Technical Debt: What Software Practitioners Have to Say

by

Erin Lim

B.A.Sc., University of British Columbia, 2003

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

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

The University of British Columbia

October 2012

ABSTRACT

Technical debt is a metaphor for the consequences that software projects face when they make trade-offs to implement a lower quality, less complete solution to satisfy business realities. While interest in the metaphor is slowly gaining traction in academic research, there already exists a significant amount of discussion in website logs (blogs). The purpose of this research is to validate the existing definitions and to enrich it with the insights, experiences and lessons learned of software practitioners working in industry. The results are based on a series of one-hour interviews conducted with nineteen software practitioners that investigates the definitions, attributes, causes, symptoms and management of technical debt. It is validated by the findings from a secondary study based on "Hard Choices", a board game designed to teach the concepts of technical debt, and a replication study conducted by Nitin Taksande at the University of Maryland, Baltimore County. The outcome of this research provides software practitioners with a set of guidelines to recognize and manage their technical debt. The guidelines state that incurring technical debt is unavoidable because software projects need to meet business goals. Instead, project teams should learn to manage technical debt by developing effective communication skills that bring visibility to its existence in order to enable all project stakeholders to take ownership in mitigating its risks.

PREFACE

This thesis presents a number of semi-qualitative studies on technical debt. The interview study was conducted in collaboration with my supervisor, Dr. Philippe Kruchten. I designed, interviewed and analyzed the data from the interviews, while Dr. Kruchten provided guidance on the methodology, reviewed the analysis of my results, and edited the manuscript. The game study was designed and conducted in collaboration with Ipek Ozkaya, Robert Nord and Nanette Brown of the Software Engineering Institute at Carnegie Mellon University and Dr. Philippe Kruchten. I contributed to the design of the game board, “Hard Choices”, and its rules, organized and ran game play sessions. My data analysis also examined my collaborators’ observations from the game play sessions that they ran. Nitin Taksande from the University of Maryland, Baltimore County, conducted a replication study in collaboration with his supervisor, Dr. Carolyn Seaman. The methodology of his replication study was based on my design of the interview study.

The results of the studies described in this thesis resulted in a number of publications:

1. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C. B., Sullivan, K., and Zazworka, N., "Managing Technical Debt in Software-Reliant Systems," Paper presented at the *Future of Software Engineering Research (FoSER 2010) Workshop,* part of FSE 2010, Santa Fe, New Mexico, USA, 2010, ACM, pp.47-52. DOI: 10.1145/1882362.1882373

2. Brown, N., Kruchten, P., Lim, E., Ozkaya, I., and Nord, R., "The Hard Choices Game Explained,” Pittsburgh: Software Engineering Institute, 2010. Available at: http://www.sei.cmu.edu/library/abstracts/whitepapers/hard-choices-game-explained-v1-0.cfm

3. Brown, N., Nord, R., Ozkaya, I., Kruchten, P., and Lim, E., "Hard Choice: A game for balancing strategy for agility," presented at the *24th IEEE Computer Society Conference on Software Engineering Education and Training (CSEE&T 2011),* Honolulu, HI, USA, 2011, IEEE Computer Society. DOI: 10.1109/CSEET.2011.5876149

4. Lim, E., Taksande, N., and Seaman, C. B., "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software*, vol. 29 (6), 2012 (to appear).

Furthermore, this study was reviewed by the University of British Columbia Behavourial Research Ethics Board to ensure ethical considerations were addressed. The board approved of the study under certificate number H11-00062.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to acknowledge and thank the following people for their contribution towards the completion of my thesis:

- Dr. Philippe Kruchten, my supervisor, who very patiently guided me through the research and development of this thesis;

- Ipek Ozkaya, Robert Nord and Nanette Brown of the Software Engineering Institute (SEI) at Carnegie Mellon University, who provided me with the opportunity to participate in their research project;

- Nitin Taksande and Carolyn Seaman of the University of Maryland, Baltimore County, who validated the results of my research and collaborated with me on a paper publishing our findings in the upcoming special issue of *IEEE Software*;

- Alan Carruthers, Scott Chan, Lawrence Croft, Alastair Foreman, Matthew Gerde, Joe Gnocato, Michael Joy, Dave Kauffman, AJ Leitch, Liezel Lorico, Edmund Low, Peter McDonald, Adrian Pang, Ron Starling, Nelson Siu, Graham Smith, Steve Smith, Brian Tomic, and Judith Vosko, who took time out of their busy lives to talk to me about their experiences with technical debt;

- Most importantly, my parents, Rita and Jerry, my sister, Nicole, and my brother, Brian, who supported, encouraged, and stood by me every step of the way.

To my Dad:

Experience is a good teacher, but you are an even better one. Thank you for being my greatest mentor. I am a better engineer because of you.

CHAPTER 1
INTRODUCTION

In his 1992 OOPSLA (Object-Oriented Programming, Systems, Languages and Applications) experience report [1], Ward Cunningham introduced a concept that caught the attention of the software development community:

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite…The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*

This concept, which Cunningham coined as "technical debt", aptly describes the consequences that software projects face when they make trade-offs to implement a lower quality, less complete solution in order to meet budget and schedule constraints imposed by business realities. The outcome of incurring technical debt is, often, increased maintenance cost and effort.

Technical debt is not bad; when incurred strategically and managed well, it enables projects to achieve a huge advantage over its competition by being able to deliver to market quickly. However, allowing too much technical debt to accumulate can be problematic by slowing down the project's forward progress, eventually bringing development to a standstill. The purpose of this study is to identify guidelines to help software practitioners recognize and manage technical debt on their software projects.

1.1 Significance

The technical debt metaphor has caught the attention of the software development community for its ability to effectively communicate to non-technical stakeholders an issue that technical stakeholders must deal with on a daily basis. While interest in the metaphor is slowly gaining traction in academic research, there already exists a significant amount of discussion about technical debt in website logs (blogs).

The results of this study will validate the existing definitions of technical debt and enrich it with the insights that software practitioners in industry have gained from their experiences working with technical debt. This study will provide a more comprehensive definition that also captures the motivations for incurring, the impact of accumulating and the practices followed for managing technical debt on a software project. Furthermore, learning from the successes and failures of software practitioners in dealing with technical debt will help the software development community improve and grow their strategies for managing technical debt.

1.2 Research Goals

The goal of this research is to identify the characteristics of technical debt and the strategies for managing it based on the experiences and lessons learned of software practitioners from industry. This study will answer the following questions:

- How do software practitioners define and characterize technical debt?
- What are the motivations for incurring technical debt on a software project?
- What are the strategies for managing technical debt on a software project?

1.3 Organization of this Thesis

This thesis is organized as follows:

- **Chapter 2** surveys and reviews the related work on technical debt;
- **Chapter 3** describes the methodology of the interview study, the primary study of this thesis. The interview study consists of a series of one-hour semi-structured interviews with software practitioners in industry to explore their experiences and lessons learned with technical debt;
- **Chapter 4** presents the findings of the interview study;
- **Chapter 5** describes the methodology and presents the findings of the game study, the secondary study of this thesis. The game study used the board game, “Hard Choices”, to validate the findings of the interview study;
- **Chapter 6** analyzes and discusses the findings of both studies to answer the research questions raised by this thesis;
- **Chapter 7** presents the conclusions and recommendations for future work.

1.4 Contributions of this Thesis

My contributions in this thesis:

- Define and characterize technical debt based on the insights, experiences and lessons learned of software practitioners from industry;
- Examine strategies for managing technical debt on a software project;
- Identify a set of guidelines to help software practitioners recognize and manage technical debt.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

The definition of technical debt has evolved and expanded since Cunningham first introduced the metaphor at OOPSLA '92. While interest in exploring the issues around technical debt in academic research is growing, there already exists a large number of on-going discussions and commentary on the subject in blogs. This chapter presents the definition of technical debt, explores its characteristics and identifies existing management practices based on a survey of the current literature.

2.1 Technical Debt Metaphor

Cunningham created the "technical debt metaphor" to help his boss understand why the software development team needed to re-factor code [2]. In his explanation, Cunningham compared writing immature code to going into financial debt. When a person goes into financial debt, he is borrowing money to allow him to do something sooner than would have been possible if he did not have the money. The borrowed money collects interest until the loan is repaid. If the borrower waits too long to pay back his loan, the interest accumulates to the point where the borrower's income is not sufficient enough to repay the loan and he must declare bankruptcy.

Similarly, a software development team goes into technical debt so that they can deliver their software to market quickly in order to gain experience and domain knowledge. The interest on the technical debt is the cost to update the code so that it reflects the development team's current understanding of the problem that they are trying to solve. The development team pays off the interest by revising and re-writing the code to consolidate their domain knowledge with the knowledge captured in the code base. As interest accrues on the technical debt, the progress of the development team slows down because "excess quantities will make the program unmasterable, leading to extreme specialization of programmers and finally an inflexible product" [1]. Eventually, the accumulation of interest over time leads to technical bankruptcy [3], when any income earned from adding new features is spent towards paying interest on the debt [4].

There are several instances where the technical debt metaphor breaks down. Firstly, unlike financial debt, technical debt is not quantifiable. Developers do not have a means of measuring the value of the principal or the amount of the interest so they do not have any indication of how much the owe and when their debt is completely repaid [4, 5]. Secondly, the interest rate is unknown and variable, usually higher if the debt is taken in frequently exercised parts of the code and lower, or even non-existent if the debt is incurred in rarely used parts of the code [5]. Moreover, unlike financial debt, technical debt ceases to exist when the software project is retired so development teams are less inclined to repay its debt as the project begins to near the end of its lifecycle [4]. Lastly, sometimes, development teams may not realize that they have taken on technical debt until they examine the software in hindsight [6], unlike financial debt, which is usually the product of an upfront, informed decision.

2.2 Types of Technical Debt

Cunningham did not elaborate on the technical debt metaphor beyond his OOPSLA experience report leaving its scope largely undefined. Steve McConnell and Martin Fowler independently developed technical debt taxonomies to classify the different types of technical debt. Both McConnell and Fowler divide technical debt into two primary categories: unintentional (inadvertent) and intentional (deliberate) debt. Other bloggers have divided technical debt into similar categories but with different names.

2.2.1 McConnell's Technical Debt Taxonomy

According to McConnell, technical debt is an obligation that a development team assumes when they compromise on the software system's quality by choosing quick, short-term designs and construction approaches that result in increased cost and complexity in the long-term [4]. McConnell created a technical debt taxonomy to capture the different types of technical debt:

- I. *Unintentional Debt*
- II. *Intentional Debt*
  - A. *Short-Term Debt*
    - 1. *Focused Short-Term Debt*
    - 2. *Unfocused Short-Term Debt*
  - B. *Long-Term Debt*

He divided technical debt into two categories: unintentional and intentional debt. Unintentional debt is non-strategic debt that a team assumes when they do a poor job, make impulsive and poorly thought out decisions, or unknowingly inherit debt from another source. Intentional debt is debt that the team takes on when they have made a strategic decision to optimize for the present instead of for the future. This type of debt is often motivated by the development team’s need to get the current release shipped in order to meet business objectives.

There are two types of intentional debt – short-term debt and long-term debt. Short-term debt is debt that is incurred at the last minute to allow the development team to deliver the release on schedule. The expectation for this type of debt is that it will be paid off quickly. On the other hand, long-term debt is debt that is deliberately taken by the team with the expectation that it will be carried around by the project for several years, much like a mortgage. For example, a development team incurs long-term intentional debt when they decide to support only one operating system because there is no expectation in the next five years to support other operating systems.

McConnell further broke down short-term debt into unfocused and focused debt. Unfocused short-term debt is comparable to credit card debt, comprised of shortcuts that are taken throughout the code. This makes unfocused short-term debt difficult to track and manage so it is easy for the development team to incur more debt than they realize. Consequently, McConnell stated that this type of debt should be avoided because the project does not benefit from incurring it. Focused short-term debt is easier to track and manage because the debt is incurred in large chunks throughout the code base, such as a hacked-together feature. McConnell compared this type of debt to a car loan.

Finally, McConnell classified certain types of incomplete work, such as backlogged, deferred or cut features as non-debt because they do not require interest payments.

2.2.2 Fowler's Technical Debt Quadrant

Fowler defined technical debt as the consequence of when a development team decides to take "quick and dirty" design approaches in order to take advantage of a market opportunity [7]. The outcome of such a decision is interest payments in the form of extra effort in the future to improve the design through refactoring. However, as Fowler noted, technical debt becomes problematic for development teams who let their debt grow so uncontrollably that the interest payments cripple the forward progress of the team. Consequently, before taking on technical debt, development teams need to evaluate whether the value of delivering early is worth the extra costs later on of paying down the principal and interest on their technical debt.

Fowler categorized the types of technical debt into a quadrant [6]:

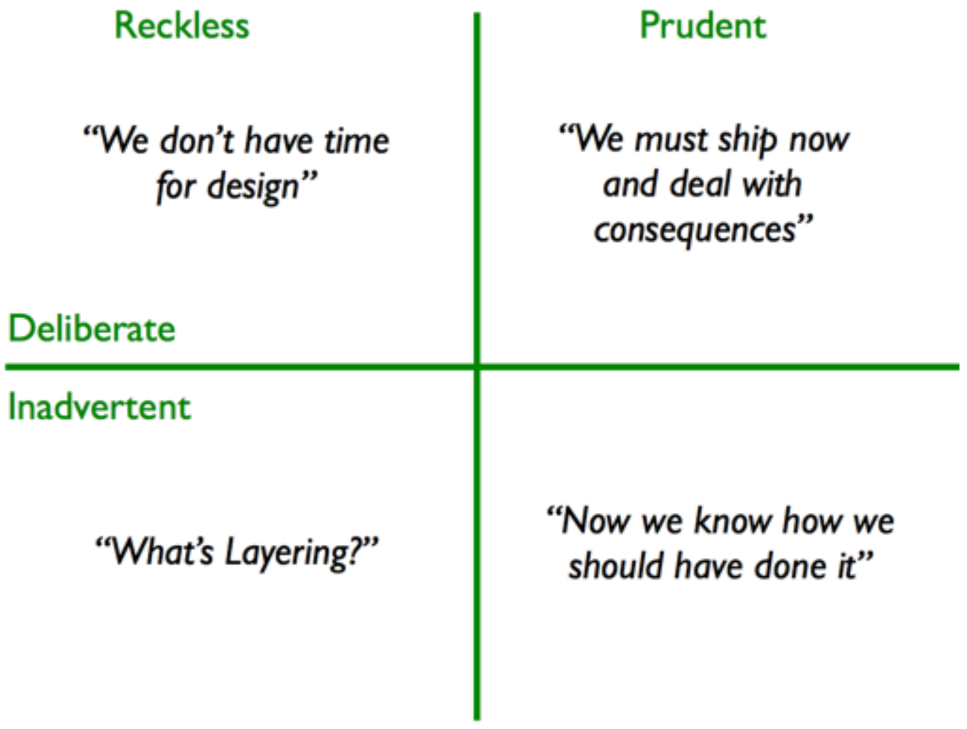

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 1: Technical Debt Quadrant [6]

He divided technical debt into reckless and prudent debt. A development team incurs reckless debt when they make a mess in the code, resulting in high interest payments. On the other hand, a development team incurs prudent debt when they choose to have design flaws in the code. Within each type of debt, Fowler distinguished between inadvertent and deliberate debt. Reckless-inadvertent debt occurs when the development team produces code without any awareness of good design practices, impacting the overall future productivity of the team. Conversely, a development team takes on reckless-deliberate debt when they decide to take a "quick and dirty" approach because they do not think they have the time to write the code cleanly. Prudent-deliberate debt is the outcome of a development team making informed decisions to incur technical debt after evaluating its benefits and consequences. Prudent-inadvertent debt usually occurs in hindsight when a development team realizes what the best design approach should be after gaining some experience and knowledge working with the code.

There are similarities between the classification of technical debt described in Fowler's technical debt quadrant and McConnell's technical debt taxonomy. For example, McConnell's definition of unintentional debt refers to Fowler's definition of reckless-inadvertent debt. Both types of debt are not beneficial to a software project and should be avoided. McConnell's intentional short-term debt can be mapped to Fowler's reckless-deliberate debt and McConnell's intentional long-term debt coincides with Fowler's prudent-deliberate debt. The only type of debt missing from McConnell's taxonomy, but which is addressed in Fowler's quadrant, is prudent-inadvertent debt. This type of debt is also the same type of debt that Cunningham referred to in his original description of the technical debt metaphor [6].

2.2.3 Other Technical Debt Taxonomies

Other bloggers have also divided technical debt into categories that are similar in characteristics to McConnell's unintentional and intentional debt and Fowler's reckless and deliberate debt. Churchville grouped technical debt into conscious and unconscious debt [8]. Conscious debt is debt that a development team takes with full awareness of its shortcomings. Unconscious debt is incurred due to "mistakes, short-sightedness, and yes, stupidity" [8]. On the other hand, Marick categorized technical debt into frivolous debt and debt-as-investment [9]. He defined frivolous debt as debt taken from ignorance resulting in unnecessary interest payments that could be avoided and debt-as-investment being debt that a team assumes because they are motivated by business pressures to meet market opportunities.

Theodoropoulos classified technical debt as strategic and non-strategic debt [10]. He defined strategic debt as "situations where tradeoffs are made in a proactive and measured way" [10] and non-strategic debt as debt created by "situations where gaps are created without stakeholder approval and the implications to strategic objectives aren't properly considered" [10]. Theodoropoulos also stated that strategic debt requires the appropriate stakeholders to be involved in the assessment and decision-making process and that a management strategy must be in place to ensure that the debt gets paid back within a reasonable time.

Cartwright characterized technical debt as transparent or opaque [11]. Transparent debt is debt that the development team knowingly incurs whereas opaque debt is debt that the development team is unaware of incurring. He also identified six types of technical debt in his blog:

- *Ignore* – avoid doing something that the development team knows is needed (e.g. leaving out requirements);
- *Get surprised* – caused by a change of direction usually because the customer has new requirements;
- *Disregard some data* – ignore signs that the project is starting to buckle under the strain of carrying so much technical debt;
- *Repeat a mistake* – repeat a previous mistake, which, unknowingly, creates technical debt for the project;
- *Guess* – applies to non-functional requirements; occurs when the development team makes the assumption that something will work without verifying that it actually does;
- *Phantom* – introduce technical debt during code refactoring; Cartwright labels this type of debt as "egotistical refactoring" [11].

2.3 Characteristics

Surveying the blogs revealed a number of characteristics attributed to technical debt. These characteristics recognize both the benefits and consequences of incurring technical debt on a software project, including its impact on business strategy, project risks, and software quality.

2.3.1 Shortcuts

Cunningham never intended for technical debt to be an excuse for writing poor code [2]:

> *A lot of bloggers at least have explained the debt metaphor and confused it, I think, with the idea that you could write code poorly with the intention of doing a good job later and thinking that that was the primary source of debt. I'm never in favor of writing code poorly, but I am in favor of writing code to reflect your current understanding of a problem even if that understanding is partial.*

Other bloggers, like "Uncle" Bob Martin, agree [12]:

> *A mess is not technical debt. A mess is just a mess. Technical debt are decisions made based on real project constraints. They are risky but they can be beneficial. The decision to make a mess is never rational, is always based on laziness and unprofessionalism, and has no chance of paying of (sic) in the future. A mess is always a loss.*

However, Cunningham's and "Uncle" Bob's opinions are not equally shared by all bloggers. A number of bloggers associated technical debt with "slapdash software architecture and hasty software development" [13], "quick and dirty choices" [14-16] and "design shortcuts and shortcomings you put into something" [17]. They defined technical debt as a measure of "the amount of time, money or effort it takes to work around, manage, and fix bad decision/implementation decisions" [18]. Thus, they generally regarded technical debt poorly and advised fellow software developers to avoid incurring it at all costs.

Additionally, some bloggers extended the technical debt metaphor further to introduce the concept of technical bankruptcy. Technical bankruptcy occurs when the "levels of technical debt become so overbearing that the project is effectively doomed" [5]. In this situation, the development team has allowed the amount of technical debt on their project to accumulate to the point where most of the effort is spent on addressing their debt rather than adding new features. As a result, many developers are reluctant to work on such technical debt-ridden code. Moreover, in some instances, development teams throw out the code base and re-write it completely from scratch.

2.3.2 Business Strategy

Conversely, many bloggers viewed technical debt as "…the engineering trade-offs that software developers and business stakeholders must often make in order to meet schedule and customer expectations. In short, you may need to use suboptimal designs in the short term, because the schedule does not allow longer term designs to be used" [12]. These bloggers saw technical debt as a contribution to an organization's business strategy. They argued that by taking shortcuts to deliver the software faster, development teams are able to gain a competitive advantage in the market, break new ground or develop a prototype or proof-of-concept to collect customer feedback [19]. Furthermore, they pointed out that using technical debt as a business strategy is particularly beneficial to start-ups whose primary objective is to find customers and learn what they want in order to build a business [15, 20]. It allows a development team to be responsive to customer needs while avoiding investing hours of effort into developing a good architecture without knowing what the customer wants. For example, a development team may choose to implement a sub-optimal design, to have limitations in the functionality, or to take a shortcut and use third-party or open source software over custom-written code in their product because they can leverage the existing software in order to deliver more features with less effort. As the development team figures out its customers' needs, it can eliminate technical debt incurred in unwanted features that are thrown out and avoid repaying debt incurred in rarely used parts of the code. Eventually, the only debt worthwhile for the development team to pay back is technical debt incurred in features that are valuable to the customer [15].

2.3.3 Measure of Quality

Gat defined technical debt as "essentially a quantifiable measure of how good the quality of your code is" [21]. Often, as Gat stated, day-to-day business pressures force development teams to incur technical debt by taking shortcuts; however, "every time a decision is made, the overall quality, maintainability, complexity and stability of a software system goes up or down" [22]. If a development team is not actively paying back its debt, the technical debt accumulates exponentially, gradually leading to the decay of the software [23].

According to Gat, the amount of technical debt in a software project is measured as the total dollar value (cost and effort) required to restore the quality of the code [21, 23]. Highsmith illustrated the relationship between the cost of technical debt and code quality using the technical debt curve [24], as shown in Figure 2:

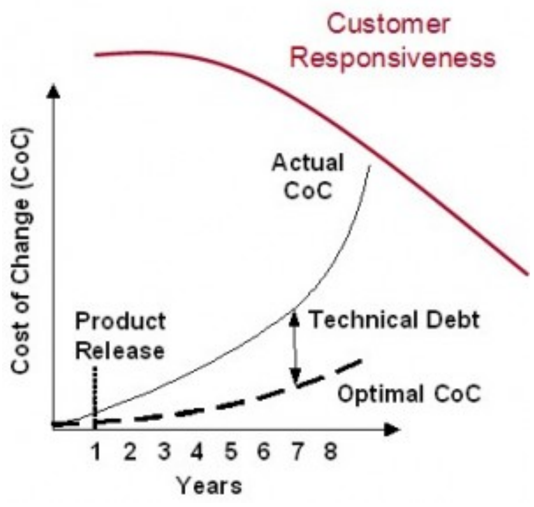


Figure 2: Technical Debt Curve [24]

Highsmith pointed out that the time axis of the technical debt curve can be viewed as both a long-term (software lifetime) and short-term (iteration) representation of the impact of technical debt on code quality over time. For both time periods, the impact of reducing code quality by accruing technical debt is an exponential increase in the cost of making changes to the code and an exponential decrease in the velocity of the development team. Thus, the development team needs to find a "threshold of acceptable pain" [25] that defines how much technical debt they can willingly live with and still be productive yet disciplined in repaying their debt.

2.3.4 Lack of Visibility

The difficulty with technical debt, as a number of blogs pointed out, is that developers are the only stakeholders who really care about code quality because they must work with the code on a daily basis. Thus, "complaints of how quality issues hinder development progress are not taken seriously enough until it presents visible business challenges" [26]. To bring visibility to the existence of technical debt, Gat developed the "Technical Debt Assessment and Valuation", which examines architecture, design, code, testing and documentation deficits using static and dynamic analytics to illustrate the cost of technical debt and the quality of the code on a software project [27]. Likewise, Barton and Sterling suggested using open source and commercial tools (such as Sonar's Technical Debt plug-in) to build a "software debt dashboard" to monitor the amount of technical debt in the software [28]. The indicators on the dashboard show "information about code complexity, percentage of duplication in the code, automated test code coverage and potential defects (including security issues). It may even provide a rollup of code metrics, giving a numerical cost for cleaning up technical debt based on existing indicators" [28].

2.3.5 Risk

Some of the literature also recognized that there is uncertainty associated with technical debt because it is difficult to assess its impact on future development in advance. As Ries pointed out: “the biggest problem with technical debt is not its impact on value or earnings, but its impact on predictability…as technical debt increases, development speed decreases, customer satisfaction decreases and predictability of results decreases” [24]. Therefore, technical debt “can be considered as a particular type of risk in software maintenance and the problem of managing technical debt boils down to managing risk and making informed decisions on what tasks can be delayed and when they need to be paid back” [29]. A number of blogs described following risk management practices documented in standards like ISO-31000 to manage their technical debt [30-34].

2.4 Management

Managing technical debt is a subject of much discussion amongst both the software development and the academic community. The focus of this area of research in technical debt is divided into three topics: identification, measurement and payback strategies.

2.4.1 Identification

The most commonly suggested approach to identifying a project’s technical debt was to compile a list by asking development team members to identify problematic and painful areas through team meetings, brainstorming sessions and water cooler discussions [30, 31, 33, 35]. Other ways included examining design documents, conducting code reviews, inspecting team operations (e.g., how up-to-date is the development platform? Are there any tasks that should be automated? Is there any system health monitoring?) [32], and using tools to gather software metrics, such as test coverage and areas in the code base with a high number of changes [31]. Many bloggers tracked their technical debt by using Wiki pages, white boards or code comments [30] so that the debt is visible and accessible to everyone on the team. Kruchten suggested tracking the technical debt by creating bug reports in the project's bug tracking system and differentiating it from bugs and feature requests through the use of color [36].

2.4.2 Measurement

Much of the existing literature commonly described techniques for measuring technical debt in terms of cost because it is easily relatable to the financial undertones of the metaphor. For instance, a CAST report described measuring technical debt in terms of the architectural and implementation quality of the code [37]. Using CAST proprietary software, a source code analysis is performed to determine the technical debt density, or, violations per thousand lines of code (KLOC). The violations are classified as high, medium and low severity and the amount of technical debt is calculated as the cost (in dollars) of addressing 50% of the high severity, 25% of the medium severity and 10% of the low severity violations. In the report, CAST claimed that the cost of technical debt in a typical application of 374 KLOC is more than $1 million.

Chin et al. suggested calculating the total amount of technical debt (in dollars) as the sum of the principal, recurring interest and compounding interest [22]. The principal is the cost, measured in dollars, to fully service the debt; its value is determined using empirical (e.g. unit test coverage, code duplication and static code analysis) and qualitative metrics. Recurring interest is specific to the organization and is determined at the product or function level. It is the measure of the maintenance and support costs as a result of having technical debt in the software. Lastly, compounding interest is the multiplicative cost that is incurred when additional technical debt is incurred in areas where it already exists; in other words, it is the growth rate of technical debt over time.

Nugroho et al. developed an approach to measuring technical debt that is based on estimating the amount of re-work (in man-months) needed to improve the software to a higher level of quality [38]. Each level of quality of the software is a snapshot taken using the SIG/TÜiT's software quality assessment method, a layered model that uses the quality characteristics defined in ISO/IEC 9126 to measure and rate the technical quality of a software system. Nugroho et al. measured the amount of interest using estimates of the amount of extra effort (in man-months) that must be spent to maintain the software because the development team chose not to improve its level of quality. By finding the difference in maintenance cost between two levels of quality, the development team can also calculate the savings and return on investment (ROI) of improving the software quality.

Guo and Seaman proposed an approach that uses portfolio management theory [29]. Portfolio management theory is a financial risk reduction strategy that determines when the assets in a portfolio should be invested or divested to maximize the return on investment and minimize the risk of the portfolio. In their approach, portfolio management theory is used to create a portfolio of technical debt items that should be paid back in the next release by selecting the items that are most likely to produce the greatest net benefit to the project. A technical debt item is measured (in person-days) in terms of its principal (effort required to complete the task), interest (extra work needed if the debt is not repaid), interest standard deviation (measure of uncertainty of whether or not the extra work is needed to address the debt) and correlation to other technical debt items.

2.4.3 Payback Strategies

The blogs also suggested a number of strategies to reduce the amount of debt. One strategy is by slowing down the development pace [30] and limiting the amount of work in progress to concentrate on outputting quality and not quantity [34]. Another strategy is to allocate time during the iteration to repay the debt. For example, a development team can allocate a percentage of time and effort during the iteration to address their technical debt without compromising the number of new features or value added for the iteration [34]. They can also build some slack into the schedule and estimates for new features so that they can pay back the technical debt that exists in the same area of code where the new feature will be added [30]. A more visible strategy is to involve the customers and product owners by communicating to them the value of paying down the project's technical debt and asking them to prioritize it alongside new feature requests [31, 36]. Hilton [30] disagreed with this approach because he felt that technical debt should remain invisible to non-technical stakeholders since it is the development team's responsibility to manage the code. However, Kruchten [36] agreed with the strategy because he strongly advocates that technical debt should be made visible to all of the project stakeholders.

2.5 Concepts Related to Technical Debt

The ideas behind the technical debt metaphor are not new concepts to the software engineering community. These concepts have been studied, characterized and examined since 1968, when Lehman began his study of software programming processes for IBM, which eventually led to the development of Lehman's Laws of Software Evolution [39, 40]. Over the years, the concepts behind technical debt have been recognized by other metaphors including software erosion, rot, decay, drift and aging.

Lehman developed the laws of software evolution based on his analysis of metrics taken from across twenty-six releases and sub-releases of IBM's OS/360 over a twenty-year span [39, 40]. The premises behind the technical debt metaphor are addressed by several of Lehman's laws. Law I (Continuing Change) described the fact that software systems constantly evolve as a result of a mismatch between the current software functionality and the operational domain. Law II (Increasing Complexity) addressed the idea that successive changes to a software system leads to an unstructured increase in interactions and dependencies, which makes it more difficult to maintain the system unless the complexity is constrained. Law V (Conservation of Familiarity) stated that the development team's progress slows down as more changes are made to the software system because developers need more time to understand how these changes affect future development work. Finally, Law VII (Declining Quality) addressed the fact that a software system's quality declines unless the development team is active in identifying and fixing areas of the code that were written under assumptions, perceptions or understandings which are no longer valid or justified.

Other researchers have examined technical debt under different terminology. Perry and Wolf used the terms architectural drift and architectural erosion [41]. They defined architectural drift occurring when developers make changes to the software without awareness of the existing architecture and architectural erosion occurring when the software architecture is violated. van Gurp and Bosch studied the causes and characteristics of software erosion [42]. They showed that software erosion is inevitable because software is developed iteratively in order to support constantly changing requirements. As a result, previously valid design decisions may no longer contribute towards an optimal design for the current version of the system. Eick et al. observed from their experiences working on a large real-time telecommunications software system that code decays over time, making maintenance more difficult and expensive [43]. They defined code decay as code that is more difficult to make changes to than is necessary, in terms of cost (changes are larger and more expensive), quality (changes are more likely to introduce faults and reduce modularity) and duration (changes take longer to make). Finally, Parnas introduced the phenomenon of software aging, which he described as occurring when either the software is not modified to meet the changing needs of the operational domain or when developers make changes to the software without understanding the original design [44].

2.6 Klinger's Interview Study

Klinger et al. conducted an interview study to examine how enterprise organizations view, evaluate and leverage technical debt [45]. The goal of their study was to test the premise that technical debt is analogous to financial leverage by enabling enterprises to pursue options that they would otherwise not have available. Klinger et al. interviewed four technical architects on different projects at IBM to explore the motivations behind the projects' decisions to incur technical debt and the resulting benefits and consequences.

Their study revealed that projects often make decisions that benefit themselves, but do not necessarily benefit other stakeholders in the enterprise. They also found that projects rarely quantify their technical debt and that quantifying technical debt is difficult. Lastly, Klinger et al. discovered that organizational "gaps" between business, operational and technical stakeholders often contribute to incurring technical debt.

While our interview study is similar to the interview study conducted by Klinger et al., there are some differences between our approaches and findings. For example, Klinger et al. focused on studying an enterprise organization's motivation for incurring technical debt; however, this study is focused on a finer level of understanding why software practitioners incur technical debt. Additionally, our interview study uses a larger and broader sample set of nineteen software practitioners from numerous organizations of varying sizes and roles whereas Klinger et al. limited their study to four technical architects at IBM. Moreover, our interview study results were validated through a replication study by Nitin Taksande at the University of Maryland, Baltimore County [46].

Our research also aims to define and characterize technical debt through a scientific process that involves sampling from a data set and extracting patterns and anti-patterns. This improves on the existing literature because many of the surveyed blogs are based solely on the opinions and experiences of the author. Thus, the findings of our research will help to validate the definitions and characterizations described in blogs.

2.7 Conclusion

Cunningham introduced the concept of technical debt to provide a tangible handle on an issue that is only visible to the development team. Taking on technical debt allows a project team to speed up development in order to achieve a greater gain, such as market share and customer feedback; however, its consequence is that future development will be slower because of the increased complexity in the code. Many bloggers in the software development community, including McConnell and Fowler, have classified technical debt into two categories: intentional and unintentional debt. Intentional debt is debt that is taken strategically whereas unintentional debt is debt that is taken carelessly. A survey of software development blogs reveals that the characteristics of technical debt have both positive and negative impact on a project's business strategy, project risks and software quality. Furthermore, many bloggers agree that managing technical debt is similar to managing project risks. Although the concepts of technical debt are not novel – they have been studied in software engineering as one of the aspects of software evolution as well as under different terminology, such as software erosion, rot, decay, drift and aging – the technical debt metaphor provides technical stakeholders with a more meaningful tool to communicate with non-technical stakeholders because the metaphor relates the issue to a concept that everybody understands.

CHAPTER 3
INTERVIEW STUDY METHODOLOGY

The purpose of the interview study is to investigate how software practitioners in industry characterize and manage technical debt. The outcome of this study was achieved through the use of qualitative research. Qualitative research aims to answer questions that uncover "why", "how" or "in what way" things are the way they are, which are not easily measurable [47]. The research is focused on learning from the experiences, feelings and opinions of its interview subjects. Information is gathered through direct interaction with the subjects through interviews, focus groups, case studies and observations.

For the interview study, I conducted a series of one-hour semi-structured interviews to explore the insights, experiences and lessons learned with technical debt of the software practitioners who participated in this study. The data collected from the interviews were analyzed using content analysis and organized into key categories [48]. The goal of this interview study is to identify how practitioners define, characterize and manage technical debt and what the motivations are for incurring technical debt. This chapter describes the steps that were taken to collect and analyze the information for this study.

3.1 Participants

In total, nineteen software practitioners were interviewed for this study between November 2009 and April 2010. The interview participants were selected from my personal contacts and co-workers as well as from contacts provided by my supervisor, Philippe Kruchten.

Information about the interview participants is detailed in Table 1. Each interview participant is assigned a code number to maintain his or her confidentiality.

Subject	Role	Industry	Organization Size	Years Experience
S01	Software Developer	IT Consultant	1	6
S02	Principal Engineer	Imaging and Printing	30,000	28
S03	Software Developer	Financial Services	300	13
S04	Software Developer	Business Intelligence	53,000+	7
S05	Software Architect	Commodity Trading	40	19
S06	Development Team Lead	3D Simulation	100	10
S07	Development Team Lead	Imaging and Printing	30,000	16
S08	Project Manager	Imaging and Printing	30,000	22
S09	Business Architect	e-Health	2000	33
S10	Development Team Lead	Imaging and Printing	30,000	18
S11	Project Engineer	Imaging and Printing	30,000	27
S12	Software Developer	Imaging and Printing	30,000	24
S13	Software Developer	Imaging and Printing	30,000	22
S14	Development Team Lead	Imaging and Printing	30,000	24
S15	Software Architect	Information Solutions	3,300	>20
S16	Project Manager	Information Solutions	3,300	>20
S17	VP Development	Hydrology	25	19
S18	Software Architect	Financial and Tax Preparation	8000	16
S19	Software Architect	Financial and Tax Preparation	8000	15

Table 1: Interview Subjects' Background Information

Interview participants were selected based on the criteria that they needed to have more than three years of work experience and that they needed to be currently working in the software or information technology (IT) industry. The selection criteria were designed to exclude Co-op students, junior developers, and non-practicing software engineers because they would not have current, real-world, hands-on experience to provide examples of working with technical debt on a software project while answering the interview questions. The interview participants selected for this study consisted of three women and sixteen men who have between five and thirty-five years of experience. The distribution of roles amongst the group included three project managers, seven software architects and nine software developers. With the exception of S01, who is an IT consultant, the remaining interview participants work in product development, eleven of whom are building web-based systems. Furthermore, the organization size of their employers includes five small companies (less than 1000 employees), three medium companies (1000 – 10,000 employees) and two large companies (more than 10,000 employees).

3.2 Data Collection

Data was collected for this study through interviews. Each interview was forty to sixty minutes long, conducted in person, on Skype or over the telephone. Most of the participants were interviewed individually, with the exception of S07 and S08, who were interviewed together, as were S12, S13 and S14.

The initial interviews with S01 and S02 were unstructured and designed to explore the general perception of software practitioners on the topic of technical debt. The goal of these interviews was to get a sense of how familiar the concept of technical debt was amongst software practitioners and to identify areas that could be pursued in future interviews. During their interviews, S01 and S02 shared their opinions on and personal experiences with, identified qualities, attributes and issues of and management strategies for technical debt on a project.

The information gathered at the initial interviews with S01 and S02 led to the development of a set of questions for future interviews on the subject. The questions asked the interview participants to focus on a specific software project that they have worked on, which incurred technical debt, as they answered the following questions:

- Why did the project take on technical debt?
- What factors contributed to the decision to take on technical debt on the project?
- What was the impact of the technical debt on the project?
- How was the technical debt managed on the project?
- What lessons were learnt from taking on the technical debt?

These questions became part of the interview guide (included in Appendix B) that was used for subsequent interviews. The interview guide contained a series of questions that was divided into two parts. The first part of the guide asked questions in a structured format to provide some context or background of the interview participant. These questions included

- How many years of experience do you have?
- What is your role on the project?
- What are your responsibilities?
- How many years have you worked on this project?

The second part of the interview guide focused on technical debt and included a series of open-ended questions in a semi-structured format, as described earlier. The design of these questions was to allow the flow of the interview to be guided by the responses of the interview participant, with the opportunity to explore some areas in more detail.

A number of the interview participants wanted to know the definition of "technical debt" before proceeding with the interview so they could have some context. Thus, if the interview participant requested it, the interview began with a brief general description of technical debt; the interview guide was later updated to include the technical debt description. As the interviews moved onto the topic of technical debt, some participants drew from a few specific examples to answer questions while others provided information by extracting insight from their past experiences across several projects.

The interview participants were invited by e-mail to participate in the study, which included a brief description of the technical debt concept, the goals of this study, the intent of the interview and a consent form. When the interview participant agreed to be interviewed and returned the signed consent form, I arranged a suitable date, time and location to conduct the interview. Many of the in-person interviews were conducted at the offices of the interview participants and a few were conducted at nearby coffee shops. Before interviewing S18 and S19, I was required to sign their company's non-disclosure agreement.

During the interviews I took notes while digitally recording the conversation, which I transcribed afterwards. The interviews with S01 and S02 were not recorded. The transcriptions were stored on a password-protected computer and sensitive information was removed. At the end of each interview, I added my impressions, in a different colored pen, to the notes that I took during the interview.

3.3 Data Analysis

A set of preset codes was determined before the interviews were started. These codes were defined as:

- *Attributes* – characteristics used to recognize or identify technical debt;
- *Causes* – reasons or motivations for incurring technical debt;
- *Symptoms* – impact or consequences of having technical debt;
- *Management* – techniques used to track and manage technical debt;

Following each interview, the data was categorized according to the preset codes. Then, the transcript was analyzed again using open coding to identify emergent codes. Any interesting patterns that arose from the open coding were then incorporated into the questions for the next interview. When all of the interviews were completed, axial coding was applied to develop categories from the emergent codes. During axial coding, the role and number of years of experience of the interview participants and the organization size of their company were also considered. The list of codes for this study is shown in Table 2:

Preset Codes	Emergent Codes	Emergent Sub-Codes
Definition	Trade-offs	
	Shortcuts	
	Unavoidable	
Attributes	Risks vs. Defects	
	Poor visibility	
	Difficult to measure	
Causes	Business reality	Time pressures
		Changing priorities
		Unclear objectives
	Inexperience	
	Attitudes	
	Software age	
Symptoms	Poor quality	
	Uncertainty	
	Deliver to market quickly	
Management	Risks	
	Expectations	
	Communication	

Table 2: Codes from Analyzing Interview Data

The "Definition" codes consist of words that were frequently repeated when the interview participants were asked to define technical debt. Some of the participants also used the word "necessary" to describe technical debt, but the meaning of "necessary" is close enough to *unavoidable* that I combined the codes together. During the analysis of the interview data, I found that participants saw technical debt as either a project risk or a defect, depending on his role (management or technical) in the project. Thus, I created the code *risk vs. defects*. I extracted the "Causes" codes by comparing and contrasting examples of why interview participants incurred technical debt. In particular, under the *business reality* code, I created the sub-codes *time pressures* (e.g. schedule constraints, deadlines), *changing priorities* (e.g. new requirements, new market direction), and *unclear objectives* (e.g. vague requirements, lack of product vision) to represent the different dimensions of how business decisions impact a project team's decision to incur technical debt. The "Symptoms" codes capture common themes of how technical debt impacted the interview participant's software project, which, generally, was related to not satisfying non-functional requirements. Finally, the "Management" codes, *risks, expectations* and *communication*, were developed by identifying patterns from the interview participants' responses on how they managed their technical debt; interestingly, the trend that emerged demonstrated that the management of technical debt often focused on managing the external stakeholder interfaces.

3.4 Validity

In their conference report, Knutson et al. [49] acknowledged that the results of empirical research in software engineering are often challenged on its validity because of

1. *A lack of independent validation of empirical results;*
2. *Contextual shifts in Software Engineering practices or environments since the time of the original research studies; and*
3. *Limited data sets at the time of the original research studies.*

One approach to validating empirical results in software engineering is through replication studies. A replication study is a study that is repeated following the objectives and design of the original study [50]. The aim of a replication study is to verify or expand on the results of the original study. A successful replication study, as described by Mantyla et al. [51], focuses on widely available software engineering data, such as code measures (defects, effort, size).

This study was repeated in a replication study conducted by Nitin Taksande [46] at the University of Maryland, Baltimore County. Taksande also sought answers to the same research questions that are posed in Chapter 1 of this thesis. Like this interview study, Taksande chose interview participants from his personal contacts, as well as from the contacts of his supervisor, Carolyn Seaman, using the same selection criteria as this study. General background information about the interview participants in Taksande's study are shown in Table 3:

Characteristic	Categorization	# of Interview Participants
Gender	Men	16
	Women	0
Years Experience	3 - 35	16
Roles	Managers	4
	Developers	8
	Quality Assurance	4
Industry	IT Services	2
	Web-based Development	2
	Financial/Banking Services	3
	Telecommunications	3
	Network Service	1
	Health Insurance	1
	Aeronautical Research	1
	Data Management	1
	Document and Imaging Services	1
Organization size	Small (< 1000 employees)	3
	Medium (1000-10,000 employees)	2
	Large (> 10,000 employees)	11

Table 3: Background Information of Interview Participants in Taksande's Replication Study

The interview participants were invited to participate in Taksande's study by e-mail. Taksande used the interview guide from this study to conduct the interviews, of which thirteen were by telephone, two were in person and one was a written response. The telephone and in person interviews were audio recorded and transcribed.

In his data analysis, Taksande also applied the preset codes from this study and used constant comparison and cross-case analysis to identify emergent sub-codes. Throughout his data analysis, Taksande gave consideration to characteristics of the interview participants, such as the number of years experience and their role on the project to examine if these characteristics affected the way the interview participants responded to the interview questions. A comparison of the results from Taksande's replication study and from this study will be discussed in Chapter 6 of this thesis. In addition, a paper describing the results from both interview studies has recently been accepted for publication in an upcoming special issue of *IEEE Software* on technical debt.

3.5 Conclusion

The goal of this interview study is to investigate how technical debt is defined in industry. This chapter describes how these objectives are met. Using qualitative research techniques, nineteen software practitioners were interviewed and asked to describe their experiences, opinions, and lessons learned on dealing with technical debt in their software projects. The outcomes of these interviews were analyzed using content analysis to identify a set of emergent categories that describe how software practitioners define, characterize and manage technical debt. The results of this study are validated by a replication study conducted by Taksande, who followed the same design model as this study; the results from both studies will be published in an upcoming special issue of *IEEE Software* on technical debt.

CHAPTER 4
INTERVIEW STUDY FINDINGS

This chapter summarizes the findings of the interviews conducted for this study following the methodology described in Chapter 3. The findings are organized by the preset codes identified in the previous chapter: definition, attributes, causes, symptoms and management.

4.1 Definition

Many of the practitioners interviewed for this study gave definitions of technical debt that closely matched Cunningham's definition. Their definitions frequently acknowledged that technical debt is essentially a balance between software quality and business reality. Examples of definitions included

- S10: *"...what shortcuts could you take that maybe gets you out to the markets faster versus, you know, that might cause you long term pain."*
- S04: *"...you can't get everything you want done and therefore, you only do the things that are not necessarily the most important but the things that will give you the most payoff."*
- S18: *"...some shortcut that has been taken or some less than desirable implementation has been done or we've de-scoped something for one reason or another and so we've acquired some amount of the work that can or should be done at some point in the future. The reason why I believe it comes into being is simple, you know, business situations. We don't have an unlimited number of resources, we don't have an unlimited amount of time, and so, the business reality forces us to make choices at points in time to be able to get the broader outcomes of, you know, delivering a solution."*

In each of the examples, the responses suggest that the decision to incur technical debt was strongly influenced by the project schedule and business priorities. These examples also highlight the fact that the words “shortcut” and “trade-off” are two words that were commonly used to describe technical debt.

There were some definitions of technical debt that were broader than Cunningham’s original definition. S19 described technical debt as a form of short-term thinking or decision-making. Similarly, S03 defined technical debt as “Trade-offs, which is, what is ideal versus what is practical or what is achievable, I guess is a better word, given the constraints.” Both definitions imply that technical debt includes hacks, workarounds, shortcuts or band-aid solutions. However, Cunningham did not consider messy code as technical debt; he saw technical debt as a reflection of the project’s current understanding of the problem. Cunningham’s definition is also narrower in scope because it describes technical debt as a short-term, temporary solution to help a project achieve its long-term goals.

When the interview participants were explained that the technical debt metaphor represented the consequences of making the trade-off between implementing a fully architected, sustainable solution and taking shortcuts in order to deliver a solution quickly to the customer, all of them recognized and understood concept immediately. As S13 succinctly put it, “Familiar with it? We live with it every day!” Most of the interview participants, like S06, acknowledged that incurring technical debt is unavoidable on a project: “…I think you’re always going to incur some level of debt because you’re never going to release a product that doesn’t have issues; I think that’s an impossibility.”

Generally, the interview participants agreed with S19 that technical debt should not be avoided on a project, but managed.

4.2 Attributes

The results of the interviews revealed a number of characteristics that practitioners attribute to technical debt. These characteristics include differing perspectives between project managers and developers on incurring technical debt, being less visible to external stakeholders than to internal stakeholders and being difficult to measure.

4.2.1 Risks vs. Defects

The interviews revealed that developers and project managers had different perspectives on technical debt. Developers were more likely to view technical debt as bugs, defects, issues and deficiencies in the code whereas project managers saw technical debt as project risks. As a result, project managers were more willing to accept incurring technical debt. On the other hand, developers generally preferred not incurring any debt in the code at all. In fact, there were a few, like S18's co-worker, who used technical debt "…usually in a very negative connotation that, you know, we're short-changing stuff yet again and that we're, you know, not doing the best that we can." S19 pointed out that the differing goals between developers and project managers contributed towards the tension around incurring technical debt on a software project:

> *Engineers don't like it because they want to create perfect software. As a technical leader, I'm probably about 60% technical and 40% business. I also recognize you have to get things to market. You have to hit your windows, right? You've got to look, who is my target customer, right? You've got to factor all these things.*

S06 clarified this point by explaining that developers want perfect code because they need to work with the code base on a day-to-day basis; thus, they are more closely aware of the potential issues that may arise in the future from having unaddressed technical debt in the code base.

4.2.2 Poor Visibility

The interviews also highlighted the fact that technical debt is not as visible to external stakeholders, such as customers and management, as it is to developers who must work with the code daily. One of the reasons, S15 pointed out, is, typically, software projects are more likely to acquire technical debt in its infrastructure than its user interface

> *…because it's so in-your-face it never suffered because it always had to work, right, because that's what the customer sees. The stuff underneath is where it breaks, right?...all of that infrastructural stuff we would pay in the business layer…that's probably where, because you could bury it there, right? Because you could do it bad and no one would be none the wiser. As long as the output was right then it didn't matter, right?*

Consequently, as S08 and S12 both pointed out, management often does not recognize the value in addressing a software project's technical debt unless doing so either results in a tangible reward for management or is paid for by the customer. S15 also observed that customers are not willing to provide time or money for the development team to repay its technical debt unless they can derive business value from it.

4.2.3 Difficult to Measure

Many of the technical leaders (team leaders, software architects and project engineers) that were interviewed wanted a way to measure the amount of technical debt that existed on their software project. However, they recognized that measuring technical debt was not easy because the impact of all technical debt is not equal. As S15 noted,

> *You can't measure how much extra work you're causing by doing what you're doing…all we could do is to do our best to make sure that design-wise, we did the best we could with what we had and not break the process.*

S18 questioned:

> *"…at some point, if it [technical debt] works and it's meeting the needs and there's relatively little or no impact to customers or productivity or maintainability or any of the –ilities or anything that's important to us as a business, then, is there value in doing it [paying back the debt]?"*

S11 felt that measuring technical debt was difficult because its impact seemed to grow exponentially over time as the amount of technical debt on his project increased. To illustrate his point, S11 used the metaphor of placing a frog in cold water. In the metaphor, when a frog is placed in a pot of hot water, it will immediately jump out of the pot. However, if a frog is placed in a pot of cold water and the pot is slowly heated up, the frog, a cold-blooded creature, will die from over-heating because it does not recognize the gradual increase in temperature until it is too late. For S11, a software project is the frog that is placed into a boiling pot of technical debt hot water:

> *And it's kind of like technical debt, you creep into it. Like, it just gets gradually worse and worse; it's only gradually worse than it was the previous day. So it's always easy to say, well, I'm going to put this off until tomorrow or until next release or whatever because you can always do that. But, at some point, you get overwhelmed by it.*

Without having a means to measure technical debt accurately, many of the interview participants found that they were left to guess the amount of debt they were carrying forward with them to the next release. Furthermore, not knowing made it difficult for project teams to estimate the amount of effort required to make future changes to the system and to predict the likelihood that customers would encounter effects of having technical debt present in the system.

4.3 Causes

One of the focuses of the interviews was to understand the reasons why participants decided to incur technical debt on their projects. Business reality, attitudes, inexperience and aging were the most frequently mentioned reasons for projects to incur technical debt.

4.3.1 Business Reality

All of the interview participants identified business reality as the primary motivation behind their decision to take on technical debt. As S18 explained, "…the business reality forces us to make choices at points in time to be able to get to the broader outcomes of, you know, delivering a solution." S19 added, "I don't think you should try to avoid it [technical debt] because I think it's necessary. You know, and because that's just business reality." The business realities for software projects include schedule constraints, unclear requirements and changing priorities.

4.3.1.1 Time Pressures

Many interview participants acknowledged that they were motivated to incur technical debt because of time pressures. For example, S04 remarked that if his project did not take on technical debt, it ran the risk of not being able to deliver at all. Likewise, S15 reflected that

> *"…some decisions are made…to say, well, we could do that better or we could do it the short way. And the short way won because of time pressures. I wouldn't say that there was a conscious effort to sacrifice integrity or the structure of the software or the maintainability of the software over get to market. I don't think it was a real conscious decision…it was more of the dev*

team saying, ok, well, if you want to meet those things we're going to have to do it this way and that's all because the timelines weren't moving."

The interview participants provided a number of examples to illustrate how time pressures influenced their team's decision to take on technical debt. S17's project took on technical debt because they had committed to selling their system to the customer before it was built; consequently they were contractually obligated to release under a tight deadline. S04's project accrued technical debt because it needed to deliver on time in order to integrate with another product. S06's project decided to incur technical debt because they needed to deliver in time for an upcoming trade show that marketing felt was a "massive opportunity" for the company. S18's and S19's project teams used technical debt to help them deliver their software by the first week of November to coincide with the two-week period when their customers were in the market to shop for new software. S19 said that if his project team missed this two-week window, they might as well not launch the product at all because they would have missed their target market completely. Lastly, technical debt allowed S10's project to develop a working prototype of the software in order to secure funding from investors.

4.3.1.2 Changing Priorities

S19 commented that his project team sometimes incurred technical debt in reaction to an unexpected situation or to receiving new market information. He remarked that even design decisions that were made with the best information available could quickly become technical debt when the project team learned of and responded to new information. S06 described it as

> *You have sort of this scenario where over time, your priorities change and you need, for whatever reason, you get new information from the market or whatever, so some new information entered the system and you decide that you need other changes to what you thought you needed or you need something new.*

S07 commented:

> *But it's also, you make a decision, you do it the best you can, right? You incurred this debt and it's look how the market goes, right? Like these things change in a month and then your debt isn't as bad as it could have been. It's these external factors.*

S08 provided an example to illustrate this scenario. In her example, S08 described how her project invested in developing a desktop application for their product. However, the market changed direction away from desktop applications to web-based applications because IT departments did not want users installing software onto their computers for security reasons. Consequently, "…the differentiating features [of the desktop application]…it sounded like the winning formula for this product at the beginning, turned out to be wrong." Fowler described this type of technical debt as prudent-inadvertent debt; McConnell did not include this type of debt in his taxonomy.

4.3.1.3 Unclear Objectives

A number of interview participants attributed incurring technical debt to having both internal and external stakeholders who did not fully understand the objectives or deliverables of the project. S17 reflected that one of the reasons his project incurred technical debt was because he had developers on his team who did not understand how to use the software from the customer's point-of-view. Consequently, it was difficult for him to rely on his team members to contribute to the decision making process because they were unable to appreciate how their design decisions could impact their customers. S06 made a similar observation and pointed out that technical debt accumulated because "…so much stuff was just thrown in without any oversight by people who didn't really know what they were doing." Part of the problem, as S06 pointed out, and S10 agreed, was that some developers did not understand the history of the project – how the software evolved, what has been built, and what the pressures were at the time the decisions were made.

From his experiences, S11 believed the lack of product vision contributed to incurring technical debt. He felt that "it's really hard to get perfect information about what people think they want today, right, even if it might be in someone's head, it's really hard to translate that into a system and so that's hard, but it's impossible to know where you're going to want to be in five years". As a result, S11 believed the reasons his projects incurred technical debt were because the project team did not have a clear product roadmap and because the team lacked an organization structure that designated someone to be responsible for overseeing the development of the project.

Additionally, several interview participants said they incurred technical debt because customers often did not know what they needed the system to do. For example, S17's project team was contractually obligated to build a customized solution for a customer. However, the customer was "incredibly unclear" about what they wanted the system to do and kept changing their minds up until weeks before delivery. Similarly, S11 discovered that "you don't know what the real requirements are until you get them in front of the customer." He described a situation where his project team spent a significant amount of time trying to capture requirements from a customer who did not have a clear picture of their expectations for the finished system. Unfortunately, when the system was delivered, the customer rejected it because it was not what they wanted, despite the fact that the system satisfied all of the requirements. Both S10 and S15 found that customers needed to play with the product to know what they want; as a result, S10's and S15's project incurred technical debt in order to enable their project teams to deliver their product early to solicit feedback.

4.3.2 Inexperience

Some of the interview participants attributed incurring unintentional (inadvertent-reckless) technical debt to having junior developers and Co-op students on the team. S06, S10 and S15 stated that inexperienced developers on the team were likely to incur technical debt because, as S06 commented, they "didn't really know the difference between good software and bad software…". They explained that junior developers required more time and more supervision to produce the same quality work as other members of the development team. However, due to schedule constraints, the development team often did not have the time to offer junior developers the level of support they needed; consequently, their results were sloppier leading to unintentional technical debt.

S04 identified that he incurred technical debt because of his project team's unfamiliarity with the technology that they were using. In his example, S04 and his team were unfamiliar with developing software for the Linux operating system. They had assumed that using Java, a cross-platform development language, would allow their application to work on both Windows and Linux operating systems without additional effort. Furthermore, they decided to prioritize addressing issues on the Windows operating system because the team was more familiar with the environment and it would be easier to support. However, when S04 and his team began testing their software on Linux, they realized that there were many fundamental issues that they had overlooked which made supporting their software on Linux impossible. As a result, S04 and his team unintentionally incurred technical debt as a result of their inexperience developing for the Linux platform.

4.3.3 Attitudes

The interviews also revealed that the attitudes of high-level technical/non-technical leaders and project managers had a strong influence towards the project team's decision to incur technical debt. Teams that had technical leadership and management who had "just get stuff done" attitudes were more likely to incur technical debt compared to teams who had leaders who were focused on what the software needed to do. For instance, S07 described a situation where the project engineer on her team was so focused on making the product the best offering in the market that he drove the team to develop features quickly but just as quickly abandoned them when the customer response was poor, in order to move onto something else. Unfortunately, this resulted in a number of half-finished, abandoned features that became a maintenance issue for the development team. S06 had similar experiences when his technical leadership decided to create three products with the same source code on three different code branches. Incurring this technical debt significantly increased his team's maintenance overhead because whenever they fixed an issue, they needed to merge the changes to three code branches, perform three code reviews and test three code bases. On the other hand, S11 felt that his project team incurred technical debt because they allowed their priorities to be driven by their largest customer, who favored developing new features over making bug fixes.

S12 also observed that having "rogue programmers" on the team were a source of technical debt. He described these developers as "smart people who would go off and do stuff on their own without even surfacing to check-in with the rest of the team." Likewise, S07 explained how her company's management team's goal to retain top talent by carving out side projects so that developers could experiment with new technologies and languages ultimately resulted in a "Frankenstein" architecture for the company's flagship product. She also found that, as a result of the company's low turnover rate, the project teams incurred technical debt because there was a lower need to document design decisions and enforce processes since there was minimal knowledge transfer to new employees.

4.3.4 Software Age

A few of the interview participants described a type technical debt that is caused by software aging and evolution. As S11 remarked:

> *And I find that even if your architecture is really good, after 10 years, it's not so good anymore because you've had so many new features that you just didn't know on day 1 and so if you don't put continual effort into building…after a while, it starts, the old architecture, even if it is great, it starts to sort of creak under its own weight…*

One reason, S11 pointed out, is that systems are usually architected based on what was possible to build at the time when the requirements were specified. He explained that it is difficult for the project team to predict how the system will grow and to account for this growth because they do not have insight into the future. Furthermore, S11 remarked that "if you architect it [the software] for everything you could think of on day 1, you'd have some huge beast of an architecture that would be 80% wrong because 80% would be directed towards features that you'll never build."

Another reason is that the technology used gets old, as S12 explained:

> *And for me, one example of technical debt is just over time, technologies gradually become obsolete or superseded by other things and so, over time, you build up a technical debt that involves using stuff that's now of sort of legacy technologies…I don't see any way of avoiding that…*

S12 described a project that he worked on where the project team had made a decision many years earlier to use an inter-process communication technology that, presently, is old. He acknowledged that the technical debt was in the technology itself, because there are now better and newer communications technologies available that are more familiar to everyone on the development team and more easily maintainable. However, S12 explained that replacing the technology was expensive because it would require changes at the infrastructure level of the system and because it is used everywhere. Nonetheless, at some point, S13 added, the project team will have no choice but to upgrade the technology because it will no longer be supported by the available computing platforms.

4.4 Symptoms

This study also aimed to capture the impressions of the interview participants on how technical debt impacted their projects. Their responses revealed that, to a large degree, technical debt negatively impacted software projects, causing instability, maintenance and performance issues and compromises. However, technical debt also had a significant positive impact on their projects by allowing their development teams to deliver their product quickly to market to meet their business goals.

4.4.1 Poor Quality

All of the interview participants stated that the outcome of incurring technical debt was poor quality, including increased complexity, poor performance, low maintainability, and fragile code. Consequently, the poor quality created a long-term "pain" for the developers. S08 described the situation as "what you ended up with is some code that is arcane, I'd say, like, very antiquated, hard to maintain, fragile, like easy to break, and bugs would fall through" that left only a small number of developers who were able to maintain the code. Thus, as S10 pointed out, it was stressful for some engineers to work with a debt-laden code base.

Some of the interview participants described how technical debt also inflicted "pain" on their customers. Particularly, technical debt often impacted the performance, reliability and stability of the product, creating poor customer perceptions and making customers unhappy and angry. In S11's case, the customer was so upset with the product that S11's company not only risked losing their customer, but also was sued by the customer for failing to deliver what was promised to them. Some participants reflected that having technical debt increased the amount of effort and cost required to support their customers, especially because it was less efficient and more expensive to address technical debt issues found at a customer site.

When the interview participants incurred technical debt, they compromised on the quality of their software in different ways. For instance, S18's project team reduced the scope of their delivered solution to support only a subset of functionality. S03 decided to implement a "hack" to work around a limitation in the Java Runtime Environment (JRE) in order to implement a new feature that could detect if the user was outside the bounds of the application. However, the "hack" prevented quality assurance (QA) from being able to verify S03's solution using test scenarios that reflected normal, daily user interaction (QA could only verify the solution by putting the mouse upside down on the table). Both S06's and S15's projects relaxed on enforcing software development processes, such as up-front design and code and design reviews. As S06 explained it:

> *…they [engineering] ended up getting to a point where it wasn't going to be done in time so they basically started having to really cut corners just to make things work. So whether it was a good solution or something that was scalable or something that was likely to be expanded on, it was just completely thrown out the window, the design wasn't necessarily followed and then…you had this thing that sort of worked at the end of the day and did a lot of the things that the unofficial requirements said it needed to do but did it in such a fashion that put in the hands of real users in real scenarios turned out to be severely lacking.*

Thus, S06's project team ended up releasing a product to market that did not work reliably for customers and resulted in a code base that was a hybrid of fragile, old code that the developers were afraid to work with and new code that did not work 100% of the time.

4.4.2 Uncertainty

Several interview participants described how technical debt impacted system stability and caused unexpected behavior. S08 provided an example of an application that her project team built whose original architecture was not designed to support its current state. Unfortunately, upper management could not find the justification to invest in re-writing the application, leaving it in a state where "most things would work but some customer would try some odd ball thing in a new release and it wouldn't work."

Technical debt also created uncertainty for developers making changes to the code base. S06 explained how his project's technical debt created a number of implicit dependencies in the code. Over time, his team grew nervous about making changes to the code to fix bugs because they were unsure if their changes would cause additional problems in the system: "…the amount of time that it took to fix any known issue was getting, I wouldn't say exponential, but it had that feeling where every time…you feared that any time you made a change, you were going to cause something else to go wrong." Furthermore, his management began to lose confidence in bug fixes to the code, especially under time pressures, because they were afraid that the fixes would cause problems that would not be detected by QA before the software was delivered to the customer.

For S11, having technical debt made it difficult for his project team to make accurate work estimates. He found that technical debt skewed work estimates in two ways. Firstly, the added complexity introduced by technical debt meant it took longer to develop new features or fix bugs. Secondly, the increased customer liability as a result of having technical debt meant the development team spent more time addressing customer calls and fixing customer issues, especially if they were required to visit a customer site. Thus, when a feature or bug fix exceeded its work estimate, S11 could never be sure if the work took longer because of the unanticipated complexity of the feature or because of the added complexity from the technical debt.

4.4.3 Deliver to Market Quickly

When asked if incurring technical debt was worthwhile, most of the interview participants expressed that it was because it allowed their project team to deliver software to market quickly. S08 felt that incurring technical debt enabled her project team to win a huge corner of the market. S10's project team incurred technical debt for similar reasons - his project team released a first-to-market product and wanted to beat out its competitors. Furthermore, S10's project team wanted to deliver their software to market quickly so they could collect customer feedback sooner. This was particularly important to S10's project team because they were not very familiar with their domain; thus, they could learn what their customers really wanted and how they planned to use the software, allowing them to direct their focus on what to build next and where to improve first. Likewise, S18 stated that taking on technical debt to deliver to market quickly helped his team avoid spending too much time in areas of the system that did not provide business value.

However, a few of the interview participants felt that some of the technical debt that they incurred to deliver their software quickly to market was not necessary. S11 commented:

> *…the stuff we were doing is not going to drive sales for another year and so, spending another week shoring it up could have been the right thing to do, but we were always so focused on I want to get this release out this week or this month…And I also looked at features we'd developed…and we'd sort of got them half done…and they were kind of OK but no one ever used them because they weren't great and actually, they didn't need them all that much anyways."*

S06 had the same opinions as S11. He felt that his project unnecessarily took on technical debt in order to meet the hard delivery deadlines set by marketing; however, "people weren't beating down our door for this product so it really had to be pushed at people to buy it." Furthermore, his project team spent six months following the product's release to build a service pack that re-architected and fixed many features in the product. Thus, as S06 pointed out, incurring the technical debt was not worthwhile to the success of the product; instead, S06 believed that if the release had been delayed, the development team could have had more time to develop the product and marketing could have spent more time building up market interest.

4.5 Management

Most of the responses to the question of how the interview participants manage technical debt on their projects involved some form of communication, both externally, with customers, marketing, and management and internally, within the development team. However, several interview participants suggested that doing nothing was also an approach to managing technical debt. They felt that it was better to passively wait for customer feedback instead of being proactive at paying back technical debt. Their reason is because the product does not gain any value from paying back technical debt that did not cause problems for the customer in the first place. Furthermore, they also pointed out that waiting for customer feedback helped the project team avoid reducing the wrong kind of debt or taking the wrong approach to reduce the debt. Nonetheless, the interviews also captured some more proactive approaches to managing technical debt, including analyzing risks, managing expectations and documentation.

4.5.1 Risks

One suggestion from the interview participants for managing technical debt was to manage it as project risks. S10's team prioritized their list of technical debt to pay off by comparing the priority, value, cost and risk of each debt item. Technical debt items with high risk and low cost were pushed back, and debt items with high cost, low risk were addressed immediately. Other debt items in between were evaluated against other factors, such as bottlenecks, customer requests and long-term plans for the product. S11 gave an example of a team leader on his project team who saw feature requests as also an opportunity to address technical debt in the same area of code. Thus, the team leader would factor in extra time to his work estimates for the feature, because "…the outside world doesn't know if this feature should take 20 days or 10 days so if you take 20 days and re-factor it, no one outside has to know…" On the other hand, S06 and S15 negotiated with management to convince them to allocate 5-10% of the total effort and cost of each release to paying off their technical debt.

S06 found that actively monitoring the code changes that were checked into source control also helped to limit the amount of unintentional or unnecessary technical debt introduced to the source code. Similarly, S15 and S18 enforced software quality processes and standards on their development team as a way of mitigating technical debt risks.

Lastly, S17 believed that building a team who shared his philosophy and attitude towards technical debt helped him manage his project's technical debt. Thus, he made some key hires that brought in new influences to change and improve the team's perspective on incurring technical debt strategically by spending more up-front time planning and designing.

4.5.2 Expectations

Another approach to managing technical debt that S15, S16 and S18 followed was to manage customer expectations by making them aware that technical debt existed on their projects. S16 coined the approach as "under-promise, over-deliver." For S15, this included exposing all of the technical debt issues in the software to the customer because

S15 felt that “we’re all equal partners in making a mess so we have to be equal partners in cleaning it up.” Both S15 and S16 maintained open dialog with their customers by highlighting the existence technical debt on the project and its impact on the future cost of implementing new features if time is not allocated to address it. Similarly, S18 brought visibility by describing technical debt to the non-technical stakeholders as the risks they would face while trying to meet their deliverables. This approach benefitted from putting technical debt into a context that non-technical stakeholders understood and to which they could easily relate.

4.5.3 Communication

The most common approach to managing technical debt was to communicate its presence through documentation. S03 and S18 used in-line comments and TODOs to mark the places where technical debt was incurred in the code. S19’s project team kept track of all of its project’s technical debt on a Wiki page that is accessible to everyone on the team. S15 tracked his team’s technical debt as items in their bug tracking system.

Moreover, many of the interview participants documented the technical debt on their project by hosting a brainstorming session with all of the developers at the end of each release. In the brain storming session, the developers were asked “If there was stuff that you could re-do in the software, what would it be?” Each of the areas of code that were identified during the brain storming session were recorded and tracked as technical debt. There are several benefits to this approach. Firstly, it quickly highlighted the key areas in the code where technical debt should be addressed based on how often that part of the code is mentioned by the developers. Secondly, it took advantage of the insights of the developers, who work with the code daily, to identify technical debt in the system. Lastly, this approach constantly not only reminded the development team that technical debt existed but also made developers acutely aware of the amount of technical debt in the system.

4.6 Conclusion

Interviews conducted with nineteen practitioners in the software industry revealed that technical debt is not a new concept to them but something that they are familiar with and work with on a daily basis. In general, interview participants described technical debt as “trade-offs” and “shortcuts”. Most instances of technical debt on a project were incurred as a result of business reality – meeting budget and schedule constraints – but other factors, such as familiarity with project objectives, experience and software age played a factor too. The impact of technical debt, as described by the interview participants, was largely negative and painful, affecting performance, stability, and overall software quality. However, the participants also acknowledged that incurring technical debt was worth the pain because it helped their project teams deliver their software to market quickly. Finally, all of the interview participants pointed out that the key to managing technical debt on a software project is communication – constantly bringing visibility to the existence of technical debt and making both technical and non-technical stakeholders aware of the consequences of carrying technical debt forward to the next release.

CHAPTER 5
GAME STUDY – "HARD CHOICES"

The purpose of the game study is to validate the findings of the interview study by investigating how software practitioners characterize and manage technical debt through game play. Game play is a learning technique that Agile coaches often employ to teach new concepts. It is an interactive and fun teaching tool that allows students an opportunity to actively engage in gaining insights and hands-on experience from applying the concepts being taught to simulated situations. At times, game play can be a more effective means of delivering ideas instead of the traditional methods of lecturing and reading books.

I worked with researchers in the Communicating the Benefits of Architecting Within Agile Development project at the Software Engineering Institute (SEI) at Carnegie Mellon University in Pittsburgh, PA to develop a board game around the concepts of technical debt, called "Hard Choices" [52]. The goal of the board game is to help players recognize, and identify strategies for managing technical debt throughout the software development process [53]. This chapter describes the "Hard Choices" board game, the methodologies that were undertaken to conduct this study and the results derived from the discussions that were generated after playing the game with several groups of software practitioners.

5.1 Game Board

In January 2010, we attended a one-day workshop on "Creating Agile Learning Games" by Chris Sims of the Agile Learning Labs [54] to learn about the game creation process and to brainstorm ideas for games. At the workshop, we came across the “Short Cut” board game by Quality Tree Software, Inc. [55]. Since the “Short Cut” board game contained many of the game design elements that we wanted for our technical debt board game, we decided to enhance and expand on the “Short Cut” game to create “Hard Choices.” The game design elements that we used to communicate the concepts around technical debt in “Hard Choices” are shown in Table 4:

Technical Debt Concept	Game Design Element
Risk, Uncertainty	Chance, Turns
Trade-offs	Strategy, Choice, Unbalance, Knowledge
Impact	Competition, Ending, Winning, Scoring

Table 4: Technical Debt Concept to Game Design Element Mapping

The game board (see Appendix C) consists of a winding path from “START” to “END”; the path is a metaphor for the software development release cycle. Certain squares along the path are marked with symbols to represent decisions that the development team must make during the release, as shown in Table 5:

Symbol	Name	Action
	Hard Choices	Decision point: players must decide whether to take a shortcut (i.e. cross the bridge) to finish the game faster, but incur a penalty, or, continue and advance slowly through the game.
	Bridge	Shortcut: Players crossing the bridge collect a “Bridge” card. The bridge represents a shortcut or trade-off decision, which allows players to finish faster. The “Bridge” card carries a penalty – players must subtract 1 from every subsequent dice roll. Players can get rid of their “Bridge” cards by skipping a turn in the future.

Symbol	Name	Action
🪛 🪚 🔨	Tool	Design/Architecture: Players landing on a tool collect a “Tool” card. The tools represent a decision to spend time to implement good design/architecture. The “Tool” card carries a reward – players receive 1 point for every “Tool” card that they collect. Players can trade their “Tool” card for a free turn.

Table 5: “Hard Choices” Path Symbols

The goal of the game is to cross the “END” square, which is equivalent to the end of a software release cycle, with the highest number of points. In addition to gaining points by collecting “Tool” cards, players collect extra points for crossing the finish line first (i.e. being the first to deliver the software to market).

5.2 Rules

“Hard Choices” is designed for two to four players. To play the game, the player rolls a dice to determine the number of squares he advances (a player can move in any direction and/or change direction). With each dice roll, the player is required to make choices on whether or not to move in a certain direction, cross a bridge, land on a tool, skip a turn to get rid of a “Bridge” card, or, trade in a “Tool” card for an extra free turn in order to cross the “END” with the highest number of points. Each choice that the player makes has benefits and consequences. For example, the benefit of crossing a bridge is that the player may reach the “END” square quicker, which could potentially help him earn extra points. However, the consequence of crossing a bridge is that the player must subtract 1 from every subsequent dice roll, which could result in slowing down his progress towards the “END” square. Likewise, moving along the path with the aim of landing on a tool has the benefit of collecting a point, but carries the consequence of slower progress towards the "END" square. The player's decision at every dice roll depends on his current position on the board, the position of other players on the board and his game strategy.

When the first round of "Hard Choices" is over (i.e. all of the players have reached the "END" square), the players are informed that they will need to play a second round. The purpose of the second round, which represents the second release of a software product, is to help players appreciate the impact of the decisions that they made in the previous round, or release. In the second round, the players retain all of the "Bridge" and "Tool" cards that they collected in the first round of the game. Additionally, "Game Changer" cards are introduced into the round to mimic the sudden changes in business goals or customer demands that can happen during product development. "Game Changer" cards can affect the valuation of the "Bridge" and "Tool" cards and introduce additional elements of randomness and uncertainty into the game, for example:

- For each "Saw" card, +1 to the dice roll;
- For each "Bridge" card, -2 to the dice roll;
- Players may return one "Bridge" card for every "Hammer" card that they own.
- Players may cross one bridge for every "Screwdriver" card that they own without incurring a penalty.

Multiple rounds of "Hard Choices" are played until the players feel that they fully understand the concepts of managing risk, uncertainty, options and the impact of carrying technical debt during software development. Generally, two rounds of the game are sufficient enough for the players to grasp the objectives of "Hard Choices."

5.3 Methodology

This game study is based on the findings that I gathered from three game play sessions in which I attended, as listed in Table 6:

Session	Location	Number of Participants	My Role
Agile Vancouver meeting	Vancouver, BC, Canada	15	Organizer and Facilitator
Agile Development and Software Architecture: Hard Choices Game Event at the SATURN Conference	Minneapolis, MN, United States	16	Workshop Participant
"Dev-Squared" meeting at Aquatic Informatics, Inc.	Vancouver, BC, Canada	15	Organizer and Facilitator

Table 6: Information about "Hard Choices" Sessions

Although this study is only based on the three sessions that I attended, the "Hard Choices" board game has been played by over one hundred people at a number of conferences, meetings, workshops and classrooms led by members of the Communicating the Benefits of Architecting Within Agile Development project team.

The game study provided an opportunity to explore how the participants view and manage technical debt in a uniform environment. In this study, all of the participants shared the same experiences of learning about technical debt through playing "Hard Choices", regardless of their familiarity with the topic. Subsequently, this provided everyone with the same context to draw from during the discussions when they were asked to share their lessons learned. Furthermore, the game allowed for the examination of the participants' decision making process throughout a software development cycle in a very compressed amount of time. The participants in the study played two rounds of the game to capture how the decisions they made in the previous round affected their decisions in the current situation; each round was approximately twenty minutes.

5.3.2 Data Collection

Participants were invited to attend the “Hard Choices” game play sessions through e-mail sent to the organization’s mailing list and, in the case of the SATURN conference, through word-of-mouth. The participants who attended the sessions included software developers, business analysts, software architects, requirements engineers and project managers.

Before game play started, the participants were given a short five-minute presentation that briefly defined the concept of technical debt and explained the rules of the game. Then, the participants were randomly divided into groups of three or four players and given the board game, game pieces and game rules. While the groups played “Hard Choices”, I was available as a facilitator to answer any questions that arose. When each group finished the first round of the game, I asked the groups to wait until everyone in the room finished playing. Figure 3 is a photo of participants playing “Hard Choices”, taken from one of the game play sessions:

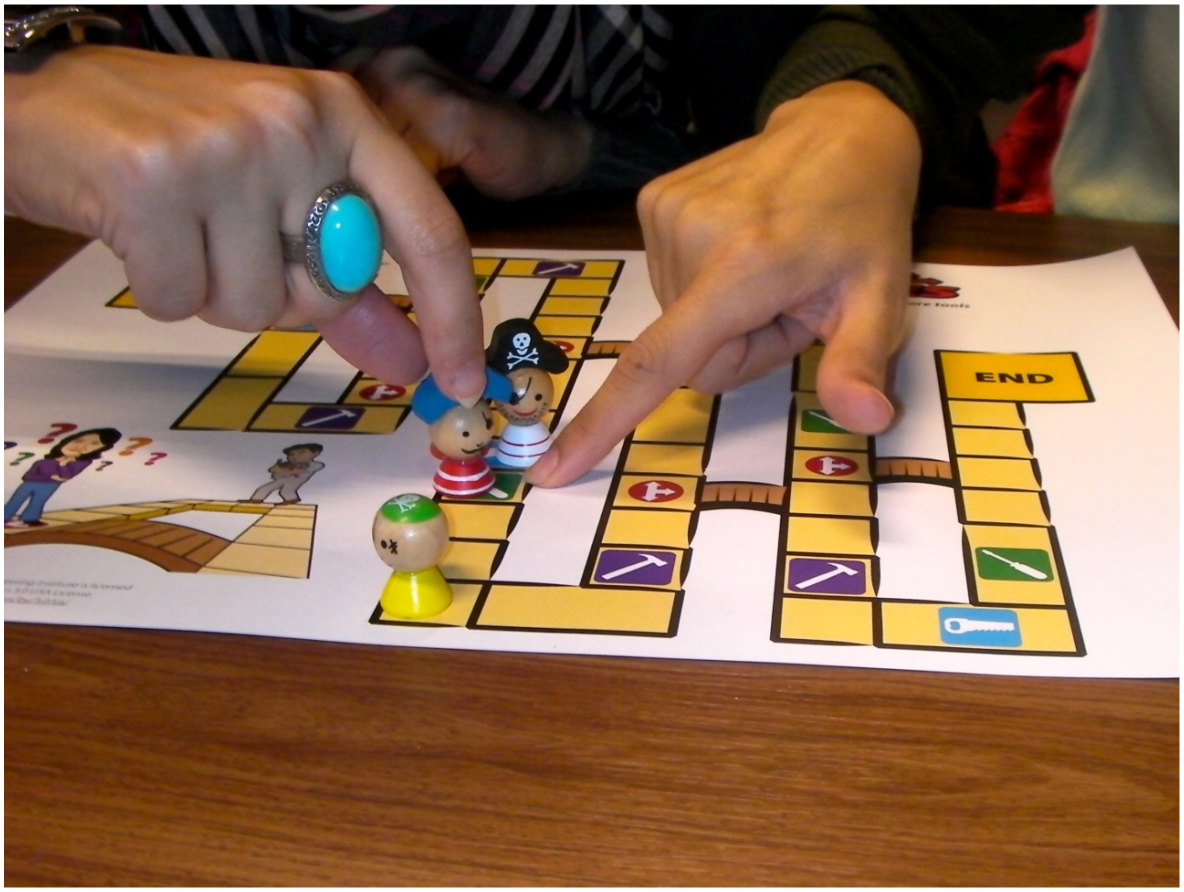


Figure 3: Playing the "Hard Choices" Board Game

When all of the groups completed round one, I introduced the "Game Changer" cards and told the groups to play round two. Following round two, I led the participants in a debrief session to discuss what the players learned from playing "Hard Choices", including strategies and how elements of the game mapped to the technical debt concept. Furthermore, I also asked the participants to provide me with feedback on the game and suggestions for how the game could be improved.

During the debrief session, I took notes on the insights and feedback that the participants shared. Immediately after the end of the session, I supplemented the notes with additional observations and comments that I had heard during the game play. These notes were shared through e-mail with the members of the Communicating the Benefits of Architecting Within Agile Development project team.

5.3.3 Data Analysis

The data from the “Hard Choices” game play sessions were also analyzed using content analysis. Unlike the interview study, the game study did not have any preset codes. The game data was coded using open coding to identify emergent codes. Axial coding was applied to develop categories from the emergent codes. Then, the game codes were compared to the interview codes to find similarities and differences and to extract themes. The list of codes for the game study is shown in Table 7:

Codes	Sub-Codes
Calculated Risks	Opportunities
	Drawbacks
Adaptability	
Luck	

Table 7: Codes from Analyzing Interview Data

All of the codes were derived from words frequently used by the participants in their feedback during the debrief sessions. The “Calculated Risks” code describes the participants’ process when evaluating their options for their next move. The outcomes of their decisions generally classified into two groups – *opportunities* and *drawbacks*. The participants pointed out that some decisions helped to create opportunities that allowed them to move ahead quickly and contributed to helping them win the game, whereas other decisions created drawbacks that tended to slow down their ability to progress forward. The “Adaptability” code reflects on an attribute that the participants felt was necessary to be successful at playing “Hard Choices.” Many of the participants believed that having the flexibility to change their strategy while they were playing given their current situation was an important element in winning the game. Lastly, the “Luck” code derives from the participants’ observation that chance was a factor in their outcome in “Hard Choices”.

5.4 Findings

The following section summarizes the findings of the feedback provided by the participants following a “Hard Choices” game playing session. The findings are organized by the codes defined in the previous section.

5.4.1 Calculated Risks

A number of participants recognized that the key to successfully managing their technical debt was taking calculated risks. Participants found that deciding on the type of risk to take and when to take the risk, evaluating the outcome and being responsive to how other players played contributed towards their success in the game.

5.4.1.1 Opportunities

The participants discovered a number of strategies for incurring technical debt that created opportunities for them to pull ahead of the other players in the game. Some players found that a successful strategy was to cross a lot of bridges at the beginning of the game to get ahead of the other players. When there was enough distance between them and the other players, they began playing back the debt by skipping turns because they had enough time to pay off their debts before the other players caught up to them. A number of participants developed the strategy of crossing bridges only as they got closer to the “END”. They felt that they could afford to be riskier towards the end of the game because it was not as important to them how many bridges they had left over as long as they could cross the finish line first. One participant even observed that if there were two bridges close together, it was worth crossing both bridges because it got him much further ahead of the other players. On the other hand, other players discovered that in order to keep moving forward in the game, it was more beneficial to skip a turn to pay off some of their debt rather than to lose a turn because they had accumulated too much debt. Some also found that they could take advantage of tools they collected to help them move ahead because they could trade in their tools for an extra turn.

In fact, all of the participants realized by the end of the game that taking bridges was unavoidable. One participant made an interesting comment about her strategy with regards to taking bridges. Her initial strategy was to play the game cautiously so she avoided crossing any bridges as she advanced towards the "END" square on the game board. However, when she saw that she was getting further and further behind the other players, she realized that she needed to cross more bridges in order to catch up with the other players in the game.

Interestingly, business analysts, and project managers (i.e. non-technical stakeholders) seemed to play the game more aggressively than developers and testers (i.e. technical stakeholders). In particular, business analysts and project managers tended to cross bridges whenever they could, especially in the first round. On the other hand, developers and testers were hesitant about crossing bridges and preferred to follow the game path and collect tools. However, both groups realized by the second round that they moved the fastest through the game when they were able to find a balance between crossing bridges, collecting tools and skipping a turn to pay back a bridge.

5.4.1.2 Drawbacks

Likewise, participants also uncovered strategies that were detrimental to their success at crossing the finish line. For example, one of the players had played aggressively, crossing all of the bridges in the first round in order to collect the extra points rewarded for crossing the "END" square first. Another player in the same game played a much more conservative game, avoiding bridges and collecting tools, and ultimately, finishing last in the first round. However, when the second round began, the aggressive player suddenly found that he could not move as quickly through the game because he was held back by all the debt that he had accumulated from the first round. In the end, the conservative player, although she moved more slowly in the first round, finished the second round faster, collected extra points, and won the game.

5.4.2 Adaptability

Throughout the game, the participants recognized that they needed to adapt their strategy depending on the current context of the game. A few participants observed that their strategy depended upon the location of the other players on the game board. For example, several participants found that once another player in the game crossed the "END" square first to collect the extra points, their motivation to continue playing aggressively to finish faster diminished. At that point, the participants preferred to take their time and collect more tools to increase their chances of winning the game because there was no longer a reason to reach the "END" square quickly.

Almost all of the participants agreed that they changed their strategy in the second round to reflect what they had learned from playing the first round. Similarly, with the introduction of the "Game Changer" card in the second round, the participants adjusted their strategy to maximize the benefit or minimize the consequence of the card. These players found that, as the game progressed, they adapted their strategy to benefit from repeating successful approaches used by other players in the game and to avoid repeating their mistakes.

Nonetheless, many of the participants remarked that their overall strategy would have been different if they knew in advance that there would be multiple rounds of the game; however, they realized that, like in industry, it is impossible to predict the direction of a software project until the project team receives customer feedback.

5.4.3 Luck

In addition to strategy, some players observed that luck played a part in being able to successfully manage technical debt. One player observed that he depended on luck when the "Game Changer" card was selected for the second round because the instructions on the card could either work in his favor or work against him. Similarly, another player successfully finished second in both rounds of the game without needing to cross any bridges because he rolled a number of 6's on the dice. Another player commented that even though she did everything right – she took minimal shortcuts and collected tools – she did not advance very far on the game board because she rolled low dice values. She came up with the analogy that her low dice rolls was like having a development team who does everything "by the books" but whose productivity is slow because of poor team dynamics or having a lack of experienced developers on the team. Eventually, the player had to change her strategy and took more shortcuts in order to keep up with all of the other players on the game board.

5.5 Feedback

During debrief, there were many suggestions for improving “Hard Choices”. For instance, a player described how he felt elements of “Hard Choices” matched up to Scott Ambler’s iron triangle of software development [56], shown in Figure 4:

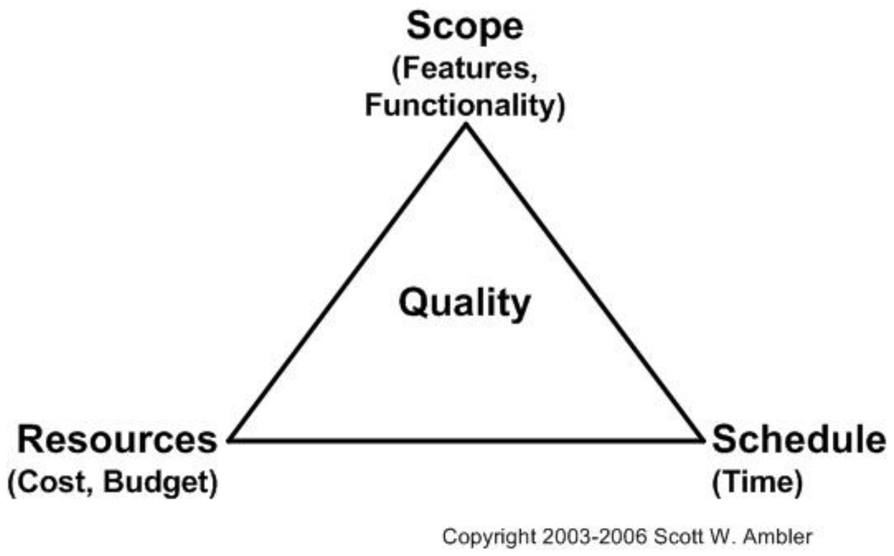

Scope
(Features, Functionality)
Quality
Resources
(Cost, Budget)
Schedule
(Time)
Copyright 2003-2006 Scott W. Ambler

Figure 4: The Iron Triangle of Software Development [56]

He identified that the tools represented the scope and the game path and the bridges represented the schedule. However, he pointed out that “Hard Choices” was missing elements to represent the resources, located in the third corner of the iron triangle. He felt having an element in the game that represented resources would make players be more strategic about acquiring tools and crossing bridges.

There were also suggestions to make the game board more reflective of reality. A number of participants felt that risk (the bridges) was represented too linearly, which would not be the case in real-life, where risks grow exponentially. They also felt it would be more realistic for the bridges closer to the “START” square to have a higher penalty than the bridges closer to the “END” square and for the penalty on the bridges acquired in the first round to increase significantly if they are carried into the second round. Another participant pointed out that the game failed to reflect the Agile tenet of building features only when needed (to avoid architecting) as is the common industry practice. Yet another participant recognized that “Hard Choices” allowed players to complete the game without collecting any tools whereas, in industry, it would be almost impossible for an organization to make money from delivering a product without features.

Nonetheless, many of the participants agreed that “Hard Choices” was useful for demonstrating the concepts and impact of technical debt, particularly from a technical perspective. Several thought that the game would also be useful as a tool for understanding and showing the different perspectives that software engineers and business managers have with respect to taking risks. One of the problems in software development that the participants pointed out was that software engineers and business managers have different goals: software engineers are focused on delivering quality software whereas business managers are focused on delivering features. They felt that “Hard Choices” would allow both software engineers and business managers to gain insight and better understanding into each other’s decision making process for taking risks as a means of helping to improve communication between them.

5.6 Conclusion

This chapter describes the methodology and findings of a game study suing the "Hard Choices" board game. "Hard Choices" was developed by the Communicating the Benefits of Architecting Within Agile project group as a tool to teach players about the concept of technical debt. It is a metaphor for the software release cycle, where bridge and tool playing squares represent the trade-offs that practitioners make while trying to ensure that the release is delivered on schedule. The purpose of this game study is to validate the findings of the interview study on technical debt through game play (results presented in Chapter 6). The insights, feedback, and discussions generated by the participants, which included developers, project managers, testers and business analyst, provided opportunities to further investigate how software practitioners characterize and manage technical debt.

CHAPTER 6
DISCUSSION

This study set out to identify guidelines to help software practitioners recognize and manage technical debt. The discussion in this chapter will focus on identifying those guidelines by answering the research questions of this thesis using the findings from the interview and game studies.

6.1 Defining and Characterizing Technical Debt

The participants from both studies strongly associated technical debt with making trade-offs. This association also agreed with the findings from the literature review. In "Hard Choices", the game players made trade-offs between moving ahead as quickly as possible and collecting the most points to win the game. The factors that the game players considered when making these trade-offs were directly related to how they would move in their next turn. On the other hand, the interview participants had more variables and uncertainties to consider when evaluating their trade-offs. In particular, the interview participants had to evaluate not only the technical implications but also the impact on the delivered business value. Such trade-offs included, for example, deciding if it was better to postpone the release to build a higher quality product, which would require the sales team to change their messaging to the customers, or to release the product immediately in order to capture the market share. Thus, project teams benefitted from involving all project stakeholders (technical and non-technical) in the decision process to incur technical debt.

In the literature, technical debt is classified as unintentional and intentional debt [4]. However, both studies were unable to find many instances where the participants unintentionally incurred technical debt, like sloppy or messy code. In fact, participants from both studies primarily incurred technical debt because of project decisions. When the participants made poor decisions, they incurred "unintentional" debt. These poor decisions resulted in creating risks, such as slowing down progress and causing pain, including poor quality, poor performance and poor maintainability. On the other hand, when the participants made good decisions, they created opportunities, which brought them success in reaching their goals, including winning "Hard Choices" and delivering their product quickly to market. Therefore, another way to classify technical debt is as risks and opportunities.

Both studies also found that the participants recognized that technical debt was unavoidable because of business reality. The participants acknowledged that, without incurring technical debt, they would move too slowly to keep up with their competition.

Lastly, the interview participants reported that technical debt was difficult to measure because the value of the debt was different depending on where it was incurred in the code. In some cases, the value of the technical debt was worthless because it was incurred in an infrequently used piece of code. Furthermore, external factors, such as the market direction and customer demand also impacted the value of the technical debt and could make a worthless debt suddenly acquire value. The game players did not make similar observations because every bridge was assigned a quantifiable penalty and the penalty was the same regardless of where the player crossed the bridge on the game board. However, the game players recognized this deviation from reality in their feedback on the game.

6.2 Motivations to Incur Technical Debt

Both studies demonstrated that the decisions and strategies adopted by the participants were strongly motivated by their desire to "win". For the game players, this meant collecting the most points whereas for the interview participants, it was capturing a market share. In both cases, "winning" was also associated with coming in first - being the first player to cross the "END" square to collect extra points or being the first company to offer a product to market. Consequently, time constraints were always the primary reason for incurring technical debt. Nonetheless, participants in both studies faced hurdles along the way that included randomness and unpredictability, such as the roll of a dice, “Game Changer” cards, the change in market direction, working in a new domain and unclear requirements. Interestingly, the hurdles that the participants faced were not directly related to the work that they needed to do to achieve their win. Rather, these hurdles were tangential risks that the participants needed to take into consideration in order to ensure their success.

The participants in both studies used technical debt to mitigate their tangential risks. But, they found that when they incurred their debt and how they incurred it greatly affected their successes. For example, participants from both groups reported that incurring a large amount of technical debt at the beginning to gain separation from the competition before repaying the debt worked as a strategy to help them get ahead. The game players also found that if another player beat them to crossing the "END" square first, they changed their strategy to win by collecting more tools and earning more points. This strategy, although not mentioned by the interview participants from this study, has proven to be successful for products such as the iPod from Apple and Facebook.

Observations from the two studies also suggest that there is a difference in attitudes on incurring technical debt between developers and management. In the interviews, project managers were more likely to take on technical debt because they realized that their project needed to also meet its business goals. Conversely, developers were less likely to take on technical debt because they realized that they would have to deal with the resulting maintenance issues on a daily basis. Similarly, when playing "Hard Choices", the project managers and business analysts were more inclined to cross bridges in order to help them get ahead as fast as possible. However, the developers and testers were more reluctant about crossing the bridges and preferred to take the longer route and collect tools. The developers and testers were also better at putting in the effort to get rid of their bridges than the project managers and business analysts. Nevertheless, all of the participants agreed that incurring technical debt was worthwhile because it helped them win, despite the pain it inflicted, including slowing down their forward progress, poor performance and increased complexity and maintenance issues.

6.3 Strategies for Managing Technical Debt

Finally, the findings from both studies showed that managing technical debt is much like managing risks. Like risk management, technical debt management is a balancing act that strives to achieve a level of debt that enables the project to achieve its goals while mitigating its failures. The participants in the game study found that their success depended on their ability to find a strategy to incur just the right amount of debt; if they incurred too much or too little debt or incurred debt too early, they would slow down their progress towards the "END" square. One strategy for maintaining this balance that both groups of participants recognized was to pay back the technical debt as frequently and as early as possible.

Managing risks require that all of the stakeholders become owners in mitigating the possible failures. This is supported by findings from the interview study where a number of interview participants recognized the importance of using communication to increase the visibility of technical debt to both technical and non-technical stakeholders. Communication was also important to convince all the stakeholders to buy into the same strategy for managing technical debt. As a result, the stakeholders were able to work together to find solutions because everyone could see how the technical debt impacted each other's goals. The game study did not report similar findings because the participants worked individually to play the game. In this case, the game does not accurately reflect real-life scenarios where product development always requires a team effort.

In both studies, a common approach to managing technical debt was to deliberately make the effort to re-pay the debt. For example, in the game study, the game players who won the game were the ones who tended to skip a turn to get rid of their bridges when they had the opportunity to do so rather than allowing the bridges to accumulate to the point where they were losing turns. Likewise, in the interview study, a number of participants managed their technical debt by specifically allocating 5-10% of each release cycle towards paying back their technical debt. There were also other participants who incorporated extra time in their effort estimates to pay off their debt or used features as an opportunity to address technical debt in the same area of code.

Lastly, an important aspect of risk management involves using past experiences and lessons learned to create heuristics that can be applied to mitigate similar risks in the future. This was especially evident in the findings from the game study, where the game players identified that they adjusted their game strategy between the two rounds based on their mistakes from playing the first round. The players found that in doing so, they were more successful in playing the second round of the game. Although this finding was not strongly identified in the interview study as an approach for managing technical debt, many interview participants included comments in their responses to the interview questions that showed they had evaluated their decisions in hindsight and reflected on how they would have done things differently. As a result, investigations such as this one are beneficial to software practitioners in managing their technical debt by aiming to capture and document the insights, experiences and lessons learned that can be applied to future software projects.

6.4 Validation from the Replication Study

The findings from Taksande's replication study [46] strongly correlated with the results of the interview study conducted for this thesis. For instance, Taksande's findings agreed with this study's findings that technical debt is viewed as unavoidable trade-offs that projects make in order to meet budget and schedule constraints. In addition, Taksande reported that the interview participants in his study also regarded technical debt as a compromise on the functionality, quality and correctness of the software. Both studies discovered that the interview participants commonly included hacks, workarounds and sloppy coding as part of their definition of technical debt, contrary to Cunningham's assertion that messy code is not a form of technical debt.

Similar to this study, Taksande found that the interview participants generally did not incur technical debt unintentionally; rather, they incurred technical debt from making decisions. Taksande classified these decisions into good decisions and poor decisions. He explained that good decisions were intentional decisions made to meet budget and time constraints whereas poor decisions led the organization to incur huge losses. This study broadened this classification to describe technical debt as either project opportunities or risks.

Furthermore, both studies reported that the most common reason for incurring technical debt is time constraints because commitments to the customer always take precedence. Taksande's findings validated this study's findings that other reasons for incurring technical debt include changing requirements, usually due to customers not knowing what they want or management receiving new crucial, market information, and, decisions that were once optimal but have proven to be problematic over time, especially as the project team becomes more familiar with their domain. However, unlike this study, Taksande did not find any interview participants who reported inexperienced software developers as a source of technical debt.

The findings from Taksande's study also agreed with this study that software practitioners feel incurring technical debt is worthwhile because the opportunity cost from delivering on time and in budget greatly outweighed the negative symptoms of reduced quality, increased complexity and poor performance. In both studies, the interview participants described the impact of incurring technical debt on their project to include increased cost and resources required to support and maintain the software and the risk of making customers unhappy, potentially losing their goodwill and faith.

Furthermore, Taksande's replication study reported management strategies analogous to those described in this study. Particularly, Taksande's findings strongly emphasized making the existence of technical debt visible through documentation. He reported a number of interview participants who conducted regular "architectural audits" and who tracked technical debt in their project backlog or task board. The same techniques were also uncovered in this study. Both studies also found that another approach to managing technical debt is to maintain an open dialog with non-technical stakeholders (e.g. project managers, customers) so that they understand why technical debt exists and its impact on the project. However, Taksande's study did not find any interview participants who managed their technical debt by specifically dedicating time in their release cycle to paying off their debt like some of the interview participants in this study.

Finally, Taksande's study found that many of the interview participants felt that working with technical debt was a good learning experience for the developers on their software projects. The interview participants felt that the past experiences with technical debt taught them what to do and what not to do the next time they needed to incur technical debt for similar reasons. Although the interview participants for this study did not make the same observation, the game players in the game study reported that they gained insight from their current experiences with technical debt to improve their strategy in the future.

6.5 Conclusion

The findings in this thesis suggest that technical debt is more than just a technical issue; it is a three-dimensional problem defined by technical decisions, delivered business value and time. Software projects are constantly trying to make technical decisions that will deliver the greatest business value in the shortest amount of time. However, these decisions turn into technical debt if, at some point in the future, it slows down the development team's ability to continue delivering business value.

But, not all technical decisions are technical debt. If a project makes decisions that do not deliver business value at any point during the lifecycle of the software project, such as writing poor quality code, these decisions are not technical debt. Likewise, technical deficiencies, such as defects and unimplemented features, are also not technical debt because they do not hinder the team's ability to continue delivering business value in the future.

Nonetheless, as the findings in this thesis shows, technical debt is necessary for projects to survive in the business of software. Thus, software practitioners would benefit from having models, tools and processes that help them forecast the future business value of their present technical decisions. Having the ability to predict the long-term business outcomes of short-term technical decisions would help software practitioners choose the right kinds of technical debt to incur and avoid those that would cause too much pain, for both software practitioners and their customers, in the future.

The following are a set of guidelines identified from the results of this thesis to help software practitioners recognize and manage technical debt:

G1: Technical debt is trade-offs between technical and business issues. A project must make these trade-offs throughout its development cycle. As such, the decision to incur technical debt should involve input from both technical and non-technical stakeholders.

G2: Technical debt is unavoidable. Without it, a software project progresses too slowly to keep up with its competition. Project teams should not avoid technical debt; they should manage it.

G3: Technical debt can create opportunities or risks. When a project incurs technical debt as a result of making good decisions, it creates *opportunities* for the project. When a project incurs technical as a result of making poor decisions, it creates *risks* for the project.

G4: Project teams incur technical debt to *win* (i.e. being first to market, capturing a large market share). Project teams use technical debt to mitigate against tangential risks, which include time and budget constraints, changing or unclear requirements and uncertainty over the market direction. These tangential risks affect a project's chances of winning.

G5: Developers are more reluctant to incur technical debt than project managers because of differing goals. Project management is more motivated to incur technical debt because they want to meet their business targets whereas development is less motivated because they have to maintain the code on a daily basis.

G6: Manage technical debt like managing risks. To manage technical debt, a project needs to find the right amount of technical debt to incur, to make all project stakeholders owners in mitigating the possible failures, and to learn from past experiences.

G7: Managing technical debt successfully requires making an effort to pay back debt regularly. This avoids allowing the technical debt to accumulate out of control and eventually, overwhelm the project.

6.6 Limitations of This Study

There are several limitations to this study. The study focuses on the perspectives of software practitioners who develop application- and web-based systems. It does not include the views of software practitioners who work on safety critical systems, real-time systems, embedded firmware, mobile applications, and open source software. Furthermore, most of the participants interviewed for this study are software developers and architects; thus, this study primarily examines design debt and does not address other forms of technical debt, such as testing, documentation, defect debt. Additionally, the majority of participants interviewed for this study are male, so the study was unable to determine if gender played a factor in how technical debt is perceived.

From a methodological point-of-view, approximately 50% of the interview participants in the primary study worked for the same organization. All of these interview participants had worked together on the same project at one point, although, at the time of the interview, they were working on a number of different projects. In addition, the researcher was also an employee at the same organization. Secondly, the analysis of the data was conducted by a single researcher who is new to qualitative research. As a result, the analysis may be biased by the researcher's perspective and may have limited her ability to search for additional data. Thirdly, the data collected from playing "Hard Choices" omitted to include data from additional session that were conducted by other researchers in the Communicating the Benefits of Architecting Within Agile Development project group, which could affect the comprehensiveness of the findings of this study. Lastly, the findings of both studies were collected from interviews, discussion groups and feedback. As a result, the findings are subjected to the opinions of the participants and rely on the interpretation of the researcher. Moreover, the accuracy of the comments from the discussion group and feedback from the secondary study are limited by the researcher's memory and her ability to accurately record the comments.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis identified guidelines to help software practitioners recognize and manage technical debt. The findings from a series of one-hour long interviews with nineteen software practitioners in industry and from playing the game "Hard Choices" with three groups of software practitioners revealed a set of guidelines that characterize technical debt. This chapter summarizes the research contributions of this thesis and makes suggestions for future work.

7.1 Research Goals Summary

Very little academic research has been completed to study technical debt. The existing literature on the subject comes primarily from contributions of the software development community through blogs and on-line discussions. The goal of this research is to identify the characteristics of technical debt and the strategies for managing it. To achieve this goal, this thesis aimed to answer the following questions:

- How do software practitioners define and characterize technical debt?
- What are the motivations for incurring technical debt on a software project?
- What are the strategies for managing technical debt on a software project?

The research is based on applying qualitative research techniques to analyze the experiences and lessons learned of software practitioners from industry. The findings from two studies involving software practitioners from industry largely correlated with each other and with the information found in the literature review. Moreover, the findings corroborated with the results of a replication study. The outcome of this research resulted in a set of guidelines that characterize technical debt, which software practitioners can use to help them recognize and manage technical debt on their projects.

7.2 Contributions of This Work

My contributions in this thesis:

- Surveyed and reviewed the existing literature on technical debt to examine how it is currently perceived by the software development community;
- Developed an interview process using qualitative research to investigate how software practitioners in industry perceive technical debt. This included creating a semi-structured interview guide focused on exploring a software practitioner's insights, experiences and lessons learned with technical debt;
- Conducted a series of one-hour long interviews with nineteen software practitioners. The outcomes of the interviews were analyzed using constant comparison analysis to understand the definition, attributes, causes, symptoms and management of technical debt;
- Designed the board game, "Hard Choices" with researchers in the Communicating the Benefits of Architecting Within Agile Development project at the SEI at Carnegie Mellon University in Pittsburgh, PA to introduce players to the concept of technical debt;
- Ran a series of sessions at conferences and workshops to play "Hard Choices" with software developers, business analysts, project managers and QA testers. The comments and feedback from the debrief session were examined to validate the findings from the interviews;

- Developed a set of guidelines based on the findings of both the interview and game studies that aim to help software practitioners recognize and manage their technical debt on future software projects. These guidelines characterize technical debt as unavoidable trade-offs that can create opportunities or risks for a software project. Project teams can successfully manage their technical debt by openly communicating its existence and its impact with all of the project stakeholders so that everyone becomes an owner in mitigating its risks. Furthermore, project teams should regularly and actively make the effort to pay back their technical debt to avoid becoming overwhelmed by it.

The work in this thesis is validated by a replication study conducted by Nitin Taksande [46] at the University of Maryland, Baltimore County. Furthermore, the outcomes of this research have resulted in a number of publications [52, 57-59] (see Appendix A), including a recently accepted paper for an upcoming special issue of *IEEE Software* on technical debt [59]. The paper explains that technical debt is a large and complex balancing act between various short-term and long-term concerns using the outcomes from the interview study described in this thesis and from Taksande's replication study. It describes the interview study methodology using information from Chapter 3 of this thesis. Moreover, the findings are based on comparing and merging the results from our study (Chapter 4) and from Taksande's study and identifying those that had the most support from both interview data sets.

7.3 Future Work

Future work in this area of research includes:

- Conducting additional studies to address the limitations of this study and to further validate and verify the findings described in this thesis;
- Investigating how business practitioners view technical debt and assessing the similarities and differences with how software practitioners view technical debt;
- Developing better tools to measure and track the technical debt on a software project. Currently, tools like Sonar's technical debt plug-in, CAST proprietary software and SIG/TÜiT's software quality assessment method have developed proprietary techniques for measuring and tracking technical debt. However, their approaches focus on evaluating the amount of technical debt in a software project through code metrics (e.g. lines of code, code complexity, duplication, code coverage). My findings suggest that business issues such as growth, consumer responsiveness, and market trends also contribute towards the "value" of technical debt in a software project. Therefore, an area of investigation is to examine techniques that model the amount and value of technical debt relative to all of the possible changes to the software and its impact on the technical and business aspects of the project;
- Investigating strategies to manage technical debt. One possible approach is to leverage the financial undertones of the metaphor and develop strategies that are grounded in financial and economic theories, such as net present value (NPV), real options, opportunity costs, and return on investment (ROI). Guo and Seaman have already begun work in this area by using portfolio management theory [29].

7.4 Conclusion

Like any business, software development is driven by the need to capture market share through satisfying customer needs and defeating competitors. These business realities make it impossible for software projects to survive without incurring technical debt. Thus, software project teams should not avoid incurring technical debt even though they will need to make compromises on their technical quality. Instead, software project teams should use technical debt as a strategic tool to help them create opportunities to achieve their business goals. The key to successfully using technical debt lies in the software project team's ability to manage it. Managing technical debt requires effective communication skills because technical debt is neither easily seen nor understood by non-technical stakeholders, yet it relies on having all of the project's stakeholders take ownership in mitigating its risks. As a result, this thesis aims to identify a set of guidelines to help software practitioners recognize and manage technical debt on their project. The contributions of this thesis have expanded the existing knowledge, largely captured on blogs and on-line discussion groups, to include the insights, experience and lessons learned of software practitioners in industry. Nevertheless, there is still a large amount of future work to be done in this subject area to develop strategies and tools that can help project teams successfully manage their technical debt.

REFERENCES

1. Cunningham, W. *The WyCash Portfolio Management System*. OOPSLA '92 Experience Report [Wiki Article] 1992 March 26 [cited 2010 February 3]; Available from: http://c2.com/doc/oopsla92.html.

2. Cunningham, W. *Ward Explains Debt Metaphor*. [Wiki Article] 2009 September 5 [cited 2010 February 3]; Available from: http://c2.com/cgi/wiki?WardExplainsDebtMetaphor.

3. McKay, J. *When technical debt becomes technical bankruptcy*. james mckay dot net [Blog] 2009 April 6 [cited 2010 February 11]; Available from: http://jamesmckay.net/2009/04/when-technical-debt-becomes-technical-bankruptcy/.

4. McConnell, S. *Managing Technical Debt*. Best Practices White Paper [PDF] 2008 June 1 [cited 2010 November 29]; Available from: http://www.construx.com/Page.aspx?cid=2801.

5. Berteig, M. *Technical Debt*. Agile Advice [Blog] 2006 [cited 2010 February 9]; Available from: http://www.agileadvice.com/archives/2006/12/technical_debt.html.

6. Fowler, M. *Technical Debt Quadrant*. Martin Fowler [Blog] 2009 October 14 [cited 2010 February 3]; Available from: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html.

7. Fowler, M. *Technical Debt*. Martin Fowler [Blog] 2009 February 26 [cited 2010 February 3]; Available from: http://www.martinfowler.com/bliki/TechnicalDebt.html.

8. Churchville, D. *Technical Debt: Deficit Spending for Geeks*. Agile Project Planning [Blog] 2005 March 20 [cited 2010 February 9]; Available from: http://www.extremeplanner.com/blog/2005/03/technical-debt-deficit-spending-for.html.

9. Marick, B. *Technical debt: debt and history*. Exploration Through Example [Blog] 2008 June 22 [cited 2010 February 9]; Available from: http://www.exampler.com/blog/2008/06/22/technical-debt-paying-it-down/.

10. Theodoropoulos, T. *Technical Debt - Part 1: Definition*. Acrowire Blog [Blog] 2010 November 12 [cited 2011 March 11]; Available from: http://blog.acrowire.com/technical-debt/technical-debt-part-1-definition/.

11. Cartwright, I. *Five kinds of technical debt*. what i still don't know [Blog] 2009 January 5 [cited 2010 February 10]; Available from: http://blog.iancartwright.com/2009/01/five-kinds-of-technical-debt.html.

12. Martin, R. *A Mess is not a Technical Debt.* [Blog] 2009 September 22 [cited 2010 February 3]; Available from: http://blog.objectmentor.com/articles/2009/09/22/a-mess-is-not-a-technical-debt.

13. *Technical Debt*. [Wikipedia] 2011 May 1 [cited 2011 June 5]; Available from: http://en.wikipedia.org/wiki/Technical_debt.

14. Webb, D. *How to Avoid Technical Debt and Increase Systems Flexibility*. Application Development [Article] 2008 March 20 [cited 2010 February 10]; Available from: http://www.eweek.com/c/a/Application-Development/How-to-Avoid-Technical-Debt-and-Increase-Systems-Flexibility/.

15. Dyson, P. *Technical Debt and the Lean Startup*. Paul Dyson's Blog [Blog] 2011 August 15 [cited 2011 November 12]; Available from: http://pauldyson.wordpress.com/2011/08/15/technical-debt-and-the-lean-startup/.

16. Dyson, P. *An ecomonic model of technical debt?* Paul Dyson's Blog [Blog] 2011 [cited; Available from: http://pauldyson.wordpress.com/2011/08/18/an-economic-model-of-technical-debt/.

17. Savage, B. *Paying Down Technical Debt*. BrandonSavage.net [Blog] 2009 March 23 [cited 2010 February 3]; Available from: http://www.brandonsavage.net/paying-down-technical-debt/.

18. Casey, K. *Managing Technical Debt*. Keith Casey's blog [Blog] 2009 March 2 [cited 2010 February 3]; Available from: http://caseysoftware.com/blog/technical-debt.

19. Tosi, J. *Technical Debt and the Boogie Monster*. Agile Management Blog [Blog] 2010 July 19 [cited 2011 September 5]; Available from: http://blogs.versionone.com/agile_management/2010/07/19/technical-debt-and-the-boogie-monster/.

20. Ries, E. *Embrace Technical Debt*. Lessons Learned [Blog] 2009 July 29 [cited 2010 February 3]; Available from: http://www.startuplessonslearned.com/2009/07/embrace-technical-debt.html.

21. Gat, I., *Opening Statement.* Cutter IT Journal, 2010. **23**(10): p. 3-10.

22. Chin, S., et al., *The Economics of Tecnical Debt.* Cutter IT Journal, 2010. **23**(10): p. 11-18.

23. Gat, I. *Technical Debt*. SPaMCAST 112 [Podcast] 2010 December 12 [cited 2011 March 24]; Available from: http://www.spamcast.libsyn.com/s-pa-mcast-112-israel-gat-technical-debt.

24. Highsmith, J. *The Financial Implications of Technical Debt*. Jim Highsmith.com [Blog] 2010 October 19 [cited 2011 March 24]; Available from: http://www.jimhighsmith.com/2010/10/19/the-financial-implications-of-technical-debt/.

25. Steve. *Technical Debt: The Threshold of Acceptable Pain*. Technoetic [Blog] 2006 September 19 [cited 2010 February 10]; Available from: http://blog.technoetic.com/2006/09/19/threshold-of-pain/.

26. Sterling, C., *Managing Software Debt: Building for Inevitable Change*. 2011, Boston, MA: Pearson Education Inc. 244.

27. *Technical Debt Assessment and Valuation*. 2011 [cited; Available from: http://www.cutter.com/consulting-and-training/technical-debt-assessment.html.

28. Barton, B. and C. Sterling, *Managing Project Portfolios More Effectively by Including Software Debt in the Decision Process.* Cutter IT Journal, 2010. **23**(10): p. 19-24.

29. Guo, Y. and C.B. Seaman, *A portfolio approach to technical debt management*, in *Second International Workshop on Managing Technical Debt (MTD 2011)*. 2011, ACM: Waikiki, Honolulu, HI, USA. p. 31-34.

30. Hilton, R. *When To Work on Technical Debt*. Absolutely No Machete Juggling [Blog] 2011 July 22 [cited 2011 November 12]; Available from: http://www.nomachetejuggling.com/2011/07/22/when-to-work-on-technical-debt/.

31. Laribee, D. *Using Agile Techniques to Pay Back Technical Debt*. MSDN Magazine [Article] 2009 [cited 2011 February 23]; Available from: http://msdn.microsoft.com/en-us/magazine/ee819135.aspx.

32. Brown, K. *Paying back technical debt*. Comment lines by Kyle Brown [Blog] 2010 [cited; Available from: http://www.ibm.com/developerworks/websphere/techjournal/1001_col_brown/1001_col_brown.html.

33. neilj. *How I Manage Technical Debt*. Fragile [Blog] 2011 January 8 [cited February 25 2011]; Available from: http://fragile.org.uk/2011/01/how-i-manage-technical-debt/.

34. Ropa, S. *Manging Technical Debt*. Agile Management Blog [Blog] 2011 July 11 [cited 2011 November 12]; Available from: http://blogs.versionone.com/agile_management/2011/07/11/managing-technical-debt/?mkt_tok=3RkMMJWWfF9wsRoguqzJZKXonjHpfsX77u0sXbHr08Yy0EZ5VunJEUWy3YADT9QhcOuuEwcWGog81glKCemaco9O%2Fw%3D%3D.

35. Theodoropoulos, T. *Technical Debt - Part 2: Identification*. Acrowire Blog [Blog] 2010 December 6 [cited 2011 March 11]; Available from: http://blog.acrowire.com/technical-debt/technical-debt-part-2-identification/.

36. Kruchten, P. *What Colours is your Backlog? (revisited)*. Agile Vancouver Conference [Powerpoint Slides] 2009 November 3 [cited 2009 November 3]; Available from: http://philippe.kruchten.com/talks/.

37. CAST. *How to Monetize Application Technical Debt*. [PDF] 2011 [cited 2011 March 8]; Available from: http://www.castsoftware.com/campaigns/how-to-monetize-application-technical-debt?gad=otd.

38. Nugroho, A., J. Visser, and T. Kuipers, *An Empirical Model of Technical Debt and Interest*, in *Proceeding of the 2nd Working on Managing Technical Debt*. 2011, ACM: Waikiki, Honolulu, HI, USA.

39. Lehman, M.M., *Laws of Software Evolution Revisited*, in *Fifth European Workshop on Software Process Technology (EWSPT '96)*. 1996, Springer-Verlag: Nancy, France. p. 108-124.

40. Lehman, M.M., et al., *Metrics and Laws of Software Evolution - The Nineties View*, in *Fourth International Symposium on Software Metrics (METRICS '97)*. 1997: Albuquerque, NM. p. 20-32.

41. Perry, D.E. and A.L. Wolf, *Foundations for the Study of Software Architecture.* ACM SIGSOFT Software Engineering Notes, 1992. **17**(4): p. 40-52.

42. van Gurp, J. and J. Bosch, *Design Erosion: Problems and Causes.* Journal of Systems and Software, 2002. **61**(2): p. 105-119.

43. Eick, S.G., et al., *Does Code Decay? Assessing the Evidence from Change Management Data.* IEEE Transactions on Software Engineering, 2001. **27**(1): p. 1-12.

44. Parnas, D.L., *Software Aging*, in *Sixteenth international Conference on Software Engineering (ICSE '94)*. 1994, IEEE Computer Society Press: Sorrento, Italy. p. 279-287.

45. Klinger, T., et al., *An Enterprise Perspective on Technical Debt*, in *Second International Workshop on Managing Technical Debt (MTD 2011)*. 2011, ACM: Waikiki, Honolulu, HI, USA. p. 35-38.

46. Taksande, N., *Empirical Study on Technical Debt as Viewed by Software Practitioners*, Master's Thesis, 2011. Department of Information Systems, University of Maryland, Baltimore County, 107.

47. Hancock, B. *Trent Focus for Research and Development in Primary Health Care: An Introduction to Qualitative Research*. [PDF] 1998 [cited 2012 March 15]; 1-31]. Available from: http://faculty.cbu.ca/pmacintyre/course_pages/MBA603/MBA603_files/IntroQualitativeResearch.pdf.

48. Taylor-Powell, E. and M. Renner. *Analyzing Qualitative Data*. [PDF] 2003 [cited 2012 March 15]; Available from: http://learningstore.uwex.edu/assets/pdfs/g3658-12.pdf.

49. Knutson, C.D., et al., *Report from the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER 2010).* SIGSOFT Software Engineering NOTES, 2010. **35**(5): p. 42-44.

50. Shull, F., et al., *Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem*, in *International Symposium on Empirical Software Engineering (ISESE '02)*. 2002, IEEE Computer Society: Nara, Japan. p. 7-16.

51. Mantyla, M., C. Lanssenius, and J. Vanhanen. *Rethinking Replication of Software Engineering: Can We See the Forest for the Trees?* in *First International Workshop on Replication in Empirical Software Engineering Research (RESER 2010)*. 2010. Cape Town, South Africa.

52. Brown, N., et al. *The Hard Choices Game Explained*. [PDF] 2010 May [cited 2010 May]; Available from: http://www.sei.cmu.edu/library/abstracts/whitepapers/hard-choices-game-explained-v1-0.cfm.

53. Lim, E. *From The Trenches: Hard Choices Game*. SATURN Network Blog [Blog] 2010 March 10 [cited 2010 March 10]; Available from: http://saturnnetwork.wordpress.com/2010/03/10/from-the-trenches-hard-choices-game/.

54. Hendrickson, E., H. Johnson, and C. Sims, *Creating Agile Learning Games for Coaches & Consultants*. 2009. p. 1-16.

55. Hendrickson, E. *Short Cut*. [Board Game] 2004-2010 [cited 2012 February 25]; Available from: http://agilistry.com/resources/shortcut/.

56. Ambler, S.W. *The "Broken Iron Triangle" Software Development Anti-pattern*. Ambysoft [Article] 2006-2009 [cited 2009 November 30]; Available from: http://www.ambysoft.com/essays/brokenTriangle.html.

57. Brown, N., et al. *Managing Technical Debt in Software-Reliant Systems*. in *FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. 2010. Santa Fe, New Mexico.

58. Nord, R., et al., *Hard Choice: A game for balancing strategy for agility*, in *Software Engineering Education and Training (CSEE&T)*. 2011, IEEE Computer Society: Honolulu, HI, USA. p. 553.

59. Lim, E., N. Taksande, and C. Seaman, *A Balancing Act: What Software Practitioners Have to Say about Technical Debt.* IEEE Software, 2012. **29**(6).

APPENDIX A – PUBLICATIONS

Four publications resulted from our research:

1. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C. B., Sullivan, K., and Zazworka, N., "Managing Technical Debt in Software-Reliant Systems," Paper presented at the *Future of Software Engineering Research (FoSER 2010) Workshop,* part of FSE 2010, Santa Fe, New Mexico, USA, 2010, ACM, pp.47-52. DOI: 10.1145/1882362.1882373

   This position paper was submitted to a one-time international working conference focused on exploring the future of software engineering research (part of the 8th International Symposium on the Foundations of Software Engineering). The paper presented the results of the "Technical Debt Workshop" held June 2-3, 2010 at the Software Engineering Institute (SEI).

2. Brown, N., Kruchten, P., Lim, E., Ozkaya, I., and Nord, R., "The Hard Choices Game Explained," Pittsburgh: Software Engineering Institute, 2010. Available at: http://www.sei.cmu.edu/library/abstracts/whitepapers/hard-choices-game-explained-v1-0.cfm

   This SEI whitepaper provides a complete definition of the board game, "Hard Choices". The board game is made available under the Creative Commons license BY-NC-SA. "Hard Choices" (board game, playing pieces, instructions) can be downloaded from: http://www.sei.cmu.edu/architecture/tools/hardchoices/

3. Brown, N., Nord, R., Ozkaya, I., Kruchten, P., and Lim, E., "Hard Choice: A game for balancing strategy for agility," presented at the *24th IEEE Computer Society Conference on Software Engineering Education and Training (CSEE&T 2011),* Honolulu, HI, USA, 2011, IEEE Computer Society. DOI: 10.1109/CSEET.2011.5876149

   This poster was submitted to an IEEE conference, presenting the "Hard Choices" board game as an educational support tool (see Appendix C).

4. Lim, E., Taksande, N., and Seaman, C. B., "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software*, vol. 29 (6), 2012 (to appear).

   This paper has been accepted for a special issue of *IEEE Software* on technical debt. It explains that technical debt is a large and complex balancing act between various short-term and long-term concerns using the outcomes of the interview study conducted for our research and for Taksande's replication study. The paper describes the interview study methodology using information from Chapter 3 of this thesis. Moreover, the findings are based on comparing and merging the results from our study (Chapter 4) and from Taksande's study and identifying those that had the most support from both interview data sets.

APPENDIX B – INTERVIEW GUIDE

Ward Cunningham termed "technical debt" as a metaphor for representing the consequences of making the trade-off between designing/implementing a fully-architected, sustainable software system and taking shortcuts to deliver the system quickly to the customer. The purpose of this interview is to identify how projects characterize, decide to incur and manage technical debt.

1. **Context**
   - 1.1. Describe yourself:
     - 1.1.1. How many years of experience do you have?
     - 1.1.2. What is your role on the project?
     - 1.1.3. What are your responsibilities?
     - 1.1.4. How many years have you worked on the project?

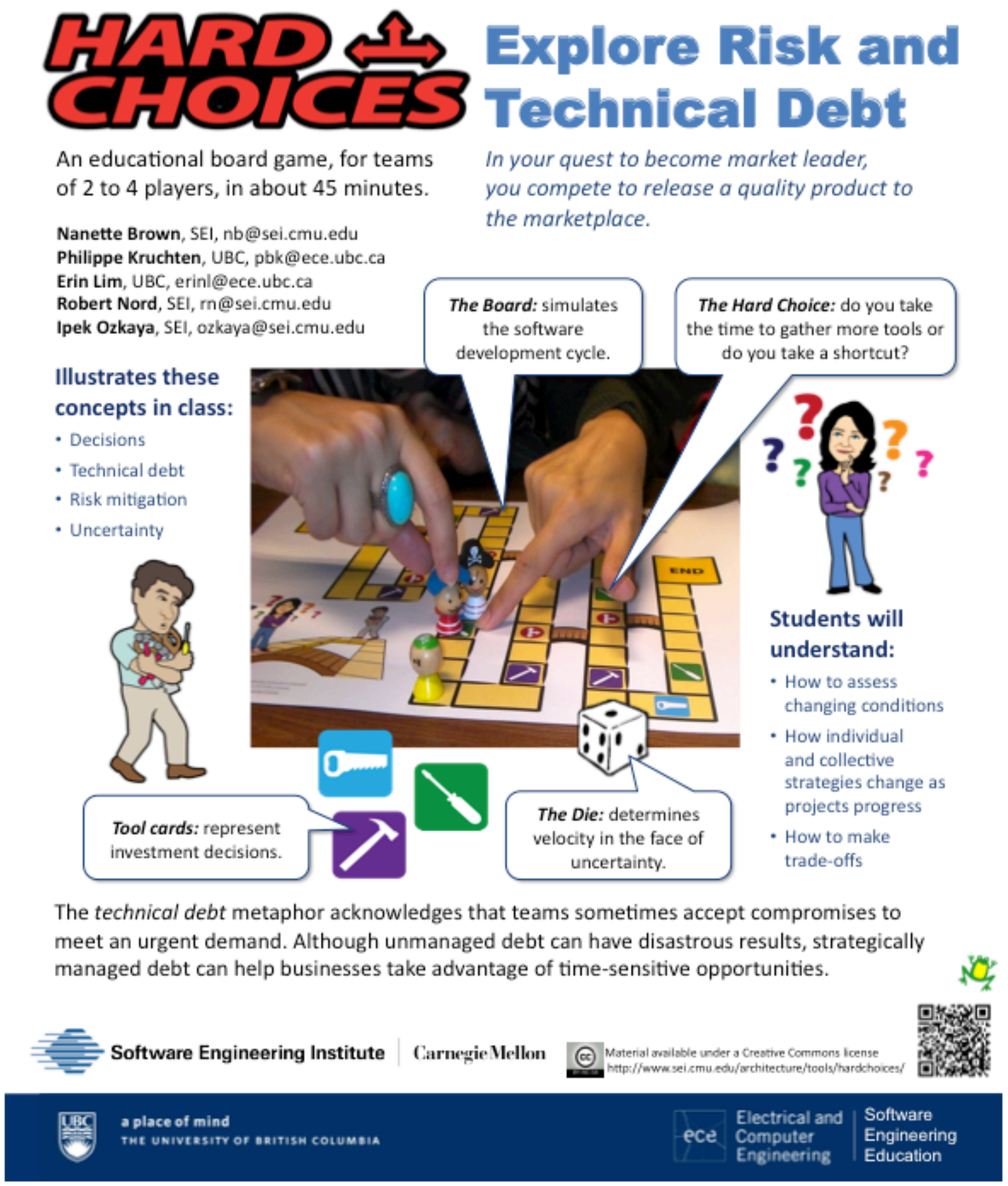
   - 1.2. Describe your project:
     - 1.2.1. What does the system do?
     - 1.2.2. How old is the system?
     - 1.2.3. Who are your customers (market segment)?
     - 1.2.4. What is the software development process on the project?
     - 1.2.5. What is the release planning process on the project?

2. **Technical Debt**
   - 2.1. Describe an example of technical debt on your project.
     - 2.1.1. What happened?
     - 2.1.2. Why do you consider it to be "technical debt"?
     - 2.1.3. Did you plan to incur the "technical debt"?
       - 2.1.3.1. If yes…
         - 2.1.3.1.1. Who made the decision to incur the debt?
         - 2.1.3.1.2. When did you decide to incur the debt?
         - 2.1.3.1.3. Why did you decide to incur the debt?
         - 2.1.3.1.4. How did you decide to incur the debt?
       - 2.1.3.2. If no…
         - 2.1.3.2.1. Who identified that this is a debt?
         - 2.1.3.2.2. When did you discover the debt?
         - 2.1.3.2.3. Why did you decide that is a debt?
         - 2.1.3.2.4. How did you discover the debt?
     - 2.1.4. What is the impact of the technical debt on your project…
       - 2.1.4.1. For the customers?
       - 2.1.4.2. For development?
       - 2.1.4.3. For future modifications to the system?
     - 2.1.5. Do you know how much of the system is affected by the debt? If so, how do you measure the size of the debt?
     - 2.1.6. How are you managing the debt? Are there any plans to pay off the debt?

2.1.7. How do you communicate the debt to your customers? Your managers? Your developers?
2.1.8. What are the benefits of incurring the debt?
2.1.9. What are the consequences of incurring the debt?
2.1.10. Was incurring the debt worthwhile? Would you do it again?
2.1.11. What did you learn from this experience? What would you do the same/differently?

Are there any other thoughts, comments, suggestions, and/or lessons learned from your experiences with technical debt that you would like to share?

APPENDIX C – HARD CHOICES POSTER



HARD CHOICES

Explore Risk and Technical Debt

An educational board game, for teams of 2 to 4 players, in about 45 minutes.

In your quest to become market leader, you compete to release a quality product to the marketplace.

Nanette Brown, SEI, nb@sei.cmu.edu
Philippe Kruchten, UBC, pbk@ece.ubc.ca
Erin Lim, UBC, erinl@ece.ubc.ca
Robert Nord, SEI, rn@sei.cmu.edu
Ipek Ozkaya, SEI, ozkaya@sei.cmu.edu

Illustrates these concepts in class:

- Decisions
- Technical debt
- Risk mitigation
- Uncertainty

The Board: simulates the software development cycle.

The Hard Choice: do you take the time to gather more tools or do you take a shortcut?

Tool cards: represent investment decisions.

The Die: determines velocity in the face of uncertainty.

Students will understand:

- How to assess changing conditions
- How individual and collective strategies change as projects progress
- How to make trade-offs

The *technical debt* metaphor acknowledges that teams sometimes accept compromises to meet an urgent demand. Although unmanaged debt can have disastrous results, strategically managed debt can help businesses take advantage of time-sensitive opportunities.

Software Engineering Institute | Carnegie Mellon

UBC a place of mind THE UNIVERSITY OF BRITISH COLUMBIA

ece Electrical and Computer Engineering | Software Engineering Education