in the development cycles. Participants from Organization C mentioned that their teams ran on budgets allocated to them which limited what they could do.

4.5.4 Attitudes & Practices
Participants felt that developers’ attitudes and development practices within the development team had caused or worsened some of their ATDs.

SI18 and SI09 both mentioned that their development teams did not have the practice of properly documenting developers’ knowledge. Others also commented on poor documentation habits as a contribution towards technical debt issues. Technology trap, as discussed in 2.4.1.1, is most commonly caused by the practice of using the greatest and the latest technology available. Also, some participants said that they tended to choose the simplest and quickest solution to a problem which was vulnerable to ATDs.

4.5.5 Software Aging
All software systems are affected by their age in one way or another; this is an unavoidable natural cause. A software system in development for years faces obsolete components and changes in its operating environment. Furthermore, years of active development can cause
architecture erosion where the architecture is no longer clean like when it was designed. All ATD cases discussed in 2.4.3 were caused by the age of the software system.

4.6 Debt Payment

We looked at participants’ responses from both the interviews and questionnaire to understand their current practices of managing architectural technical debt repayment. After ATD has been incurred or identified, software developers should, ideally, take their time to develop a plan for handling the debt. Based from our observations, there are three different approaches participants and their teams use to tackle ATD: active approach, reactive approach, and ignoring approach.

4.6.1 Active Approach

A development team that uses an active approach acknowledges the existence of ATD and creates a plan to actively manage the debt. An active debt reduction effort may or may not fully repay the debt, but the ultimate goal is to mitigate the ATD’s impacts on the development team and productivity. An active approach can be either an ongoing effort that is stretched across many development cycles or it can be a development focus within a short period of time.

Our participants used an active strategy when they faced two situations. The first was when the software system no longer met the requirements and expectations due to the present of suboptimal architectural design, and when any additional workarounds would not overcome the issue. In this type of situation, the participants immediately dedicated their resources and development time to correcting the suboptimal architecture. For example, the participants from
Organization A who often experienced requirement changes were required to often redesigns and rewrites on their existing software systems.

The second situation is when the participants understood the potential troubles of not paying off their debt and allowing its debt interest to get out of control. In this case, they took preventive actions to reduce the debt. However, they commented that this work was only possible if their resources and time allowed and the management had approved the needed extra work. Both SQ02 and SQ06 commented on their ongoing efforts with debt management. SQ02 said there were “still ongoing debt payments, slowly diminishing in size over time.” SQ06 commented that they were looking at “design alternatives…” that offers “a roadmap out of the technical debt.”

4.6.2 Reactive Approach
Our study found that the reactive approach was the most common strategy that participants used when handling ATDs. With this approach, software developers accept the presence of the ATDs but only manage the debt when it stands in the way of developers and repayments can no longer be delayed. Participants commented that they often used this approach because they saw no immediate significant consequences in not managing the debt and adding new features usually has a higher priority than fixing problems that were not yet critical.

Participants such as SQ05, SQ08, SI16, and S17 pointed out that they had acknowledged the presence of some of their debts but had made no plan to manage them unless the debt created obstacles in the development. SI18 and SQ09 said they had begun to repay some their debts
because, after not managing them for some period of time, the debt interest had grown too high such that repayment plans must be made to increase their productivity again.

4.6.3 Ignoring Approach

With an ignoring approach, the developers accept the presence of the ATDs, but no further steps will be taken to reduce the debt. In other words, if the cost for paying off the debt is greater than the benefits of simply reducing it, it makes more sense to accept the ATD penalties rather than taking any action. Based from our observation, there are two situations which our participants used this approach.

The first situation is when the debt became impossible to reduce without significant development cost. For example, SI14 and SI15’s software system had been in development since the early 1990’s, and both participants commented that much of the original architecture was obsolete and efforts to replace it would be too costly. Their current plan was to not manage the debt and wait for the eventual retirement of their software system. SI16 mentioned that their database, designed years ago, was not meeting current standards. A new database had never been planned because database replacement was a time-consuming job and data migration would be a painful process as well.

The second situation is when the debt interest was minimal or not growing. This occurs when the suboptimal architecture is noncritical and the debt itself does not create much risks to productivity. In this situation, the developers can easily escape from debt repayment and simply accept the minimal development penalties. On example is that SI15 said they developed a
memory managed code to facilitate with their system in the 1990s. This piece of code could have been replaced with a better built-in function today, but the participant felt that such replacement was unnecessary because the old code still worked correctly and the debt had not impacted their productivity.

4.6.4 Tracking ATD

In our interview and questionnaire, we asked our participants to discuss the current practice of tracking any technical debt issues, and the identity of the tool used to assist such tracking. The only reported technical debt tool, if using, was SonarQube. SonarQube is a software specializing in static code analysis to inspect code quality and manage technical debt. As a companywide development practice, participants from Organization A had been using SonarQube to track and manage technical debt. SQ09 and SQ03 have reported that their team using SonarQube as a tracking tool as well. However, SQ03 felt that SonarQube “[did] not track the big design issues”, and they had had to develop a separate list for those issues. Jira, an issue and project tracking software, was discussed by some participants as their TD tracking tool but this tool relies on its users to identify the TD problems first. SQ08 said their ATD tracking strategy was having weekly architecture meetings to discuss problems with existing architectural design. Some also reported using informal tracking processes, such as developing a list of known debts, putting debts in a backlog, or simply making mental note of the debts’ existence.

Other participants reported that their development teams did not track technical debt because either they were unfamiliar with how it should be tracked (SQ04), the development teams did not
have the awareness of technical debt (SQ05), or they did not believe it was necessary to do so (SQ14 and SQ15).

4.7 Conclusion

This chapter describes the findings after looking at 53 ATD items from the 15 different non-trivial software systems from eight different organizations, as reported by the participants. From the data collected, we created two classifications related to ATDs based on their themes and similarities. The first classification is on the source of ATD: technology choice, requirement, obsolete system, knowledge loss, recycling & old code integration, suboptimal architecture, poor build & deployment method, and lack of testing plan. Here, participants seemed to be most troubled by their choice of technology and the uncertainty of their requirements. The other classification, drivers of ATD, is: business plan, lack of experience or awareness, development resources, attitudes & practices, and software aging. We observed that business plan was either the main driver or the contributor towards the decision of incurring ATD. Furthermore, we analyzed the participants’ technical debt payment strategies for all 53 ATD cases and found three distinct debt payment approaches were being used: active, reactive, and ignoring. The most common debt payment approach is reactive where development teams worked around the debt until the debt payment could no longer be delayed. Finally, we reported the current practice of tracking technical debts and found the participants usually used their issue tracking software to management technical debt issues, with SonarQube as the most commonly used TD specific tool.
Chapter 5

Discussion

This study set a goal to examine industry software practitioner experience with architectural technical debt to assist practitioners and future researchers in understanding ATD accumulations and causes better. In this chapter, we discuss the findings of this study, answer our research questions, and present the insights gained from this study.

5.1 Taxonomy of ATD

In this study, we created two taxonomies related to the architectural technical debt. One is based on its source, while the other is based on the drivers that lead to incurring ATD. First, from the participant responses, we discovered that the cause of architectural technical debts can be grouped together in the following way: technology choice, requirement, obsolete system, knowledge loss, recycling & old code integration, suboptimal architecture, poor build & deployment method, and lack of testing plan (Table 5). The most common sources of ATD reported are technology choice and requirement which were discussed by participants from different projects. Another interesting observation is that the source of ATD is influenced mostly by the organization practices, background, and its software rather than developers. For example, the participants from a start-up company discussed more on requirement uncertainty issues because they often built the system first before knowing the exact software system specification details from the customers. On the other hand, the participants who worked with matured
systems tended to discuss more frequently on unexpected major requirement changes due to shifts in client needs or business goals, and obsolete components due to software aging.

<table>
<thead>
<tr>
<th>Source</th>
<th>Description of the source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology Choice: Technology Trap</td>
<td>Software relies on particular technologies to be functional</td>
</tr>
<tr>
<td>Technology Choice: Early Adopter</td>
<td>Decision to use the latest and greatest technology available</td>
</tr>
<tr>
<td>Technology Choice: Suboptimal Technology Choice</td>
<td>The chosen technology does not fully meet the requirements</td>
</tr>
<tr>
<td>Requirement: Incorrect &amp; Unclear Requirement</td>
<td>Requirements that are not properly defined</td>
</tr>
<tr>
<td>Requirement: New Purpose</td>
<td>Matured software that received major requirement changes</td>
</tr>
<tr>
<td>Obsolete System</td>
<td>Software system that faces aging issue</td>
</tr>
<tr>
<td>Knowledge Loss</td>
<td>Developer knowledge not passed on to others</td>
</tr>
<tr>
<td>Recycling &amp; Old Code Integration</td>
<td>1. Recycling: Reusing code developed previously to speed up development</td>
</tr>
<tr>
<td></td>
<td>2. Old Code Integration: Fitting two components that were developed at different times</td>
</tr>
<tr>
<td>Suboptimal Architecture</td>
<td>Software architectural designs that are made either intentionally or unintentionally</td>
</tr>
<tr>
<td>Poor Build &amp; Deployment Method</td>
<td>Quick and dirty ways of handling system build and deployment</td>
</tr>
<tr>
<td>Lack of Testing Plan</td>
<td>Testing strategy is not considered in early stage of software development which results in an architectural design that does not support proper testing</td>
</tr>
</tbody>
</table>

Table 5. Sources of ATD

To better understand each taxonomy, here are examples for each category:

a) **Technology Choice: Technology Trap**

To save time during software development, a development team integrated a third-party software component that possessing all the feature sets needed for their core client-server communication
service. However, this third-party software component is developed and maintained by another party who has recently chosen to stop actively updating this software. Now, the development team needs to perform a major rewrite to move away from this third-party component, or they are stuck with a software system that rarely receives new features and bug fixes.

b) **Technology Choice: Early Adopter**

A development team has chosen to use a new framework that is the successor of a popular state-of-the-art framework. However, this new framework is still in Beta and has many bugs and immature features. The development team has chosen this framework because they want to have a first-hand experience on the new features and a jump-start on a framework that will eventually replace the current one. Because of this choice, the development has faced system-breaking issues each time the Beta software has a major update and must live with unresolved bugs that persist within the system.

c) **Technology Choice: Suboptimal Technology Choice**

A development team has chosen to use a framework that they believed to be sufficient for their throughput requirements. Following integration of this framework, they realized it did not produce the performance result they had expected. Now, the most logical choice for the development team is to replace this suboptimal framework with another that meets the requirements.
d) **Requirement: Incorrect & Unclear Requirement**

A development team was tasked by the management team to create a prototype for a potential market. This newly development system will be pitched to a few interested buyers. Since the functional and non-functional requirements, including the software architecture, were established at the best guesses of both the developers and management team without potential buyer input, the software system is likely to require rewrites and architectural redesigns to fit the buyer’s requirements later on down the road.

e) **Requirement: New Purpose**

A custom-built data management software was originally designed for a specific customer. Now, the management team is interested in selling this software to other customers who would also find this software system useful. After the announcement of this new requirement, the system immediately becomes suboptimal because the architectural design of this software system was only intended to support the current customer’s system and operating environment. The development team expects to deal with architectural redesigns on the software system.

f) **Obsolete System**

A software system was developed 20 years ago with the best practices and state-of-the-art technologies. However, those technologies are obsolete by today’s standard and they are no longer supported by their vendors anymore. This software system is now facing software evolution issues because the old components do not support new technologies well and lots of workarounds have to be made when developing new features.
g) **Knowledge Loss**

A development team has been through many generations of developers. These developers have formed various architectural decisions during their time on the team, most of which were not properly documented. As a result, when the current generation of developers try to understand some of the old architecture and update it, they were uncertain if their changes would have negative impacts on the system’s quality and functionalities.

h) **Recycling & Old Code Integration**

A development team has just started working on a new software system. During the planning phase, they discovered they could borrow and integrate some of the components they have developed in a previous project. While they understood these components will need modifications and might not work well with the new system, they have decided to integrate them regardless to save development effort.

i) **Suboptimal Design**

A development team did not seriously consider modularity when they first developed a software system. When the client’s requirements were updated and the some of the architectural designs needed to be changed, they found the tightly coupled components were difficult to modify and extra development effort were made to correct the modularity issues.

j) **Poor Build & Deployment Method**

A development team has chosen to save development effort by allowing their patching executable to completely replace components that needed to be updated instead of modifying
them. This method is being used since the development team does not want to spend development effort to make the system components work with the patching executable and they can simply “copy and replace all” to save time.

**k) Lack of Testing Plan**

A development team had not created a testing strategy in the early phase of the software development. Now, they are trying to create a thorough testing plan, but have found that the architectural design of their system does not allow easy access to its internal data for testing. To solve this problem, the development team has to redesign some of its components to allow the testing framework to access the internal data.

For our second taxonomy, after analyzing the each reported technical case, we observed that business plans, lack of experience or awareness, development resources, attitudes & practices, and software aging are the main drivers of incurring ATD (Table 6). Here, two categories, business plans and development resources, we labeled them as business related factors. A business plan is made by the management to seek profits and happy customers. Development resources are allocated for the development team by management to make sure each dollar is spent effectively and no budget overrun. Therefore, in these two categories, the management is the one causing ATD rather than developers. From this, we conclude that since business people have an influence on incurring ATD, technical debt researchers must look at both business side and the developer side when conducting research so that they can include all stakeholders.
<table>
<thead>
<tr>
<th>Driver</th>
<th>Description of the driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Plans: Time</td>
<td>Time pressure and time-to-market impose limits on the developers</td>
</tr>
<tr>
<td>Business Plans: Changes &amp; Uncertainties</td>
<td>1. Objective and specification changes make the current implementation suboptimal.</td>
</tr>
<tr>
<td></td>
<td>2. Uncertainties creates unstable requirements for the developers</td>
</tr>
<tr>
<td>Lack of Experience or Awareness</td>
<td>Developers were unfamiliar with chosen technology or not aware of potential ATD they can introduce</td>
</tr>
<tr>
<td>Development Resources</td>
<td>Scarcity of resources such as number of developers and skillsets impose limits on the development team</td>
</tr>
<tr>
<td>Attitudes &amp; Practices</td>
<td>The cultural aspect of the development team such as “Everyone here does things this way”</td>
</tr>
<tr>
<td>Software Aging</td>
<td>All software ages naturally and becomes suboptimal</td>
</tr>
</tbody>
</table>

Table 6. Why Was ATD Incurred?

Note that it is very common for an architectural technical debt item to have multiple sources or drivers. In this study, we have chosen to use the best-fit strategy for simplicity purpose when categorizing each ATD item reported and let them only have one main source and one main driver.

Originally, we wish to create an ATD classification that takes into account the architectural designs such as the types of database, design patterns, architectures, and technologies used. The goal was to see if we could discover recurring architectural decisions that are vulnerable to ATDs. After analyzing all the collected data, we could not establish one because each ATD case described by the participants used different technologies and faced different technical challenges,
and there were also cases which a combination of technical issues that created one unique ATD case.

5.2 Average Time to Pay Off ATD

A generalization cannot be made on the average effort required to pay off an architectural technical debt from a particular origin. Participants reported a wide range of estimated developer efforts from a few weeks to a few months. One possible explanation is that each ATD has its own unique challenges and manifests itself differently. A more focused, larger scale research on a particular ATD issue might help to identify whether the average effort would be generalizable. Another possible explanation we have is that many participants had not yet started developing debt payment plans. Therefore, the efforts reported were very rough estimates based on their best guesses. The actual development effort could be very different than the estimated effort. A better approach to tackle this problem in a future research would be to only look at ATDs that have been fully repaid, or having an ongoing observation type research which the researcher would observe the exact development effort spent each week on debt payment.

5.3 Is ATD Avoidable?

We asked participants if they believed the ATD issues they reported were avoidable in the first place and what they could have done differently. The majority of the participants believed they could have done something differently, such as using another third-party framework, refining their designs, and so on to prevent their current problems from occurring. Unfortunately, as SI18 said “Hindsight is always 20/20.” So, is ATD avoidable? To some extent, it is. We saw many participants admitted that, had they spent more time performing on initial analysis, research, and
planning on their designs, they would not have been facing these troubles today. On the other hand, those who replied unavoidable either believed it was impossible to foresee the potential problems or felt that their designs were the only available solution that made sense at the time due to budget, deadline, technologies, and other factors. In conclusion, ATDs can be prevented in some cases. However, based from what have been reported, taking extra time to review design decisions and doing some related research are the best “insurance” against ATD.

5.4 Managing Technical Debt

We discovered that majority of participants were aware of technical debt in their software systems and used issue tracking software such as Jira to manage it. Issue trackers are great for monitoring and communicating the presence of technical debt between developers by recoding the problems and sharing them among the developers. Even though they are useful for making records, these tools do not help with the technical debt identification process. During the interviews, the participants have expressed interest in having a tool that specialized in technical debt identification. For those who are already using a TD tool that has an automated identification feature, all of them have been utilizing SonarQube. As discussed previously, SonarQube is not effective for ATD identification since, as a static code analyzer, its automation cannot see beyond code-level issues. ATD at design-level will still need to be identified and managed manually. From this, we believe that a more powerful TD tracking and identification tool that can look beyond source code still need to be developed.

One interesting observation is that interviewees who have expressed interest in using a TD tool in the future were concerned that their management will not see the value of having one if such
tool does not distinguish itself from other project management software. Since technical debt, like a software bug, is merely another issue in software systems, a strong case is required on why having a specialized tool is important and what value it provides.

5.5 Business Side is Important

One insight gained from the study is that the management plays an important role in technical debt. In the previous discussions, we note that ATDs can be caused by the business decisions and not simply those of developer. From the taxonomies we created, business plan, and development resources are all business-related causes. Secondly, the participants have mentioned that they do not usually have a say on whether a plan to manage technical debt should be made, since reducing technical debt is not a trivial task. This kind of additional work will require reviewing and approval by managers: In order to start an ATD management process, the business side must acknowledge that ATD is an issue, and then a commitment to mitigate the problem will need to be made. Only following this could the developers propose a plan and a development resource budget to manage ATD.

Traditionally, architectural technical debt payment has been focusing on development productivity and meeting deadlines. In this study, we found that participants, especially those who also work with company management, seem to be very interested in learning more about the advantages and disadvantages to business of paying off technical debt. According to them, paying off debts that are troublesome definitely brings benefits to the development team, but this might not be the case for the business side. In one particular example, SI14 and SI15 commented that their medical software system was regulated by the government and significant changes to
the system would require it to be reviewed and approved by relevant agencies again. Even though paying off debt will increase the productivity significantly for SI14 and SI15’s team, the benefits did not exceed the negative impact of having an additional validation process that the system has to go through. Furthermore, some participants have reported that the main challenge of having a debt reduction plan was to demonstrate the business advantages it could bring. Therefore, knowing the advantages and disadvantages to the business of paying off the debt would make it much easier for the management to make an informed decision on debt reduction commitment.

5.6 Takeaway for developers

What can software developers do with the information discussed here?

- To better understand and create awareness of architectural technical debt in developers’ software systems, they can do the following:
  1. Use the source of ATD taxonomy (Table 5) to describe the ATD items when trying to identify and document whether current or potential future debt items are present within the software system. A development team can also put these categories into their issue tracker’s vocabularies to adopt these terms.
  2. Use the drivers of ATD (Table 6) when documenting the reasons of why debts were incurred in 1.
  3. Use the results of 1 and 2 in meetings with other developers to spread information on the debt items identified and educate them about the categories they fall into.
• To mitigate the possibilities of developing ATD issues, developers should spend more time on research and understand uncertainties of their system to avoid possible ATD issues in the future.

• Developers should hold regular meetings with management teams to understand current and long-term business goals regarding the software being developed, so that the developers can prepare and design the system to be ready for those goals.
Chapter 6

Conclusion

This thesis presents architectural technical debt taxonomies that would help software practitioners and technical debt researchers to understand ATD better. We also present the findings of current ATD management practices in the industry. In total, we analyzed 53 architectural technical debt issues in 15 non-trivial software intensive projects collected from 18 interview participants and 10 questionnaire participants in eight organizations. This chapter will summarize the answers to the research questions, the contributions of this study, and future work suggestions.

6.1 Answering Research Questions

The existing literature on architectural technical debt contributes mostly in terms of on technical debt management strategies, code-level ATD discovery, and ATD understanding. The goals of this study are to examine architectural technical debt’s causes and accumulations and the current practices of managing architectural technical debt. In this section, we will answer the questions stated in Chapter 1.

A. RQ1: How is architectural technical debt introduced in a software system?

We answer this question by first listing out the sources of ATD (Table 5). The two most common causes are technology choices and requirement related issues. In our study, these two issues made
up of 28 out of a total of 53, or 53% of all the ATD cases reported. Then, to understand better, we looked at the reasons why such designs were made for each case and developed the list shown in Table 6. Based on our observations, business decision is the most common factor that leads to the incursion of architectural technical debt, especially in cases involving on time pressure, unexpected changes, and uncertainties. The second most common reason is the developer’s lack of development experience and awareness on potential issues. This often relates to developers using a technology that they have never worked with before and being unaware of the potential issues that it could bring.

B. *RQ2: How do software practitioners manage architectural technical debt?*

To answer RQ2, we examined the two major efforts of technical debt management: tracking, and debt payment. In Section 4.6, we found software practitioners use one of the three strategies: active, reactive, and ignoring, to handle debt payment with the reactive approach being the most common. However, as discussed in Section 5.2, we could not generalize the average time required to pay off a specific type of architecture debt. As for tracking, issue tracking software is the most popular and SonarQube is the most used TD specific tool for ATD identification purposes.

6.2 Contributions of This Study

The contributions of this thesis include:

- The development of two taxonomies (Tables 5 and 6) on the causes and origins of design-level architectural technical debt experienced by software practitioners in the industry.
• Reporting, in Chapter 5, the discussions taking place with study participants and the insights gained.

• Identifying, in Section 5.4, the need for better architectural technical debt specific tools that includes design-level ATD identification ability.

• The conclusion that the non-technical stakeholder (business management) is an important decision maker in technical debt management.

• Suggesting future works on questions that this study did not answer and new questions for future researchers to answer.

The findings of this study have agreed or revealed similar results to other studies on technical debt:

• Our taxonomy on the sources of technical debt is very similar to the one presented by Ernst et al. (2015).

<table>
<thead>
<tr>
<th>Our Taxonomy</th>
<th>Similar categories from Ernst et al.’s Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology Choice: Technology Trap</td>
<td>Dependencies on external team’s code</td>
</tr>
<tr>
<td></td>
<td>Dependencies on external software packages</td>
</tr>
<tr>
<td>Technology Choice: Early Adopter</td>
<td></td>
</tr>
<tr>
<td>Technology Choice: Suboptimal Technology Choice</td>
<td>Bad architecture choices</td>
</tr>
<tr>
<td>Requirement: Incorrect &amp; Unclear Requirement</td>
<td></td>
</tr>
<tr>
<td>Requirement: New Purpose</td>
<td></td>
</tr>
<tr>
<td>Obsolete System</td>
<td>Obsolete technology</td>
</tr>
<tr>
<td></td>
<td>Obsolete code</td>
</tr>
<tr>
<td>Knowledge Loss</td>
<td>Lack of code documentation</td>
</tr>
<tr>
<td>Recycling &amp; Old Code Integration</td>
<td></td>
</tr>
<tr>
<td>Suboptimal Design</td>
<td>Poor Build &amp; Deployment Method</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Lack of Testing Plan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Ernst et al.’s taxonomy but not used in this table:
Overly complex code, Inter module dependencies, Code duplication or repetitive edits, and Other

<table>
<thead>
<tr>
<th><strong>Table 7. Comparison of Sources of ATD taxonomy to Ernst et al.’s</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Our taxonomy</strong></td>
</tr>
<tr>
<td>Business Plans: Time</td>
</tr>
<tr>
<td>Business Plans: Changes &amp; Uncertainties</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Lack of Experience or Awareness</td>
</tr>
<tr>
<td>Development Resources</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Attitudes &amp; Practices</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Software Aging</td>
</tr>
</tbody>
</table>

From Martini et al.’s categories, but not used here:
Business factor: Priority of features over product, Parallel development, and Effects uncertainty.

| **Table 8. Comparison of Driver of ATD to Categories from Martini et al.’s Study** |
• We identified that the reactive approach, that is, waiting until debt payment can no longer be delayed, is the most common strategy taken to manage technical debt; this agrees with the “Crisis-based ATD management” claim by Martini et al. (2014).

• Our results agree with the findings made by Martini, Besker, and Bosch (2018) that software practitioners use mostly issue trackers and static analyzers to manage TD and better TD specific tools are still needed.

• Our results agree with Klinger et al. (2011) that there is a communication gap between technical stakeholders (developers) and non-technical stakeholders (management), and the non-technical stakeholders are often the ones making technical debt decisions.

6.3 Future Work

Some suggestions, including the ones discussed in the previous chapters, for further investigation into this area are:

• Refine the questionnaire and interview questions we developed and replicate the data collection process with more software practitioners from different organizations to validate this study’s findings and also investigate further.

• In this study, we discovered the importance of non-technical stakeholders in managing ATD in this study. However, we did not investigate further into this finding since that is beyond our scope. Currently, there are only a handful of studies that involving non-technical stakeholders as research participants or discussing their roles and involvement in ATD. Further studies that involve non-technical stakeholders and look into their influence on architectural technical debt decisions and management will need to be made.
Our study tried to identify recurring architectural implementations that often lead to ATDs, but no valid conclusion could be drawn from our data analysis. This is mostly due to the lack of evidence that any two architectural implementations are similar. A dedicated study should be made on this topic in the future.

We were unable to find the average development effort required to pay off a specific type of architectural technical debt, since most participants have not started payment yet, making the data unreliable. A future research that only examines at ATDs that have been fully repaid or conducts an ongoing observation of the exact development efforts spent would be a more appropriate approach. Also, it would be interesting to know the effort needed to reduce debt to an acceptable level and compare this with effort required to fully pay off a debt.

Our findings suggested there is still lack of well-rounded architectural technical debt specific tools available for software development teams to effectively identify and manage ATD at non-code-level. Future research and development for such tool is still necessary.
References


Bavota, G., & Russo, B. (2016). *A large-scale empirical study on self-admitted technical debt*. Proceedings of the 13th International Conference on Mining Software Repositories, IEEE, 315-326. doi: [http://dx.doi.org/10.1109/MSR.2016.040](http://dx.doi.org/10.1109/MSR.2016.040)


https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Proceedings of the 2nd Workshop on Managing Technical Debt, 31-34.
doi:http://dx.doi.org/10.1145/1985362.1985370

IEEE International Conference on Software Maintenance (ICSM), 528-531.
doi:http://dx.doi.org/10.1109/ICSM.2011.6080824

Guo, Y., Spinola, R. O., & Seaman, C. (2016). Exploring the costs of technical debt management
— a case study. Empirical Software Engineering, 21(1), 159-182.
doi: http://dx.doi.org/10.1007/s10664-014-9351-7

study in locating the architectural roots of technical debt. Proceedings of the 37th
International Conference on Software Engineering-Volume 2, 179-188.
doi: http://dx.doi.org/10.1109/ICSE.2015.146

technical debt. Proceedings of the 2nd Workshop on managing technical debt, IEEE CS,
35-38, doi: http://dx.doi.org/10.1145/1985362.1985371

Kosti, M. V., Ampatzoglou, A., Chatzigeorgiou, A., Pallas, G., Stamelos, I., & Angelis, L.
of the 43rd Euromicro Conference on Software Engineering and Advanced Applications
(SEAA), 2017, 329-333, doi: http://dx.doi.org/10.1109/SEAA.2017.59


Lim, E., Taksande, N., & Seaman, C. (2012). A balancing act: what software practitioners have to say about technical debt. *IEEE software, 29*(6), 22-27. doi:[http://dx.doi.org/10.1109/MS.2012.130](http://dx.doi.org/10.1109/MS.2012.130)


doi: [http://dx.doi.org/10.1109/MTD.2013.6608671](http://dx.doi.org/10.1109/MTD.2013.6608671)


Appendix A – Interview Guide and Questions

Interview Guide

Technical Debt Short Summary:

The metaphor of technical debt was coined by Ward Cunningham in 1992 describing taking shortcuts or using immature and imperfect code during software development due to lack of time and resources is similar to taking on debts. In the short term, taking on these debt items can speed up the development speed to meet deadlines, but these debt items must eventually be repaid to promote easy software evolution and maintenance. The initial usage of technical debt was to address code quality issues, but in recent years, the usage of the metaphor has been extended to the whole software lifecycle.

Question main question to ask:
During development, what decision you or your team has made that you regretted?
We are looking at ONE project

Things to lookout for:
1. There might be a difference between development decision and Technical Debt
2. Intentional vs Unintentional
3. There might be multiple TD inside one discussed case.

TD Item:
TD Principal: The optimal solution that was suppose to happen
TD Interest: Time spent on non-optimal solution
TD Payment: The act of repaying the debt. Also ask how much time was spent
TD Status: What is the current status of the debt. Repaid? Still there? Partially paid? Or Doesn’t need to be paid at all.

Interesting things found in previous interviews:

1.

2.
Questions & Guide:

1. Pre-interview steps
   1.1. Introduction
   1.2. Repeat again participant rights
   1.3. Make sure consent form is well understood and signed

2. Introductory Questions
   2.1. Questions about the subject
       2.1.1. How many years of experiences you had in software industry?
       2.1.2. What roles and how long you have spent on each role?
       2.1.3. What responsibility you had in each role?
       2.1.4. Are you familiar of TD? What is your understanding of it?
       2.1.5. Spend time describe TD

3. Technical Debt Questions
   Can you describe a project that you have seem TD with? Name? How big is it? Years of development?
   Selecting one specific software system X:
   3.1. Elicit potential elements of technical debt
       3.1.1. What early design decisions are you regretting now in project X (choice of technology, or process, of structure, of framework, …)?
   3.2. Describe technical debt occurred in project X (Discovering the TD)
       3.2.1. What happened? why did you make this (unfortunate) choice at the time?
       3.2.2. Who identified the problem? Yourself or others?
       3.2.3. How does this particular tech debt item manifest itself (description)?
       3.2.4. What was the reason you have to incurred debts?
       3.2.5. Did you realize the existence of debt at that moment?
       3.2.6. Where alternatives considered at that time?
       3.2.7. Any follow up questions
   3.3. Describe the consequences of debt in project X
       3.3.1. What was the consequences and cost? What is it that you cannot do, or at a too high cost?
       3.3.2. Is it being managed or planned to be fixed?
       3.3.3. What did you regret?
       3.3.4. What could have been done instead?
       3.3.5. Any follow up questions
       3.3.6. If another project, Y, is planned to be discussed, repeat from 2.2 on project Y
   3.4. Describe technical debt tools
       3.4.1. Do you know any technical debt management tool?
       3.4.2. Have you used it? if no, why not?
       3.4.3. If yes
3.4.3.1. Is it effective?
3.4.3.2. To best of your knowledge, does it find all the debts?
3.4.3.3. What is the cost of using it?
3.4.3.4. Will you continue use it?
3.4.4. If no
   3.4.4.1. Will you be interested in knowing what it does?
   3.4.4.2. Will you be interested in using it?

4. Closing
   4.1. Are there any thoughts, comments, or questions you might have?
   4.2. Are there anything related to technical debt that you might want to share but that is not directly related to this study?
   4.3. Thank you for your time.
Appendix B – Copy of Questionnaire

Technical Debt Survey

Page 1

About this study

The metaphor of technical debt was coined by Ward Cunningham in 1992 describing taking shortcuts or using immature and imperfect code during software development due to lack of time and resources is similar to taking on debts. In the short term, taking on these debts can speed up the development speed to meet deadlines, but these debts must eventually be repaid to promote easy software evolution and maintenance. The initial usage of technical debt was to address code quality issues, but in recent years, the usage of the metaphor has been extended to the whole software lifecycle.

In a recent 2016 technical debt seminar in Dagstuhl, technical debt is defined as follow:
"In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."

The purpose of this study is to investigate technical debt elements in large, non-trivial software system. In particular, we want to look at technical debt outside of source code level such as issues related to requirement, architecture, software choices, and etc.

This survey would take at least 30 minutes of your time. It is highly recommended to use the save feature if you wish to save the survey for later. Please note that results of your participation will be reported without any reference to you specifically. All data will be treated confidentially and your identity will not be revealed in reporting the study results.
Page 2

Enter your participant ID

Please refer to your invitation email

Type here

https://survey.abc.co/s/survey/print?token=zJnjuUEOggzAMBp_3M4wGxXEQ5QoAvAjxNRxWqQxg6rOPW4s6vZL-QzebDYx100HSQ98SGyQ-UDxKzv82k...  2/13
Page 3

Background

Your organization

- 1 - 10
- 11 - 50
- 51 - 200
- 201 - 500
- 501 - 1000
- 1001+
- Private
- Public
- Non-profit
- Government
- Others

If organization type is "others", please specify

Type here

Does your organization describe itself as a "startup company"?

- Yes
- No

Industry

- Aerospace
- Agriculture
- Chemical
- Computer/Software
- Construction
- Defense
- Education
- Energy
- Entertainment
- Financial Services
- Food
- Health Care
- Hospitality
- Information
Technical Debt Survey - 100%

- Manufacturing
- Mass Media
- Telecommunications
- Transportation
- Water
- Others

If industry is "others", please specify: Type here

**Yourself**

- 0 - 5
- 5 - 10

Years of experience in the industry

- 10 - 15
- 15 - 20
- 20+

- 0 - 5
- 5 - 10

How many years have you been working in your organization

- 10 - 15
- 15 - 20
- 20+

Your current role / title in the organization: Type here

Responsibility of your role

Type here

How familiar are you with the term technical debt?

- Very familiar
- Familiar
- Somewhat familiar
- Heard the term a few times
- Never heard of it

In your own words, prior to this survey, how would you describe the term technical debt to a developer who has never heard of it before?

https://surveyabc.xai/tdsurvey/print?token=Z31hLEOgzAMBP_3M48G6lX4Ze9Qn3cJJoNIRxWkGk6tOPw4s6yfZL-QxcD2Yz0f0HDSQ855YQ-QDxKzv52k...
Page 4

Project with technical debt

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

In the study, we want to focus on one software intensive project that you have worked on recently and faced issues with technical debt outside of source code level. Please consider ONE software intensive project that we can explore and UP TO THREE different design decisions you or your team has regretted the most.

Please use the save feature and come back later if you want to take some time to think about what project/cases to discuss.

Note: Software bugs usually are not considered as a type of technical debt.
Page 5

Project background

Please describe ONE software system you have worked on recently where technical debt manifests itself:

- <500k
- 500k - 1M
- 1M - 2.5M
- 2.5M - 5M

Software size (in lines of code):

- 5M - 10M
- 10M - 50M
- 50M+
- I don’t know

- 0 - 2
- 2 - 5
- 5 - 10

Software age (years):

- 10 - 15
- 15 - 20
- 20 - 30
- 30+

- 0 - 5
- 5 - 10
- 10 - 15
- 15 - 20
- 20 - 30
- 30+

Estimated remaining life (years):

Development team size at the peak of this project (if of people):

- 1 - 10
- 11 - 20
- 21 - 50
- 51 - 100
- 101 - 200
- 201 - 500
Purpose of this software

Type here

12/3/2017
Technical Debt Survey - 100%

500+
Question: In the software system you just described, what design decisions have led to significant technical debt that you or your team has regretted the most?

Please provide up to three cases. If you want to spend some time to think about your responses, you can utilize the save feature to come back later.

**Case 1**

<table>
<thead>
<tr>
<th>What was the design decision you or your team regretted?</th>
<th>Type here</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why this design choice was made? Was the team aware of the potential issues?</td>
<td>Type here</td>
</tr>
<tr>
<td>Was there another design choice available at the time? If yes, why it was not chosen?</td>
<td>Type here</td>
</tr>
<tr>
<td>What are the consequences of this choice? What are the development penalties the team has to pay?</td>
<td>Type here</td>
</tr>
<tr>
<td>If to quantify each month, how many extra hours you or your team has to spend on maintenance effort and workarounds because of this design decision?</td>
<td>Type here</td>
</tr>
<tr>
<td>What is the current state of this technical debt? Is it being managed? Are there any debt payment?</td>
<td>Type here</td>
</tr>
<tr>
<td>If the debt still manifests, how many weeks in development time would the team need to eliminate this issue?</td>
<td>Type here</td>
</tr>
<tr>
<td>Do you think this was avoidable? What could have done differently?</td>
<td>Type here</td>
</tr>
</tbody>
</table>
Page 7

Question: In the software system you just described, what design decisions have led to significant technical debt that you or your team has regretted the most?

Please provide up to three cases. If you want to spend some time to think about your responses, you can utilize the save feature to come back later.

Optional: Case 2

If you don't have more cases to share, please leave everything blank on this page and proceed to the next page

<table>
<thead>
<tr>
<th>What was the design decision you or your team regretted?</th>
<th>Type here</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why this design choice was made? Was the team aware of the potential issues?</td>
<td>Type here</td>
</tr>
<tr>
<td>Was there another design choice available at the time? If yes, why it was not chosen?</td>
<td>Type here</td>
</tr>
<tr>
<td>What are the consequences of this choice? What are the development penalties the team has to pay?</td>
<td>Type here</td>
</tr>
<tr>
<td>If to quantify, each month, how many extra hours you or your team has to spend on maintenance effort and workarounds because of this design decision?</td>
<td>Type here</td>
</tr>
<tr>
<td>What is the current state of this technical debt? Is it being managed? Are there any debt payment?</td>
<td>Type here</td>
</tr>
<tr>
<td>If the debt still manifests, how many weeks in development time would the team need to eliminate this issue?</td>
<td>Type here</td>
</tr>
<tr>
<td>Do you think this was avoidable? What could have done differently?</td>
<td>Type here</td>
</tr>
</tbody>
</table>

https://surveyabc.ax/s/hdburwv/print?token=ZnjiUE0ogAAMBP_3M4bGa1XEZ6sQ6xArAJeN8wWqgQ0dOPW4svvZL-Qc8FBY1a0HED5Q985GyQ-UDxKzt62... 10/13

88
Page 8

Question: In the software system you just described, what design decisions have led to significant technical debt that you or your team has regretted the most?

Please provide up to three cases. If you want to spend some time to think about your responses, you can utilize the save feature to come back later.

Optional: Case 3

If you don’t have more cases to share, please leave everything blank on this page and proceed to the next page.

What was the design decision you or your team regretted?

Type here

Why this design choice was made? Was the team aware of the potential issues?

Type here

Was there another design choice available at the time? If yes, why it was not chosen?

Type here

What are the consequences of this choice? What are the development penalties the team has to pay?

Type here

If to quantify, each month, how many extra hours you or your team has to spend on maintenance efforts and workarounds because of this design decision?

Type here

What is the current state of this technical debt? Is it being managed? Are there any debt payment?

Type here

If the debt still manifests, how many weeks in development time would the team need to eliminate this issue?

Type here

Do you think this was avoidable? What could have done differently?

Type here
Technical debt management method

Does your team actively track technical debt in projects?

☐ Yes  ☐ No

If you answered "yes", how is technical debt tracked? If "no", why not?

Type here