Georgia State University

ScholarWorks @ Georgia State University

Computer Information Systems Dissertations | Department of Computer Information Systems

12-16-2020

Managing Technical Debt in Agile Software Development Projects

Maheshwar Boodraj
Georgia State University

MANAGING TECHNICAL DEBT IN AGILE SOFTWARE DEVELOPMENT PROJECTS

BY

MAHESHWAR BOODRAJ

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree

Of

Doctor of Philosophy

In the Robinson College of Business

Of

Georgia State University

GEORGIA STATE UNIVERSITY
ROBINSON COLLEGE OF BUSINESS
2020

ACCEPTANCE

This dissertation was prepared under the direction of the Maheshwar Boodraj Dissertation Committee. It has been approved and accepted by all members of that committee, and it has been accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Business Administration in the J. Mack Robinson College of Business of Georgia State University.

Richard Phillips, Dean

DISSERTATION COMMITTEE

Dr. Mark Keil (Chair)
Dr. Lars Mathiassen
Dr. Likoebe Maruping
Dr. Narayan Ramasubbu
Dr. Yolande Chan

ABSTRACT

MANAGING TECHNICAL DEBT IN AGILE SOFTWARE DEVELOPMENT PROJECTS

BY

MAHESHWAR BOODRAJ

December 1, 2020

Committee Chair: Dr. Mark Keil

Major Academic Unit: Computer Information Systems

One of the key reasons that agile software development methods have gained popularity in recent years is because they enable organizations to produce software quickly to meet the needs of various stakeholders. However, this focus on delivering software quickly often encourages practitioners to incur *technical debt* – design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Worldwide, technical debt is estimated to be a trillion-dollar problem. This has prompted significant interest from both researchers and practitioners. In this dissertation, I present two essays that advance our knowledge of the causes of technical debt in agile software development projects and that offer potential solutions to manage the most important of these causes of technical debt. In my first essay, I conduct a ranking-type Delphi study of information technology (IT) project managers and software developers to identify and prioritize the most important causes of technical debt in agile software development projects. The findings from this study provide a verified list of 55 causes of technical debt in agile software development projects and offer 13 potential techniques to manage the causes of technical debt that were most important to the IT project managers and software developers in this study. In my second essay, I conduct a randomized experiment to examine the impact of software developers' *construal level*, a cognitive process, on the unintentional accumulation of technical debt in software development projects. The findings from this experiment suggest that software developers at a high construal level are more likely to focus on developing the architecture or design than software developers at a low construal level. Collectively, the findings from these two essays deepen our understanding of the intentional and unintentional causes of technical debt in agile software development projects. Further, the findings offer potential techniques to manage the most important causes of technical debt for IT project managers and software developers.

DEDICATION

To my wife Truanna and our sons Alexander and Benjamin, who sacrificed so much so that I could pursue my dream of obtaining a Ph.D. from Georgia State University.

ACKNOWLEDGEMENTS

First, I would like to thank the members of my dissertation committee: Dr. Mark Keil (Chair), Dr. Lars Mathiassen, Dr. Likoebe Maruping, Dr. Narayan Ramasubbu, and Dr. Yolande Chan. It has been the privilege of my lifetime to work with and learn from Dr. Keil. Words cannot adequately express my gratitude to Dr. Keil for accepting me into the Ph.D. program at Georgia State University, for serving as my advisor for the past five years, for pushing me when I needed it, and for supporting me at every stage of the Ph.D. program.

My internal committee members, Dr. Mathiassen and Dr. Maruping, have profoundly changed my thinking both in and out of the classroom. Dr. Mathiassen gave me a deep appreciation for action research and for doing work that is relevant to practice, and Dr. Maruping introduced me to digitally-enabled collectives and multi-level analysis, which have sparked several ideas for future research. In addition, Dr. Mathiassen and Dr. Maruping generously wrote numerous letters of recommendation to support a successful job search.

My external committee members, Dr. Ramasubbu and Dr. Chan, provided many insightful suggestions on how to improve my dissertation. As an expert on technical debt, Dr. Ramasubbu directed me to the classical literature in this space and helped me to better position my work as an information systems researcher. As an expert on qualitative research, Dr. Chan helped me to get the most out of the participant interviews in my first essay. In addition, Dr. Ramasubbu and Dr. Chan met with me several times along the way and provided invaluable career advice.

Second, I would like to acknowledge the generous support of the Project Management Institute (PMI), which awarded me with a *Thesis Research Grant*, without which this dissertation research would not have been possible. I would also like to thank the many participants, who gave so generously of their time and expertise.

Third, I would like to thank my fellow doctoral students at Georgia State University for making the journey so much more enjoyable. I would particularly like to thank my good friends and co-authors, Dr. Kambiz Saffarizadeh and Dr. Tawfiq Alashoor, for the many conversations we have had about life and research methods. I would also like to thank Dr. Amrita George, Dr. Christine Abdalla Mikhaeil, and Mrs. Yukun Yang for co-authoring with me and for being a source of support and encouragement.

Last, and in no way least, I would like to thank my entire family. I am especially grateful to Mr. Collins and Mrs. Jennifer Forman for supporting my family and me in more ways than I can mention. I am also thankful to my favorite cousin, Miss Amanda Forman, for the motivating postcards and paintings from Oxford. Of course, I would not be on this path if it were not for my parents, Dr. Girjanauth and Mrs. Neermattie Boodraj, who are both educators. They filled my childhood home with love and books, drove me to and from numerous extracurricular classes, and fostered my curiosity in all things' science and technology – even after I permanently disassembled our only radio.

Table of Contents

List of Tables

List of Figures

1 Introduction

Every year, the Project Management Institute (PMI) – the world's leading project management organization – conducts a global survey of project, program, and portfolio managers to identify the major trends in project management. In 2018, the results from this global survey revealed that almost one in four projects completed within the prior year used agile methods, and a similar number used hybrid methods that included agile elements (Project Management Institute, 2018). Agile methods are particularly attractive for several reasons: they enable project teams to develop products quickly, they enable developers to elicit early feedback to better meet customer requirements, and they enable organizations to more readily respond to today's constantly changing marketplace (Beck et al., 2001; Maruping, Venkatesh, & Agarwal, 2009), among other benefits.

Unfortunately, in agile software development projects, this focus on delivering software quickly often encourages practitioners to incur *technical debt* – design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible (Avgeriou, Kruchten, Ozkaya, & Seaman, 2016; McConnell, 2008). Incurring technical debt is not necessarily a problem if the debt is repaid promptly or if the system is designed for short-term use. However, when this debt is not repaid promptly, or a system remains in use long after its intended lifetime, there can be several undesirable consequences. For example, too much technical debt in a software application can increase the difficulty and cost to add new functionality and maintain the existing functionality (Brown et al., 2010).

Technical debt is a real and pressing business challenge (Brown et al., 2010; Kruchten, 2019). One study provides a conservative estimate of $361,000 of technical debt for every 100,000 lines of code in a typical software application (Curtis, Sappidi, & Szynkarski, 2012). Another study provides an estimate of $1 trillion of technical debt in the global maintenance backlogs for information technology software (Kruchten, Nord, & Ozkaya, 2019; Kyte, 2010). While research on managing technical debt has steadily advanced, it is far from complete, which presents a timely opportunity for me to contribute to the academic discourse on managing technical debt in agile software development projects.

To be able to manage technical debt effectively, one of the first and most important steps is to identify the causes of technical debt (Kruchten, Nord, & Ozkaya, 2012). However, to the best of my knowledge, there has been no systematic attempt to identify and prioritize the causes of technical debt in agile software development projects. Consequently, this dissertation includes two essays to advance our knowledge in this area. Further, in this dissertation, I suggest potential techniques for managing the causes of technical debt that are most important to IT project managers and software developers. In sum, the essays in this dissertation address the following two overarching research questions:

1. *What are the most important causes of technical debt in agile software development projects?*
2. *What are some potential techniques for managing technical debt in agile software development projects?*

In my first essay, I conduct a Delphi study of experienced IT project managers and software developers to identify and prioritize the most important causes of technical debt in agile software development projects. In my second essay, I conduct an experiment to examine the role of software developers' *construal level* on the unintentional accumulation of technical debt in software development projects. Construal level refers to the degree to which we perceive an object as being psychologically distant (Trope & Liberman, 2010). This psychological distance can take the form of temporal distance, spatial distance, social distance, or hypothetical distance (Trope & Liberman, 2010).

I studied construal level for several reasons. First, I believe that this cognitive process is the root cause of other more conspicuous factors that contribute to the accumulation of technical debt in software development projects (e.g., inadequate code review). Second, to the best of my knowledge, the impact of our cognitive processes on the accumulation of technical debt, though important, is not well-understood. Third, establishing a causal link between our cognitive processes and the accumulation of technical debt would open the door for well-known techniques to manipulate construal level to serve as potential techniques for mitigating technical debt in software development projects.

To address my research questions, I used the Delphi method and the experimental method. In my first essay, I use the Delphi method following the approach used by Schmidt (1997) to solicit expert opinions on the most important causes of technical debt. This approach is supported by Brown et al. (2010), who suggested that elicitation from practitioner experts is a fruitful direction to identify the relative importance of different sources of technical debt. In my second essay, I use the experimental method, which is especially powerful for making causal inferences due to its high internal validity (Shadish, Cook, & Campbell, 2002; Trochim, Donnelly, & Arora, 2016).

Through this dissertation, I make several meaningful contributions to research on and practice of managing technical debt in agile software development projects. First, I offer a verified list of 55 causes of technical debt in agile software development projects. Second, I identified and prioritized the 15 causes of technical debt that a majority of the IT project managers in this study indicated were the most important and the 12 causes of technical debt that a majority of the software developers in this study indicated were the most important. Third, by interviewing a select group of IT project managers and software developers, I identified 13 potential techniques for managing the eight causes of technical debt that were a priority for both IT project managers and software developers. Further, I offer four plausible explanations for the differences between the rankings by IT project managers and the rankings by software developers. Finally, I demonstrate that software developers' construal level can play a role in the unintentional accumulation of technical debt in software development projects. Specifically, I offer empirical evidence that software developers at a high construal level are more likely to focus on developing the architecture or design than software developers at a low construal level.

2 Essay One: A Delphi Study of IT Project Managers and Software Developers

Abstract

One of the reasons that agile software development methods such as Scrum, extreme programming, and feature-driven development are so popular is because they enable frequent software delivery. However, this focus on delivering software frequently results in an increased tendency to take shortcuts to meet ambitious requirements or aggressive deadlines. This often results in *technical debt* – design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. In this essay, I conduct a ranking-type Delphi study of experienced IT project managers and software developers to identify and prioritize the most important causes of technical debt in agile software development projects. Further, I examine how these causes of technical debt vary in importance for IT project managers and software developers. Finally, I conduct follow-up interviews with a select group of IT project managers and software developers to identify potential techniques for managing the most important causes of technical debt that were common to both groups.

Keywords: technical debt, agile software development, IT project management, Delphi study

2.1 Introduction

One of the key reasons why software development projects typically adopt agile methods such as Scrum, extreme programming, and feature-driven development is to enable frequent software delivery. While there are numerous advantages to delivering software frequently, such as capturing market share, meeting contractual obligations, and collecting early customer feedback (Lim, Taksande, & Seaman, 2012), there are also serious disadvantages. One key disadvantage is an increased tendency to take shortcuts to meet ambitious requirements or aggressive deadlines (Baham, 2017). These shortcuts often result in *technical debt* – "a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible" (Avgeriou et al., 2016, p. 112).

Technical debt is a pervasive challenge (Brown et al., 2010; Kruchten, 2019) with significant financial consequences. A typical software application with one million lines of code is estimated to have $3.61 million of technical debt (Curtis et al., 2012). Currently, the technical debt in the global maintenance backlogs for information technology software is estimated at $1 trillion (Kruchten et al., 2019; Kyte, 2010). This has prompted significant interest from both researchers and practitioners on how to manage technical debt effectively. For example, Kruchten, Nord, Ozkaya, and Falessi (2013, p. 51) argued that a "better understanding of the concept of technical debt, and how to approach it, both from a theoretical and a practical perspective is necessary to advance its state of the art and practice." Further, Alves et al. (2016, p. 118) argued that "it is necessary to conduct further studies in the area to investigate new techniques and tools that could support developers with the control of [technical debt]." According to Rolland, Mathiassen, and Rai (2018, p.

426), “interest in technical debt has recently been intensified as failure to manage it appropriately can adversely affect a software system’s long-term maintainability, evolvability, and quality.”

To be able to manage technical debt effectively, one of the first and most important steps is to identify the causes of that technical debt (Kruchten et al., 2012). While several studies have tackled the causes of technical debt in software engineering projects (Rios, de Mendonça Neto, & Spínola, 2018), few have examined the causes of technical debt in *agile* software development projects. To the best of my knowledge, none has systematically attempted to identify and prioritize the most important causes of technical debt in agile software development projects. Being able to prioritize the causes of technical debt is especially important for agile software development projects, which have to rapidly respond to short-term needs while still developing a product that can be easily maintained and evolved in the long term.

Left unmanaged, technical debt can lead to several negative consequences. For example, technical debt can reduce the pace of software development (Fowler, 2003; Letouzey & Ilkiewicz, 2012; Yang & Boodraj, 2020), cripple the ability to meet customer requirements (Kruchten et al., 2019), and increase software maintenance difficulty and costs (Bavani, 2012; Brown et al., 2010; Z. Li, Avgeriou, & Liang, 2015). Also, technical debt can affect product performance, reliability, and stability (P. Li, Maruping, & Mathiassen, 2020; Lim et al., 2012; Ramasubbu & Kemerer, 2016), which negatively impacts users of the software. Further, technical debt can increase software complexity, which makes systems rigid (hard to change), fragile (each change breaks something else), viscous (doing things right is harder), and opaque (hard to understand) (Brown et al., 2010).

Given the increasing use of agile methods and the multitude of negative consequences that can result from unmanaged technical debt, learning more about managing technical debt in agile software development is worthy of further examination. In this essay, my objective is to generate a comprehensive list of causes of technical debt in agile software development projects and offer potential techniques for managing the most important of these causes. To achieve this objective, I used a ranking-type Delphi study (Schmidt, 1997) to solicit feedback from experienced IT project managers and software developers. Including both IT project managers and software developers increased my chances of generating a comprehensive list of causes of technical debt. Further, by considering these two distinct but critical roles in the software development process, I was able to identify areas of concordance and discordance that might warrant further examination.

My findings make several novel contributions to research and practice. The first contribution is the creation of a verified list of 55 causes of technical debt in agile software development projects. The second contribution is the identification of the 15 causes of technical debt that a majority of IT project managers in the study agree are the most important and the 12 causes of technical debt that a majority of software developers in the study agree are the most important. The third contribution is the identification of four potential explanations for the differences in the causes of technical debt that IT project managers and software developers view as most important. The fourth contribution is the identification of 13 potential techniques for managing the most important causes of technical debt that were common to both IT project managers and software developers.

2.2 Background

2.2.1 Agile Software Development

Agile software development methods such as Scrum, extreme programming, and feature-driven development (Abrahamsson, Salo, Ronkainen, & Warsta, 2002) embody the principles outlined in the Agile Manifesto: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan (Beck et al., 2001). By embodying these principles, agile software development methods aim to deliver frequent releases of software functionality. In contrast, traditional software development methods, such as the waterfall method, aim to deliver software functionality all at once. To accomplish this, traditional software development methods proceed sequentially through several stages – analysis, design, development, testing, implementation, and maintenance – with the output from one stage serving as the input to the next (Balaji & Murugaiyan, 2012).

There is a natural tension between agile software development methods and traditional software development methods, which is succinctly captured in the Agile Manifesto. Specifically, while agile software development methods promote individuals and interactions, working software, customer collaboration, and responding to change, traditional software development methods promote processes and tools, comprehensive documentation, contract negotiation, and following a plan. These differences impact how technical debt is incurred for agile software development methods versus traditional software development methods. For example, while traditional software development methods necessitate that requirements be thoroughly considered before development begins, agile software development methods welcome changing requirements, which may increase the complexity of the architecture and subsequent architectural debt. Also, while traditional software development methods require documentation at the end of each stage, agile software development methods place less focus on documentation, which may increase overall documentation debt. Of course, traditional software development methods are also susceptible to technical debt. In fact, by waiting to conduct testing until the development stage is complete, many technical debt items can go unresolved when traditional software development methods are used.

Researchers have acknowledged that agile software development methods are susceptible to technical debt in different ways from traditional software development methods (Guo, Spínola, & Seaman, 2016), which make them worthy of further examination. This has prompted research focused specifically on technical debt in agile software development. For example, Behutiye, Rodríguez, Oivo, and Tosun (2017) conducted a systematic literature review of technical debt in agile software development, Holvitie et al. (2018) conducted an industry practitioner survey on technical debt and agile software development practices and processes, and Caires et al. (2018) investigated the effects of agile practices and processes on technical debt in the Brazilian software industry. Also, to promote further research in this area, the theme of the *Ninth International Workshop on Managing Technical Debt*, where the leading researchers on technical debt gather, was focused on "Technical Debt in Agile Development" (Fontana et al., 2017).

2.2.2 Technical Debt in Agile Software Development

Over the past several years, researchers have begun to synthesize the causes of technical debt in agile software development projects to advance our knowledge in this area. For example, in a recent systematic literature review of 38 primary studies, Behutiye et al. (2017) identified ten causes of technical debt: emphasis on quick delivery, architecture and design issues, inadequate test coverage, lack of understanding of the system being built or the requirements, overlooked and delayed solutions and estimates, inadequate or delayed refactoring, duplicate code, parallel development, resource constraints, and organizational gaps among business, operational, and technical stakeholders.

Since then, researchers have identified additional causes of technical debt in agile software development projects through surveys and case studies. For example, the findings from an industry practitioner survey by Holvitie et al. (2018) identified several additional causes of technical debt such as inadequate documentation, code complexity, violation of best practices or style guides, and defects or bugs while providing additional evidence to support previously-identified causes of technical debt such as poor architecture or design, requirements for new features and functions, and inadequate testing. Similarly, the findings from a case study by Bjärås and Ericsson (2018) identified additional causes of technical debt such as violation of naming conventions or design patterns and overly complex code while providing additional evidence for known causes of technical debt such as deadline pressure, duplicate code, and poor architecture or design.

While we have learned much about the causes of technical debt in agile software development from previous studies such as these, none of these studies explicitly attempted to generate a comprehensive list of causes of technical debt. Any attempt to generate a comprehensive list by including a single group of stakeholders (such as software developers) runs the risk of being incomplete. However, including multiple groups of stakeholders (such as IT project managers, software developers, customer representatives, and end-users) runs the risk of not converging on the most important causes of technical debt. Therefore, I chose to focus on the two groups of stakeholders that are most involved in the software development process: IT project managers and software developers. In an action research study of Scrum projects, Oliveira, Goldman, and Santos (2015) found that the development team and the ScrumMaster were most responsible for identifying technical debt. Therefore, by surveying IT project managers and software developers, I seek to address my first research question:

> *RQ1: What are the primary causes of technical debt in agile software development projects?*

Given that agile software development projects typically operate in resource-constrained environments where schedule and budget are limited, it is not just important to identify the primary causes of technical debt but to prioritize those causes so that organizations can make the best use of their limited resources. To the best of my knowledge, there are no previous studies that attempt to identify and prioritize the most important causes of technical debt in agile software development projects. I attempt to do this by using a ranking-type Delphi study (Schmidt, 1997), which has been successfully employed by previous researchers to achieve consensus in rankings. Therefore, using a ranking-type Delphi study, I seek to address my second and third research questions:

> *RQ2: What are the most important causes of technical debt in agile software development projects for IT project managers and software developers?*
>
> *RQ3: In what ways do the causes of technical debt in agile software development projects vary in importance for IT project managers and software developers?*

Once I have identified and prioritized the most important causes of technical debt in agile software development projects, the next logical step is to identify potential techniques for managing these causes. In their studies, Behutiye et al. (2017), Holvitie et al. (2018), and Bjärås and Ericsson (2018) identified several strategies for managing technical debt in agile software development projects: implement coding standards, analyze and refactor the code, use test-driven development and automated testing, use continuous integration, promote mutual understanding collective code ownership, plan in advance for technical debt, enhance the visibility of technical debt, prioritize technical debt, improve estimation techniques, agree on a definition of done, communicate about technical debt with business stakeholders, and establish an acceptable level of technical debt. By conducting follow-up interviews with a select group of IT project managers and software developers, I attempt to identify what techniques are perceived to be helpful for managing technical debt in agile software development projects. Therefore, using semi-structured interviews, I seek to address my fourth research question:

> *RQ4: What are some potential techniques for managing technical debt in agile software development projects?*

2.3 Methodology

To address my research questions, I solicited input from IT project managers and software developers with extensive experience working on agile software development projects. Naturally, involving many experts results in varied opinions and perspectives. However, one of the best ways to solicit expert opinion while fostering consensus is to use a Delphi study, which provides participants with feedback from previous rounds to help foster consensus in subsequent rounds.

Delphi studies have been used by information systems (IS) scholars to rigorously study a variety of problems. For example, using a Delphi study, Daniel and White (2005) explored the nature of future inter-organizational system linkages, Nevo and Chan (2007) explored the roles and scope of knowledge management systems in organizations, Kasi, Keil, Mathiassen, and Pedersen (2008) identified the most important barriers to conducting post mortem evaluations of IT projects, Keil, Lee, and Deng (2013) identified the most critical skills for managing IT projects, and Sambhara, Rai, Keil, and Kasi (2017) identified buyer and supplier perspectives on risk factors associated with internet-enabled reverse auctions.

2.3.1 Expert Panels

I recruited participants through direct emails and LinkedIn messaging. The email addresses for the direct emails were either scraped from agile certification registries or purchased from email marketing firms. After exchanging numerous emails with potential panelists, only those persons who were committed to the Delphi process and had at least five years’ experience working on agile software development projects in the United States were invited to participate. This resulted in a convenience sample of 86 experienced agile software development practitioners.

On average, participants had 8.5 years of experience working on agile software development projects. Participants were highly educated. Approximately 93% achieved undergraduate degrees, and 49% achieved graduate degrees (six had doctoral degrees). Further, approximately 79% held at least one of the following relevant certifications: Certified Scrum Developer (CSD), Certified ScrumMaster (CSM), PMI Agile Certified Practitioner (PMI-ACP), and Project Management Professional (PMP).

2.3.2 Data Collection

I collected the data for this study using a ranking-type Delphi study following guidelines proposed by Schmidt (1997). These guidelines prescribe three phases: (1) discovery of issues, (2) identification of the most important issues, and (3) ranking of the most important issues. To provide the ability to compare the responses from IT project managers and software developers, all panelists collectively participated in the first phase. However, in subsequent phases, the IT project managers and software developers participated in separate panels. *Table 2.1* provides a summary of the Delphi process, after which I offer a detailed description of each of the three phases.

Table 2.1: Summary of the Delphi Process

Phase 1: Discovery of the Issues	• Collect at least six primary causes of technical debt from each panelist. • Consolidate the causes of technical debt from all panelists. • Ask panelists to verify that the consolidated list accurately reflects their input.
Phase 2: Identification of the Most Important Issues	• Create separate panels for IT project managers and software developers. • Ask each panelist to independently identify the 20 most important causes of technical debt from the verified list. • Trim the lists for each panel to include only those causes of technical debt selected by 50% or more of the panelists on that panel.
Phase 3: Ranking of the Most Important Issues	• Ask each panelist to independently rank the causes of technical debt on the trimmed list. • Determine whether an acceptable level of consensus has been achieved for each panel. • Repeat the ranking exercise until an acceptable level of consensus has been reached or consensus levels off in two successive rounds.

In Phase 1, panelists were asked to provide an unordered list of at least six primary causes of technical debt in agile software development projects (see Appendix A). Allowing each panelist to provide six or more items increased the chances of unearthing the most important causes of technical debt (Schmidt, 1997). Panelists provided a total of 579 causes of technical debt. After removing duplicates and consolidating similar causes, I produced a consolidated list of 57 causes of technical debt. Subsequently, each panelist was asked to review the consolidated list to verify that it accurately reflected their input. After incorporating feedback from panelists, the list was further reduced to 55 causes of technical debt.

In Phase 2, two separate panels were created. The 26 participants with only IT project management experience were assigned to one panel, and the 22 participants with only software development experience were assigned to the other panel. Participants with experience in both IT project management and software development were excluded to maintain the homogeneity of the two panels. Each panelist was then asked to independently identify the top 20 causes of technical debt from the verified list of 55 items generated in Phase 1 (see Appendix B). Asking panelists to identify the top 20 items was consistent with prior studies, such as Keil et al. (2013) and Sambhara et al. (2017), and helped to generate a manageable list for further analysis. I retained the 15 items from the IT project manager panel and the 12 items from the software developer panel that were selected by 50% or more of the panelists on that panel.

In Phase 3, each panelist was asked to independently rank the most important causes of technical debt identified in Phase 2 (see Appendix C). Panelists were also asked to provide a short paragraph explaining the reason for selecting their top-ranked cause of technical debt. I then determined whether an acceptable level of consensus was achieved using Kendall's coefficient of concordance, W (Kendall & Gibbons, 1990), which is widely used for measuring agreement (Schmidt, 1997). In total, Phase 3 was repeated four times (see Appendix D) to achieve a moderate to high level of consensus (i.e., Kendall's W > 0.5).

Completing these three phases helped to identify the primary causes of technical debt in agile software development projects and to determine the ways in which these causes vary in importance between IT project managers and software developers. However, to identify strategies for managing the most important of these causes, I conducted follow-up interviews with 14 select panelists (see Appendix E). Panelists were selected based on their level of engagement throughout the Delphi process and their willingness to participate in an interview. The interviews were conducted over Zoom and lasted for approximately 50 minutes on average. The audio recordings were then transcribed using an online transcription service.

2.4 Results and Discussion

2.4.1 Complete List of Causes of Technical Debt

At the end of Phase 1, I was able to answer my first research question: *What are the primary causes of technical debt in agile software development projects?* Specifically, by surveying the panelists, I was able to generate a list of 55 causes of technical debt in agile development projects. I initially tried to sort these causes using the four major areas proposed by Kruchten et al. (2019), which were not specific to agile software development projects. These areas were *nature of the business, change in context, development process,* and *people and team*. However, some of the causes did not fit well under these areas (e.g., *code complexity* and *code duplication*). Consequently, I adapted the four major areas by Kruchten et al. (2019) and added a fifth area to develop a classification that was more appropriate for the causes of technical debt in agile software development.

The five major areas in my classification include: causes of technical debt that are external to the organization ("External"), causes of technical debt that are within the organization but outside of the team's control ("Organizational"), causes of technical debt that are at the team or individual level ("People"), causes of technical debt that are related to processes, practices, and standards ("Process"), and causes of technical debt that are related to the software product or code (“Product”). *Table 2.2* presents the complete list of causes of technical debt arranged into these five areas.

Table 2.2: Causes of Technical Debt in Agile Software Development Projects

ID	Cause of Technical Debt	Description
External		
1.1	**Lower Standards by Contractors**	Contractors adopting a lower standard than the organization that hired them.
1.2	**New or Evolving Technology**	Not keeping up with new technology or planning for the impact it can have on an application, not having the expertise to use new technology properly, or using new technology without a strong justification to do so.
1.3	**Risks or Unknowns**	Having to deal with unknown technical elements or unexpected events or conditions.
1.4	**Unowned Dependencies**	Integrating with applications or services that you have no control over. This includes changes in third-party applications or services.
Organizational		
2.1	**Business-Technology Challenges**	Includes challenges between business representatives and the technical team, such as a lack of trust and power imbalances. This also includes business representatives exerting undue influence over technical decisions and business representatives not understanding the technology or the technical implications of their decisions.
2.2	**Changing, Ill-defined, or Missing Requirements**	Includes constantly changing, poorly defined, or missing requirements. This also includes ambiguous or vague requirements, shifting priorities, and scope creep.
2.3	**Deliberate Choices**	Incurring technical debt for valid reasons such as demos or trade shows.
2.4	**High Turnover**	Includes a loss of organizational knowledge and inconsistent coding practices resulting from high turnover. This also includes a change in priorities due to leadership changes.

2.5	**Lack of Psychological Safety**	Creating an environment where teams do not feel safe discussing technical debt.
2.6	**Lack of Resources**	Includes inadequate, shared, or overcommitted human resources. This also includes insufficient financial resources and tools.
2.7	**Lack of Stakeholder Involvement**	Lack of involvement from key stakeholders such as the Product Owner or not including key stakeholders such as the testing team.
2.8	**Lack of Vision or Roadmap**	Not having a clear vision or roadmap for the application.
2.9	**Misaligned Incentives**	Using incentives that promote feature delivery over minimizing technical debt.
2.10	**Not Prioritizing Technical Debt**	Prioritizing other work such as the delivery of new features over avoiding or resolving technical debt. This pressure can come from management, leadership, customers, or business partners.
2.11	**Poor Development Infrastructure**	Not having the appropriate hardware and software to facilitate effective software development.
2.12	**Poor Organizational Learning**	Not using the feedback from one sprint or project to improve the next sprint or project.
2.13	**Poor Physical Workspace**	Working in a physical space that is not conducive to high-quality work.
2.14	**Poor Planning**	Includes lack of careful planning, poor estimation, overly ambitious plans, or over planning.
2.15	**Schedule Pressure**	Includes schedule pressure due to unrealistic deadlines, time constraints, management directives, or market competition.
2.16	**Short-Term Focus**	Prioritizing the short term without consideration for the long term.
People		
3.1	**Developers Lack of Authority***	Not empowering developers or giving them autonomy over their work.
3.2	**Inexperienced Team Members**	Using team members who are unfamiliar with relevant coding standards and best practices or who lack requisite skills and training.

3.3	**Lack of Knowledge**	Lack of knowledge stemming from team members not sharing or transferring their knowledge with new or junior team members. This also includes a lack of mentorship.
3.4	**Lack of Ownership**	Not having or wanting ownership of or the responsibility to maintain the code after it has been developed.
3.5	**Mundane Nature of Technical Debt**	Not resolving technical debt because team members find it boring and prefer to work on other things such as new enhancements.
3.6	**Not Understanding Big Picture**	Includes a lack of understanding of the end goal or the general problem.
3.7	**Poor Collaboration or Communication**	Poor collaboration or communication among team members.
3.8	**Self-Serving Motivations**	Introducing technical debt for self-serving reasons such as job security or appearing clever.
3.9	**Stressed Teams***	Pushing teams beyond a sustainable pace of delivery or requiring teams to produce something every day.
3.10	**Team Dynamics**	Making sub-optimal decisions because of issues such as groupthink and power imbalances.
3.11	**Too Many Interruptions***	Teams having to deal with too many interruptions. This includes a break in work during a sprint.
Process		
4.1	**Improper Scoping***	Not properly scoping the minimum viable product (MVP) or the requirements for a sprint.
4.2	**Inadequate Code Review**	Inadequate or no code review. This includes not identifying or removing dead code.
4.3	**Inadequate or Inappropriate Code Refactoring**	Inadequate or delayed code refactoring. This includes premature code optimization.
4.4	**Inadequate Testing or Quality Assurance**	Includes inadequate testing, lack of automated testing, not updating tests, not testing edge cases, and not using test-driven development. This also includes poor quality assurance.
4.5	**Lack of Coding Standards**	Includes not having clearly defined standards, adhering to existing standards, or refactoring code written before standardization. This also includes a lack of modularity and hard coding.

4.6	**Lack of Comments or Documentation**	Includes a lack of comments in the code as well as poor or missing documentation such as design documents and technical specifications.
4.7	**Lack of Continuous Integration**	Not detecting quality issues early and often.
4.8	**Lack of Focus on Non-Functional Requirements**	Not focusing on non-functional requirements such as usability, reliability, scalability, performance, and security. This includes not complying with established security standards or best practices and not addressing security vulnerabilities.
4.9	**Misunderstanding Agile**	Not understanding or faithfully following agile practices.
4.10	**Not Tracking Technical Debt**	Not identifying or measuring technical debt. This also includes a lack of metrics.
4.11	**Overlooking Acceptance Criteria or Definition of Done**	Includes poorly defined or missing acceptance criteria and definition of done. This also includes not checking each user story against its acceptance criteria and the definition of done.
4.12	**Poor Backlog Refinement***	Not refining the backlog to reflect current requirements in enough detail.
4.13	**Rigidity of Processes***	Adopting rigid processes such as fixed sprint lengths when they are inappropriate for the task at hand.
4.14	**Siloed Development**	Developing software in teams that isolate themselves from each other or exclude stakeholders that could provide additional information or a different perspective. This includes separating the development of related features.
4.15	**Too Much Overhead***	Holding too many meetings or implementing cumbersome processes to address simple issues. This includes poorly planned or managed meetings.
4.16	**Unresolved or Hastily Resolved Bugs**	Postponing or hastily resolving bugs and defects.
	Product	
5.1	**Code Complexity**	Writing complex code that is difficult to understand and maintain.
5.2	**Code Duplication**	Having the same code in multiple places.
5.3	**High Volume of Issues**	Having to deal with a lot of issues because an application is unstable.

5.4	**Inappropriate Coding Choices**	Using inappropriate development frameworks, tools, or abstractions.
5.5	**Inappropriate Software Reuse**	Reusing poor quality code or shoehorning existing code. This includes an over-reliance on libraries.
5.6	**Legacy or Monolithic Code**	Using legacy or monolithic code that is difficult to understand or refactor. This also includes using deprecated or obsolete features.
5.7	**Natural Evolution of a System**	Adding new features to an application over time, which leads to product entropy or disorder.
5.8	**Poor Architecture or Design**	Creating a software architecture or design that is not carefully planned or that does not follow established standards and best practices. This includes creating an architecture or design that is fragile, overly complex, not easily scalable, difficult to maintain, or not flexible enough to accommodate emerging technologies.

****Relatively unexplored causes of technical debt that are unique to or more salient in agile software development projects.***

A close examination of this list shows that it includes all the causes of technical debt in agile software development projects uncovered during my review of the relevant literature. Further, it reveals additional causes of technical debt in agile software development projects that have received little or no attention in previous research. Most notably, these causes of technical debt fall under two areas: *people* and *process*. Unexplored causes of technical debt related to people include *developers lack of authority*, *stressed teams*, and *too many interruptions*, and unexplored causes of technical debt related to the process include *improper scoping*, *poor backlog refinement*, *rigidity of processes*, and *too much overhead*.

The list also highlights causes of technical debt that – while not unique to agile software development projects – are often overlooked. For example, it draws attention to external causes of technical debt that are typically outside of the organization's control, such as *lower standards by contractors* and *unowned dependencies*. It also surfaces interesting organizational causes of technical debt, such as *lack of psychological safety*, and people causes of technical debt, such as *self-serving motivations*. Further, it reminds us to carefully consider process causes of technical debt, such as a *lack of focus on non-functional requirements,* which can have substantial implications for the usability, reliability, scalability, performance, and security of the software.

2.4.2 Most Important Causes of Technical Debt

At the end of Phase 2, I was able to answer my second research question: *What are the most important causes of technical debt in agile software development projects for IT project managers and software developers?* Specifically, through the Delphi process, I was able to identify 15 causes of technical debt that most IT project managers in this study viewed as important and 12 causes of technical debt that most software developers in this study viewed as important. The results are presented in *Table 2.3* (in alphabetical order).

Both IT project managers and software developers agreed on eight causes of technical debt: *inadequate code review*, *inadequate or inappropriate code refactoring*, *lack of coding standards*, *not prioritizing technical debt*, *poor architecture or design*, *poor collaboration or communication*, *schedule pressure*, and *stressed teams*. This agreement would suggest that these eight causes are salient regardless of role and should be given adequate attention when managing technical debt.

There were notable differences between the two panels, however. IT project managers prioritized *lack of continuous integration*, *lack of ownership*, *lack of vision or roadmap*, *not tracking technical debt*, *overlooking acceptance criteria or definition of* *done*, *short-term focus*, and *siloed development* while software developers did not. And software developers prioritized *changing, ill-defined, or missing requirements*, *code duplication*, *inadequate testing or quality assurance*, and *unresolved or hastily resolved bugs* while IT project managers did not. These differences would suggest that IT project managers and software developers view technical debt and its causes differently.

Table 2.3: Most Important Causes of Technical Debt

	IT Project Managers	**Software Developers**
Changing, Ill-defined, or Missing Requirements		X
Code Duplication		X
Inadequate Code Review	X	X
Inadequate or Inappropriate Code Refactoring	X	X
Inadequate Testing or Quality Assurance		X
Lack of Coding Standards	X	X
Lack of Continuous Integration	X	
Lack of Ownership	X	
Lack of Vision or Roadmap	X	
Not Prioritizing Technical Debt	X	X
Not Tracking Technical Debt	X	
Overlooking Acceptance Criteria or Definition of Done	X	
Poor Architecture or Design	X	X
Poor Collaboration or Communication	X	X
Schedule Pressure	X	X
Short-Term Focus	X	
Siloed Development	X	
Stressed Teams	X	X
Unresolved or Hastily Resolved Bugs		X

2.4.3 IT Project Manager and Software Developer Rankings

At the end of Phase 3, I was able to answer my third research question: *In what ways do the causes of technical debt in agile software development projects vary in importance for IT project managers and software developers?* Specifically, after four rounds of ranking, I was able to achieve a moderate to high level of consensus on the relative importance of the causes of technical debt for each panel. According to Schmidt (1997), a Kendall’s W of 0.5 indicates moderate agreement, and a Kendall’s W of 0.7 indicates strong agreement. I achieved a Kendall’s W of .656 (p < .001) for the IT project manager panel and .629 (p <. 001) for the software developer panel. The results for each panel are presented in tables *Table 2.4* and *Table 2.5*, and the final rankings for both panels are presented in *Table 2.6*. It is noteworthy there was no attrition during the four rounds of ranking. The number of participants, k, was 26 for the IT project manager panel and 22 for the software developer panel.

Table 2.4: Mean Ranks from IT Project Manager Panel

	Round 1 (k=26)	**Round 2 (k=26)**	**Round 3 (k=26)**	**Round 4 (k=26)**
Schedule Pressure	5.00	2.81	3.00	2.23
Lack of Vision or Roadmap	6.42	3.96	3.62	3.50
Poor Architecture or Design	5.81	3.88	4.62	4.08
Short-Term Focus	6.50	5.38	5.19	4.15
Not Prioritizing Technical Debt	6.54	5.50	5.58	5.73
Poor Communication or Collaboration	8.15	7.73	6.96	5.96
Lack of Coding Standards	7.38	7.31	7.85	7.62
Not Tracking Technical Debt	8.62	8.35	8.04	8.15

Overlooking Acceptance Criteria or Definition of Done	8.88	8.42	8.46	8.35
Lack of Ownership	8.42	8.69	8.81	9.62
Lack of Continuous Integration	9.04	10.23	10.50	11.19
Inadequate Code Review	9.12	10.88	11.00	11.73
Stressed Teams	10.69	13.00	11.96	11.81
Siloed Development	9.23	11.38	12.00	12.54
Inadequate or Inappropriate Code Refactoring	10.19	12.46	12.42	13.35
Kendall's W	**.137**	**.515**	**.493**	**.656**
Significance of W	**< .001**	**< .001**	**< .001**	**< .001**

Table 2.5: Mean Ranks from Software Developer Panel

	Round 1 (k=22)	**Round 2 (k=22)**	**Round 3 (k=22)**	**Round 4 (k=22)**
Changing, Ill-defined, or Missing Requirements	4.23	3.36	2.36	2.00
Poor Architecture or Design	4.36	3.73	3.36	2.59
Schedule Pressure	5.27	4.32	4.23	3.64
Not Prioritizing Technical Debt	5.91	5.77	5.09	5.05
Unresolved or Hastily Resolved Bugs	6.14	6.09	5.95	5.91
Poor Collaboration or Communication	6.77	6.27	6.36	6.18
Inadequate Testing or Quality Assurance	7.23	7.18	6.91	6.36
Inadequate Code Review	7.23	7.68	7.50	7.73
Lack of Coding Standards	6.68	6.77	7.77	8.91
Stressed Teams	7.41	8.05	9.00	9.36
Inadequate or Inappropriate Refactoring	8.14	9.36	9.36	9.23
Code Duplication	8.64	9.41	10.09	11.05
Kendall's W	**.147**	**.308**	**.451**	**.629**
Significance of W	**< .001**	**< .001**	**< .001**	**< .001**

Table 2.6: Final Rankings for IT Project Managers and Software Developers

	IT Project Managers	**Software Developers**
Schedule Pressure	1	3
Lack of Vision or Roadmap	2	
Poor Architecture or Design	3	2
Short-Term Focus	4	
Not Prioritizing Technical Debt	5	4
Poor Collaboration or Communication	6	6
Lack of Coding Standards	7	9
Not Tracking Technical Debt	8	
Overlooking Acceptance Criteria or Definition of Done	9	
Lack of Ownership	10	
Lack of Continuous Integration	11	
Inadequate Code Review	12	8
Stressed Teams	13	10
Siloed Development	14	
Inadequate or Inappropriate Code Refactoring	15	11
Changing, Ill-defined, or Missing Requirements		1
Unresolved or Hastily Resolved Bugs		5
Inadequate Testing or Quality Assurance		7
Code Duplication		12
TOTAL (19 items)	**15 items**	**12 items**

Both IT project managers and software developers ranked *schedule pressure*, *poor architecture or design*, and *not prioritizing technical debt* in their top five most important causes of technical debt. This would suggest that agile software development projects should manage these three causes of technical debt very closely.

IT project managers, however, included *lack of vision or roadmap* and *short-term focus* in their top five most important causes of technical debt, while software developers did not. And software developers included *changing, ill-defined, or missing requirements* and *unresolved or hastily resolved bugs* in their top five most important causes of technical debt while IT project managers did not. These differences suggest that IT project managers consider having a vision or roadmap and adopting a long-term focus to be important while software developers consider having clear requirements and taking the time to resolve bugs to be important. These differences may be due to the nature of the roles of IT project managers and software developers, which are complementary. Specifically, IT project managers are typically responsible for project planning (which involves a vision or long-term focus), while software developers are typically responsible for coding (which involves delivering requirements and resolving bugs).

2.4.4 Potential Techniques for Managing Technical Debt

After conducting the follow-up interviews, I was able to answer my fourth research question: *What are some potential techniques for managing technical debt in agile software development projects?* Specifically, by interviewing a select group of panelists, I was able to identify potential techniques for managing the most important causes of technical debt for IT project managers and software developers (i.e., those causes of technical debt that both panels had in common). I discuss these techniques below, highlighting relevant quotations from my interviews.

Schedule Pressure. One technique for managing schedule pressure is to have a clear understanding of what the business needs from the project and demonstrate how an aggressive schedule can jeopardize that objective. Another (fundamentally different) technique for managing schedule pressure is to shift the conversation away from the schedule altogether. Below are illustrative quotations.

> *"While we were doing this [Delphi study], I was going through a project where schedule was really killing us. And the technique I use is to understand why there is schedule pressure and what [the business] needs from the end goal, and give examples of how cutting [corners] or this technical debt that you're leaving, will not be worth it in the long run." – IT Project Manager #1*

> *"This is a big deal for us. I'm trying to work with the product owners and product managers to stop marching toward a date, to do scope-based releases. That relieves the pressure, the schedule pressure from the team and basically, they have no room for technical debt. When you're marching toward a date, it's all about features and getting the features done. And we've seen it, we've lived it for two years. And we're actually in the process of changing." – IT Project Manager #2*

Poor Architecture or Design. One technique for managing poor architecture or design is to acknowledge the need for it and plan to create one. This architecture or design does not need to be overly complex but rather needs to fit the project. Another technique for managing poor architecture or design is to make incremental improvements over time, recognizing that the architecture or design will never be perfect. Below are illustrative quotations.

> *"I think the actionable idea here is to be serious about the need for architecture or design, and to plan for it. Allocate time and energy for it. Yeah, and be willing not to move forward until the good architecture or design is clear." – Software Developer #4*

> *"We always have to take steps to improve the architecture of what we're working on, and teams that don't do that, the debt just grows and grows, they don't even realize it. So yes, if we can have a mentality of refactoring as we go about changing things, even big things, even fundamental things in our architecture, we'll have a better time managing this, because this never goes away. You never get a perfect architecture. Architecture is always degrading unless you are upgrading it as you go." – Software Developer #5*

Not Prioritizing Technical Debt. One technique for prioritizing technical debt is to allocate time to resolve technical debt items in each sprint. Another technique for prioritizing technical debt is to provide the business with data on the real-world impacts of technical debt. Below are illustrative quotations.

> *"I was living it at the time [of the Delphi study]… I've coached the team to prioritize technical debt… to make sure we do at least one tech debt story per sprint per team, and to make sure we stay on top of it." – IT Project Manager #2*

> *"I did a lot of coming up with the numbers and showing the performance difference between a store with 1,000 products and one that had 100,000 products and the number of timeouts and the delays in getting a product into the system and all sorts of things like that. And once I could quantify what the technical debt was, I was a lot more successful in getting the time, resources, and people to actually work on it." – Software Developer #3*

Poor Collaboration or Communication. One technique to improve collaboration and communication is to build in additional events (or ceremonies) in your agile method. Another technique to improve collaboration and communication is to create sub-teams within the larger agile team. Below are illustrative quotations.

> *"We put in a couple of extra ceremonies in our backlog refinement. We call it the Three Amigos. We have different representatives from the team come together and refine the stories [requirements] at a high level. We also do a tech sync on a weekly basis where the developers get together. And we also do a technical debt review every other week. So those are really the three ceremonies where we always talk about tech debt." – IT Project Manager #2*

> *"So, the one strategy that seems to work well, is just to be paired up in a smaller team." – Software Developer #6*

Lack of Coding Standards. One technique for addressing a lack of coding standards is to use automated tools to identify and potentially fix violations of the coding standards. Below are illustrative quotations.

> *"Lack of coding standards is a common thing where we're rushed. We have a ton of automation with all our different ways to check Java code. So, before you check it in, it has to run scans on your code. So, we have a fair amount of automation. That's one of the things we do. I think we run 150,000 automation scripts every night." – IT Project Manager #6*

> *"Some of the tools can automatically apply coding standards and point out failures. We did this with our SharePoint code base, which was quite a mess, which was to apply ReSharper. And we curated the set of coding standards that we wanted to enforce, and we set it up initially in the project so that it can highlight, give us warnings for all of them. And over time, as we addressed a lot of the coding standards violations, we got to a point where we then made the actual compile fail on that. And that did actually better than anything I've ever found to enforce coding standards without taking anybody's time. And it's really nice when it fails the compile because it's just sort of nags the developer for you." – Software Developer #3*

Inadequate Code Review. One technique for improving code reviews is to have multiple software developers review the code. This can be done through peer programming or having multiple independent code reviewers. Another technique for improving code reviews is to motivate software developers to conduct code reviews because of the personal benefits that they derive. Below are illustrative quotations.

> *"At my last job, the working agreement was that we had to have two sets of eyeballs on the code review; not only did we pair program, but we had to have two sets of eyeballs with code review. So, you got at least four people looking at the code. And we were very good about scrutinizing the code in every code review… And that resulted in very low tech debt." – Software Developer #1*

> *"I think that just helping people understand that there is value in the quality of your code, that the quality of your skills as a developer actually improve by not only getting code reviews but giving them as well. Helping people understand this will actually help their career is how I've approached folks who either don't want to do code reviews or don't find the value in them." – IT Project Manager #5*

Stressed Teams. One technique for dealing with stressed teams is to identify the root cause of the stress. While this is sometimes due to schedule pressure, there can often be other causes. Below are illustrative quotations.

> *"The team is stressed because they don't have clear priorities, or they have unreasonable demands usually. Therefore, they're going to cut corners and incur tech debt. But if a team is stressed for other reasons, you got to figure out why, and maybe they're stressed because the product is of questionable value and there's a threat that they're going to get laid off. Well then, nobody cares about tech debt. You're just going to do whatever you can do to prove your value so you can stay on. And if tech debt isn't appreciated, if solving tech debt isn't appreciated, then you will probably continue to grow in your problems there." – Software Developer #5*

> *"There's a lot of trust-building, a lot of time, and a lot of investment into identifying the root causes of the stress, because what you might see on the surface may not actually be the stressor, and that takes time to dig down into. And then from there you can frequently find themes and start to try to focus on sorting those things out to try to relieve some of that pressure." – IT Project Manager #5*

Inadequate or Inappropriate Code Refactoring. One way to improve code refactoring is to educate stakeholders on what refactoring is and why we need to do it. Software developers should also be trained on how to do appropriate refactoring. Below are illustrative quotations.

> *"You can deal with it through code reviews and coaching and teaching because a lot of developers I've noticed they just don't know a better way. So, they don't refactor stuff, or they refactor it in ways that just aren't really very impressive." – Software Developer #3*

> *"There are a lot of people who still don't understand refactoring. They don't understand how to do it; they don't understand why to do it. But with refactoring, well, a lot of developers know where it needs to be done. Refactoring is very important. If you want to keep the code in such a way that it's viable and we can modify it, we can work with it, we can enhance it, and make it adapt to our changing business requirements. But I think number one step is just to educate people on what refactoring is and why we need to do it because there's lots and lots of programmers who've never even heard the term." – Software Developer #7*

In the follow-up interviews, I also delved deeper into why some causes of technical debt were ranked highly by IT project managers but not at all by software developers. Specifically, *lack of vision or roadmap* and *short-term focus* were in the top five for IT project managers but not in the top 12 for software developers. And, *changing, ill-defined, or missing requirements* and *unresolved or hastily resolved bugs* were in the top five for software developers and not in the top 15 for IT project managers.

One potential explanation for this difference is that each group is placing the responsibility on the other group. The following quotations illustrate this point.

> *"I feel like that's finger-pointing. The developers are saying, 'Hey, we don't get good requirements.' And the people who are giving the requirements are saying, 'Our requirements are fine. What are you talking about?'" – Software Developer #1*

> *"Although developers blame themselves for much of that goes wrong, we also blame other people. It's human nature. So that one doesn't particularly surprise me because that's I think one of those footballs that gets thrown back and forth between IT and project management is, 'Well, we gave you the requirements.' 'Well, they're not good enough.' And I've seen that a lot." – Software Developer #3*

> *"I guess people tend to externalize blame. So maybe it's easier for the developers to see that [requirements] as an issue because it's somebody else's problem normally." – Software Developer #4*

Another potential explanation for this difference is that software developers are more aware of and focused on things that impact the quality of the product while IT project managers are more aware of and focused on delivering the product. The following quotations illustrate this point.

> *"It's a case of who's incentivized to care about quality. The project manager tends not to be because they don't know the quality, they don't use the product, nor do they implement the product. The team implements the product, but they don't use it. The business uses the product, but they don't implement it. Someone has to care about quality, and it probably can't be the project manager." – Software Developer #5*

> *"Well, as far as the unresolved or hastily resolved bugs, I don't think project managers are down to that level of granularity. I think that's below the [big] picture. They're looking more at the big picture. They're not looking down at that level." – Software Developer #7*

> *"I have a theory, and it's just my own theory, that those are very specific and low-level type issues that software developers think about a lot. I think project managers are probably thinking a little bit more high level, as you can see because I can tie missing requirements right into roadmap or overlooking acceptance criteria. I would think of it at a higher level. So, I'm thinking the software developers are thinking down at a low level where they live. They live in requirements; they live in bugs and defects. So, it doesn't surprise me that they would put those in the top five." – IT Project Manager #2*

> *"I'm not super surprised because I think that developers like to build things, so they may not care sometimes as much about the roadmap, especially if it's such an abstract roadmap it doesn't matter. I feel like the architecture and design inform a lot of their day to day more than the product vision. So, I'm not entirely surprised by that." – IT Project Manager #5*

A third explanation for these differences is that IT project managers and software developers use different terminology to refer to the same concepts. The following quotations illustrate this point.

> *"I think that changing, ill-defined, or missing requirements is equal to short-term focus and lack of vision or roadmap. And I'm thinking roadmap, and they're thinking code. I think that's the equivalent." – IT Project Manager #1*

> *"We're calling it different things. A project manager will talk in terms of vision or roadmap. Whereas a developer might talk in terms of architecture or design. But both are painting the big picture of where we're going." – Software Developer #4*

A fourth explanation for the differences between IT project managers and software developers is that IT project managers tend to focus on "what" needs to be done and "why" it needs to be done, while software developers tend to focus on "how" it should be done. The following quotations illustrate this point.

> *"We care about the 'what,' along with the Product Owners, the 'what' and the 'why.' But the 'how,' we don't really care about. I'm not saying we don't care about it, but it's really up to the team to decide how they'll do it." – IT Project Manager #2*

> *"It's sort of like the difference just between managers and developers, in general: managers want to know 'why' something needs to be done or 'what' should be done, but developers are concentrated on 'how' to do it. They're asking two different questions. Like I said, that's true of managers and developers anyway. It has just been that way forever. I see it the same way with a project manager and a developer assigned to the team. The developer wants to know 'how,' the project manager wants to know 'what' and 'why.'" – Software Developer #7*

2.5 Conclusion

By conducting a ranking-type Delphi study of 86 experienced IT project managers and software developers, I produced a verified list of 55 causes of technical debt in agile software development projects. From that list, the panel of IT project managers identified 15 causes of technical debt that were important to most of them, and the panel of software developers identified the 12 causes of technical debt that were important to most of them. Subsequently, I interviewed a select group of IT project managers and software developers to identify potential techniques for managing the eight causes of technical debt that were common across both panels. In total, I presented 13 potential techniques for managing these eight causes. Further, I offered four potential explanations for the different rankings across the IT project manager and software developer panels.

As with any Delphi study, the panels are not meant to be statistically representative. Rather, they are meant to capture the views of relevant experts, which I readily accomplished in this essay. To make the process manageable, I focused on the two most critical roles in software development projects: IT project managers and software developers. Future research could include additional roles, such as end-users and customer representatives, who are also core to the agile software development process (Maruping & Matook, 2020). I also limited the panels to IT project managers and software developers in the United States. Future research could include additional countries to determine whether there are meaningful cultural differences in managing technical debt. Finally, I interviewed the subset of panelists who were most engaged and who indicated a willingness to participate in the interview process. After conducting the first dozen interviews, little additional insight was gained from each subsequent interview, so I concluded the interview process. Future research could test the efficacy of the potential techniques identified in these interviews.

2.5.1 Implications for Research

Researchers can use the verified list of 55 causes of technical debt identified in this essay when referring to technical debt in agile software development projects. This will make it easier to synthesize findings across multiple studies. In addition, researchers can focus on those causes of technical debt that are unique to the agile context (e.g., *poor backlog refinement* or *lack of continuous integration*) to help us better understand the nuances of technical debt in agile software development projects. Also, researchers can examine those causes of technical debt that surfaced in this essay that were not previously discussed in the extant literature on agile software development projects (e.g., *improper scoping* and *rigidity of processes*). Further, by using the prioritized list that was generated, researchers can spend finite resources investigating the causes of technical debt that are most problematic for IT project managers and software developers. Finally, researchers can probe further into the different perspectives of IT project managers and software developers to help improve our understanding of technical debt in agile software development projects.

2.5.2 Implications for Practice

Practitioners can use the verified list of 55 causes of technical debt identified in this essay as a checklist when managing technical debt in their agile software development projects. This will make it easier to address the causes of technical debt that might otherwise go unnoticed. Also, by using the prioritized list that was generated, IT project managers and software developers can focus their time and resources on managing those causes of technical debt that are likely to be most problematic. Further, practitioners can use the different rankings to better understand the priorities and challenges that IT project managers and software developers face. Finally, practitioners can implement the 13 potential techniques for managing technical debt identified during the interviews to test whether these techniques are effective in their agile software development projects.

In closing, I would like to encourage IT project managers not just to think about ‘what’ tasks need to be accomplished and ‘why,’ but also to think about the challenges that software developers face when trying to decide ‘how’ best to accomplish those tasks. And I would like to encourage software developers not just to think about ‘how’ best to accomplish the tasks that you are working on but also to think about ‘what’ the end goal is and ‘why’ the organization is working towards that goal.

2.6 Appendix A: Survey for Phase 1

In Phase 1, all participants were asked to complete the task below.

In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible.

Please provide **at least six** items that you consider to be the primary causes of technical debt in agile software development projects. For each item, please provide a brief (one or two sentences) description to help us match your items with similar items from other panelists. Please note that the items do not have to be in any particular order.

Item 1

Name	
Description	

Item 2

Name	
Description	

Item 3

Name	
Description	

…

Item 10

Name	
Description	

2.7 Appendix B: Survey for Phase 2

In Phase 2, all participants were asked to complete the task below. However, the participants in the software developer panel were asked to "adopt the perspective of a software developer" as they completed the task. The order of the 55 causes of technical debt was randomized for each participant.

Please identify the most important 20 causes of technical debt in agile software development projects from the list below by placing a check mark beside each item.

Once again, by "most important" we mean those causes of technical debt which you would handle on a priority basis if you only had enough time and resources to manage 20 of them.

As a reminder, we ask that you adopt the perspective of an IT project manager as you complete this task.

Number of Items Selected: 0 / 20

☐ BUSINESS-TECHNOLOGY CHALLENGES: Includes challenges between business representatives and the technical team, such as a lack of trust and power imbalances. This also includes business representatives exerting undue influence over technical decisions, and business representatives not understanding the technology or the technical implications of their decisions.

☐ CHANGING, ILLDEFINED, OR MISSING REQUIREMENTS: Includes constantly changing, poorly defined, or missing requirements. This also includes ambiguous or vague requirements, shifting priorities, and scope creep.

☐ CODE COMPLEXITY: Writing complex code that is difficult to understand and maintain.

…

☐ UNRESOLVED OR HASTILY RESOLVED BUGS: Postponing or hastily resolving bugs and defects.

2.8 Appendix C: Survey for Phase 3 (First Round)

In the first round of Phase 3, all participants were asked to complete the task below. However, the participants in the software developer panel were asked to rank the “12 causes of technical debt” that were identified as important by a majority of their panel. The order of the 15 (or 12) causes of technical debt was randomized for each participant.

Please rank the following 15 causes of technical debt from most important (at the top) to least important (at the bottom). You can simply drag and drop the items to reorder them.

Inadequate Code Review: Inadequate or no code review. This includes not identifying or removing dead code.

Inadequate or Inappropriate Code Refactoring: Inadequate or delayed code refactoring. This includes premature code optimization.

Lack of Coding Standards: Includes not having clearly defined standards, adhering to existing standards, or refactoring code written before standardization. This also includes a lack of modularity and hard coding.

…

Stressed Teams: Pushing teams beyond a sustainable pace of delivery or requiring teams to produce something every day.

2.9 Appendix D: Survey for Phase 3 (Subsequent Rounds)

In the subsequent rounds of Phase 3, all participants were asked to complete the task below. However, the participants in the software developer panel were asked to re-rank the "12 causes of technical debt" that were identified as important by a majority of their panel.

Please re-rank the following 15 causes of technical debt from most important (at the top) to least important (at the bottom). You can simply drag and drop the items to reorder them.

At the beginning of each item, in parenthesis, is the average rank of that item from the previous round. At the end of each item, in brackets, is the percentage of participants from the previous round that placed the item in the top half of their list. Please consider these two pieces of additional information as you re-rank the items.

Please note that the list is initially ordered by the average rank of each item from the previous round, where lower ranks represent higher importance (e.g., 1st place).

(5.00) Schedule Pressure: Includes schedule pressure due to unrealistic deadlines, time constraints, management directives, or market competition. **[77% placed in top half of their list]**

(5.81) Poor Architecture or Design: Creating a software architecture or design that is not carefully planned or that does not follow established standards and best practices. This includes creating an architecture or design that is fragile, overly complex, not easily scalable, difficult to maintain, or not flexible enough to accommodate emerging technologies. **[81% placed in top half of their list]**

(6.42) Lack of Vision or Roadmap: Not having a clear vision or roadmap for the application. **[62% placed in top half of their list]**

…

(10.69) Stressed Teams: Pushing teams beyond a sustainable pace of delivery or requiring teams to produce something every day. **[19% placed in top half of their list]**

2.10 Appendix E: Semi-Structured Interview Protocol

During the Zoom interviews, IT project managers were presented with the final ranking of the 15 causes of technical debt (see *Figure 2.1*), and software developers were presented with the final ranking of the 12 causes of technical debt (see *Figure 2.2*) identified as most important by their respective panels.

The following questions were then used to guide this portion of the interview:

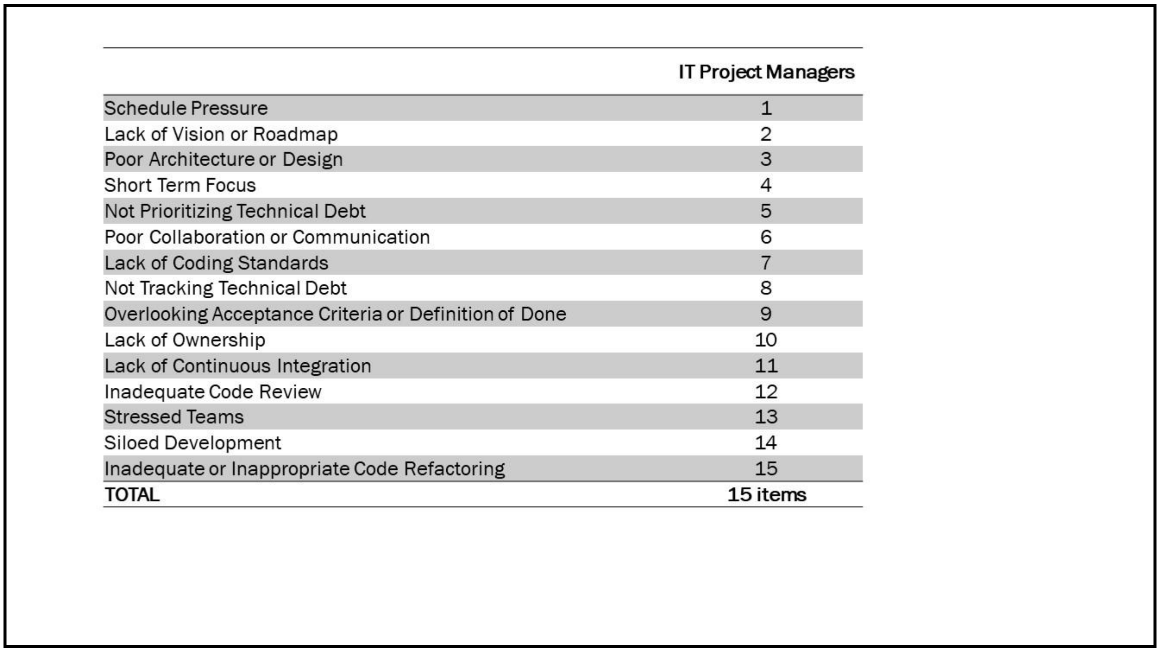
1. Are there any causes of technical debt on the list for which you have developed effective strategies to manage them?

2. If so, could you tell me what strategies you have found to be most effective in managing those causes of technical debt?

3. Could you share an example of when a particular strategy was used? How was this strategy implemented? How effective was this strategy?

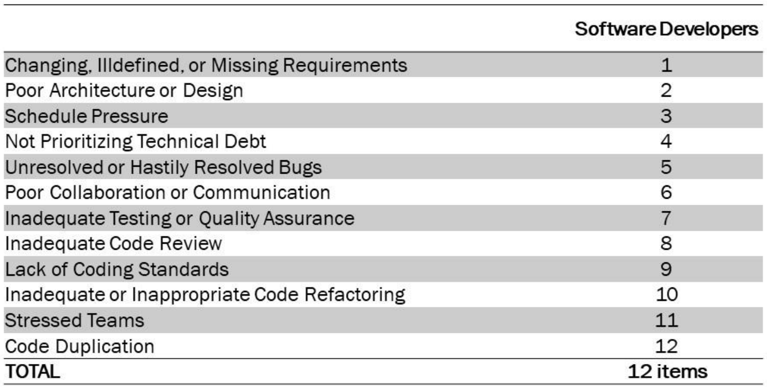
4. Are there causes of technical debt that you see as more problematic in that there is no effective strategy to overcome it?

5. Are there any causes of technical debt on the list (or not on the list) that you found interesting or surprising?



	IT Project Managers
Schedule Pressure	1
Lack of Vision or Roadmap	2
Poor Architecture or Design	3
Short Term Focus	4
Not Prioritizing Technical Debt	5
Poor Collaboration or Communication	6
Lack of Coding Standards	7
Not Tracking Technical Debt	8
Overlooking Acceptance Criteria or Definition of Done	9
Lack of Ownership	10
Lack of Continuous Integration	11
Inadequate Code Review	12
Stressed Teams	13
Siloed Development	14
Inadequate or Inappropriate Code Refactoring	15
TOTAL	**15 items**

Figure 2.1: Final Ranking for IT Project Manager Panel



	Software Developers
Changing, Illdefined, or Missing Requirements	1
Poor Architecture or Design	2
Schedule Pressure	3
Not Prioritizing Technical Debt	4
Unresolved or Hastily Resolved Bugs	5
Poor Collaboration or Communication	6
Inadequate Testing or Quality Assurance	7
Inadequate Code Review	8
Lack of Coding Standards	9
Inadequate or Inappropriate Code Refactoring	10
Stressed Teams	11
Code Duplication	12
TOTAL	12 items

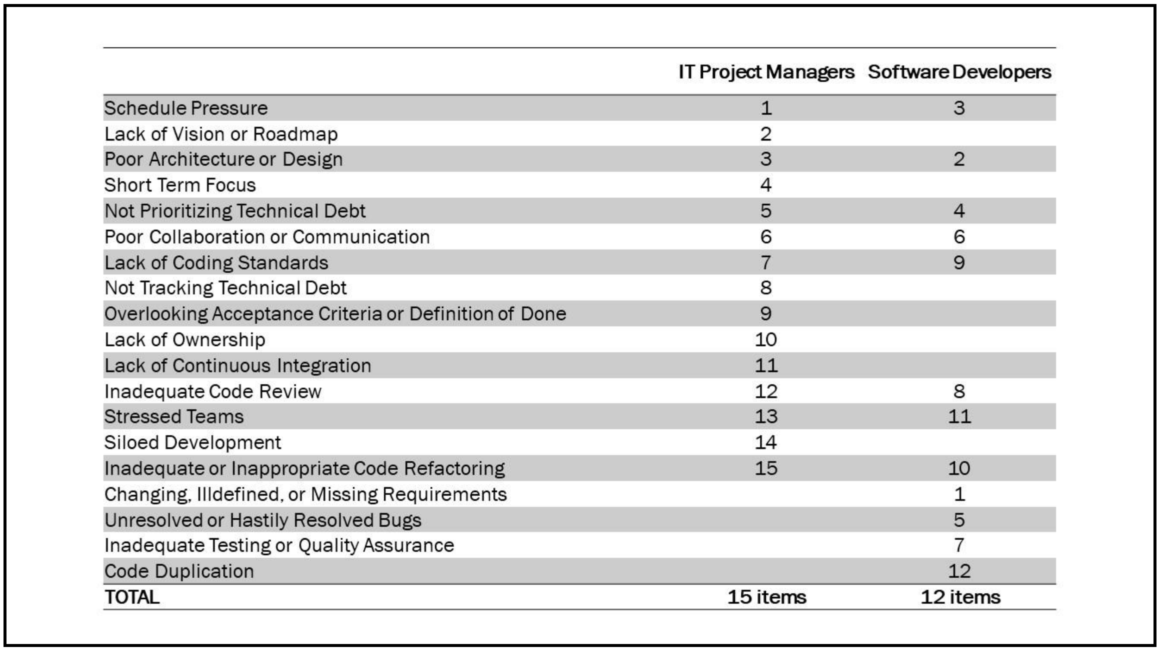
Figure 2.2: Final Ranking for Software Developer Panel

IT project managers were then presented with the software developers rankings (see *Figure 2.3*), and software developers were then presented with the IT project manager rankings (see *Figure 2.4*).

The following questions were then used to guide this portion of the interview:

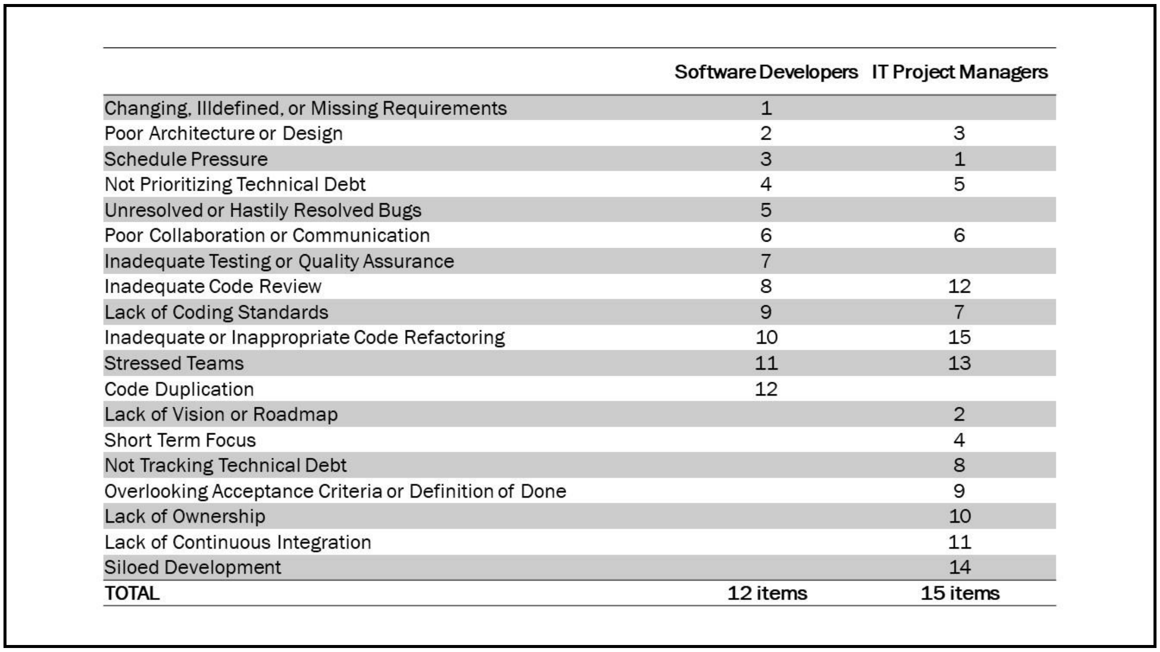
6. Why do you think some causes of technical debt were ranked highly by IT project managers and not at all by software developers (and vice versa)?

7. Is there anything in the different rankings that you found interesting or surprising?



	IT Project Managers	Software Developers
Schedule Pressure	1	3
Lack of Vision or Roadmap	2	
Poor Architecture or Design	3	2
Short Term Focus	4	
Not Prioritizing Technical Debt	5	4
Poor Collaboration or Communication	6	6
Lack of Coding Standards	7	9
Not Tracking Technical Debt	8	
Overlooking Acceptance Criteria or Definition of Done	9	
Lack of Ownership	10	
Lack of Continuous Integration	11	
Inadequate Code Review	12	8
Stressed Teams	13	11
Siloed Development	14	
Inadequate or Inappropriate Code Refactoring	15	10
Changing, Illdefined, or Missing Requirements		1
Unresolved or Hastily Resolved Bugs		5
Inadequate Testing or Quality Assurance		7
Code Duplication		12
TOTAL	**15 items**	**12 items**

Figure 2.3: Comparison of IT Project Manager and Software Developer Rankings



	Software Developers	IT Project Managers
Changing, Illdefined, or Missing Requirements	1	
Poor Architecture or Design	2	3
Schedule Pressure	3	1
Not Prioritizing Technical Debt	4	5
Unresolved or Hastily Resolved Bugs	5	
Poor Collaboration or Communication	6	6
Inadequate Testing or Quality Assurance	7	
Inadequate Code Review	8	12
Lack of Coding Standards	9	7
Inadequate or Inappropriate Code Refactoring	10	15
Stressed Teams	11	13
Code Duplication	12	
Lack of Vision or Roadmap		2
Short Term Focus		4
Not Tracking Technical Debt		8
Overlooking Acceptance Criteria or Definition of Done		9
Lack of Ownership		10
Lack of Continuous Integration		11
Siloed Development		14
TOTAL	**12 items**	**15 items**

Figure 2.4: Comparison of Software Developer and IT Project Manager Rankings

3 Essay Two: The Impact of Software Developers' Construal Level on Technical Debt

Abstract

Software developers faced with ambitious requirements and deadline pressure often take shortcuts to deliver all the requirements on time. These shortcuts often result in *technical debt* – design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Worldwide, technical debt is estimated to be a trillion-dollar problem. To increase our knowledge of the factors that contribute to technical debt, I conducted a randomized experiment to examine the impact of software developers' *construal level* on the unintentional accumulation of technical debt. Construal level, a cognitive process that we all use, refers to the degree to which we perceive an object as being psychologically distant. The results of my experiment suggest that software developers at a high construal level are more likely to focus on developing the architecture or design than software developers at a low construal level. This finding is particularly important since architectural debt has the highest cost of ownership compared to other forms of technical debt.

Keywords: technical debt, software development, mindset, construal level theory, experiment

3.1 Introduction

"Short cuts make long delays."
J.R.R. Tolkien

A software developer facing an impending deadline with many tasks yet to be completed has several options. One option is to rescope the remaining work to include only include those high-value tasks that can be reasonably completed by the deadline. A second option is to negotiate a new deadline that would allow sufficient time to complete all the remaining tasks. A third option is to dedicate additional resources to complete the remaining tasks by the original deadline. A fourth option is to take shortcuts to deliver the incomplete tasks by the original deadline. The last option – which is all too common – is the focus of this essay.

In software development, taking shortcuts often result in *technical debt* – "a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible" (Avgeriou et al., 2016, p. 112). For example, faced with an impending deadline, a software developer might overlook good design principles, conduct inadequate testing, or hastily resolve bugs to meet the deadline. However, by taking these shortcuts, software developers are likely to experience more difficulties maintaining or evolving the software in the future (Brown et al., 2010), which will only compound over time if the technical debt remains unresolved.

There are, of course, situations where it makes sense to incur technical debt to obtain some immediate benefit, such as gaining a first-mover advantage or receiving early customer feedback. This debt is intentional and worth incurring with a plan to resolve it later. However, there are times when software developers incur technical debt unintentionally (McConnell, 2008), often based on inexperience and bad design choices, which may not be obvious until later.

Worldwide, technical debt is estimated to be a trillion-dollar problem (Kruchten et al., 2019; Kyte, 2010), which has resulted in numerous scholars calling for further research in this area. For example, Kruchten et al. (2012, p. 21) argued for "more tools and methods to identify and manage debt." In addition, Kruchten et al. (2013, p. 51) argued that a "better understanding of the concept of technical debt, and how to approach it, both from a theoretical and a practical perspective is necessary to advance its state of the art and practice." Further, Alves et al. (2016, p. 118) argued that "it is necessary to conduct further studies in the area to investigate new techniques and tools that could support developers with the control of [technical debt]." In this essay, I answer their call-to-action by investigating whether software developers' *construal level* contributes to the unintentional accumulation of technical debt.

The core idea is that two software developers presented with the same requirements may construe the requirements differently based on their construal level such that software developers at a low construal level will focus on the details of the requirements, and software developers at a high construal level will focus on the abstract aspects of the requirements. Construal level theory has shown that different construal levels can have varying impacts on decision making. Currently, there is a paucity of studies addressing the behavioral antecedents of technical debt and none that examine the impact of software developers' construal level on technical debt. To effectively manage technical debt, we must first understand the causes of technical debt (Kruchten et al., 2012), and this essay seeks to advance our knowledge in this area.

In my first essay, I found that while IT project managers tend to focus on ‘what’ work needs to be done and ‘why’ that work needs to be done, software developers tend to focus on ‘how’ to get the work done. This ‘how’ focus has been shown to induce a low construal level (Freitas, Gollwitzer, & Trope, 2004). Further, Liberman and Trope (1998) showed that a low construal level prompts people to focus on feasibility concerns (what *can* be done) over desirable outcomes (what *should* be done). I, therefore, argue that software developers at a low construal level will be more likely to incur technical debt due to their focus on feasibility concerns such as developing the features quickly, whereas software developers at a high construal level will be less likely to incur technical debt due to their focus on desirable outcomes such as a well thought out architecture and design.

To test my hypothesis, I conducted a randomized experiment with 65 experienced software developers. The findings from this experiment suggest that software developers’ construal level may, in fact, play a role in the unintentional accumulation of technical debt. Specifically, software developers at a high construal level indicated a greater likelihood of focusing on good planning practices than software developers at a low construal level. However, the way in which software developers’ construal level impacts technical debt may be more nuanced than initially hypothesized.

3.2 Background

3.2.1 Technical Debt

Technical debt has long been a problem in software development and was initially discussed within the context of *software evolution* and *software maintenance* (Brown et al., 2010; Kruchten et al., 2019). However, there was no single concept that adequately captured this phenomenon until Ward Cunningham introduced the technical debt metaphor in 1992 (p. 30) when he said:

> *“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.”*

Over time, the technical debt metaphor gained popularity and evolved to include aspects outside of the software code (Brown et al., 2010; Kruchten et al., 2012), such as the architecture and production environment (Kruchten et al., 2019). The current definition of technical debt, which I adopt in this essay, reflects these changes and provides increased conceptual clarity (Avgeriou et al., 2016, p. 112):

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

Technical debt is a rich concept that can be understood from several different perspectives. Ampatzoglou, Ampatzoglou, Chatzigeorgiou, and Avgeriou (2015) likened technical debt to financial debt, which has a principal and interest. According to Camden

(2013), paying back the principal involves implementing the correct replacement, while paying the interest takes the form of time and resources spent working around the incorrect implementation. The difference between technical debt and financial debt, however, is that the interest associated with technical debt may or may not need to be repaid (Guo et al., 2016).

McConnell (2008) argued that there were two kinds of technical debt: *intentional debt* and *unintentional debt*. Intentional debt occurs when organizations make conscious decisions to focus on the short term instead of the long term. Examples of intentional debt include postponing documentation until later or accepting poorly written source code with the intention of tidying it up afterward. Unintentional debt occurs because of doing a poor job. Examples of unintentional debt include code produced by inexperienced programmers or selecting design approaches that turn out to be a bad choice. Unintentional debt occurs quite often (Cha, Dong, & Vogel-Heuser, 2018) and is typically more problematic than intentional debt (Klinger, Tarr, Wagstrom, & Williams, 2011). In this essay, I argue that unintentional debt can also result from software developers' construal level.

Fowler (2009) proposed the technical debt quadrant, which is illustrated in *Figure 3.1* below.

	Reckless	Prudent
Deliberate	A	B
Inadvertent	C	D

Figure 3.1: Technical Debt Quadrant

Quadrant A encompasses those instances in which a team may have the knowledge and skills to write clean code or make good design choices but decide to incur technical debt anyway because they do not think that they have the time to do it right. Quadrant B encompasses those instances in which a team chooses to incur technical debt strategically after thorough consideration of the costs and the benefits (e.g., to capture market share). Quadrant C encompasses those instances in which a team inadvertently incurs technical debt that is not beneficial to the project (e.g., due to inexperience), and Quadrant D encompasses those instances in which a team inadvertently incurs technical debt that is beneficial to the project (e.g., learning what the best design approach should have been).

Regardless of whether technical debt is intentional or unintentional, reckless or prudent, “technical debt takes different forms in different types of development artifacts, such as the code, the architecture and the production infrastructure” (Kruchten et al., 2019, p. 20). Technical debt in the code includes violation of coding standards, improper naming, duplicate code, misleading or incorrect comments, and unnecessary code complexity. Technical debt in the architecture includes the platform chosen, middleware used, communication technologies adopted, and user interface designs created. Technical debt in the production environment includes build scripts, test suites, and the deployment infrastructure. In this essay, I explore the impact of software developers’ construal level on different types of technical debt.

Unmanaged, technical debt can lead to several negative consequences. For example, technical debt can reduce the development pace (Fowler, 2003; Letouzey & Ilkiewicz, 2012), cripple the ability to meet customer requirements (Kruchten et al., 2019), and increase software maintenance difficulty and costs (Bavani, 2012; Brown et al., 2010;

Z. Li et al., 2015). In addition, technical debt can affect product performance, reliability, and stability (Lim et al., 2012; Ramasubbu & Kemerer, 2016), which negatively impacts users of the software. Further, technical debt can increase software complexity, which makes systems rigid (hard to change), fragile (each change breaks something else), viscous (doing things right is harder), and opaque (hard to understand) (Brown et al., 2010).

However, not all consequences of technical debt are negative. For example, incurring some technical debt can help to capture market share, meet contractual obligations, collect early customer feedback (Lim et al., 2012), and increase the speed of software delivery (Ramasubbu & Kemerer, 2014) and scalability (Kruchten et al., 2019). In addition, incurring technical debt strategically can help to preserve limited capital and delay development expenses that may or may not need to be repaid later (McConnell, 2008).

3.2.2 Construal Level

Construal level refers to the degree to which someone perceives an object as being psychologically distant (Trope & Liberman, 2010). Psychological distance refers to the distance between an object and the self; this distance can be in time, space, social distance, or hypotheticality (the likelihood of an event occurring) (Maglio, Trope, & Liberman, 2013; Trope & Liberman, 2010). A week from now, 3 miles away, a friend, and high likelihood would represent small psychological distances when compared to a year from now, 3,000 miles away, a stranger, or a low likelihood (Soderberg, Callahan, Kochersberger, Amit, & Ledgerwood, 2015).

People typically use construal when they have incomplete information about a particular situation (Ross, 1987), as is often the case in software development projects where requirements are constantly changing (Maruping et al., 2009). This, of course, means that software developers may interpret and respond to the same situation differently based on their construal level. Construal level theory states that people use higher construal levels to represent an object as their psychological distance from the object increases (Trope & Liberman, 2010).

When high-level construal is used, objects are represented in abstract terms that consist of general, superordinate, and decontextualized features (Liberman & Trope, 1998). When low-level construal is used, objects are represented in more concrete terms that consist of specific, subordinate, and contextualized features (Liberman & Trope, 1998). For example, a software developer at a high construal level may think of a module in terms of its purpose (e.g., accept credit card payments), whereas a software developer at a low-construal level may think of the same module in terms of its components (e.g., user input form, card validation process, payment confirmation screen, etc.).

Construal level theory provides an interesting lens, which has been used to study a variety of problems. For example, construal level theory has been used to study consumer behavior (Dhar & Kim, 2007; Liberman, Trope, & Wakslak, 2007), decision making (Liberman & Trope, 2003; Wan & Agrawal, 2011), and self-control (Fujita, 2008; Fujita & Carnevale, 2012). Construal level theory has also been used to study information systems phenomena such as IT project risk management (Lee, Keil, & Shalev, 2019) and online password use (Kaleta, Lee, & Yoo, 2019).

Construal level has also been manipulated in a number of ways. For example, Freitas et al. (2004) primed some participants to a high construal level by asking them to consider “why” they would engage in a particular activity and primed other participants to a low construal level by asking them to consider “how” they would engage in the same activity. As another example, Fujita, Trope, Liberman, and Levin-Sagi (2006) presented participants with 40 words and asked those in the high-level construal condition to generate superordinate category labels for each word (e.g., one category for dog could be animal) and those in the low-level construal condition to generate subordinate exemplars for each word (e.g., one example of a dog would be poodle).

Construal level theory is particularly appropriate for this study given its applicability to decision-making contexts such as the ones that software developers face in their daily routines and the relevance of why/how thinking to software development (discussed next).

3.2.3 Hypothesis

One of the key findings from the interviews in my first essay was that while IT project managers tend to focus on the big picture (‘what’ work needs to be done and ‘why’), software developers tend to focus on the details (‘how’ to get the work done). As discussed, focusing on ‘how’ to perform a task induces a low construal level while focusing on ‘why’ to perform a task induces a high construal level (Freitas et al., 2004). Further, we know that a low construal level promotes a focus on feasibility concerns while a high construal level promotes a focus on desirable outcomes (Liberman & Trope, 1998). Therefore, when faced with a choice between feasibility concerns such as developing all the required features by an impending deadline and desirable outcomes such as minimizing technical debt, I expect software developers at a low construal level to favor feasible options over desirable outcomes. Specifically, I expect software developers at a low construal level to be less likely to focus on good software development practices when compared to software developers at a high construal level.

3.3 Methodology

I opted to use the experimental method to test my hypothesis because of its high internal validity (Shadish et al., 2002; Trochim et al., 2016). This approach is supported by numerous prior studies on construal level that have successfully used experiments to test their hypotheses (e.g., Freitas et al., 2004; Henderson, Fujita, Trope, & Liberman, 2006; Liberman & Trope, 1998; Maglio et al., 2013; Wakslak & Trope, 2009). Specifically, I used a *basic randomized design comparing two treatments* (Shadish et al., 2002). In studies on construal level, it is common to use two treatments: a high construal level manipulation and a low construal level manipulation. It is also common to use randomization to minimize or eliminate several common threats to validity, including selection, history, maturation, testing, instrumentation, and regression (Trochim et al., 2016). Random assignment helps to minimize or eliminate these threats to validity by creating probabilistically equivalent groups that should be impacted by these threats similarly. Therefore, we can reasonably infer that observed differences between the groups are not due to these factors but rather are due to the experimental manipulations.

3.3.1 Preliminary Study Design

The literature on construal level theory highlights several ways to manipulate and measure construal level. One way to manipulate construal level involves presenting participants with different scenarios, each intended to induce either a low construal level or a high construal level. For example, Fujita, Henderson, Eng, Trope, and Liberman (2006) and Henderson et al. (2006) presented one group of participants with a scenario 3 miles away and another group of participants with the same scenario 3,000 miles away. Another way to manipulate construal level involves presenting participants with a list of 40 items and asking one group of participants to identify a *general category* for each item (e.g., animal would be a general category for dog) and the other group of participants to provide a *specific example* of each item (e.g., poodle would be a specific example of dog) (Fujita, Trope, et al., 2006).

One way to measure construal level is to use the Behavior Identification Form (BIF) – a list of 25 common behaviors (e.g., voting), each followed by two different ways in which that behavior might be identified (e.g., influencing the election or marking a ballot) (Vallacher & Wegner, 1989). The number of high-level identifications is then summed to provide a measure of construal level. A variation of the BIF is the Work-Based Construal Level (WBCL) – a list of 18 common work activities, such as "preparing a report" or "using a computer," each followed by a low-level description and a high-level description (Reyt & Wiesenfeld, 2015). A third way to measure construal level is to present participants with several scenarios and ask them to identify the likelihood that each scenario will occur. For example, Wakslak and Trope (2009) presented participants with seven hypothetical scenarios and asked them to rate the likelihood that each scenario would occur on a scale from 1 (not likely) to 7 (very likely).

As part of my preliminary study design, I tested several combinations of the abovementioned manipulations and measures on Amazon Mechanical Turk (AMT). Using AMT provided the ability to test numerous combinations of manipulations and measures at a relatively low cost. These initial tests suggested that using an *integral manipulation* (such as a scenario on software development) would increase my chances of observing an effect – if one was present – compared to using an *incidental manipulation* (such as the word task). To elaborate, an integral manipulation occurs within the context of the decision task, whereas an incidental manipulation occurs outside the context of the decision task (Lee et al., 2019). Armed with this insight, I proceeded to run two pilots using experienced software developers. The first pilot was run using participants recruited through Qualtrics, and the second pilot was run using participants recruited through Upwork – a reputable freelancer website. After comparing the results from these two pilots, I decided to collect the data for the main study through Upwork.

3.4 Results and Discussion

3.4.1 Pilot Study 1

In this pilot study, participants were presented with a scenario in which they were asked to imagine that they worked for a consulting company and were assigned to develop an online store for a client (see Appendix A). Then, participants in the high construal level manipulation group were asked to briefly describe “why” they would review, test, and document the software code for the online store, and participants in the low construal level manipulation group were asked to briefly describe “how” they would review, test, and document the software code for the online store. Afterward, participants were asked to respond to six items (see *Table 3.1*) intended to measure their intention to avoid technical debt. Subsequently, participants were asked to respond to several items intended to rule out rival explanations: the difficulty of the manipulation task, the effort required to complete the manipulation task, the willpower required to complete the manipulation task, conscientiousness, self-control, positive affect, and negative affect. Finally, participants were asked to provide some demographic information.

Table 3.1: Dependent Variables for Pilot Study 1

Construct	Item	Reliability
Code Review	I would carefully review the software code to ensure that it follows relevant best practices.	.977
	I would rigorously review the software code to ensure that it meets applicable coding standards.	
Testing	I would conduct extensive testing to ensure that the software code performs as expected.	.996
	I would perform extensive testing to ensure that the software code does not have any bugs.	
Documentation	I would write comprehensive documentation to ensure that I am able to modify the software code in the future.	.996
	I would create comprehensive documentation to ensure that my team members are able to understand how the software code works.	

Qualtrics provided 19 usable responses (from over 1,000 attempted or completed responses) for this pilot study over a 25-day period. All participants had three or more years of software development experience, and almost all had college degrees (N = 18). The median survey completion time was approximately 9 minutes. Descriptive statistics are presented in *Table 3.2*.

Table 3.2: Descriptive Statistics for Pilot Study 1

	Treatment*	**N**	**Mean**	**Std. Deviation**
Code Review	Low CL	10	6.25	1.87
	High CL	9	6.06	1.94
Testing	Low CL	10	6.40	1.90
	High CL	9	6.06	1.94
Documentation	Low CL	10	6.20	1.87
	High CL	9	5.83	1.94

Construal Level (CL).

Given the small sample size, a non-parametric Mann-Whitney U test (Mann & Whitney, 1947) was run to determine if there were differences in intention to review, test, and document the code between participants in the high construal group and participants in the low construal group. Code review scores were not statistically significantly different between high construal level (Mdn = 7.00) and low construal level (Mdn = 7.00) participants, U = 38.50, z = -.615, p = .604, using an exact sampling distribution for U. Testing scores were not statistically significantly different between high construal level (Mdn = 7.00) and low construal level (Mdn = 7.00) participants, U = 31.00, z = -1.477, p = .278, using an exact sampling distribution for U. Documentation scores were not statistically significantly different between high construal level (Mdn = 6.50) and low construal level (Mdn = 7.00) participants, U = 34.50, z = -.960, p = .400, using an exact sampling distribution for U. These results were not surprising given the small sample size and large standard deviations.

3.4.2 Pilot Study 2

In this pilot study, I administered the same survey that was used in the Qualtrics pilot (see Appendix A). *Table 3.3* presents the reliabilities of the dependent variables. I recruited 20 participants for this pilot study over a 12-day period. All participants had three or more years of software development experience, and almost all had college degrees (N=18). The median survey completion time was approximately 19 minutes. Descriptive statistics are presented in *Table 3.4*.

Table 3.3: Dependent Variables for Pilot Study 2

Construct	Item	Reliability
Code Review	I would carefully review the software code to ensure that it follows relevant best practices.	.640
	I would rigorously review the software code to ensure that it meets applicable coding standards.	
Testing	I would conduct extensive testing to ensure that the software code performs as expected.	.923
	I would perform extensive testing to ensure that the software code does not have any bugs.	
Documentation	I would write comprehensive documentation to ensure that I am able to modify the software code in the future.	.754
	I would create comprehensive documentation to ensure that my team members are able to understand how the software code works.	

Table 3.4: Descriptive Statistics for Pilot Study 2

	Treatment*	**N**	**Mean**	**Std. Deviation**
Code Review	Low CL	10	6.15	0.78
	High CL	10	6.10	0.70
Testing	Low CL	10	5.80	1.49
	High CL	10	6.35	0.63
Documentation	Low CL	10	6.10	0.74
	High CL	10	5.55	0.72

Construal Level (CL).

Given the small sample size, a non-parametric Mann-Whitney U test (Mann & Whitney, 1947) was run to determine if there were differences in intention to review, test, and document the code between participants in the high construal group and participants in the low construal group. Code review scores were not statistically significantly different between high construal level (Mdn = 6.00) and low construal level (Mdn = 6.25) participants, U = 47.50, z = -.195, p = .853, using an exact sampling distribution for U. Testing scores were not statistically significantly different between high construal level (Mdn = 6.25) and low construal level (Mdn = 6.00) participants, U = 59.00, z = .706, p = .529, using an exact sampling distribution for U. Documentation scores were not statistically significantly different between high construal level (Mdn = 5.50) and low construal level (Mdn = 6.00) participants, U = 27.50, z = -1.747, p = .089, using an exact sampling distribution for U. These results were not surprising given the small sample size.

3.4.3 Main Study

There were several important differences between the two pilot studies. First, the participants from Upwork spent more time engaging with the survey than the participants from Qualtrics. Specifically, the participants from Upwork spent approximately 19 minutes on average completing the survey, whereas the participants from Qualtrics spent approximately 9 minutes on average completing the survey. Second, the participants from Upwork wrote approximately 140 words on average for the manipulation task, whereas the participants from Qualtrics wrote approximately 66 words on average for the manipulation task. Third, the standard deviations for the dependent variables for Upwork were much smaller than the standard deviations for the dependent variables for Qualtrics. Qualtrics did, however, have higher reliabilities for the dependent variables. Fourth, the data collection period for Qualtrics was twice the data collection period for Upwork. Finally, the quality of participants for Qualtrics was suspect given the extremely low qualifying rate (approximately 1.9%). After careful consideration of these factors, I decided to collect the data for the main study from Upwork.

Study Participants. I recruited 65 participants for this study. Participants had between 3 years and 40 years of software development experience with an average of 12.5 years of software development experience. Most participants had college degrees (N = 51), with four holding doctorates. The median survey completion time was approximately 24 minutes. All participants were compensated for their time.

Manipulation and Measures. While the scenario for the main study remained the same as the one used in the pilot studies, the manipulation task and dependent variables were revised (see Appendix B). Specifically, I revised the manipulation task to ask participants either a series of four “how” questions or four “why” questions (see *Figure*

3.2). I made this change to strengthen the manipulation since it required participants to delve deeper into the details of a single task (i.e., minimize technical debt) with each subsequent "how" question or to think more abstractly with each subsequent "why" question, whereas the previous manipulation task with three separate "why" and "how" questions did not necessarily push participants to think more concretely or abstractly with each subsequent question. This approach was introduced by Freitas et al. (2004) and has been used successfully by Ho, Ke, and Liu (2015) to study user acceptance of a new e-learning system and Kaleta et al. (2019) to study online password use and intended password choice.

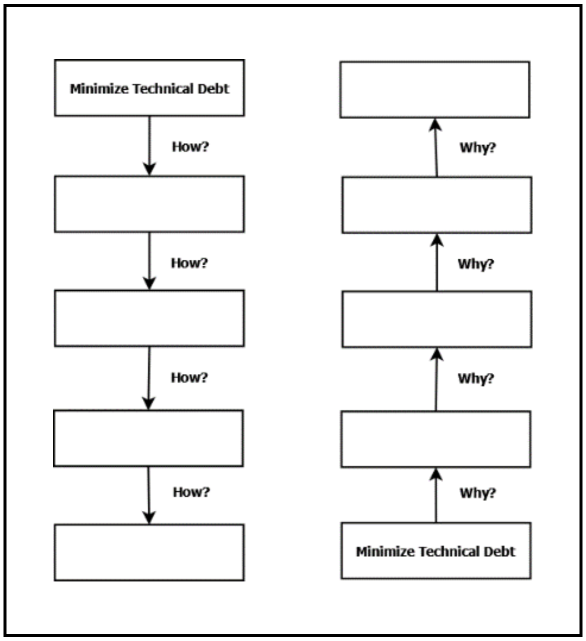


Figure 3.2: How/Why Manipulations

The dependent variables were revised to leverage the findings from my first essay. Specifically, nine of the most important causes of technical debt identified by software developers in the Delphi study were used to create dependent variable items for this study.

The one reversed coded item ("I would focus on developing the features.") performed poorly in the initial factor analysis (loading < .4) and was, therefore, excluded from subsequent analysis. The remaining items are presented in *Table 3.5*.

Table 3.5: Dependent Variables for Main Study

Factor	**Items**	**Reliability**
Factor 1: Coding	I would focus on resolving any bugs.	.761
	I would focus on testing and quality assurance.	
	I would focus on conducting code reviews.	
	I would focus on avoiding duplicate code.	
	I would focus on meeting coding standards.	
Factor 2: Refactoring	I would focus on refactoring the code where required.	N/A
Factor 3: Planning	I would focus on defining the requirements.	.619
	I would focus on developing the architecture or design.	

Examination of the factors and their items suggest that Factor 1 ("Coding") is more action-focused, whereas Factor 3 ("Planning") is more planning-focused. Factor 2 ("Refactoring") lies somewhere in between; to refactor, we must plan ahead while taking action on the code. While the reliability of Factor 1 is above the commonly accepted cutoff value of .7, the reliability of Factor 3 is somewhat below this cutoff. However, when dealing with psychological constructs, values below .7 can realistically be expected because of the diversity of the constructs being measured (Field, 2013; Kline, 2000). Consequently, I computed the means of Factor 1 and Factor 3 and used them in subsequent analysis. For comparison, I also present my analysis of the individual items in Factor 3.

Manipulation Check. To test whether the manipulations were effective, I followed the approach used by Kaleta et al. (2019). Specifically, I evaluated the four responses that each participant provided for the manipulation task. If a response addressed the question "why minimize technical debt," it was scored +1; if a response addressed the question "how to minimize technical debt," it was scored -1; and if a response did not address either, it was scored 0. I then summed the scores for each participant to create a construal level index that ranged from -4 to +4 with a higher positive score indicating a higher construal level. A Welch t-test was run to determine if there were differences in the construal level index between treatments. The construal level index for the high-level treatment (M = 3.74, SD = .898) was higher than the construal level index for the low-level treatment (M = -3.10, SD = 1.375), a statistically significant difference (t(51) = -23.475, p < .001). The manipulations were, therefore, effective.

In addition, I tested whether there were any differences in the perceived difficulty of the manipulation task, or the effort and willpower required to complete the manipulation task. Further, I tested whether the manipulation task may have had any unintended effects on self-control, positive affect, or negative affect (see Appendix C for items). Results from an independent samples t-test indicated that there were no statistically significant differences between the participants in either group. Further, inspecting the means show that participants found the manipulation tasks relatively easy and indicated that it only took low to moderate effort and willpower to complete them. The results are presented in *Table 3.6*.

Table 3.6: Results from Independent Samples T-Test

	Reliability (α)	**Low CL* Mean (SD)**	**High CL* Mean (SD)**	**Significance (2-tailed)**
Task Difficulty	N/A	2.29 (.643)	2.44 (.746)	.388
Effort	N/A	2.58 (.765)	2.71 (.906)	.551
Willpower	N/A	2.52 (.851)	2.56 (.960)	.851
Self-Control	.768	3.57 (.485)	3.74 (.623)	.211
Positive Affect	.912	3.23 (.890)	3.34 (.808)	.627
Negative Affect	.875	1.44 (.518)	1.42 (.585)	.860

Construal Level (CL).

Hypothesis Test. To establish a relationship between software developers' construal level and technical debt, the results would need to show a statistically significant difference between the means of the dependent variables in the high construal level group and the low construal level group. My expectation is that participants in the high construal level group will demonstrate a higher willingness to focus on good software development practices compared to participants in the low construal level group. My assumption is that a higher willingness to focus on good software development practices will lead to less technical debt. This assumption is based on the findings from Essay One, which showed that a lack of focus on good software development practices is among the most important causes of technical debt.

An independent-samples t-test was run to determine if there were differences in coding, refactoring, and planning between participants in the high construal level group and participants in the low construal level group. For coding, the difference between participants in the low construal level group (M = 4.32, SD = 1.03) and the high construal level group (M = 4.12, SD=1.25) was contrary to my prediction though insignificant (t(63)

= .697, p = .488). For refactoring, the difference between participants in the low construal level group (M = 3.58, SD = 1.54) and the high construal level group (M = 3.68, SD = 1.68) though in the predicted direction was insignificant (t(63) = -.238, p = .812). For planning, the difference between participants in the low construal level group (M = 5.66, SD = 1.12) and the high construal level group (M = 6.13, SD = 1.14) was in the predicted direction and marginally significant (t(63) = -1.680, p = .098) for a two-tailed test. For a one-tailed test, which is appropriate for one-directional hypotheses, the p-value for planning is significant (p < . 05).

Unpacking Factor 3 and running an independent samples t-test on the individual items reveals a consistent difference in the means in the predicted direction. Specifically, for defining the requirements, the difference between participants in the low construal level group (M = 6.06, SD = 1.48) and the high construal level group (M = 6.32, SD = 1.37) was in the predicted direction though not statistically significant (t(63) = -.728, p = .469), and for developing the architecture or design, the difference between participants in the low construal level group (M = 5.26, SD = 1.32) and the high construal level group (M = 5.94, SD = 1.13) was in the predicted direction and statistically significant (t(63) = -2.254, p < .05). This finding is particularly important since architectural debt has the highest cost of ownership compared to other forms of technical debt (Kruchten et al., 2019; Nord, 2018).

Qualitative Responses. While the primary objective of this essay was to examine the impact of software developers' construal level on technical debt, examining the written responses to the manipulation tasks provided additional insight on why software developers think it is important to minimize technical debt and how they typically minimize technical debt. A review of the written responses to the questions on “why minimize technical debt” revealed several reasons why experienced software developers minimize technical debt: to create code that can be easily expanded and maintained, to reduce the time and effort spent dealing technical debt in the future, to increase the speed of software development in the future, to make it easier for themselves (and other software developers) to operate in the future, and to deliver software that is less likely to have bugs or require “a bunch of patches here and there.”

A review of the written responses to the questions on “how to minimize technical debt” revealed several techniques that experienced software developers use to minimize technical debt: have a clearly-defined goal and a well-thought-out plan upfront, simplify the requirements by decomposing them into smaller achievable tasks, start with a minimum viable product that addresses the high-priority needs of the customer, ensure that the code is well documented and easily understood, leverage automated testing to ensure that the code is well tested, and follow best practices and standards established by others.

3.4.4 Discussion

While I initially hypothesized that software developers at a low construal level would be more willing to incur technical debt than software developers at a high construal level, the results present a more nuanced perspective. Specifically, the results of the factor analysis suggest that there are two main foci when thinking about technical debt – an action focus and a planning focus – and software developers respond to these in different ways. To elaborate, a low construal level seems to promote an action focus, while a high construal level seems to promote a planning focus. These findings are not necessarily inconsistent with construal level theory when we consider that a low construal level leads to a concrete mindset focused on the details of the ‘here and now’ (typical of coding), and a high construal level leads to an abstract mindset focused on the big picture of the ‘there and then’ (typical of planning).

Regulatory focus theory provides an interesting alternate lens with which to interpret the results. Crowe and Higgins (1997) distinguish between a promotion focus and a prevention focus in decision making. Specifically, they argued that a promotion focus was concerned with avoiding errors of omission, and a prevention focus was concerned with avoiding errors of commission. Stated simply, errors of omission refer to situations where we failed to act when we should have, and errors of commission refer to situations where we acted and we were wrong (Viswanathan, 2016). In our context, not planning for the future would represent an error of omission and writing poor quality code would represent an error of commission. Interpreting the results through this lens would, therefore, suggest that inducing a high construal level leads to a promotion focus (i.e., creating a good architecture and design) whereas inducing a low construal level leads to a prevention focus (i.e., avoid writing bad code). Examining the exact interplay between construal level and regulatory focus on technical debt might be a fruitful avenue for future research.

3.5 Conclusion

While I found limited evidence that software developers are likely to incur technical debt unintentionally based on their construal level, I did find evidence to support the relationship between a high construal level and an increased focus on developing the architecture or design. Further, the results of the factor analysis of the dependent variables suggest that software developers think about the causes of technical debt along a continuum with an action focus on one end and a planning focus on the other end. Future research can examine the nuance of these factors. As I have highlighted, regulatory focus theory might prove to be a useful theoretical lens for this exercise.

There were some limitations of this study, which may have contributed to the non-significant findings. First, it was difficult to recruit a large number of experienced software developers, which may have limited the power of the experiment to detect small (but potentially real) effects. A post hoc power analysis using G*Power (Faul, Erdfelder, Lang, & Buchner, 2007) revealed that I would need a sample size of approximately 144 participants to detect an effect size of 0.417 (which was observed for the "planning" factor) and a sample size of approximately 820 participants to detect an effect size of 0.174 (which was observed for the "coding" factor) for a one-tailed independent samples t-test at an alpha of 0.5 and power of 0.8. Second, there may have been a social desirability bias by the participants, who might not have wanted to admit that they would take shortcuts. Third, in the experiment, participants could indicate a high level of focus on all the dependent variables; however, in a real software development project, they would likely be forced to make actual trade-offs between meeting deadlines and minimizing technical debt.

3.6 Appendix A: Treatment and Measures for Pilot Studies

All participants were initially presented with the scenario task below.

SCENARIO TASK

Please read the scenario below and answer the questions that follow.

Imagine that you are a software developer for a consulting company in the United States. You have been assigned to work on a project for a client to write the software code for an online store.

The online store should allow your client to:

- upload high-resolution photos and videos of items along with their prices,
- showcase items by department, and
- provide special offers on select items.

The online store should allow customers to:

- create wish lists, purchase items, and browse related items,
- post reviews, and
- return defective or unwanted items.

You only have one month to develop an initial version of the software code for the online store that can be delivered to the client. However, based on your experience, one month is not enough time to develop the software code for all the required features and ensure that the software code for each feature is thoroughly reviewed, tested, and documented. Therefore, you will likely have to make some important trade-offs.

Then participants were exposed to either a low construal level treatment or a high construal level treatment. Participants in the low construal level treatment group were asked "HOW" and participants in the high construal level treatment group were asked "WHY."

Please describe in a few sentences HOW you would review the software code for the online store.

Please describe in a few sentences HOW you would test the software code for the online store.

Please describe in a few sentences HOW you would document the software code for the online store.

Afterward, all participants were presented with the dependent variables below. Items were measured on a 7-point scale from *(1) Strongly disagree* to *(7) Strongly agree*.

Code Review	1. I would carefully review the software code to ensure that it follows relevant best practices. 2. I would rigorously review the software code to ensure that it meets applicable coding standards.
Testing	3. I would conduct extensive testing to ensure that the software code performs as expected. 4. I would perform extensive testing to ensure that the software code does not have any bugs.
Documentation	5. I would write comprehensive documentation to ensure that I am able to modify the software code in the future. 6. I would create comprehensive documentation to ensure that my team members are able to understand how the software code works.

3.7 Appendix B: Treatment and Measures for Main Study

All participants were initially presented with the scenario task below.

SCENARIO TASK

Please read the scenario below carefully and answer the questions on the following pages.

Imagine that you are a software developer for a consulting company in the United States. You have been assigned to work on a project for a client to write the software for an online store.

The online store should allow your client to:

- upload high-resolution photos and videos of items along with their prices,
- showcase items by department, and
- provide special offers on select items.

The online store should allow customers to:

- create wish lists, purchase items, and browse related items,
- post reviews, and
- return defective or unwanted items.

You only have one month to develop an initial version of the software for the online store that can be delivered to the client. However, based on your experience, **one month is not enough time** to develop the software with all the required features and ensure that the code for each feature is thoroughly reviewed, tested, and documented**. Therefore, you will likely have to make an important trade-off between time spent developing the features and time spent minimizing technical debt.**

As a reminder, “in software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible.”

Then participants were exposed to either a low construal level treatment or a high construal level treatment. Participants in the low construal level treatment group were presented with the following thought exercise.

THOUGHT EXERCISE

Please complete the thought exercise below.

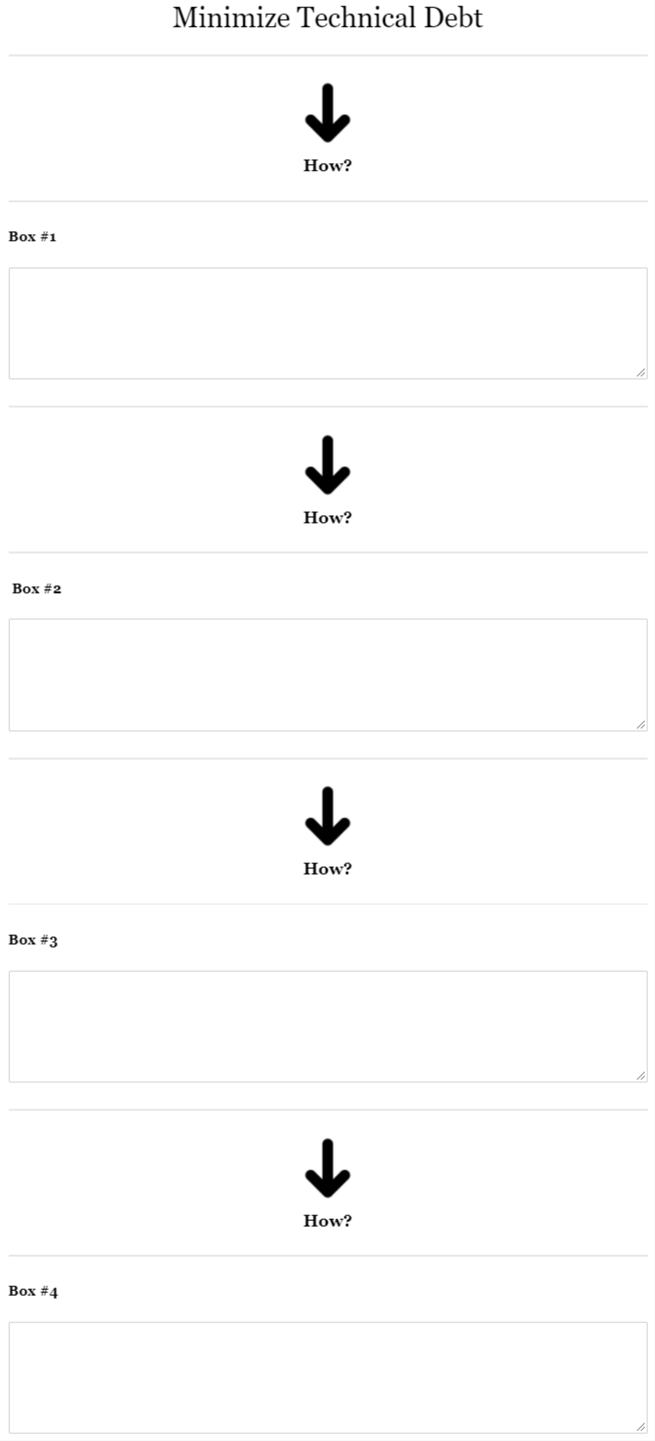
For everything that we do, there is always a process of ***how*** we do it. Moreover, we can often follow our broad life-goals down to our very specific behaviors.

For example, like most people, you want to experience life fully. How can you do this? Perhaps by seeing the world. How can you see the world? Perhaps by increasing your savings. How can you increase your savings? Perhaps by earning extra money. How can you earn extra money? Perhaps by completing online surveys such as this one.

Research suggests that you can improve your overall life satisfaction by engaging in thought exercises like the one above - in which you think about how your ultimate life goals can be expressed through specific actions.

In this online survey, we are testing such a technique. Our thought exercise is intended to focus your attention on ***how*** you do the things you do on your software development projects. For this thought exercise, please consider the following activity: ‘minimizing technical debt.’

Please complete Box #1, Box #2, Box #3, and Box #4 in that order each time asking “how” to your response in the previous box.



Minimize Technical Debt

↓

How?

Box #1

↓

How?

Box #2

↓

How?

Box #3

↓

How?

Box #4

Participants in the high construal level treatment group were presented with the following thought exercise.

THOUGHT EXERCISE

Please complete the thought exercise below.

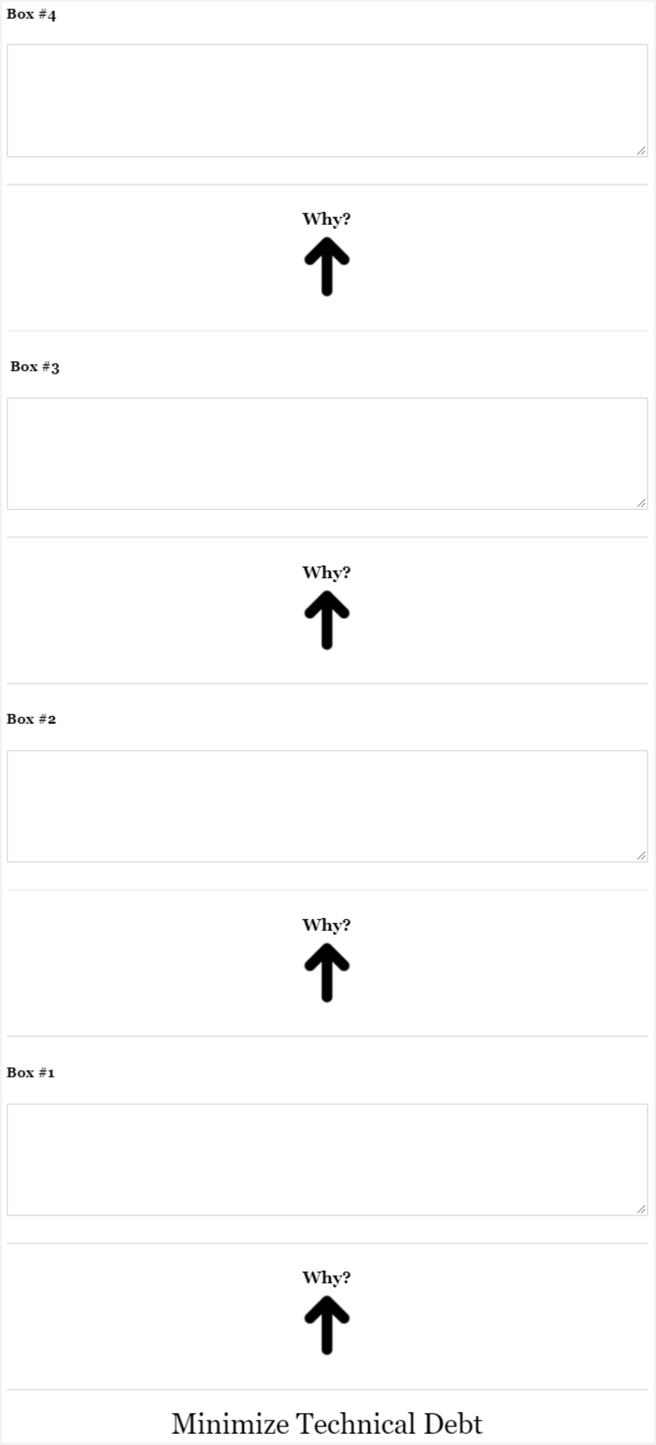
For everything that we do, there is always a reason ***why*** we do it. Moreover, we can often trace the causes of our behavior back to broad life-goals that we have.

For example, you are currently participating in an online survey. Why are you doing this? Perhaps to earn extra money. Why earn extra money? Perhaps to increase your savings. Why increase your savings? Perhaps you want to see the world. Why see the world? Perhaps you want to experience life fully.

Research suggests that you can improve your overall life satisfaction by engaging in thought exercises like the one above - in which you think about how your actions relate to your ultimate life goals.

In this online survey, we are testing such a technique. Our thought exercise is intended to focus your attention on ***why*** you do the things you do on your software development projects. For this thought exercise, please consider the following activity: ‘minimizing technical debt.’

Please complete Box #1, Box #2, Box #3, and Box #4 in that order each time asking "why" to your response in the previous box.



Box #4

Why?

Box #3

Why?

Box #2

Why?

Box #1

Why?

Minimize Technical Debt

Afterward, all participants were presented with the dependent variables below.

INSTRUCTIONS

Given that **you only have one month to develop an initial version of the software** for the online store that can be delivered to the client, please indicate where you would spend most of your time by answering the following questions.

1. I would focus on defining the requirements.
2. I would focus on developing the architecture or design.
3. I would focus on developing the features.
4. I would focus on testing and quality assurance.
5. I would focus on resolving any bugs.
6. I would focus on conducting code reviews.
7. I would focus on meeting coding standards.
8. I would focus on refactoring the code where required.
9. I would focus on avoiding duplicate code.

Items were measured on a 7-point scale from *(1) Strongly disagree* to *(7) Strongly agree*.

Finally, all participants were asked to answer questions related to task difficulty, effort, willpower, self-control, and positive and negative affect. The constructs and measures are presented below.

Task Difficulty (Self-Developed)	Task difficulty was measured using a single item "How difficult was the scenario task for you?" Responses were recorded on a 5-point scale from *(1) Very easy* to *(5) Very difficult*.
Effort (Self-Developed)	Effort was measured using a single item "How much effort did it take you to complete the scenario task?" Responses were recorded on a 5-point scale from *(1) Very low* to *(5) Very high*.
Willpower (Self-Developed)	Willpower was measured using a single item "How much willpower did it take you to complete the scenario task?" Responses were recorded on a 5-point scale from *(1) Very low* to *(5) Very high*.
Self-Control (Tangney, Baumeister, & Boone, 2004)	Self-control was measured using 13 items on a 5-point scale from *(1) Not at all* to *(5) Very much*: 1. I am good at resisting temptation. 2. I have a hard time breaking bad habits. (R) 3. I am lazy. (R) 4. I say inappropriate things. (R) 5. I do certain things that are bad for me, if they are fun. (R) 6. I refuse things that are bad for me. 7. I wish I had more self-discipline. (R) 8. People would say that I have iron self-discipline 9. Pleasure and fun sometimes keep me from getting work done. (R) 10. I have trouble concentrating. (R) 11. I am able to work effectively toward long-term goals. 12. Sometimes I can't stop myself from doing something, even if I know it is wrong. (R) 13. I often act without thinking through all the alternatives. (R) ***Reverse coded items are indicated with (R).***

Positive Affect and Negative Affect (Watson, Clark, & Tellegen, 1988)	Positive affect and negative affect were measured using 20 items on a 5-point scale from *(1) Very slightly or not at all* to *(5) Extremely*: 1. Interested (P) 2. Distressed 3. Excited (P) 4. Upset 5. Strong (P) 6. Guilty 7. Scared 8. Hostile 9. Enthusiastic (P) 10. Proud (P) 11. Irritable 12. Alert (P) 13. Ashamed 14. Inspired (P) 15. Nervous 16. Determined (P) 17. Attentive (P) 18. Jittery 19. Active (P) 20. Afraid ***Positive items are indicated with (P).***

4 Conclusion

Managing technical debt in agile software development projects remains a significant challenge for software practitioners. Consequently, researchers have placed increased emphasis on this area to better understand the phenomenon and provide potential solutions. In particular, researchers have argued that we must first be able to identify the causes of technical debt before we can effectively manage it (Kruchten et al., 2012). However, to date, there has been no systematic attempt to identify and prioritize the causes of technical debt in agile software development projects. This prompted my first overarching research question: *What are the most important causes of technical debt in agile software development projects?*

In my first essay, I answer this research question by conducting a ranking-type Delphi study (Schmidt, 1997) of 86 experienced software practitioners. Specifically, I generated a verified list of 55 causes of technical debt in agile software development projects. From this list, I identified and prioritized the 15 causes of technical debt that were most important to a majority of the IT project managers in this study and the 12 causes of technical debt that were most important to a majority of the software developers in this study. In my second essay, I also contribute to answering this research question by providing initial evidence that software developers' construal level can promote the unintentional accumulation of technical debt. Specifically, by conducting an online experiment with 65 experienced software developers, I demonstrated that software developers primed at a high construal level are more likely to focus on developing a good architecture or design than software developers primed at a low construal level.

Having identified the most important causes of technical debt in agile software development projects, the next logical step was to explore potential solutions for managing these causes of technical debt. This prompted my second overarching research question: *What are some potential techniques for managing technical debt in agile software development projects?*

In my first essay, I answer this research question by interviewing a select group of IT project managers and software developers from the Delphi study. Specifically, I identified 13 potential techniques for managing the eight causes of technical debt that were a priority for both IT project managers and software developers. Further, these interviews revealed four plausible explanations for the different rankings by IT project managers and software developers. In my second essay, I also contribute to answering this research question by examining the qualitative responses from software developers. Specifically, the software developers in my second essay emphasized the need to have a clearly defined goal and a well thought out plan upfront. Further, they suggested the need to simplify requirements so that we can start with a minimum viable product that addresses the high priority needs of the customer. This mitigates the pressure of trying to deliver all the requirements by the deadline, which invariably results in taking shortcuts.

Collectively, the findings from my two essays represent a meaningful contribution to the literature on managing technical debt in agile software development projects. In addition to advancing our knowledge of the causes of technical debt in agile software development projects and offering potential techniques for managing the most important of these causes of technical debt, the findings provide a strong foundation on which to conduct further research. For example, researchers could use the verified list of causes to conduct seeded Delphi studies (Keil, Tiwana, & Bush, 2002) involving other stakeholder groups such as end-users and customer representatives or other cultural groups such as IT project managers and software developers in India (a powerhouse for software development). Conducting seeded Delphi studies saves researchers valuable time and effort by eliminating the most labor-intensive phase of the Delphi study – identification of the issues. It might also be worthwhile for researchers to examine which causes of technical debt contribute the greatest amount of technical debt when objectively analyzing a portion of code. Researchers could also conduct action research, case studies, and field experiments to test the efficacy of the potential techniques for managing technical debt uncovered in this dissertation. Researchers could also delve deeper into the role of our cognitive processes and biases on the unintentional accumulation of technical debt. For example, researchers could examine the interplay of construal level (a cognitive process) and other theories, such as regulatory focus theory, on technical debt. I have already begun to examine the role of a pervasive cognitive bias – *the planning fallacy* – on technical debt in agile software development projects (see Boodraj, 2018).

The findings from my two essays also represent a meaningful contribution to practice. Software practitioners can use the verified list of causes as a checklist to identify technical debt items in their agile software development projects. This checklist can be expanded as they identify additional causes of technical debt. Further, practitioners can test the potential techniques that were uncovered to see whether they are effective in their unique organizational context. Finally, practitioners can be mindful of how their focus on the details of the task at hand may induce a low construal level, thereby causing them to miss the big picture of the architecture or design.

One of the key strengths of this dissertation was the participation of experienced software development practitioners. In essay one, the 86 participants had an average of 8.5 years of experience working on agile software development projects. This meant that the list of 55 causes of technical debt that was identified was the result of 735 years of experience in agile software development projects. In essay two, the 65 participants had an average of 12.5 years of software development experience, with several having more than 25 years of experience. Using participants with such extensive experience represented a deliberate effort on my part to increase the overall relevance of this dissertation. However, recruiting enough qualified and interested participants proved to be a significant limitation. While the findings from this dissertation represent a meaningful contribution to the literature on technical debt, they are just the beginning of what I hope will be a lifelong pursuit of advancing both the research on and practice of managing technical debt in agile software development projects.

References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile Software Development Methods: Review and Analysis*. Retrieved from https://www.vttresearch.com/sites/default/files/pdf/publications/2002/P478.pdf

Alves, N. S. R., Mendes, T. S., de Mendonça Netoa, M. G., Spínola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology, 70,* 100-121.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2015). The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, *64*, 52-73.

Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing Technical Debt in Software Engineering. *Dagstuhl Reports, 6*(4), 110–138.

Baham, C. (2017). *Exploring the Relationship between Perceptions of Agile Software Development and Technical Debt.* Paper presented at the 22nd Americas Conference on Information Systems, Boston, MA.

Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, *2*(1), 26-30.

Bavani, R. (2012). Distributed Agile, Agile Testing, and Technical Debt. *IEEE Software, 29*(6), 28-33.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org/

Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology, 82*, 139-158.

Bjärås, F., & Ericsson, P. (2018). *Practices to minimize technical debt in agile software development: A practical student project case study*. Retrieved from http://fileadmin.cs.lth.se/cs/Education/EDAN80/Reports/2018/BjarasEricsson.pdf

Boodraj, M. (2018). *Leveraging the Planning Fallacy to Manage Technical Debt in Agile Software Development Projects.* Paper presented at the 13th International Research Workshop on IT Project Management, San Francisco, CA.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., . . . Zazworka, N. (2010). *Managing Technical Debt in Software-Reliant Systems*. Paper presented at the FSE/SDP Workshop on Future of Software Engineering Research, Santa Fe, NM.

Caires, V., Rios, N., Holvitie, J., Leppänen, V., Mendonça, M. G., & Spínola, R. O. (2018). *Investigating the Effects of Agile Practices and Processes on Technical Debt - The Viewpoint of the Brazilian Software Industry*. Paper presented at the 30th International Conference on Software Engineering and Knowledge Engineering, Redwood City, CA.

Camden, C. (2013). Avoid getting buried in technical debt. Retrieved from https://www.techrepublic.com/blog/software-engineer/avoid-getting-buried-in-technical-debt/

Cha, S., Dong, Q. H., & Vogel-Heuser, B. (2018). *Preventing Technical Debt For Automated Production System Maintenance Using Systematic Change Effort Estimation With Considering Contingent Cost*. Paper presented at the 16th International Conference on Industrial Informatics, Porto, Portugal.

Crowe, E., & Higgins, E. T. (1997). Regulatory focus and strategic inclinations: Promotion and prevention in decision-making. *Organizational Behavior and Human Decision Processes, 69*(2), 117-132.

Cunningham, W. (1992). *The WyCash Portfolio Management System*. Paper presented at the Object-oriented Programming Systems, Languages, and Applications, Vancouver, Canada.

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the Principal of an Application's Technical Debt. *IEEE Software, 29*(6), 34-42.

Daniel, E. M., & White, A. (2005). The future of inter-organisational system linkages: findings of an international Delphi study. *European Journal of Information Systems, 14*(2), 188-203.

Dhar, R., & Kim, E. Y. (2007). Seeing the forest or the trees: Implications of construal level theory for consumer choice. *Journal of Consumer Psychology, 17*(2), 96-100.

Faul, F., Erdfelder, E., Lang, A.-G., & Buchner, A. (2007). G* Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods, 39*(2), 175-191.

Field, A. (2013). *Discovering Statistics Using IBM SPSS Statistics* (4th ed.). Thousand Oaks, CA: Sage.

Fontana, F. A., Chatzigeorgiou, A., Trumler, W., Izurieta, C., Avgeriou, P., & Nord, R. L. (2017). Technical Debt in Agile Development: Report on the Ninth Workshop on

Managing Technical Debt (MTD 2017). *ACM SIGSOFT Software Engineering Notes, 42*(3), 18-21.

Fowler, M. (2003). Technical Debt. Retrieved from https://martinfowler.com/bliki/TechnicalDebt.html

Fowler, M. (2009). Technical Debt Quadrant. Retrieved from https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Freitas, A. L., Gollwitzer, P., & Trope, Y. (2004). The influence of abstract and concrete mindsets on anticipating and guiding others' self-regulatory efforts. *Journal of Experimental Social Psychology, 40*(6), 739-752.

Fujita, K. (2008). Seeing the forest beyond the trees: A construal-level approach to self-control. *Social and Personality Psychology Compass, 2*(3), 1475-1496.

Fujita, K., & Carnevale, J. J. (2012). Transcending temptation through abstraction: The role of construal level in self-control. *Current Directions in Psychological Science, 21*(4), 248-252.

Fujita, K., Henderson, M. D., Eng, J., Trope, Y., & Liberman, N. (2006). Spatial distance and mental construal of social events. *Psychological Science, 17*(4), 278-282.

Fujita, K., Trope, Y., Liberman, N., & Levin-Sagi, M. (2006). Construal levels and self-control. *Journal of Personality and Social Psychology, 90*(3), 351.

Guo, Y., Spínola, R. O., & Seaman, C. (2016). Exploring the costs of technical debt management–a case study. *Empirical Software Engineering, 21*(1), 159-182.

Henderson, M. D., Fujita, K., Trope, Y., & Liberman, N. (2006). Transcending the" here": the effect of spatial distance on social judgment. *Journal of Personality and Social Psychology, 91*(5), 845-856.

Ho, C. K., Ke, W., & Liu, H. (2015). Choice decision of e-learning system: Implications from construal level theory. *Information & Management, 52*(2), 160-169.

Holvitie, J., Licorishc, S. A., Spínola, R. O., Hyrynsalmi, S., MacDonell, S. G., Mendesg, T. S., . . . Leppänen, V. (2018). Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology, 96*, 141-160.

Kaleta, J. P., Lee, J. S., & Yoo, S. (2019). Nudging with construal level theory to improve online password use and intended password choice. *Information Technology & People*.

Kasi, V., Keil, M., Mathiassen, L., & Pedersen, K. (2008). The post mortem paradox: a Delphi study of IT specialist perceptions. *European Journal of Information Systems*, *17*(1), 62-78.

Keil, M., Lee, H. K., & Deng, T. (2013). Understanding the most critical skills for managing IT projects: A Delphi study of IT project managers. *Information & Management, 50*(7), 398-414.

Keil, M., Tiwana, A., & Bush, A. (2002). Reconciling user and project manager perceptions of IT project risk: a Delphi study. *Information Systems Journal, 12*(2), 103-119.

Kendall, M. G., & Gibbons, J. D. (1990). *Rank Correlation Methods* (5th ed.): Edward Arnold.

Kline, P. (2000). *Handbook of Psychological Testing* (2nd ed.). New Fetter Lane, London: Routledge.

Klinger, T., Tarr, P., Wagstrom, P., & Williams, C. (2011). *An Enterprise Perspective on Technical Debt.* Paper presented at the 2nd International Workshop on Managing Technical Debt, Honolulu, HI.

Kruchten, P. (2019). *The End of Agile as We Know It.* Paper presented at the Proceedings of the International Conference on Software and System Processes, Montreal, Canada.

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software, 29*(6), 18-21.

Kruchten, P., Nord, R. L., & Ozkaya, I. (2019). *Managing Technical Debt: Reducing Friction in Software Development* Addison-Wesley Professional.

Kruchten, P., Nord, R. L., Ozkaya, I., & Falessi, D. (2013). Technical Debt: Towards a Crisper Definition. *ACM SIGSOFT Software Engineering Notes, 38*(5), 51-54.

Kyte, A. (2010). *Measure and Manage Your IT Debt*. Retrieved from https://www.gartner.com/en/documents/1419325/measure-and-manage-your-it-debt

Lee, J. S., Keil, M., & Shalev, E. (2019). Seeing the Trees or the Forest? The Effect of IT Project Managers' Mental Construal on IT Project Risk Management Activities. *Information Systems Research, 30*(3), 1051-1072.

Letouzey, J.-L., & Ilkiewicz, M. (2012). Managing Technical Debt with the SQALE Method. *IEEE Software, 29*(6), 44-51.

Li, P., Maruping, L. M., & Mathiassen, L. (2020). *Developing and Managing Open Source Enterprise Systems through Open Superposition: A Digital Options and Technical Debt Perspective.* Paper presented at the 25th Americas Conference on Information Systems, Virtual Conference.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software, 101*, 193-220.

Liberman, N., & Trope, Y. (1998). The role of feasibility and desirability considerations in near and distant future decisions: A test of temporal construal theory. *Journal of Personality and Social Psychology, 75*(1), 5-18.

Liberman, N., & Trope, Y. (2003). Construal Level Theory of Intertemporal Judgment and Decision. In G. Loewenstein, D. Read, & R. Baumeister (Eds.), *Time and Decision: Economic and Psychological Perspectives of Intertemporal Choice* (pp. 245-276): Russell Sage Foundation.

Liberman, N., Trope, Y., & Wakslak, C. (2007). Construal level theory and consumer behavior. *Journal of Consumer Psychology, 17*(2), 113-117.

Lim, E., Taksande, N., & Seaman, C. (2012). A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software, 29*(6), 22-27.

Maglio, S. J., Trope, Y., & Liberman, N. (2013). Distance from a distance: Psychological distance reduces sensitivity to any further psychological distance. *Journal of Experimental Psychology: General, 142*(3), 644-657.

Mann, H. B., & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics, 18*(1), 50-60.

Maruping, L. M., & Matook, S. (2020). The Multiplex Nature of the Customer Representative Role in Agile Information Systems Development. *MIS Quarterly, 44*(3), 1411-1437.

Maruping, L. M., Venkatesh, V., & Agarwal, R. (2009). A control theory perspective on agile methodology use and changing user requirements. *Information Systems Research, 20*(3), 377-399.

McConnell, S. (2008). *Managing Technical Debt*. Retrieved from http://www.construx.com/uploadedfiles/resources/whitepapers/Managing%20Technical%20Debt.pdf

Nevo, D., & Chan, Y. E. (2007). A Delphi study of knowledge management systems: Scope and requirements. *Information & Management, 44*(6), 583-597.

Nord, R. (2018). *Managing Technical Debt in Agile Environments*. Retrieved from https://apps.dtic.mil/sti/pdfs/AD1083762.pdf

Oliveira, F., Goldman, A., & Santos, V. (2015). *Managing Technical Debt in Software Projects Using Scrum: An Action Research.* Paper presented at the 2015 Agile Conference, Washington, DC.

Project Management Institute. (2018). *Pulse of the Profession 2018*. Retrieved from https://www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2018

Ramasubbu, N., & Kemerer, C. F. (2014). Managing Technical Debt in Enterprise Software Packages. *IEEE Transactions on Software Engineering, 40*(8), 758-772.

Ramasubbu, N., & Kemerer, C. F. (2016). Technical debt and the reliability of enterprise software systems: A competing risks analysis. *Management Science, 62*(5), 1487-1510.

Reyt, J.-N., & Wiesenfeld, B. M. (2015). Seeing the forest for the trees: Exploratory learning, mobile technology, and knowledge workers' role integration behaviors. *Academy of Management Journal, 58*(3), 739-762.

Rios, N., de Mendonça Neto, M. G., & Spínola, R. O. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology, 102*, 117-145.

Rolland, K. H., Mathiassen, L., & Rai, A. (2018). Managing digital platforms in user organizations: the interactions between digital options and digital debt. *Information Systems Research, 29*(2), 419-443.

Ross, L. (1987). The Problem of Construal in Social Inference and Social Psychology. In N. E. Grunberg, R. E. Nisbett, J. Rodin, & J. E. Singer (Eds.), *A Distinctive Approach to Psychological Research: The Influence of Stanley Schachter* (pp. 118-150): Lawrence Erlbaum Associates, Inc.

Sambhara, C., Rai, A., Keil, M., & Kasi, V. (2017). Risks and Controls in Internet-Enabled Reverse Auctions: Perspectives from Buyers and Suppliers. *Journal of Management Information Systems, 34*(4), 1113-1142.

Schmidt, R. C. (1997). Managing Delphi Surveys Using Nonparametric Statistical Techniques. *Decision Sciences, 28*(3), 763-774.

Shadish, W. R., Cook, T. D., & Campbell, D. T. (2002). *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Boston, MA: Houghton Mifflin Company.

Soderberg, C. K., Callahan, S. P., Kochersberger, A. O., Amit, E., & Ledgerwood, A. (2015). The Effects of Psychological Distance on Abstraction: Two Meta-Analyses. *Psychological Bulletin*, *141*(3), 525-548.

Tangney, J. P., Baumeister, R. F., & Boone, A. L. (2004). High Self-Control Predicts Good Adjustment, Less Pathology, Better Grades, and Interpersonal Success. *Journal of Personality*, *72*(2), 271-324.

Trochim, W. M., Donnelly, J. P., & Arora, K. (2016). *Research Methods: The Essential Knowledge Base*. Boston, MA: Cengage Learning.

Trope, Y., & Liberman, N. (2010). Construal-level theory of psychological distance. *Psychological Review*, *117*(2), 440-463.

Vallacher, R. R., & Wegner, D. M. (1989). Levels of Personal Agency: Individual Variation in Action Identification. *Journal of Personality and Social Psychology*, *57*(4), 660-671.

Viswanathan, V. (2016). What's worse - errors of omission or errors of commission? Retrieved from https://www.mawer.com/the-art-of-boring/blog/whats-worse-errors-omission-errors-commission/

Wakslak, C., & Trope, Y. (2009). The effect of construal level on subjective probability estimates. *Psychological Science*, *20*(1), 52-58.

Wan, E. W., & Agrawal, N. (2011). Carryover effects of self-control on decision making: A construal-level perspective. *Journal of Consumer Research*, *38*(1), 199-214.

Watson, D., Clark, L. A., & Tellegen, A. (1988). Development and Validation of Brief Measures of Positive and Negative Affect: The PANAS Scales. *Journal of Personality and Social Psychology*, *54*(6), 1063.

Yang, Y., & Boodraj, M. (2020). *Managing Code Debt in Open Source Software Development Projects: A Digital Options Perspective.* Paper presented at the 25th Americas Conference on Information Systems, Virtual Conference.