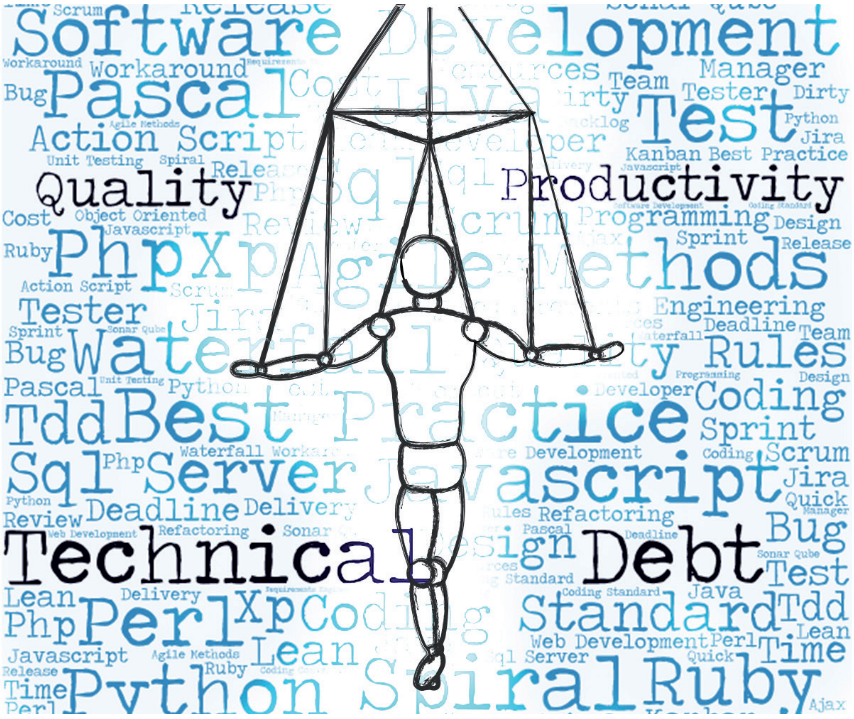
JYVÄSKYLÄ STUDIES IN COMPUTING

258

Hadi Ghanbari

Investigating the Causal Mechanisms Underlying the Customization of Software Development Methods



JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 258

Hadi Ghanbari

Investigating the Causal Mechanisms Underlying the Customization of Software Development Methods

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora-rakennuksen Lea Pulkkisen salissa
tammikuun 7. päivänä 2017 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in building Agora, Lea Pulkkinen hall, on January 7, 2017 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2017

Investigating the Causal Mechanisms Underlying the Customization of Software Development Methods

JYVÄSKYLÄ STUDIES IN COMPUTING 258

Hadi Ghanbari

Investigating the Causal Mechanisms Underlying the Customization of Software Development Methods

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2017

Editors
Marja-Leena Rantalainen
Department of Mathematical Information Technology, University of Jyväskylä
Pekka Olsbo, Ville Korkiakangas
Publishing Unit, University Library of Jyväskylä

Cover picture by Heta Kangasniemi

URN:ISBN:978-951-39-6944-8
ISBN 978-951-39-6944-8 (PDF)

ISBN 978-951-39-6943-1 (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä 2017

ABSTRACT

Ghanbari, Hadi
Investigating the Causal Mechanisms Underlying the Customization of Software Development Methods
Jyväskylä: University of Jyväskylä, 2017, 129 p.
(Jyväskylä Studies in Computing
ISSN 1456-5390; 258)
ISBN 978-951-39-6943-1 (nid.)
ISBN 978-951-39-6944-8 (PDF)
diss.

Over the last four decades, software development has been one of the mainstream topics in the Software Engineering and Information Systems disciplines. Thousands of methods have been put forward offering prescriptions for software development processes. The goal of these methods is to produce high-quality software in a systematic manner. However, it is widely known that these methods are rarely followed as prescribed – developers often modify or skip different steps, practices, or quality rules recommended by software development methods. While a group of previous studies suggests that maximizing the flexibility and leanness of software development processes is the key driver of such customizations, another group argues that the inadequacy of these methods to fulfill stakeholders' expectations is the main reason they are customized in practice. However, to the best of our knowledge, there are no theory-based and empirically grounded explanations elucidating the causal mechanisms underlying the customization of software development methods. We attempted to take the first step in overcoming this gap by conducting this doctoral research.

We first conducted an extensive Systematic Literature Review to identify the gaps in research on customization of software development methods and to clarify the need for addressing these shortcomings. Following this, we attempted to address some of the identified gaps by conducting a longitudinal field study. Collecting data from different software projects across industrial domains and using the Grounded Theory Methodology, we built a process theory called *Theory of Software Development Balance*. In this theory, we explain the mechanisms through which software development methods are customized in practice in order to maintain balance among contrasting and sometimes contradictory contextual forces associated with software projects.

Keywords: Software Development Methods, Information Systems Development, Technical Debt, Behavioral Software Engineering, Balancing, Grounded Theory, Process Theory, Systematic Literature Review

Author

Hadi Ghanbari
Department of Computer Science and Information Systems
University of Jyväskylä
Finland

Supervisors

Professor Mikko Siponen
Department of Computer Science and Information Systems
University of Jyväskylä
Finland

Professor Kalle Lyytinen
Department of Design & Innovation
Weatherhead School of Management
Case Western Reserve University
United States

Reviewers

Professor Lars Mathiassen
Center for Process Innovation
J. Mack Robinson College of Business
Georgia State University
United States

Professor Pekka Abrahamsson
Department of Computer and Information Science
Norwegian University of Science and Technology
Norway

Opponent

Professor Kieran Conboy
Lero Research Centre & Whitaker Institute
School of Business & Economics
National University of Ireland Galway
Ireland

DEDICATION

I dedicate this work to Heta who has been a great support for me, not just as a lovely spouse, but also as a caring friend. *Äzizäm*, it is very hard to describe how grateful I am to you. Thank you so much for always being ready to discuss about my research and comment on my work and at the same time listening to me complaining about how difficult it is to study Finnish. I am sure without your help it would have been impossible for me to simultaneously work on these two exhausting projects.

I also dedicate this work to my parents. Well, I don't know how I could thank them enough since without them I would not even exist. *Maman*, *Baba*, I have always tried to be a good son for you so at least I hope to deserve whatever you have done for me. I know that sometimes – or maybe many times – I made you feel worried. At least I hope that reading this text will make you happy and realize that so far I have managed to achieve many good things in my life. To you I owe all those achievements.

ACKNOWLEDGEMENTS

Preparing this doctoral dissertation has been a long process, which was initiated by my eagerness to further my education and deepen my understanding about *software and information systems development*. However, this would have not been possible without the support and contribution of many people that I would like to thank.

First of all, I want to express my gratitude to Dr. Mikko Siponen, my advisor, not only for his support and instruction during the last four years, but also for providing me with the opportunity to meet internationally known scholars and to present my work to them. I have always aimed for the highest level in my life, and seeing Mikko's achievements as a researcher has given me further motivation to aim for only the best results in my research. I would also like to thank my external advisor Dr. Kalle Lyytinen for providing me with his critical insights and constructive comments that have enabled me to think outside the box and find new ideas to improve my work.

I would like to thank a very influential colleague to me, Dr. Tero Vartiainen, who has challenged me in conducting research, thinking critically, and learning Finnish. I must also thank Dr. Jouni Similä who has been a great support for me both during and after my master's degree studies at the University of Oulu. Additionally, I would like to thank the external reviewers of my thesis, Dr. Lars Mathiassen and Dr. Pekka Abrahamsson, and all the other scholars who have provided feedback on my doctoral dissertation.

I must acknowledge University of Jyväskylä and the Finnish Funding Agency for Technology and Innovation (TEKES) for funding my doctoral research. I also appreciate the support from my colleagues at the Department of Computer Science and Information Systems, University of Jyväskylä, especially my teammates Hemin Jiang, Jungwon Kuem, Naomi Woods, Ying Li, Alain Tambe, Manja Nikolovska, and Aggeliki Tsohou. Additionally, I must thank Tapio Tammi, Seija Paananen, and Tiina Lampinen for their supports during the last four years.

Well, conducting a four year project needs plenty of motivation and energy; thus, I am very grateful to my friends, especially Heidi, Sara, Nastaran, Payam, Amin, Tolga, Viktor, Hoji, Ehsan, and Mansoor, who were always up for having fun, enjoying life, and helping me to recharge my batteries to get ready for work.

Last but not least, my deepest gratitude goes to my family for all the love and support that they have given to me.

Hadi Ghanbari
Jyväskylä, December 9, 2016

FIGURES

TABLES

CONTENTS

"It is established in the sciences that no knowledge is acquired save through the study of its causes and beginnings, if it has had causes and beginnings, nor completed except by knowledge of its accidents and accompanying essentials."

– Avicenna[1], *Canon of Medicine*, ca. 1020

[1] Iranian philosopher, scientist, and physician (Rothman 2012)

1 INTRODUCTION

Nowadays, software and information systems are intertwined with every area of human life and business world in a way that it is extremely difficult for societies to survive without them. Consequently, software development continues to be a mainstream topic in both Software Engineering (SE) and Information Systems (IS) and at the same time receives a considerable amount of attention from practitioners.

Despite all the financial and human resources that have been spent by the software community to improve both the productivity of software processes and the quality of software products, software development projects still face severe challenges (The Standish Group International, Inc. 2009). A research by Standish Group (2009) shows that 24% of system development projects failed before completion, while 44% of projects faced severe challenges such as exceeding budget and delivery time or missing specified features and functions.

To overcome these challenges, thousands of software development methods have been put forward (Conboy & Fitzgerald 2010, Huisman & Iivari 2006) offering prescriptions for performing a sequence of activities for developing software in a systematic way (Avison & Fitzgerald 2003). However, it is widely reported by scholars and practitioners that these software development methods are not followed fully but are instead customized (Conboy & Fitzgerald 2010, Iivari & Iivari 2011, Ralph 2015b). Often, in practice, certain steps and practices suggested by these methods are omitted or performed imperfectly (Ralph 2015b, Ahonen & Junttila 2003).

From the organizational viewpoint, maximizing the leanness of software development processes is a key goal of such customizations (Vartiainen & Siponen 2012, Codabux & Williams 2013, McConnell 2007, Ahonen & Junttila 2003, Baskerville & Pries-Heje 2004). However, to the best of our knowledge, there are no scientific, i.e., theory-based and empirically grounded, explanations elucidating the causal mechanisms underlying the customization of the methods. Especially, there is a lack of research to investigate the customization of software development methods from developers' perspective.

Therefore, in this doctoral dissertation, our goal is to perform an in-depth analysis of the logic behind customizing software development methods in order to provide new scientific explanations of why and how certain software development steps or practices are ignored or modified. We attempted to reach this goal by drawing on SE and IS literature and by conducting a longitudinal interpretive field study (Klein & Myers 1999) across industrial domains.

The results of this research not only reveal the conditions under which developers decide to ignore software development methods but also explain the processes through which these methods are customized as developers decide to skip or modify certain practices or steps. Such empirically grounded explanations are beneficial for both research and practice, and therefore, they enable the software community to improve software development processes while reinforcing both individual and organizational tendencies towards the use of software development methods. The contributions of this doctoral dissertation to both research and practice are discussed in later sections.

1.1 Research objectives

Over the last four decades, increasing interest has been shown in making sense of and comparing and exploring the underlying theoretical assumptions of different software development methods (Iivari 1991, Iivari & Lyytinen 1998, Hirschheim, Klein & Lyytinen 1995, Iivari, Hirschheim & Klein 1998b, Iivari 1990, Lyytinen 1987). These studies have improved our understanding of different methods and their limits and have suggested important avenues for future research.

However, it has been suggested by a group of researchers (Hirschheim & Newman 1991, Orlikowski & Iacono 2001, Ralph & Wand 2008, Truex, Baskerville & Travis 2000, Sjøberg et al. 2008, Wohlin, Šmite & Moe 2015, Johnson, Ekstedt & Jacobson 2012) that the lack of theories about software and systems development is one of the key unresolved problems in SE and IS fields. In particular, there is a very limited understanding of the usage and practicability of software development methods (Iivari & Maansaari 1998, Ralph & Wand 2008, Truex, Baskerville & Travis 2000, Wynekoop & Russo 1995).

Software development is a highly dynamic and complex phenomenon consisting of a set of interrelated processes—such as requirements engineering, software construction, software evaluation and software maintenance—of which the ultimate goal is to transform user requirements into working software (Sabherwal & Robey 1993, Slaughter et al. 2006, Truex, Baskerville & Travis 2000). Over time, as software projects started to fail or were faced with severe challenges, organizations realized a need for more-systematic and formalized approaches to manage complexity in projects and to improve the quality of software processes (Avison & Fitzgerald 2003). Consequently, a large number of software development methods were designed with the goal of providing effective means and guidelines to assist developers in performing a series of predefined steps and activities during software development processes (Iivari & Maansaari 1998, Iivari, Hirschheim & Klein 2004, Iivari, Hirschheim & Klein 1998a).

A variety of definitions for software and systems development methods has been suggested in the SE and IS disciplines. For instance, systems development methodology is defined by Hirschheim, Klein, and Lyytinen (1995, p.22) as "an organized collection of concepts, methods, beliefs, values, and normative principles supported by material resources". Alternatively, Wynekoop and Russo (1995) define systems development method as:

> A systematic approach to conducting at least one complete phase (e.g. requirements analysis, design) of systems development, consisting of a set of guidelines, activities, techniques and tools, based on a particular philosophy of systems development and the target system. (Wynekoop & Russo 1995, p.66)

This is similar to another definition suggested by the *Guide to the Software Engineering Body of Knowledge* (Bourque & Fairley 2014), which is widely accepted in the software community:

> Software development methods impose structure on software engineering with the goal of making that activity systematic, repeatable, and ultimately more success-oriented […] Methods provide an approach to the systematic specification, design, construction, test, and verification of the end-item software and associated work products. (Bourque & Fairley 2014, p.162)

Hirschheim and Klein (1992) suggest that software and systems development methods can be broken down into process and modeling notation. Process indicates how, and in which order, the development of the system is carried out (e.g., the Waterfall model), and modeling notation describes how things are presented (e.g., Object-Oriented notations).

It is argued by a group of researchers that the software and system development process to be followed is an important factor that may contribute to projects' success (Baskerville et al., 2004; Iivari, 1991; Ralph & Wand, 2008). Thus, designing more-effective and more-efficient methods has received a great deal of attention from the software community (Leppanen 2006, MacCormack et al. 2003, Banker, Davis & Slaughter 1998, Lyytinen 1989). These software and system development methods introduce a set of principles and prescriptions to system developers that help them understand what kinds of practices and techniques they should use for developing software. However, despite a plethora of software development methods suggested by the software community (Conboy & Fitzgerald 2010, Huisman & Iivari 2006), it is widely known that there is no single method that suits all kinds of software development projects (Brinkkemper 1996, Iivari 1991). Therefore, software development teams rarely follow the methods as prescribed (Ralph 2015b) but customize them (Conboy & Fitzgerald 2010, Iivari & Iivari 2011) or even do not follow them at all (Truex, Baskerville & Travis 2000). In other words, it is very common for development teams to omit or modify different steps, rules, or practices of a given method.

The majority of previous studies investigate the customization of software development methods from an organizational perspective, in which increasing productivity and leanness of software development processes are the main reasons forcing development teams to modify or ignore certain software development steps or practices (Baskerville & Pries-Heje 2004, Tom, Aurum & Vidgen 2013, Lim, Taksande & Seaman 2012). These studies argue that often, under resource constraints or in response to constant market demands, software firms have to make trade-offs between lower development costs, shorter delivery times, and software quality. Under the influence of such organizational-level trade-offs, developers are forced to customize software development methods (Tom, Aurum & Vidgen 2013, Lim, Taksande & Seaman 2012).

While software development methods and practices are adapted or selected by organizations, developers are the ones who must follow those methods and practices. Therefore, it is very important to understand how software developers deal with such organizational level decisions. However, the role of developers in customizing software development methods has been significantly ignored in previous research. This ignorance becomes more problematic when bearing in mind that a group of researchers (Lim, Taksande & Seaman 2012, Peters 2014, McConnell 2007) have suggested that software developers often have a tendency to perform high-quality work, and therefore, they might not want to be associated with such quality-compromising simplifications.

Thus, there is a significant need for scientific explanations clarifying the organizational- and individual-level mechanisms underlying the customization of software development methods. Such explanations must be able to take into account the temporal order and sequence of different steps in customizing software development methods to explain how customizing software development processes unfolds over time and why it happens in certain ways. Therefore, such explanations need to penetrate the logic behind the customization of software development methods to clarify the causal mechanisms through which software development methods are customized and to explain the rationale behind developers' behavior in omitting certain steps while performing other practices.

In this research, we aim at providing such empirically grounded explanations by using a *process theory* (Mohr 1982) approach. Process theory (Mohr 1982) is a suitable means for explaining how and possibly why a discrete set of events occurs over time (Gregor 2006, Ralph & Wand 2008, Van de Ven, A. H. 1992). Recently, there has been growing interest among SE and IS scholars in building process theories (Burton-Jones, Mclean & Monod 2011, Markus & Robey 1988, Van de Ven, A. H. 1992, Van de Ven, A. H. & Poole 2005, Ralph 2015b).

By building a process theory, we aim at clarifying the conditions under which and the causal mechanisms through which software development methods are knowingly customized in practice. We try to reach these goals by answering our main research question: *Why are software development methods customized in practice?* which is devided to the following sub-questions:

- *RQ1: What is the state of research related to the customization of software development methods?*
- *RQ2: How does the development context affect the customization of software development methods?*
- *RQ3: Under what conditions software development methods are customized in practice?*
- *RQ4: Through what causal mechanisms software development methods are customized in practice?*

These research questions are approached both theoretically and empirically through several stages, as shown in Figure 1 and as explained in the following section.

1.2 Overview of Chapters

This doctoral dissertation consists of three studies, each of which is aimed at addressing one of the research questions introduced in the previous section (see Figure 1). In the following paragraphs we provide a short overview of these studies and explain how these studies are related to each other.

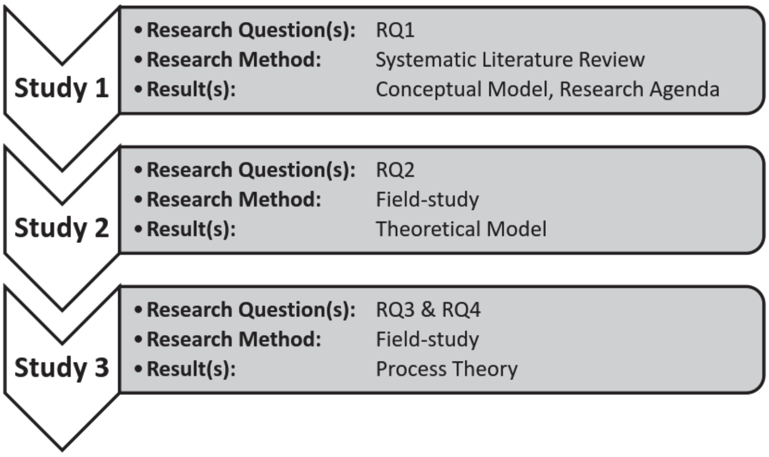

Study 1
• Research Question(s): RQ1
• Research Method: Systematic Literature Review
• Result(s): Conceptual Model, Research Agenda
Study 2
• Research Question(s): RQ2
• Research Method: Field-study
• Result(s): Theoretical Model
Study 3
• Research Question(s): RQ3 & RQ4
• Research Method: Field-study
• Result(s): Process Theory

FIGURE 1 An overview of the studies included in this dissertation

In Study 1 we approach RQ 1 by identifying and aggregating the existing research on the customization of software development methods in order to provide a clear understanding of the research problem and to emphasize the need to address this problem. In doing so, we have conducted a comprehensive Systematic Literature Review (SLR) to clarify how previous studies in the SE and IS disciplines reported and explained the customization of software development methods. By synthesizing these previous studies, we report the state of research on the customization of software development methods, and we indicate the current gaps in the literature. Additionally, by analyzing these identified studies, we build a theoretical process model to indicate how, from our perspective, the customization of software development methods happens.

According to the results of Study 1, we have realized that the act of customizing software development methods is highly influenced by the development context. Therefore, we decided to answer RQ 2 by conducting a field study (i.e., Study 2). In this study, we indicate that customizing software development methods is not limited to fast-changing and volatile environments; rather, in critical domains software development teams might also decide to customize methods in response to contextual obstacles. In Study 2, we show that different organizational constraints make software development challenging, and as a result, software developers might decide to customize methods to deal with these challenges.

Finally, in Study 3, we approach RQ 3 and RQ 4, and based on the empirical data collected from a field study conducted across development contexts, we propose a process theory called *Theory of Software Development Balance*. Our suggested theory indicates the causal mechanisms through which software development methods are customized to balance contradictory stakeholders concerns and contextual forces. In particular, in Study 3, we show how macro-level structural balance and micro-level social balance are maintained in the development context by firms and individual developers.

1.2.1 Study 1 - Omission of Quality Software Development Practices: A Systematic Literature Review and Research Agenda[2]

Research Objectives

In recent years, a significant amount of resources has been spent in software development projects, since software products have become inseparable parts of human life. Consequently, during the last four decades, the software community has tried to improvise software development and quality assurance processes by proposing different methods, best practices, and quality rules (Sommerville 2011, Poth & Sunyaev 2014). However, recent literature hints that ignoring such software development practices and performing workarounds increase the chance of producing defective software products (Ahonen & Junttila 2003, Austin 2001, Baskerville & Pries-Heje 2004, Baskerville et al. 2001, Baskerville et al. 2003, Vartiainen, Siponen & Moody 2011).

It is apparent that available software development methods and quality standards will become ineffective if they are purposefully ignored. Therefore, it is very important to understand why software professionals ignore best practices and engage in quality-compromising practices. It must be noted that the term "best practices" here means a well-defined technique or method that enables

[2] H. Ghanbari, T. Vartiainen, & M. Siponen, "Omission of Quality Software Development Practices: A Systematic Literature Review and Research Agenda", *ACM Computing Surveys (under review).*

developers to successfully complete one step of software development (Tighy 2012). These practices can be suggested by given software development methods, quality rules, and standards or by in-house procedures and guidelines prepared in a company.

In Study 1, we attempt to understand why software development teams knowingly decide to omit quality practices as previously defined. To gain such understanding, we decided to conduct an SLR study (Kitchenham & Charters 2007, Okoli & Schabram 2010) to discover the state of research on this phenomenon.

Research Results

In our SLR study, we initially found 4838 studies that were potentially relevant to the customization of software development practices. However, after an extensive and iterative review process, we have found only 19 of these studies to be relevant to our research questions. These studies use a variety of terminologies, including 'shortcutting' (Austin 2001), 'systematic omission of software tasks' (Samalikova et al. 2011), 'technical debt' (Cunningham 1992), and 'short-cycle time development' (Baskerville & Pries-Heje 2004) to report the intentional omission of quality practices.

According to the identified studies, the omission of testing and quality control activities is the most common instance of ignoring software development practices. Keeping in mind that such activities play a vital role in ensuring the quality and reliability of software products, it becomes obvious that this specific aspect of software development has received less attention from the software community. Ignoring design and implementation practices followed by documentation are the next most common instances of omission of quality software development practices. Finally, the omission of quality practices during the requirements analysis and specification phase is reported by one-fourth of the identified studies. Based on these results, it can be said that the delivery of new functional features compared to the evaluation of software quality has received more attention from software development teams.

The identified studies report the intentional omission of quality practices from both organizational and individual perspectives. From an organizational perspective, the omission of quality software development practices is the result of organizational-level decisions to gain certain business advantages or to deal with certain business obligations or common issues and challenges of software development projects. However, from an individual perspective, software developers often decide to omit quality practices in order to achieve certain personal goals.

Previous studies report a variety of factors leading to the omission of quality practices. We have divided these factors, based on their nature, into five main categories: *Business goals*, *Customers' requirements*, *Project constraints*, *Technical issues*, and *Psychological factors.* Each of these categories, originated from different levels of context, affects the omission of quality practices.

Our results show that, depending on their business environments, development teams might be pressured or encouraged to ignore quality software development practices in order to gain competitive advantages. On the other hand, developers might decide to ignore quality software development practices under the influence of different organizational level factors such as development resources and technical obstacles. Finally, we suggest that these market- and organizational-level factors might influence individuals' attitudes and cognitive tendencies and as a result managers and developers might decide to neglect certain software development practices.

Conclusions

In Study 1, our goal is twofold: first, to discover the state of research on the omission of quality practices and to understand the extent to which this phenomenon has been investigated previously; and second, to determine the root causes underlying the customization of software development methods as suggested by previous studies.

Based on the analysis of the primary studies, we hypothesize that the deliberation of omission behavior concerns the contradiction between quality and productivity and that there is a psycho-social process pertaining to omission instantiations. The current literature does not consider the omission of quality practices adequately with respect to why and how software developers make the decision to omit a quality practice and how to address this phenomenon in practice.

The results of Study 1 show that, despite previous attempts to explain the customization of software development processes, there is a need for further research to clarify several aspects of this research phenomenon. In particular, further research is needed to deeply investigate the contextual factors and conditions under which the omission of quality practices is initiated. Another area that requires further research is the psychological processes through which software professionals decide to perform such questionable practices. Furthermore, while previous studies consider several short-term consequences of the omission of quality practices, future research needs to study the long-term consequences of such questionable practices. Finally, future research must identify and suggest different interventions and solutions that could enable the software community to overcome the omission of quality practices.

1.2.2 Study 2 - Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study[3]

Research Objectives

Previous research shows that a large number of software and system development projects fail or face severe challenges (The Standish Group International, Inc. 2009). Apart from different technical and human problems, software defects are the main cause of most software vulnerabilities and failures (Fonseca & Vieira 2008, Wijayasekara et al. 2012). Therefore, identifying and fixing software

[3] H. Ghanbari (2016), "Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study", *Proceedings of 49th Hawaii International Conference on System Sciences (HICSS-49)*, pp. 5407-5416.

deficiencies, such as bugs, missing requirements or flaws in software design (Notander, Höst & Runeson 2013), have major importance in increasing the quality and reliability of software products.

As mentioned earlier, to increase the quality and reliability of software products, a large number of software development methods and tools have been put forward by the software community (Avison & Fitzgerald 2003, Iivari & Maansaari 1998). However, these software development methods are rarely followed as prescribed (Boehm & Turner 2003, Conboy & Fitzgerald 2010, Baskerville & Pries-Heje 2004). According to the results of Study 1, a group of scholars uses the metaphor of technical debt (Cunningham 1992) to explain strategic business decisions made by organizations to achieve short-term goals by compromising or fully eliminating certain software development activities, such as architectural design, documentation, and testing (Lim, Taksande & Seaman 2012, Martini, Bosch & Chaudron 2014), to speed up delivery times (Brown et al. 2010, Lim, Taksande & Seaman 2012). According to this group of primary studies, such quality-compromising trade-offs are mainly tactically and reactively made by firms under the influence of market demands to capture market share (Lim, Taksande & Seaman 2012, Lindgren et al. 2008) or fulfill their contractual obligations (Martini, Bosch & Chaudron 2014).

While minimizing development time and costs might play a key role in highly competitive markets (Eberlein & Leite, Julio, Cesar, Sampaio, do Prado 2002, Sommerville 2005), quality of software often has a higher priority in developing critical systems (Sommerville 2015). Critical systems are those systems whose failure might cause devastating financial, infrastructure, or human life loss or injuries (Sommerville 2015). Despite such criticality, there have been still a considerable number of failures in critical systems that were caused by software defects and vulnerabilities (Dalcher 1999, Alemzadeh et al. 2013, Lann 1997, Eklund, Nichols & Knutsson 2016). However, previous studies have shown that technical debt does not always occur because of bad design or development decisions but might also be due to environmental factors that cannot be controlled by development teams (Tom, Aurum & Vidgen 2013, Brown et al. 2010). Therefore, in Study 2, we decided to conduct an exploratory field study in a company active in the aerospace domain to investigate whether, and possibly why, the omission of software development practices and quality rules happens in critical domains.

Research Findings

The results of Study 2 show that ignoring software development practices and steps is not limited to highly volatile and competitive markets. Rather, even in critical domains where integrating dependability and safety requirements into software has a significant importance, development teams might decide to make quality-compromising trade-offs while facing different issues and challenges.

In Study 2, we identified four main circumstances under which performing planned development activities becomes very challenging in the case company and the development teams might thus be forced or encouraged to customize given software development methods. Particularly, in the case company, due to *Ambiguity of Requirements*, *Diversity of Projects*, *Inadequate Knowledge Management*, and *Resource Constraints*, software developers are often forced to minimize software processes by skipping certain practices or postponing certain activities.

Our data indicate that the diversity of projects in the case company makes it challenging for the development teams to follow certain methods for specifying software requirements. In some projects stakeholders' requirements are communicated vaguely, while in other projects the requirements are defined well. On the other hand, inadequate knowledge management in the case company makes it even more problematic to identify and specify a clear set of requirements. Inadequate knowledge management alongside the ambiguity of the requirements makes it very challenging for development teams to precisely estimate the necessary resources for implementing stakeholders' requirements. As a result, development teams face a lack of necessary resources later during the software project. Under the above-mentioned circumstances, following planned software processes and complying with given methods become very difficult for development teams. Therefore, they might decide to ignore or modify certain practices or activities to deal with such circumstances.

In Study 2, based on our observations and drawing from previous literature, we suggest that utilizing certain agile practices might enable development teams to avoid, or to at least identify and manage, technical debt.

Conclusions

In Study 2, upon collecting data from several projects in a company active in the aerospace domain, we discovered a set of factors that make it challenging for software development teams to rigorously follow given methods. The results of this study reveal that ignoring recommended practices and quality rules might happen even in critical software projects, where certain standards and costly software engineering processes must be followed.

In this study, we suggest that combining agile practices with plan-driven processes brings flexibility into critical software projects and, as a result, enables development teams to avoid or at least better manage technical debt in these projects. However, further research is needed to support our suggestions and to investigate the effectiveness of agile practices to manage technical debt in critical software projects.

1.2.3 Study 3 - Why Software Development Methods are Customized in Practice - A Theory of Software Development Balance[4]

Research Objectives

Thousands of software development methods have been put forward by SE and IS communities in order to better manage highly dynamic and complex soft-

[4] H. Ghanbari & M. Siponen, "Why Software Development Methods are Customized in Practice - A Theory of Software Development Balance", *IEEE Transactions on Software Engineering ("Revise and Resubmit" for 2nd round of review).*

ware processes (McLeod & Doolin 2012). The available methods mainly propose a set of predefined practices and steps for developing and maintaining software products (Iivari 1991, Iivari, Hirschheim & Klein 1998b). However, it has been widely reported by previous studies that these methods are not followed as prescribed (Truex, Baskerville & Travis 2000, Baskerville & Pries-Heje 2004, Avison & Fitzgerald 2003, Coleman & O'Connor 2007, Conboy & Fitzgerald 2010, Fitzgerald 1998, Iivari & Maansaari 1998, Huisman & Iivari 2006, Kiely & Fitzgerald 2002, Ralph 2016, Fitzgerald, Hartnett & Conboy 2006); rather, software development teams modify or skip certain steps or practices that are recommended by these methods (Highsmith & Cockburn 2001, Sommerville 2005).

Maximizing the leanness of software development processes (Baskerville & Pries-Heje 2004, Fitzgerald, Hartnett & Conboy 2006, Boehm 2002, Baskerville et al. 2001, Lim, Taksande & Seaman 2012, Lindgren et al. 2008) and a lack of universal methods suitable for all types of projects (Truex, Baskerville & Travis 2000, Iivari 1991, Brinkkemper 1996, Henderson-Sellers & Serour 2005) are the main reasons suggested for modifying or even combining different software development practices in each project (Conboy & Fitzgerald 2010, Boehm 2002, Henderson-Sellers & Serour 2005, Leppanen 2006, Iivari & Iivari 2011). However, despite the important role of individuals in implementing software development methods, there is a lack of understanding of how software developers decide to ignore or modify best practices recommended by given methods.

Until now, the majority of research on software development, especially in the SE field, has focused on identifying and improving novel methods and practices, but without firm theoretical foundations (Ralph 2016, Zhang & Budgen 2012). Therefore, there is a need to propose novel SE theories (Sjøberg et al. 2008, Wohlin, Šmite & Moe 2015, Johnson, Ekstedt & Jacobson 2012) that explain why these software development methods are widely ignored in practice.

In Study 3, we attempt to contribute to this issue by proposing a novel theory grounded in the experience of software professionals active in different contexts in order to explain the customization of software development methods as mechanisms—at both individual and organizational levels—for maintaining balance between stakeholders' multi-concerns and contradictory contextual forces over time.

Research Findings

The results of our field study show that all of the interviewees skip or modify software development practices to some extent, even though all of them think that using methods is beneficial. However, all the interviewees mentioned that these methods must be customized based on the nature of the software under development and the characteristics of the development context.

From our data, we identified a wide range of instances where software development methods and practices were ignored. Such instances include, but are not limited to, ignoring design and coding guidelines, skipping documentation and quality control activities, abandoning recommended tools, performing workarounds, and postponing external quality assurance activities. During the data analysis, we found that such changes in the software development processes are caused by different factors, such as the complexity of the software under development, the evolution of the requirements, technology advancement, and contrasting and sometimes contradictory stakeholders expectations and market forces.

In Study 3, we propose a novel grounded theory called the *Theory of Software Development Balance*. Our suggested theory indicates how developers, depending on the development context, decide to modify or omit certain software development practices or activities to deal with such complex and inconsistent contextual settings. In particular, this theory explains how the customization of software methods is initiated by the unique characteristics of software and progresses through a series of complex interactions among software stakeholders under the influence of contextual forces.

In our theory, we explain how the dispositional attributes of internal and external entities (e.g., stakeholders and regulatory authorities) involved in software projects, their structural arrangement, and their interactions lead to instability in the software development context; as a result, changes in the software development process are unavoidable. These entities may have inconsistent goals and contradictory opinions about how these goals must be achieved. For example, for some stakeholders faster delivery time has higher priority, while other stakeholders might be more concerned about the quality of the software. Because of such differences between stakeholders' goals and concerns, they might have contrasting and sometimes contradictory opinions about the necessity of utilizing software development methods.

On the other hand, since software is evolving over time, software development processes are highly dynamic and they cannot be universally predefined. Often, software development teams seek solutions not only to address constant requirement changes but also to increase their velocity and productivity. As a result, software firms may decide to ignore certain practices or activities in order to maximize the leanness of their software development processes.

In Study 3, we explain how software developers might decide to skip certain practices and activities in order to address deal with unexpected issues and evolving requirements as well as organizational-level decisions to address environmental changes. However, according to several interviewees, performing such quality-compromising workarounds is undesirable. Therefore, we argue that developers experience inconsistent social situations when their personal concerns and values are in contradiction with contextual forces. We propose that developers try to resolve such inconsistent situations either by identifying alternative solutions to perform their tasks while avoiding quality-compromising practices or by justifying such compromises to themselves.

Finally, in our theory, we propose that the lack of proper quality control and assurance mechanisms is another reason for the customization of software development methods. Since controlling the quality of software products and processes is expensive and challenging, it is very difficult for software stakeholders to identify compromises in software development processes, especially if quality management mechanisms are not appropriate. Therefore, it becomes possible for development teams to take shortcuts without fear of being noticed by other stakeholders.

Conclusions
In Study 3, we conducted a longitudinal field study and built a novel process theory. This empirically grounded theory indicates the causal mechanisms underlying the customization of software development methods in order to clarify how and under what conditions certain software development activities and practices are ignored. In this theory, we propose that the unique nature of software triggers the mechanisms underlying the customization of software development methods, while these mechanisms progress as a result of interactions among several driving forces influenced by inconsistent socio-structural settings.

In the *Theory of Software Development Balance*, we propose that development teams, depending on their context, decide to customize software development processes in order to maintain balance among contrasting or even contradictory forces imposed by stakeholders' concerns and contextual determinants. Therefore, we suggest that, if needed, software methods must be customized strategically and according to the organizational structures of software firms and their development contexts.

1.3 Publication Status

Due to the importance of the research topic and based on growing interests in building theories, especially process theories in both SE and IS domains (Burton-Jones, Mclean & Monod 2011, Markus & Robey 1988, Van de Ven, A. H. 1992, Van de Ven, A. H. & Poole 2005, Ralph 2015b), we expect that our findings receive considerable attention from software development community both in academic and industry. Therefore, based on the results of this PhD study, we have been preparing several high quality scientific papers to be published in leading SE and IS outlets. So far, three papers are prepared, as discussed in the previous section.

As it is indicated in Table 1, one peer-reviewed conference paper is published and two other papers are still under review at the time that this thesis has been published. Study 1, which is prepared as a standalone literature review paper is under review at the *ACM Computing Surveys* which is a highly ranked peer-reviewed journal published by the Association for Computing Machinery.

Study 2 is published in the proceedings of *The 49th Hawaii International Conference on System Sciences (HICSS-49)* which is an annual peer-reviewed IS conference. The paper was presented during the HICSS-49 and based on the constructive feedback provided by the audience we are planning to further improve and prepare a new version of this paper.

TABLE 1 The publication plan

Study	Author(s)	Title	Forum	Status
Study 1	Ghanbari, Vartiainen, & Siponen	Why Software Professionals Knowingly Omit Quality Software Development Practices: A Systematic Literature Review and Research Agenda	*ACM Computing Surveys*	Under Review
Study 2	Ghanbari	Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study	*HICSS-49*	Published
Study 3	Ghanbari & Siponen	Theory of Software Development Balance: A Grounded Theory	*IEEE Transactions on Software Engineering*	Revise and Resubmit for 2nd round of review

Finally, Study 3 has received a *Revise and Resubmit* decision after the first round of review at the *IEEE Transactions on Software Engineering* which is the leading journal in the SE discipline (Garousi & Fernandes 2016). It must be noted that this peer-reviewed journal is published by the IEEE Computer Society.

Even though this doctoral thesis consists of three studies presented in Table 1, at the time of publishing this thesis two more journal articles are being prepared based on the results of this research.

2 RESEARCH APPROACH

As discussed earlier in this doctoral research our goal is to seek an in-depth understanding of the mechanisms and rationale behind the customization of software development methods. In so doing, we aim to build empirically grounded theories to explain why and how software development methods are customized in practice. Since software development is a highly dynamic and complex sociotechnical phenomenon (McLeod & Doolin 2012), we need to employ a suitable research approach that enables us to precisely identify and analyze the mechanisms through which software development methods are customized. In addition, the selected research approach must enable us to explore contextual factors initiating and affecting these mechanisms and their outcomes.

2.1 Interpretive Research

It is suggested in both SE and IS literature that qualitative research is a suitable approach for developing knowledge about dynamic processes and for understanding the logic behind change over time (McLeod & Doolin 2012, Van de Ven, A. H. & Poole 2005). In particular, it is suggested that an interpretive study that is grounded in the experience of professionals involved in software development processes is useful for developing new theoretical understandings about this phenomenon (Sarker, Lau & Sahay 2000, Urquhart, Lehmann & Myers 2010, Klein & Myers 1999).

Therefore, based on my personal beliefs regarding the conduct of research, I have decided to follow an interpretive (Orlikowski & Baroudi 1991, Klein & Myers 1999) approach in this PhD study. The interpretivist paradigm (Orlikowski and Baroudi, 1991) enables a researcher to develop an understanding of a research phenomenon by using and interpreting the perceptions and experiences of those who participate in that phenomenon (Thanh & Thanh 2015). The concept of "paradigm" is widely used by scientific communities to describe the assumptions underpinning different theoretical ideas and research approaches and to classify them accordingly (Iivari 1991). In the following sections, we briefly explain the paradigmatic constituents of this doctoral research.

2.1.1 Ontology

Ontological assumptions concern the form and nature of the reality, meaning the phenomena under study (Iivari 1991, Guba & Lincoln 1994, Orlikowski & Baroudi 1991). In other words:

> Whether the empirical world is assumed to be objective and hence independent of humans, or subjective and hence having existence only through the action of humans in creating and recreating it. (Orlikowski & Baroudi 1991, p.7)

From an ontological standpoint, the interpretive approach assumes that a subjective social world is constructed and reconstructed through ongoing actions and interactions of humans. Therefore, since social relations or entities (e.g., organizations or social systems) do not exist objectively and apart from humans they can only be interpreted by researchers:

> Meaning and intentional descriptions are important, not merely because they reveal subjects' states of mind which can be correlated with external behavior, but because they are constitutive of those behaviors. (Orlikowski & Baroudi 1991, p.13)

Keeping in mind that an interpretive approach recognizes that the social reality is produced through constant human interactions, understanding of the social reality may change as researchers provide different interpretations of this reality over time.

2.1.2 Epistemology

Epistemological assumptions have to do with the nature of scientific knowledge, meaning how the researcher (i.e., knower) acquires valid knowledge about the phenomena under investigation (Iivari 1991; Guba & Lincoln, 1994; Orlikowski & Baroudi, 1991).

Epistemologically, the interpretive approach assumes that by getting involved in a social setting, researchers are able to construct understandings and interpretations of a phenomenon by accessing the subjective meanings that are constructed and assigned to that phenomenon by the participants of that particular setting.

> The intent of the research is to increase understanding of the phenomenon within cultural and contextual situations; where the phenomenon of interest is examined in its natural setting and from the perspective of the participants; and where researchers did not impose their outsiders' a priori understanding on the situation. (Orlikowski & Baroudi 1991, p.5)

As a result, since interpretive research enables researchers "to understand human thought and action in social and organizational contexts" (Klein & Myers 1999, p.67), it is suitable for developing circular or reciprocally interacting models of causality to indicate how participants perceive the social world around them and their role in this socially constructed world (Orlikowski & Baroudi 1991).

2.1.3 Methodology

Methodological assumptions concern which methods can be used by the researcher to acquire the valid empirical evidence about the phenomenon under investigation (Guba & Lincoln 1994, Orlikowski & Baroudi 1991).

As discussed earlier, interpretivism assumes that the phenomenon under study must be understood in its context and from the perspective of the actors who experience it (Orlikowski & Baroudi 1991). Therefore, qualitative research methods, such as field studies (Klein & Myers 1999), are appropriate for investigating a social phenomenon within its natural context (Orlikowski & Baroudi 1991, Stol, Ralph & Fitzgerald 2016) through qualitative data collected from different sources, including interviews, observations, and archival materials (Conboy, Fitzgerald & Mathiassen 2012). Orlikoswki and Baroudi (1991) suggest that:

> Following on the ontological belief that reality is socially constructed, the interpretive researcher attempts to derive his or her constructs from the field by in-depth examination of and exposure to the phenomenon of interest. The categories and themes that emerge out of this approach are intended to closely couple those relevant to the study's participants [as] interpretive techniques allow participants to use their own words and images, and to draw on their own concepts and experiences. (Orlikowski & Baroudi 1991, p.14)

However, it must be noted that gaining a deep understanding of the phenomenon under investigation and observing the social processes within a context is an effort-intensive task (Maxwell 2004). Therefore, interpretivist researchers must spend a considerable amount of effort and time in a social context to collect the "rich" data (Maxwell 2004) necessary for understanding the causal mechanisms within that context. Therefore, in this doctoral research, we utilized a combination of techniques to collect "rich" data and to build empirically grounded interpretations of the mechanisms through which software development methods are customized in practice.

Systematic Literature Review
To gain an understanding of the state of research on the customization of software development methods, we have conducted an SLR (Kitchenham & Charters 2007, Okoli & Schabram 2010) study to identify and analyze previous empirical studies relevant to our research phenomenon.

It has been suggested by previous studies that an SLR is a suitable methodology for aggregating and evaluating completed and recorded research regarding a certain topic of interest to both identify gaps in the body of knowledge and propose directions for conducting future research to address these identified gaps (Kitchenham & Charters 2007, Kitchenham et al. 2010, Okoli & Schabram 2010, Rowe 2014).

As suggested by Petersen et al. (2008), following a well-defined procedure in our SLR study enabled us to identify and analyze a wide range of previous studies. By providing a synthesis of these previous studies, we were able to identify and show the gaps in the literature regarding the customization of software development methods. At the same time, by analyzing data aggregated from the identified studies, we have built a theoretical model indicating our interpretation of the processes through which the customization of software development methods happens in practice. This model is explained in detail in Study 1.

Finally, based on the synthesis of our SLR study, we have suggested a research agenda for filling the identified gaps in the current knowledge. We have attempted to address some of these gaps in several steps and by conducting a longitudinal field study.

Field Study

Following our SLR, we conducted a three year field study within the context of an industrially led research and development project consisting of 17 European organizations and research institutes. During this field study we employed a combination of techniques to collect "rich" qualitative data from software development teams active in a variety of contexts. In doing so, we aimed at examining different aspects of customization of software development methods across industrial domains.

In particular, we collected data through formal semi-structured interviews (Myers & Newman 2007) and informal conversations with software development professionals (e.g., developers, designers, testers, and project managers). In addition, we collected data by observing participants during several field visits to the case companies as well as regular co-located and online project meetings with the project consortium.

Finally, we used archival materials such as project reports, software development procedures and guidelines provided by the interviewees, domain-specific standards used by case companies, and public information available on the internet as complementary data sources. It must be mentioned that we used a qualitative data analysis tool called Nvivo to store and manage the large amount of qualitative data collected during this doctoral research.

2.2 Theory Construction

As explained earlier, in this doctoral research, our aim was to identify and provide theoretical explanations of the causal mechanisms through which software development methods are customized. To reach this goal, we followed an interpretive approach to theory construction, drawing on dimensions of theory construction suggested by Markus and Robey (1998). These dimensions, namely, causal agency, logical structure, and level of analysis (Markus & Robey 1988), are shown in Figure 2.

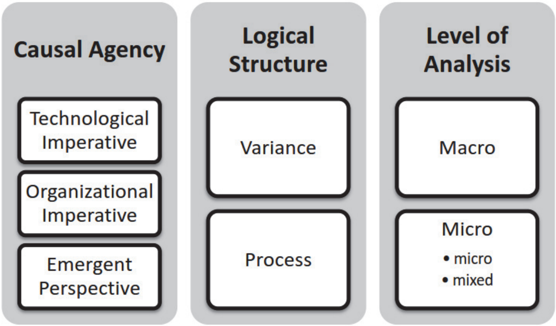


FIGURE 2 Dimensions of Causal Structure (Markus and Robey 1988)

Regarding the causal agency, in this doctoral research, we followed an emergent perspective on theory construction (Markus & Robey 1988). This is because we assume that developers' tendency to neglect software development methods and practices is shaped by both complex and unpredictable development environments as well as their intentions and social interactions.

Regarding the level of analysis, in our field study, we focused on both micro- and macro-level (Markus & Robey 1988) entities. This is because we assume that the decision to ignore software development practices is the result of interactions between individual software developers and larger social structures in the development context.

Finally, regarding the logical structure, we followed a process approach (Markus & Robey 1988) in which we aimed to identify mechanisms underlying developers' behavior in neglecting software development practices. A process theory (Mohr 1982) can explain how and possibly why a discrete set of events occurred over time (Gregor & Jones 2007, Ralph 2015a, Ralph 2016, Van de Ven, A. H. & Poole 2005, Van de Ven, A. H. 1992). Therefore, a good process theory provides richer explanations of why and how (Van de Ven, A. H. 1992) software developers take shortcuts by skipping recommended practices and quality rules.

In the following sections we briefly discuss the characteristics of a process theory and its differences from a variance theory.

2.2.1 Process vs. Variance Theory

In regard to the logical structure, theories can be divided into two types: variance and process. Table 2, which is adapted from Markus and Ruby (1989), shows the characteristics of process and variance theories and how they are different.

TABLE 2 Differences between the Logical Structure of Variance and Process Theory (Markus and Robey 1988)

	Variance Theory	Process Theory
Role of Time	Static	Longitudinal
Definition	The cause is necessary and sufficient for the outcome	Causation consists of necessary conditions in sequence; chance and random events play a role
Assumptions	Outcome will occur when necessary and sufficient conditions are present	Outcome may not occur (even when conditions are present)
Elements	Variables	Discrete outcomes
Logical Form	If X, then Y; If more X, then more Y	If not X, then not Y; cannot be extended to "more X" or "more Y"

Generally speaking, while variance theories deal with variables and their correlations, process theories mainly deal with events and the processes through which those events are connected (Maxwell 2004). However, Moher (1982) suggests that the difference between these two types of theory can be best explained in terms of necessary and sufficient conditions.

> The variance theory is a type [of theory] whose characteristics grow out of a foundation in the necessary and sufficient, whereas the characteristics of process theory grow out of a foundation in the necessary alone. (Mohr 1982, p.36)

In other terms, in a variance theory, the precursor (X) is considered to be a necessary and sufficient condition for the outcome (Y) to occur, whereas in a process theory, the precursor (X) is a necessary but not sufficient condition for the outcome (Y) to occur. Therefore, for a process theory to be satisfactory, the precursor must:

> Contain three types of elements - (1) necessary conditions and (2) necessary probabilistic processes, which together form the core of the theory, and (3) external, directional forces that function to move the focal unit and conditions about in a characteristic way, often herding them into mutual proximity. (Mohr 1982, p.45)

Since variance theory deals with correlations between variables, experimental and survey methods are commonly used to measure how changing the values of particular variables contributes to variations in the values of other variables (Maxwell 2004). On the other hand, qualitative methods are appropriate for identifying how events occur in a given context and what are the contextual causal mechanisms connecting those events (Maxwell 2004). However, since these causal processes usually cannot be observed directly, the identification of plausible and valid causal explanations requires intensive field-work to collect rich empirical evidence (Maxwell 2004).

Therefore, during our field study we attempted to collect "rich" qualitative data from different sources to identify and explain under what conditions and through which mechanisms software development methods are customized in practice. These conditions and mechanisms are extensively explained in Study 2 and Study 3.

2.2.2 Grounded Theory Method

In recent years, the Grounded Theory Method (GTM) has been used increasingly in the SE and IS disciplines (Giardino et al. 2013, Hoda, Noble & Marshall 2013, Birks et al. 2013, Urquhart & Fernandez 2013, Stol, Ralph & Fitzgerald 2016). GTM is suitable for generating "conceptual theory that accounts for a pattern of behavior which is relevant and problematic for those involved" (Glaser 1978, p.93). Such theory is generated through systematic collection and analysis of empirical data based on the experiences of humans and the contextual factors associated with a phenomenon (Hoda, Noble & Marshall 2013, Glaser & Strauss 1967).

In GTM studies a *theoretical sampling* (Glaser & Strauss 1967, Glaser & Holton 2004) strategy must be followed to collect and analyze data iteratively. This strategy enables researchers to enrich their emerging theoretical concepts by identifying shortcomings in the collected data and to address them by collecting more data (Birks et al. 2013).

To analyze the data systematically, we have performed three stages called *open coding*, *selective coding*, and *theoretical coding* (Glaser 1978, Glaser & Holton 2004, Glaser 1992, Urquhart, Lehmann & Myers 2010) as well as *memoing* (Glaser & Strauss 1967).

In the first stage, *open coding* (Glaser 1978, Glaser & Holton 2004, Glaser 1992), textual data collected from the field are broken down into small pieces and conceptual labels are assigned to conceptually similar fractures. In the second stage, *selective coding* (Glaser 1978, Glaser & Holton 2004, Glaser 1992), a core conceptual category is identified and those categories that are sufficiently related to this core category are further developed through further data collection and coding (Glaser 1978, Glaser & Holton 2004). Finally, in the last stage of data analysis, *theoretical coding* (Glaser 1978, Glaser & Holton 2004, Glaser 1992), the interrelationships between the core category and relevant categories (i.e., selective codes) are conceptualized to theorize the research phenomenon. While performing these three stages of data analysis, *memoing* must be performed by researchers in order to capture their insights regarding the emerging conceptual categories and their interrelationships (Glaser & Strauss 1967).

The GTM is suggested to be effective in generating context-based and process-oriented explanations about socio-technical phenomena (Urquhart, Lehmann & Myers 2010, Myers 1997), such as software and information systems development. Therefore, in this doctoral research, we decided to use the GTM (Glaser 1978, Glaser & Strauss 1967, Glaser & Holton 2004, Glaser 1992) to develop a process theory (Mohr 1982) that is grounded in the experience of software development professionals.

As mentioned earlier, drawing from literature and based on the results of our field study, we found that a typical scenario of ignoring recommended practices or quality rules relates to conflicts between stakeholders' concerns and contextual forces. For example, while software developers are more concerned with producing innovative and high-quality artifacts (Lim, Taksande & Seaman 2012, Katz 2005), due to market demands, managers are more concerned with increasing productivity. Therefore, we propose that such conflicts might trigger the psycho-social processes leading to the customization of software development methods.

Drawing upon the concept of balance from the field of psychology (Heider 1946, Heider 1967, Heider 1958, Cartwright & Harary 1956), we propose that the customization of software development methods is the result of individual- and organizational-level mechanisms for maintaining balance between stakeholders' multi-concerns and contradictory contextual forces. Especially from an individual perspective, we argue that due to structural pressures, software developers experience social inconsistencies and as a result they become motivated to take shortcuts or perform workarounds to resolve such tensions. In the following section we briefly explain the balance theory.

2.2.3 Balance Theory

Since the 1960s, balance theory (Cartwright & Harary 1956, Heider 1946, Heider 1958, Newcomb 1961) has been widely used in the field of psychology and the social sciences as a meta-framework for studying how the attitudes and relations between social actors influence the structural arrangements between actors in social groups (Hummon & Doreian 2003). The theory was originally suggested and developed by Heider (1946, 1958) to explain the mechanisms for changes towards balanced states from an individual view-point (micro-level). Basically, Heider proposes that in social groups, rational actors have a tendency toward balanced states (Khanafiah & Situngkir 2004).

The initial ideas on balance were later extended to the group level (macro-level) by Cartwright and Harary (1956) and Newcomb (1961). Unlike Heider, they suggest that balance occurs as the result of change processes at the group level (Hummon & Doreian 2003). To formalize Heider's original balance theory, Cartwright and Harary (1956) represented social structures as graphs. Using graph theory in their approach had a significant impact on the further development of balance theory (Khanafiah & Situngkir 2004, Hummon & Doreian 2003). More recently, balance theory has been increasingly used in other disciplines to study the relation between the behavior of social actors and their mode of thinking, especially in large social groups such as consumer networks and social networks (Bargh, Chen & Burrows 1996, Woodside & Chebat 2001).

Heider (1958, p.201) describes a balance state as "a situation in which the relations among the entities in a social system fit together harmoniously; there is no stress toward change". He categorizes the relations between separate entities in a social group into unit and sentiment relations. A group of separate entities that are perceived as belonging together represents a unit (e.g., members of a software development team). On the other hand, one's attitude towards other entities in the social group is called sentiment relation (Heider, 1958). Here, the term attitude refers to positive and negative relationships (e.g., feelings or valuation) of a person to another entity which might be a person or a non-person (i.e., an object, a situation, an event, an idea etc.). The relations between separate entities, from the perspective of perceiver (*p*), may be for dyads, triads or more-complex cases. A dyad indicates a relation between two separate entities, while a triad represents a relationship among three separate entities. In more-complex groups the relations among separate entities can be shown as a combination of several dyads and triads.

For dyads, the relation can be either between a person and a non-person (*p, x*) or between two persons (*p, o*). The relations in a triad may be between one person and two non-person entities, two persons and a non-person entity, or three persons (*p, o, x*). Heider's *pox* model (see Figure 3) shows the relations between social entities for a triad.

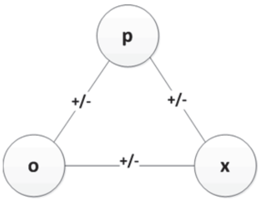


FIGURE 3 Heider's *pox* model (Khanafiah & Situngkir, 2004)

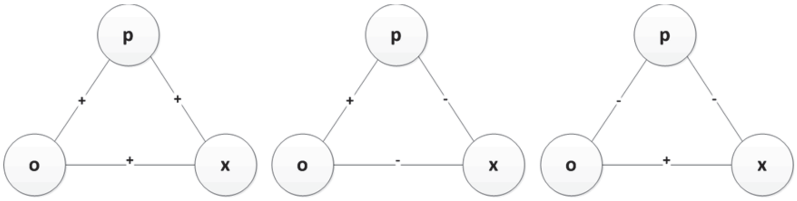


FIGURE 4 Three examples of balanced triads

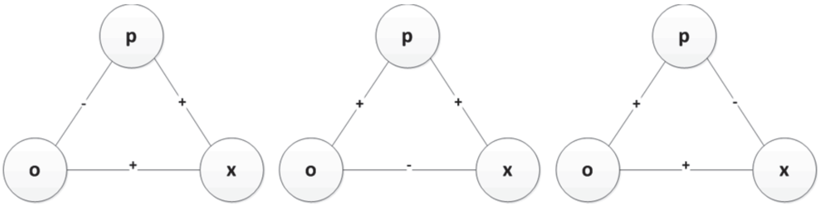


FIGURE 5 Three examples of imbalanced triads

In this model, the relations are balanced as long as the multiplication of their signs is positive (see Figure 4). As can be seen in Figure 4, all the triads are balanced. Heider (1958) argues that in balanced situations, no tension is felt by actors involved in a social structure, and therefore, there is no need for conscious thinking to occur. An imbalanced state occurs as soon as the multiplication of the signs of the relations between entities becomes negative (see Figure 5).

In such situations, actors feel tension, and thus, they tend to rearrange their social structures in order to achieve balance. For example, in the triads shown in Figure 5, to reduce tension, *p* must either change the relationship towards *x* or *o*.

As mentioned earlier, a combination of dyads and triads can be used to represent the relations between entities in large and complex social groups. However, it must be noted that change towards balance in social systems is not a straight-forward process, as it is influenced by different individual and group factors. As social actors gain higher levels of knowledge and experience their perceptions about socio-structural arrangements in their environments change over time. Therefore, at each point in time, especially in large groups, different actors might have different and even contradictory perceptions about balance. Therefore, each actor might choose different approaches towards balance based on their level of knowledge and their perception of a balanced state at the time of decision making (Hummon & Doreian 2003).

Additionally, an actor might have inconsistent attitudes toward the same entity simultaneously. This is because different relations between entities are not integrated logically (Woodside & Chebat 2001). In such situations, there is a tendency in actors to use cognitive restructuring (e.g., excuses or rationalizations) as a means for balancing inconsistent relations (Woodside & Chebat 2001). Therefore, it can be seen that maintaining balance does not necessarily need to be a physical process but it can also be a cognitive process.

In the following paragraphs, we use the *pox* model to represent an example from our empirical observations in terms of maintaining balance within the software development context.

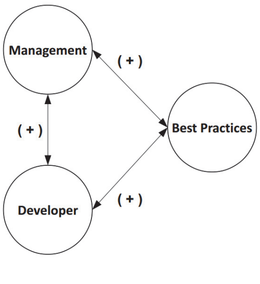


FIGURE 6 A balanced triad from Developer's perspective

In Figure 6 we show a situation in which the Developer is aware of the Best Practices recommended by the firm's software development method. The decision to select and use this software development method is made by the firm's Management team. Since the Developer is employed by the company (i.e., a unit relationship), he is aware that he is obliged to follow organizational guidelines, and therefore, he tends to follow Best Practices. At this point, this situation is balanced.

In response to a customer's request, the Management team decides to reduce the delivery time. In doing so, the Developer is requested to speed up the development by skipping or postponing certain practices Management considers unnecessary for this release. This leads to an imbalanced situation shown in Figure 7. Therefore, based on the theory of balance, the Developer tends to take physical or mental steps to maintain balance. In this case, there are three possible alternatives to resolve the balance, as shown in Figure 8.

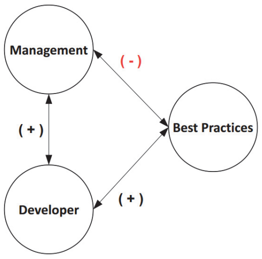


FIGURE 7 An imbalanced triad from Developer's perspective

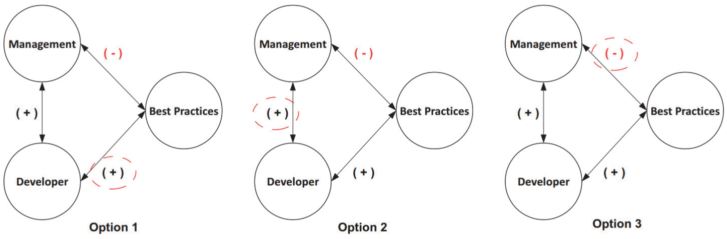


FIGURE 8 Three possible alternatives for maintaining balance

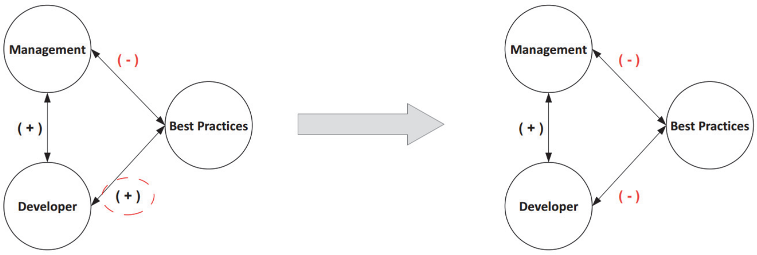


FIGURE 9 Developer decides to ignores Best Practices

In Option 1, the Developer simply follows Management's order and ignores the Best Practices (see Figure 9). Study 1 showed that this is what the majority of previous studies report, as they argue that the omission of quality practices is due to organizational-level decisions. We argue that such an argument is problematic, since it assumes that developers are fully obedient agents who simply follow orders against their personal and professional concerns.

In Option 2, the Developer decides to ignore Management's order and to follow the Best Practices (see Figure 10). In this case, the Developer might compromise his relationship with the organization, which will most likely affect his career. From our perspective, this option also sounds very simplistic, since developers have obligations to their organizations, and therefore, they cannot simply avoid organizational-level decisions. Especially when keeping in mind the high rates of change in software projects, it becomes obvious that such organizational-level decisions to change software development processes are made frequently.

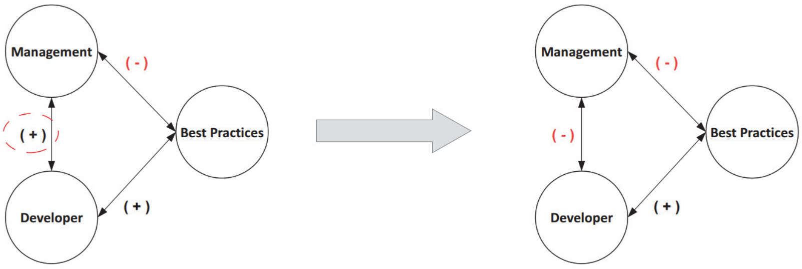


FIGURE 10 Developer decides to ignore Management's order

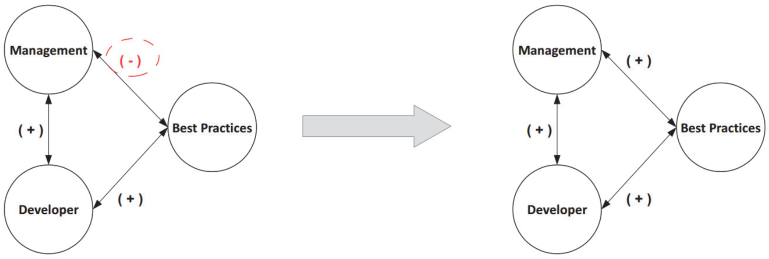


FIGURE 11 Developer convinces Management to follow Best Practices

Finally, in Option 3, the Developer tries to identify an alternative solution to perform his task and at the same time maintain balance. For instance, based on our empirical observations, the Developer may try to convince the Management team that ignoring the Best Practices is not a good idea (see Figure 11).

Another solution is that the Developer tries to voluntarily work overtime to perform his task based on the recommended Best Practices. However, if the Developer cannot identify any alternative solution and has to ignore the Best Practices, he may use cognitive restructuring as a mental solution to maintain balance. For instance, based on our data, the Developer might blame the Management team for ignoring the Best Practices and potentially compromising quality.

Each of the above-mentioned alternatives has its own effects on developers' behavior. In Study 3, we argue that if in a certain development context developers frequently experience pressure to take shortcuts and skip recommended practices and quality rules, their attitudes towards following these recommended practices will change over time. As a result, the omission of quality rules and practices will become legitimate in that context.

It must be noted that at each point in time there might be relatively stronger or even dominant relations between entities in a software development context, which will affect developers' decisions. For example, junior software developers might be more concerned about their career prospects than their personal concerns about following recommended practices. Therefore, if they are asked to take shortcuts, they might simply follow organizational-level decisions and skip given practices and quality rules, even though they have negative attitudes towards doing so.

3 STUDY 1- OMISSION OF QUALITY SOFTWARE DEVELOPMENT PRACTICES: A SYSTEMATIC LITERATURE REVIEW AND RESEARCH AGENDA[5]

3.1 Abstract

Software deficiencies are minimized by utilizing recommended software development and quality assurance practices. However, these practices become ineffective if software professionals purposefully ignore them. Conducting a systematic literature review, we discovered that while a group of studies have investigated the omission of quality practices, different aspects of this phenomenon deserve further research. Future research must investigate the conditions triggering the omission of quality practices and the consequences of such quality-compromising decisions. Additionally, because software development is a human-centric phenomenon, the psychological aspects of ignoring quality practices deserve more precise investigation. Finally, future research must suggest interventions to overcome this issue.

3.2 Introduction

Today, software has become an integrated part and parcel of everyday modern life. Not only are computers and tablets in offices and homes, but software systems have also found their way into a range of commonly used devices, from smartphones to Internet-TV. Despite all the financial and human resources that have been spent on information systems and software development projects, deficiencies in software products are widely reported in the research and practice literature (Fraser & Mancl 2008, Brooks 1995, Mancl, Fraser & Opdyke 2007).

[5] Ghanbari, H., Vartiainen, T. & Siponen, M. (Under review at ACM Computing Surveys).

Previous research has shown that such software defects are amongst the most important causes of software failures and vulnerabilities (Fonseca & Vieira 2008, Wijayasekara et al. 2012). These software deficiencies not only make information systems vulnerable but also cause extensive financial costs for software stakeholders and societies (Fonseca & Vieira 2008, Linberg 1999, Wijayasekara et al. 2012, Judy 2009). For example, in the year 2002, the cost of software deficiencies in just the United States was estimated to be almost 60 billion dollars (Judy 2009, Tassey 2002).

Minor and trivial software defects might not cause serious issues for stakeholders (Black 2012), and ordinary users might even perceive and largely accept them as technical issues, such as application or even operating system crashes and delays in services (Leveson & Turner 1993). However, because software systems deployed with critical bugs are more vulnerable to safety and security threats, they might result in devastating damages for stakeholders and societies in general (Fonseca & Vieira 2008, Leveson & Turner 1993, Eklund, Nichols & Knutsson 2016).

In response to such quality challenges during the last four decades, the software community has mainly been engaged in improving software development and quality assurance processes by proposing a variety of standards, procedures, and best practices (Sommerville 2011, Poth & Sunyaev 2014). Although utilizing these practices might enable developers to identify and resolve defects in software products, software defects might stay hidden even after delivery in some cases (Wijayasekara et al. 2012). In addition, fixing identified software deficiencies becomes more expensive and time-consuming in the later stages of projects, especially after software delivery (Banker, Davis & Slaughter 1998, Van Emden & Moonen 2002). Therefore, such deficiencies should be avoided in the first place, especially in more critical and complex systems (Leveson & Turner 1993, Wijayasekara et al. 2012).

Recent literature hints that software deficiencies might be the result of omitting proper software development practices or following "quick-and-dirty" shortcuts by development teams (Ahonen & Junttila 2003, Austin 2001, Baskerville & Pries-Heje 2004, Baskerville et al. 2001, Baskerville et al. 2003, Vartiainen, Siponen & Moody 2011). In such situations, developers may often, for example, trade software quality for short-term gains by deciding to implement a task as soon as possible rather than following best practices. In this study, we refer to such quality-compromising decisions as “omission of quality practices.”

By omitting quality practices, software professionals (e.g., requirement analysts, programmers, testers, or project managers) purposefully opt to not follow proper software development practices that are recommended by either development procedures and standards or the software community. Instead, they choose to follow a questionable practice that might compromise the software quality. For example, imagine that a programmer has a coding task that can be performed in two alternative ways: A or B. Following A, the developer spends enough time and effort to perform his task according to a certain coding standard that is recommended to improve the quality of code. Alternately, by following B, the developer knowingly ignores the coding standard and follows a "quick-and-dirty" approach to finish the task quickly. When the developer chooses to follow B while being aware of A, we call this an "omission of quality practices." Examining why software professionals engage in such questionable practices is extremely important because any quality software development practices become ineffective if software professionals purposefully ignore them.

In this article, our goal is to understand why software development teams knowingly decide to omit quality practices as previously defined. To gain such understanding, we decided to conduct a Systematic Literature Review (Kitchenham & Charters 2007, Okoli & Schabram 2010) to discover to what extent this phenomenon has been investigated by previous research. Through an extensive search performed on previous studies, only 19 studies were considered to be relevant for answering our research questions. The results of our study show that, despite its importance, several aspects of this phenomenon deserve further scholarly investigation. In particular, further research is needed to deeply investigate the contextual factors and conditions under which the omission of quality practices is initiated. Another area that requires further research is the psychological processes through which software professionals decide to perform such questionable practices. Furthermore, while previous studies consider several short-term consequences of omission of quality practices, future research needs to study the long-term consequences of such questionable practices. Finally, future research must identify and suggest different interventions and solutions that could enable the software community to overcome the omission of quality practices.

The rest of this study is structured as follows. In Section 3.3, the research methodology is presented, and different stages of the planning and conduction of the literature review are explained. In Section 3.4, the results of the literature review are reported and discussed in detail. The paper continues by outlining our proposal for the research agenda in Section 3.5. Finally, Section 3.6 summarizes the key findings.

3.3 Research method

Conducting a literature review enables scholars to identify neglected research themes and spot critical gaps in the body of knowledge that deserve further scholarly investigation (Rowe 2014). It is suggested by previous studies that a Systematic Literature Review (SLR) is a suitable methodology for aggregating and evaluating completed and recorded research regarding a certain topic of interest to both identify gaps in the body of knowledge and propose directions for conducting future research to address these identified gaps (Kitchenham & Charters 2007, Kitchenham et al. 2010, Okoli & Schabram 2010, Rowe 2014). While proposing directions for future research might not include extensive and detailed research plans, it is suggested that providing such deployment plans can be an excellent added value to SLR articles (Rowe 2014).

In SLRs, following a well-defined methodology enables researchers to identify and analyze a wider range of previous studies (Petersen et al. 2008). Often these previous studies summarized and analyzed by SLRs are referred to as primary studies (Kitchenham et al. 2010). Although conducting an SLR requires a significant amount of time and effort due to the large number of previous studies that must be identified and evaluated (Okoli & Schabram 2010, Kitchenham & Charters 2007, Petersen et al. 2008), following a well-defined and reliable process can improve the comprehensiveness and scientific rigor of the SLR while reducing researchers' biases (Okoli & Schabram 2010, Petersen et al. 2008, Rowe 2014).

At the very beginning of this research project, we conducted an initial literature review to identify previous studies related to our research questions. During this initial examination, we identified a limited number of studies relevant to our research topic. Therefore, we decided to conduct an SLR according to the guidelines suggested by Kitchenham (2004; 2007) and Okoli and Schabram (2010). Both have been widely used for conducting SLRs in the Software Engineering (SE) and Information Systems (IS) disciplines. According to the results of our initial literature review, we decided to choose a wide range of search terms to identify a larger number of studies and to cover all the potentially relevant studies. By this, we aimed to indicate the gap in the literature regarding the omission of quality software development practices and to provide directions for future research. Figure 12 shows an overview of the literature review process. In the following sections, we discuss different stages of the process through which we planned and conducted our SLR.

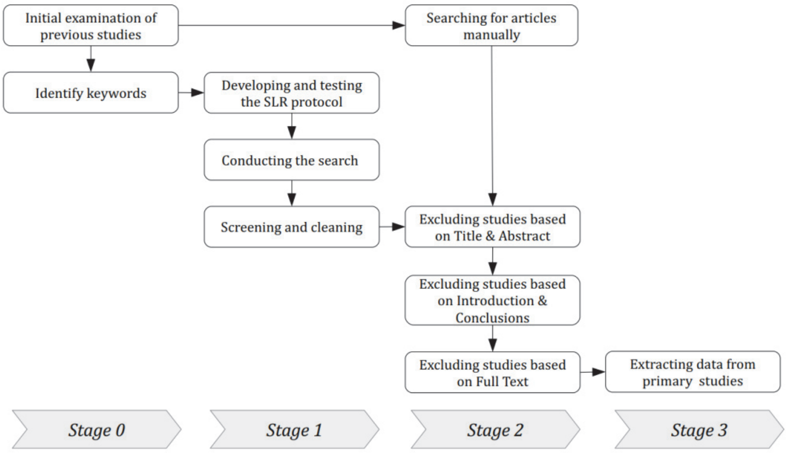

Initial examination of previous studies
Searching for articles manually
Identify keywords
Developing and testing the SLR protocol
Conducting the search
Screening and cleaning
Excluding studies based on Title & Abstract
Excluding studies based on Introduction & Conclusions
Excluding studies based on Full Text
Extracting data from primary studies
Stage 0
Stage 1
Stage 2
Stage 3

FIGURE 12 Four stages of planning and conducting the SLR

3.3.1 Initial literature review Study (Stage 0)

To evaluate the state of research on the omission of quality software development practices, we conducted an initial literature review study in which we identified several studies reporting on the omission of software development methods and practices. Following this, and using the snowball technique, we searched the lists of references of these identified papers to discover additional relevant studies. Although this preliminary literature review did not return a considerable number of relevant studies, it helped us to identify a set of keywords that have been used by previous studies and software professionals, while also noting the issues regarding the omission of software development methods and practices. These keywords were later used during Stage 1 of our SLR to search for and identify relevant literature.

3.3.2 Planning the review (Stage 1)

During the planning stage, and according to the guidelines suggested by Kitchenham and Charters (2007), we prepared a search protocol to guide our SLR and increase the rigor of the review process. This search protocol was then tested by two of the authors and improved accordingly. This protocol consisted of our research questions, our search strategy (i.e., the search terms and resources which must be searched), study selection criteria and evaluation mechanism, data extraction strategy, and review timetable. In the following sub-sections we provide more detail about the contents of the review protocol.

Research questions
According to the objectives of our research, we try to answer our main research question:

> *What is the state of research related to the customization of software development methods?*

Based on this research question, we have formed the following sub-questions to be answered:

> *RQ1: How is the omission of quality practices reported by previous studies?*

> *RQ2: What are the common instances of the omission of quality practices reported by previous studies?*

> *RQ3: Under what conditions does the omission of quality practices take place?*

Search strategy
After formulating these research questions, the search terms were chosen by identifying the keywords in the research questions and the results of our initial literature review study (see Table 3). Because we are interested in software developers' behaviors during the software development processes, "software development" and "software design" were chosen as the main keywords, which are used in the SE discipline. To extend the scope of the study to complementary fields, we also decided to use "system(s) development" and "system(s) de-

sign" as common alternatives, which are used in the IS and Computer Science disciplines.

TABLE 3 The search terms identified based on research objectives

Search Arguments	Selected words and phrases
First argument	Software development, Software design, System* development, System* design
Second argument	Omission, Omit, Questionable, Shortcut, Quick-and-dirty, Trade off, Technical debt, Dark side, Gray area, Dubious, Software quality

Additionally, based on our research questions, we used the terms "omission" and "omit" to capture all previous studies that could be relevant to our research questions. However, according to our preliminary initial literature review study, we recognized that terms such as "questionable", "shortcut", "quick-and-dirty", "trade off", "technical debt", "dark side", "gray area", "dubious", and "software quality" could also be used by those studies that are relevant to our research objectives (see Table 3).

Hence, we have combined the search arguments indicated in Table 3 to form our search string as follows:

> *("Software development" OR "software design" OR "system* development" OR "system* design") AND ("omission" OR "omit*" questionable OR shortcut OR "quick and dirty" OR "quick-and-dirty" OR "trade off" OR "trade-off" OR "technical debt" OR "dark side" OR "gray area" OR "grey area" OR "dubious" OR "Software quality")*

To identify the relevant studies, we conducted several pilot tests on major digital libraries, such as *IEEE Xplore*, *ProQuest*, *ACM Digital Library*, and *ScienceDirect*. The combination of these searches showed that the two former libraries are able to handle more complex searches and that they have a wider coverage because they retrieved considerably higher numbers of studies than the two latter libraries. For that reason, we decided to conduct the search on the *IEEE Xplore* (ieeexplore.ieee.org) and *ProQuest* (search.proquest.com) libraries. In addition, we agreed to conduct manual searches to identify extra studies missing from the automatic searches or cited by primary studies.

TABLE 4 The results of the search conducted in January 2015

Database	Total number	Date range
IEEE Xplore Digital Library	3787	1968-2014
ProQuest	1285	1978-2015
Manual search	17	1998-2014
Total	5089	1968-2015
Total after screening	4838	1968-2015

Note: Manual searches were performed mainly using *Google Scholar* (scholar.google.com).

We performed the search to identify relevant studies during January 2015. After retrieving the results, we combined them into a single spreadsheet file containing records of 5072 studies. We then went through the list to identify and modify or remove any incorrect records or duplications. At this point, we added 19 articles that were manually identified by researchers but were not retrieved by the automatic search. After this step, our list consisted of a total number of 4838 unique studies.

Selection criteria and mechanism
In our review protocol, we agreed that each study must be evaluated by at least two reviewers and based on the predefined inclusion and exclusion criteria. A study was considered to be relevant if it recognizes the problem of ignoring quality software development practices or that of software professionals engaging in questionable practices during software or information system development processes. Software professionals here refers to those individuals who are involved in any step of software development processes, including requirements engineers, analysts, designers, coders, testers, product managers, and project managers. Studies were excluded if they were not peer-reviewed journal or conference articles published in English.

Due to the large number of identified studies, we agreed to conduct the evaluation process in three consecutive rounds: first, based on title and abstract; then, based on title, abstract, introduction, and conclusion; and finally, based on the full papers. During these rounds, each paper was evaluated by at least two reviewers. The results of these independent reviews were then combined to identify relevant studies. We also agreed that, if two reviewers have contradictory opinions about the relevance of a paper, this disagreement must first be resolved through negotiation between those two reviewers, and if the disagreement cannot be resolved, then a third reviewer must evaluate the paper; based on that review, the team decides whether the paper is relevant or irrelevant.

3.3.3 Conducting the review (Stage 2)

In the first round of Stage 2, two of the authors, Reviewers 1 and 2 independently evaluated the relevance of each study by reading its title and abstract. Following Kitchenham (2004; 2007), the reviewers tried to be quite liberal in performing this evaluation to decrease the chance of excluding any relevant studies. The results of the evaluation from each reviewer were then combined, and the disagreements between them were identified. Although the majority of these disagreements were resolved by reevaluating the studies and negotiation between the two reviewers, the reviewers' evaluations were contradictory in 26 cases. Therefore, Reviewer 3 evaluated each of these 26 studies, and based on his evaluation, the disagreements between Reviewers 1 and 2 were resolved. At the end of this stage, a total of 91 studies were selected for further evaluation.

During the second round of Stage 2, Reviewers 1 and 2 evaluated the 91 studies based on their title, abstract, introduction, and conclusion sections. As in the previous round, when the reviewers' independent evaluations were completed, the results were combined, and disagreements were identified and resolved. At the end of this stage, a total of 47 studies were selected for further evaluation. Finally, during the third round of evaluation, the full texts of these studies were evaluated based on the selection criteria, and a total of 19 papers were considered to be, to some extent, relevant to our research questions and were selected as primary studies. Table 5 indicates the number of studies excluded during the selection rounds.

TABLE 5 Primary studies selected through three rounds of evaluations

Evaluation Round	Number of articles	Excluded articles	Evaluated based on
1st	4838	4747	title and abstract
2nd	91	44	introduction and conclusions
3rd	47	28	full paper

3.3.4 Data extraction and synthesis (Stage 3)

In stage 3 of the review process, data extraction, a set of relevant data items was extracted from each primary study (see Table 6) and recorded into our data extraction form, which was designed during the planning stage (i.e., Stage 1).

TABLE 6 The data items extracted from the primary studies

ID	Data item extracted	Data item description	Related RQ
DI1	Article title	The title of the primary study	Overview
DI2	Author list	The full list of authors of the primary study	Overview
DI3	Year	The year in which the primary study was published	Overview
DI4	Publication Forum	The name of the forum in which the primary study was published	Overview
DI5	Publication Type	Journal, conference, workshop, or book chapter	Overview
DI6	Research Type	Empirical or conceptual	Overview
DI7	Research Settings	Summary of the empirical research settings	Overview
DI8	Research Focus	The phenomenon under study in the primary study	RQ 1
DI9	Omission Instantiations	The type of quality practices and in which stage of software development they are omitted	RQ 1
DI10	Summary	A summary of the explanation provided by the primary study regarding the omission of quality practices	RQ 2
DI11	Factors	The factors causing the omission of quality practices	RQ 3
DI12	Development context	Is the omission of quality practices bound to any specific software development method, process or approach?	RQ 2, RQ 3

As observed from Table 6, we have extracted data items beneficial for providing an overview of the primary studies (i.e., D1- D6), as well as those necessary for answering our research questions (i.e., D7 – D12). After extracting the data from primary studies, we further evaluated the relevance of each primary study to our research objectives based on short descriptive summaries of primary studies prepared by each individual reviewer.

Finally, during the data synthesis process, each of the primary studies was carefully analyzed to identify the suggested factors leading to the omission of quality practices. In addition, we tried to identify any potential mechanism or process through which software professionals decide to ignore quality software development practices. The results from data extraction and synthesis are presented and discussed in the next section.

3.4 Results of the Literature Review

In this section, we present and discuss the results of our SLR. As mentioned earlier, our initial sample included 4838 studies, from which we have selected 19 primary studies through 3 rounds of evaluations. These primary studies include both empirical and theoretical research published in peer-reviewed journals, conference proceedings, and workshops between 1994 and 2014. An overview of these primary studies is shown in Table 7.

TABLE 7 An overview of the primary studies (PS)

ID	Author(s)	Year	Title	Type	Research Settings
PS1	Lim, Taksande & Seamn	2012	A Balancing Act: What Software Practitioners Have to Say about Technical Debt	Empirical	Data collected from 35 interviews with software professionals from the US and Canada
PS2	Ahonen & Junttila	2003	A case study on quality-affecting problems in software engineering projects.	Empirical	Interviews, modeling sessions, archival materials, and discussions with the representatives of a multinational organization.
PS3	Potdar & Shihab	2014	An Exploratory Study on Self-Admitted Technical Debt	Empirical	Data extracted from source code comments in four large open-source projects.
PS4	Martini, Bosch, & Chaudron	2014	Architecture Technical Debt: Understanding Causes and a Qualitative Model	Empirical	7 focus groups with software professionals from 5 large Scandinavian firms.
PS5	Murugesan	1994	Attitude towards testing: a key contributor to software quality	Theoretical	No empirical data
PS6	McConnel	1996	Avoiding classic mistakes [software engineering]	Theoretical	No empirical data
PS7	Shah, Harrol, & Sinha	2013	Global software testing under deadline pressure: Vendor-side	Empirical	29 interviews, informal discussions, observations, and meetings of 3 teams in India, the UK, and the US.
PS8	Çalikli & Bener	2013	Influence of confirmation biases of developers on software quality: an empirical study	Empirical	Defect log-files and survey from 5 projects in Turkish firm

ID	Author(s)	Year	Title	Type	Research Settings
PS9	Codabux & Williams	2013	Managing technical debt: An industrial case study	Empirical	Interviews, questionnaire, and ethnography were used to collect data from software development department of an industrial firm.
PS10	Seth, Taipale & Smolander	2014	Organizational and customer related challenges of software testing: An empirical study in 11 software companies	Empirical	Data were collected through 18 interviews with software professionals and CEOs from 11 companies of different sizes active in a variety of business domains.
PS11	Wang & Zhang	2010	Penalty policies in professional software development practice: a multi-method field study	Empirical	Interviews and recorded log-files were used to collect data from developers involved in developing games for one the key providers in Chinese online entertainment market.
PS12	Baskerville & Pries-Heje	2004	Short cycle time systems development	Empirical	47 interviews with technical and business staff of 12 firms of various sizes from Denmark and the US.
PS13	Holvitie, Leppänen, & Hyrynsalmi	2014	Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey	Empirical	Data were collected by questionnaire from 54 software developers employed by organizations engaged in software development in Finland.
PS14	Austin	2001	The effects of time pressure on quality in software development: An agency model	Theoretical	Conceptual modeling
PS15	Samalikova, Kusters, Trienekens, Weijters, & Siemons	2011	Toward objective software process information: experiences from a case study	Empirical	Data were extracted from software configuration management databases of 10 embedded-software projects in a Dutch industrial company.
PS16	Fleming	1999	A Fresh Perspective on Old Problems	Empirical	Observations from a project in a small/rapidly growing telecommunications company in the US.
PS17	Lindgren, Wall, Land, & Norström	2008	A Method for Balancing Short- and Long-Term Investments: Quality vs. Features	Empirical	Interviews and archival documents from 7 international firms in automation, telecommunication, and transportation domains.
PS18	Elssamadisy & Schalliol	2002	Recognizing and Responding to “Bad Smells” in Extreme Programming	Empirical	Observations over 3 years in a large project with a team of 50 software professionals from leasing industry.

ID	Author(s)	Year	Title	Type	Research Settings
PS19	Bayer & Muthig	2006	A View-based Approach for Improving Software Documentation Practices	Empirical	Observations during several experiments and case studies in industrial firms. Data are also collected from subjects participating in experiments.

In regard to the publication venues, while the majority of the primary studies (i.e., 19 studies) are published in SE journals and conference proceedings, only two of the primary studies are published in IS journals. From the 19 studies published in SE venues, 6 are journal articles, 8 are conference papers, and 3 are workshop papers.

As it can be seen from Table 8, of the only two primary studies published in IS venues, one is published in *Information Systems Research* (i.e., PS14), and the other is published in *Information Systems Journal* (i.e., PS12), which are both amongst the top IS journals. Because both of these studies were published in the early 2000s and only one of these studies is based on empirical observations (i.e., PS12), it seems that the omission of quality practices has not received enough attention from IS scholars in recent years.

TABLE 8 An overview of the publication forums

Publication Venue	Type	No	REF.ID
IEEE Software	Journal	3	PS1, PS6, PS16
Software Quality Journal	Journal	2	PS8, PS15
Information Systems Research	Journal	1	PS14
Information Systems Journal	Journal	1	PS12
Information and Software Technology	Journal	1	PS7
ACM/IEEE International Conference on Software Engineering (ICSE)	Conference	2	PS11, PS18
Euromicro Conference on Software Engineering and Advanced Applications (SEAA)	Conference	2	PS4, PS17
IEEE International Workshop on Managing Technical Debt	Workshop	2	PS9, PS13
IEEE International Conference on Research Challenges in Information Science (RCIS)	Conference	1	PS10
IEEE International Conference on Software Science, Technology and Engineering (SWSTE)	Conference	1	PS2
IEEE International Conference on Software Maintenance and Evolution (ICSME)	Conference	1	PS3
IEEE International Conference on Software Testing, Reliability and Quality Assurance	Conference	1	PS5
IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS)	Workshop	1	PS19

Alternately, from the SE studies, 8 journal articles and conference papers are published in reputable SE venues, including 3 articles in *IEEE Software* (i.e., PS1, PS6, PS16), 2 papers in *Software Quality Journal* (i.e., PS8, PS15), 1 paper in

Information and Software Technology journal (i.e., PS7), and 2 papers in the proceedings of the *ACM/IEEE International Conference on Software Engineering* (i.e., PS11, PS18). However, none of the primary studies are published in top SE journals, such as *IEEE Transactions on Software Engineering* and *ACM Transactions on Software Engineering and Methodology*. This, in addition to the number of SE studies that were published in recent years (i.e., 10 studies since 2010), indicates that, while there has been increasing interest amongst SE scholars in studying different aspects of the omission of quality software development practices in recent years, these studies lack solid theoretical foundations.

After providing a short descriptive summary of the selected primary studies, we use data extracted from these primary studies and analyses prepared by each of the individual reviewers to answer our research questions in the following sections.

3.4.1 RQ1: How is the omission of quality practices reported by previous studies?

As it can be seen in Table 9, the primary studies report the intentional omission of quality practices from both organizational and individual perspectives. While the former perspective suggests that the decision to omit quality software development practices is made at the organizational level and due to certain business motivations or obligations, the later perspective explains the omission of quality practices as a result of developers' thought processes in favor of certain personal goals.

TABLE 9 A summary of the main findings of the primary studies

ID	Research focus	Summary of findings
PS1	Technical debt	Under time pressure and based on short-term thinking, developers ignore quality practices or perform temporary workarounds while making tradeoffs between quality, time and cost.
PS2	Software Quality	Poor feasibility studies, estimation, and planning decisions cause resource constraints in projects and, in the absence of proper control mechanisms, lead to the neglect of software testing.
PS3	Technical debt	More experienced developers tend to produce more technical debt due to personal goals (which are not mentioned) regardless of release pressure or the complexity of the code.
PS4	Technical debt	Poor requirement specifications, approaching deadlines, the evolution of technology, and the splitting of development and maintenance budgets lead to violations of the architecture and ignoring refactoring, especially when firms are obliged to meet deadlines.
PS5	Challenges of software testing	Because the delivery of software, rather than its quality, has higher priority for managers when coding and design are delayed, they prefer to shortcut testing to catch up with deadlines.
PS6	Challenges of software development	Due to bad estimates, development plans and schedules are often not accurate. Thus, time pressure leads to the elimination of 'non-essential' activities, such as requirements analysis, or software design and QA activities, such as reviews, test planning, and testing.

ID	**Research focus**	**Summary of findings**
PS7	Challenges of software testing	Although testers work under more time pressure than developers and designers, their role is often underrated by managers. This might lower their motivation in performing testing, especially when they face the dilemma of missing deadlines or compromising the quality.
PS8	Software quality	Due to their confirmation bias, developers have a tendency to verify the quality of their code. Thus, they may avoid performing certain unit tests that would break the code and detect defects.
PS9	Technical debt	Developers want to ensure speedy releases and responsiveness to requirement changes. However, due to resource constraints and the evolution of technology, and in the absence of a disciplined development environment, they make trade-offs that lead to technical debt.
PS10	Software testing	Managers sometimes over-trust developers to produce high-quality software and therefore do not involve testers in planning, which leads to overlooking testing scope and underestimating necessary testing efforts. Thus, when managers decide to skip tests due to a lack of resources, this puts developers under stress and leads to low motivation for testing.
PS11	Software quality	Implementing penalty policies in software firms produces the fear of punishment among developers. As a result, software developers, especially novice ones, try to pay extra attention to maximize software quality and avoid intentional omission of quality practices.
PS12	Short-cycle time systems development	Due to evolving market demands, development cycles are compressed to enable developers to respond to constant market change. In such a context, trading software quality for the rapid delivery of high-priority features has become acceptable.
PS13	Technical Debt	Due to frequent requirement changes and scarce resources, the delivery of complete software becomes difficult, and therefore best practices or design guidelines might be violated.
PS14	Shortcutting	Due to poor resource estimation and allocation, developers may face difficulties to meet deadlines. Especially in the absence of proper control mechanisms, developers who are concerned about quality of software may decide to take shortcuts to meet deadlines and avoid negative consequences of missing deadlines on their career.
PS15	Omission of software tasks	When there is slow start-up in projects or the budget is wasted, firms face problems delivering on time and within the budget. Especially when managers do not have a commitment to the firm's official software processes, developers ease up on these processes and omit important tasks.
PS16	Challenges of software development	Due to a lack of proper understanding of software quality amongst software professionals and in the absence of clear software development guidelines, software professionals may consider QA activities to be a waste of time and take shortcuts with them, hoping to improve productivity.
PS17	Software quality	Focusing on achieving short-term goals, such as shorter delivery times and costs and higher productivity, motivates software firms to minimize software processes. Resource constraints and managers' low architectural awareness are other factors leading to the taking of shortcuts.

ID	Research focus	Summary of findings
PS18	Extreme Programming	Due to incorrect estimates, developers take shortcuts and ignore refactoring to ensure speedy development and perform minimal work, especially if they believe too much in their methods.
PS19	Omission of software tasks	Often documentation is ignored or postponed in practice due to the lack of proper documentation guidelines or due to a lack of attention to the importance of documentation.

To explain such organizational and individual decisions, the primary studies use a variety of terminologies, including 'shortcutting' (Austin 2001), 'systematic omission of software tasks' (Samalikova et al. 2011), 'technical debt' (Cunningham 1992), and 'short-cycle time development' (Baskerville & Pries-Heje 2004). In addition, there are a few primary studies that report the research problem by discussing the common issues and challenges of software development projects in general and those of performing software quality assurance and testing activities in particular.

In the following sections, we discuss each of these different viewpoints on the omission of quality practices in more detail.

Omission of quality practices under organizational constraints

The majority of the primary studies explain the omission of quality practices in terms of shortcuts taken by ignoring certain steps or activities recommended by firms' official software development processes (McConnell 1996, Fleming 1999, Ahonen & Junttila 2003, Samalikova et al. 2011, Shah, Harrold & Sinha 2014, Murugesan 1994, Seth, Taipale & Smolander 2014). Such quality-compromising shortcuts are mainly taken under resource constraints (e.g., time and money) or due to the lack of attention to certain software development tasks and activities (e.g., documentation or testing) by managers and developers.

As argued by McConnel (1996), often due to incorrect estimations at the beginning of software projects, development teams prepare inaccurate plans and overly aggressive schedules, and therefore, often during the later stages of projects, developers face scheduling problems. As a result, when a development team is under time pressure, they often eliminate certain activities that they consider to be 'non-essential,' such as requirements analysis or architectural design, or quality assurance activities, such as reviews and testing (McConnell. 1996). These findings are in line with observations reported by Fleming (1999) regarding an industrial software development and maintenance project. In this study, the author explains how software development processes, and especially quality assurance activities, such as design reviews, are ignored by managers and developers simply because they consider such activities as wastes of time, and therefore, they prefer to just concentrate on producing the "real" software (Fleming 1999).

The omission of software development activities under the influence of resource constraints is also reported by Ahonen and Junttila (2003). Conducting case studies and interviewing software developers Ahonen and Junttila (2003) suggest that development teams usually face with lack of sufficient time and resources because the early phases of software projects usually becomes longer than what has been planned. As a result of such resource constraints the quality assurance activities, such as inspection and testing, are often postponed and eventually skipped entirely (Ahonen & Junttila 2003). Another primary study that supports these findings is (Samalikova et al. 2011). This study reports that due to delays in the initial phases of software development, development teams are often faced with resource constraints. In such situations, especially when the management is not committed to the firm's official process, developers do not pay attention to the quality practices, and they might take shortcuts to address scarce resources (Samalikova et al. 2011).

An empirical study by Shah, et al. (2014) reports that test engineers often experience more pressure while performing their tasks compared to other software professionals, such as developers and designers. Such extra pressure is because when software design and development phases are delayed, testers are the ones who must accommodate such delays (Shah, Harrold & Sinha 2014). However, the importance of testing activities and consequently the contribution of testers to the software development is not highly appreciated by managers and other stakeholders (Shah, Harrold & Sinha 2014). Thus, testers usually face a dilemma: to either meet deadlines by compromising the quality of software or miss the deadline but perform their tasks in a high-quality manner (Shah, Harrold & Sinha 2014). In such situations, developers' motivations and the appreciation of testing activities by managers and other stakeholders are considered to be a key factor influencing how well testing activities are performed by testers. Such negative attitudes towards testing are also reported by two other primary studies (Murugesan 1994, Seth, Taipale & Smolander 2014). In his study, Murugesan (1994) argues that, even though testing is "a key contributor to software quality" assurance, it often receives less attention from management. Additionally, the author suggests that, for many developers, testing is like a 'cushion' that can be squeezed whenever needed during the development process. Therefore, whenever the design and coding stages take longer than planned, project managers prefer to reduce the testing time to deliver the software before the deadline (Murugesan 1994). Finally, the results from another empirical study on software testing (Seth, Taipale & Smolander 2014) suggest that project managers deliberately do not involve testers in various project activities, mainly project planning, because they believe too much in the abilities of development teams to produce high-quality software. Therefore, testing scope and necessary testing efforts are often overlooked in the contracts. Consequently, later on during the projects and due to the lack of sufficient resources, project managers decide to skip important software tests (Seth, Taipale & Smolander 2014).

As reported by the first group of primary studies, due to improper estimation and planning activities, software projects are often faced with scarce resources. In such situations, if quality practices do not receive sufficient attention and appreciation from organizations, software developers might not be motivated to perform such quality practices and, as a result, compromise software quality.

Omission of quality practices for gaining strategic competitiveness

The second group of primary studies explains the omission of quality practices in terms of strategic business decisions made by organizations to gain competitive advantages in the market environment and to achieve short-term goals. This group of primary studies uses either technical debt (Cunningham 1992) or agile software development (Fowler & Highsmith 2001) terminologies to note such strategic business decisions. These viewpoints are discussed next.

Technical Debt

A group of primary studies uses the metaphor of technical debt (Cunningham 1992) to explain the strategic omission of quality practices (Lim, Taksande & Seaman 2012, Potdar & Shihab 2014, Martini, Bosch & Chaudron 2014, Codabux & Williams 2013, Holvitie, Leppanen & Hyrynsalmi 2014, Lindgren et al. 2008). Technical debt (Cunningham 1992) denotes the consequences of producing low-quality software in situations where organizations make conscious business decisions to achieve short-term goals by compromising or fully eliminating certain software development activities, such as architectural design, documentation, and testing (Lim, Taksande & Seaman 2012, Martini, Bosch & Chaudron 2014) to speed up delivery times (Brown et al. 2010, Lim, Taksande & Seaman 2012).

According to this group of primary studies, such quality-compromising trade-offs are mainly tactically and reactively made by firms under the influence of market demands. From a business perspective, software companies are motivated to increase their productivity mainly in terms of reducing time-to-market and development costs (Lindgren et al. 2008). Alternately, software companies need to be responsive to market demands and customers changes (Codabux & Williams 2013). Therefore, in such a business environment, taking on technical debt in the short-term might be beneficial or even unavoidable for software companies (Brown et al. 2010) to catch market share (Lim, Taksande & Seaman 2012, Lindgren et al. 2008) or fulfill their contractual obligations (Martini, Bosch & Chaudron 2014). However, because such short-term decisions affect the quality of software, development teams are supposed to go back and fix such workarounds as soon as possible to maintain the quality of the software products in the long run (McConnell 2007, Brown et al. 2010). However, if the skipped tasks are not implemented during the later stages of software development (i.e., the short-term technical debt is not paid back), this leads to higher levels of software deficiency and complexity and, as a result, incurs increased maintenance costs over time (Codabux & Williams 2013).

Agile software development

Another group of primary studies reports the omission of quality practices in terms of utilizing novel software development approaches, such as Internet-speed or short-cycle time system development (Baskerville & Pries-Heje 2004) and Extreme Programming (Beck 1999).

At the turn of the millennium, the rise of electronic commerce provided firms with an opportunity to access a wider range of customers by distributing their products or services through the Internet. However, fierce competition in this fast-changing environment put firms under constant pressure to deliver new software products to market faster (Baskerville et al. 2001, Baskerville et al. 2003). As a result of such 'Internet Time' (Baskerville & Pries-Heje 2004) rush to the marketplace, companies had to shorten the length of their software development cycles (Baskerville et al. 2001, Baskerville & Pries-Heje 2004). It must be noted that Internet-speed and short-cycle time software methodologies are similar to the agile school of thought (Baskerville et al. 2003).

As suggested by Baskerville and Pries-Heje (2004), Scrum (Schwaber & Beedle 2001) and Extreme Programming (Beck 1999), which are two of the most popular agile software development methods, were developed based on the short-cycle development practices used by *Microsoft* and *Netscape* during their competition in developing web browsers. Generally speaking agile software development aim at minimizing development costs and delivery times by avoiding nonessential activities during software development processes (Martin 2003, Codabux & Williams 2013). Due to such demands for shorter development cycles, development teams might become more eager to focus on software functionality and therefore do not pay enough attention to other software activities, such as design, testing, and maintenance (Baskerville & Pries-Heje 2004, Codabux & Williams 2013, McConnell 1996). As a result, the overall complexity of the software and the likelihood of producing defective software are increased (Agrawal & Chari 2007, Gibson & Senn 1989).

Based on qualitative interviews conducted with members of 12 companies from the US and Denmark producing software for fast changing markets, Baskerville and his colleagues determined that such a fast-paced development requires development teams to follow quick and parallel release-oriented prototyping approach in where "quality is negotiable" (Baskerville & Pries-Heje 2004). Another study by (Elssamadisy & Schalliol 2002) reports similar observations from a large 3-year-long software project. In this study, the authors suggest that, following principals suggested by extreme programming in large projects, developers try to speed-up development and perform minimal work. However, due to incorrect effort estimates and their excessive belief in the processes, they have to take shortcuts and ignore refactoring to reach their goals within short development cycles (Elssamadisy & Schalliol 2002).

According to the second group of primary studies, it seems that the overemphasis of short-term goals, such as the delivery of new software features and shorter delivery times, by the software industry increases firms' eagerness to speed-up development processes and therefore might reduce developers' attention to the importance of software quality. As a result, the omission of quality practices with the hope of increasing productivity becomes acceptable within the software community.

Omission of quality practices to achieve personal goals

Finally, a group of primary studies explain the omission of quality practices in terms of developers' thought processes towards achieving certain personal goals (Austin 2001, Çalikli & Bener 2013, Wang & Zhang 2010).

The first study by Austin (2001) suggests that under time pressure and in response to unexpected difficulties during the software development processes, developers might become motivated to take quality-compromising shortcuts to stay on schedule, especially if they consider the deadline to be unachievable. In this conceptual study, the author (Austin 2001) argues that taking shortcuts is not necessarily a deliberate subversive act but rather the result of developers' strategic decisions to address the situation in the most convenient way. In making such quality-compromising decisions, developers often have two main concerns: concern for their career and concern for the quality of the software (Austin 2001). From the career perspective, developers might take shortcuts to avoid the consequences of being behind schedule and losing their professional reputation by being the only developer who cannot be on time. Alternately, from a quality perspective, developers might avoid taking shortcuts because they are concerned with being penalized for compromising the quality of the software and, as a result, endangering the success of the project (Austin 2001).

Another study by (Çalikli & Bener 2013) explains how developers' confirmation biases may cause the emergence of software defects. Confirmation bias, as explained by Çalikli and Bener (2013), is a "tendency of people to seek evidence that verifies hypotheses" rather than seeking evidence that could falsify those hypotheses (Çalikli & Bener 2013). In this empirical study, the authors found some indications that, under the influence of their confirmation biases, developers might try to provide evidence that their code is working properly. Therefore, they might only run certain unit tests that prove the code is working and avoid performing those unit tests that break the code (Çalikli & Bener 2013). As a result of such quality-compromising decisions, the defects in the code might not be discovered.

Finally, in their field of study, Wang and Zhang (2010) discuss the influence of organizational punishment on the quality of software development. In this study, the authors investigated the influence of penalty policies employed by a large Chinese software company on the quantity of software defects identified in the code. Based on this penalty policy implemented in the company, those individual developers who delivered defective software were punished by taking away a specific amount of money, per defect, from their salary. The results of the study suggest that penalty policies partly affect novice developers' performances, leading to less defective software (Wang & Zhang 2010). As an example, an interviewee explained that the penalty policy made them avoid defects that were based on carelessness.

The results of the third group of primary studies suggest that the omission of quality practices might be an individual decision privately made by developers to gain certain career-related advantages. It seems that, in such situations, if developers perceive the omission of quality practices to be beneficial for them, while there is a small chance that such quality-compromising decisions will be revealed, they might decide to ignore the quality practices. This might be the reason that implementing penalty policies could affect the avoidance of the omission of quality practices.

3.4.2 RQ2: What are the common instances of the omission of quality practices reported by previous studies?

To answer our second research question, we have identified all instances of ignoring software development tasks and activities that are reported by primary studies. We have categorized all of these instances into 6 groups according to the nature of the software development activities that are ignored (see Figure 13).

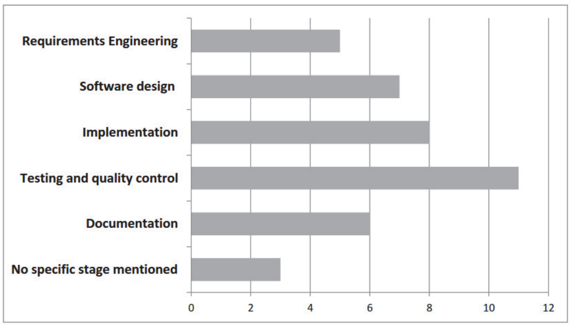


FIGURE 13 Common instances of the omission of quality practices

As observed from Figure 13, the majority of primary studies (i.e., 57%) reported at least one instance of testing and quality control activities being ignored by development teams. Such activities include, for example, planning and scoping testing activities (Lim, Taksande & Seaman 2012, McConnell 1996), writing automated unit tests (Codabux & Williams 2013), conducting formal reviews (Ahonen & Junttila 2003), and performing testing(Murugesan 1994, Baskerville & Pries-Heje 2004).

In addition to this, 42% of the primary studies mention the occurrence of quality-compromising activities during the design and implementation stages, while 31% of the primary studies report instances of ignoring documentation. Finally, 26% of the primary studies report that the omission of quality practices takes place during the requirements analysis and specification phase. It must be noted that 15% of the primary studies report the omission of quality practices in general and do not provide any specific instance nor mention the particular stages of software development that were compromised.

These results show that the omission of quality assurance and testing activities is considerably high among development teams. Keeping in mind that such activities play a vital role in ensuring the quality and reliability of software products, it becomes obvious that this specific aspect of software development has received less attention from the software community. It seems that the constant demands from the software industry and fierce competition between software companies motivate development teams to concentrate more on the delivery of new functional features rather than evaluation of the quality of the software. Such oversight can be a good explanation for high rates of software defects and project failures.

3.4.3 RQ3: Under what conditions does the omission of quality practices take place?

In response to our third research question, we identified a variety of factors during the data analysis that are reported by previous studies as affecting developers' behaviors during the software development processes and, as a result, leading to the omission of quality practices (see Table 10).

TABLE 10 Factors causing the omission of quality practices

Identified factors	Description	Instances
Business goals	From a business perspective, it is desirable or even vital for companies to increase their market share and consequently increase their revenue. As a result, organizations might ignore quality practices to achieve such short-term goals.	Eagerness to increase sales (PS2, PS9), Reduce development costs (PS2, PS3, PS8, PS17), Rapid delivery of high-priority features (PS4, PS17), Collect external funding (PS1), Capture market share (PS1, PS17), Reduce time-to-market (PS1, PS3, PS6, PS9, PS12, PS14, PS16, PS17, PS18)
Project constraints	The extent to which software activities are followed highly depends on the availability of necessary resources, such as time, budget, workforce, and the quality of official development guidelines and control mechanisms in the company.	Lack of time (PS1, PS2, PS4, PS5, PS6, PS7, PS9, PS10, PS13, PS15, PS18), Lack of human resources (PS9, PS13, PS17), Lack of financial resources (PS2 PS9, PS13), Lack of technical skills (PS2, PS9, PS13, PS17), Lack of clear process guidelines (PS4, PS9, PS10, PS19), Lack of clear architectural documentation (PS4), Lack of effective quality control mechanisms (PS14, PS15)
Customers' requirements	Customers' requirements are often not clear at the beginning of projects, which makes requirement changes unavoidable. Thus, sometimes developers might ignore quality practices to address these issues.	Collect early feedback from customers (PS1), Customers' wish lists are too long (PS1), Fuzzy requirements (PS1, PS12), Requirement changes (PS1, PS4, PS12, PS13, PS17)
Technical issues	In some situations, the performance of quality practices is ignored due to technical difficulties associated with software development.	Technology evolution (PS4, PS9, PS12), Use of legacy code (PS4), Use of third-party software (PS4)
Psychological factors	In some cases, ignoring quality practices is an individual decision made by managers, developers, or both and due to their attitudes, feelings, beliefs, or cognitive characteristics.	Lack of commitment to development processes (PS1, PS5, PS15, PS16, PS19), Lack of motivation to perform tasks (PS7, PS19), Developers' confirmation bias (PS8), Interpret requirements conveniently (PS14), No fear of punishment (PS11, PS14), Risk-taking behavior (PS10), Poor buy-in for testing (PS5, PS7, PS10)

As observed from Table 10, a variety of reasons are reported by primary studies as possibly leading to the omission of quality practices. While the majority of the primary studies emphasize the role of resource constraints as a key driver of ignoring quality software development practices, other reasons, such as constant market demands, lack of understanding of customers' requirements, individuals' attitudes and motivations, and technical difficulties associated with software development, are also suggested to cause such questionable practices.

Based on the nature and similarity of identified factors, we have divided them into five main categories: *Business goals*, *Customers' requirements*, *Project constraints*, *Technical issues*, and *Psychological factors*. These categories are succinctly described and different instances of them are reported in Table 10. We explain each of these identified factors in the next section and propose a theoretical model accordingly.

3.4.4 A Synthesis of the Literature Review

We produced a synthesis of the five categories of factors that entail the omission of quality practices. By indicating their scope of effects and interrelationships between these identified factors, our model represents the context in which the psycho-social process of the omission of quality practices is initiated and emerged overtime. We call this process psycho-social because the omission of quality practices occurs in a social context but is implemented by single individuals involved in software development and under the influence of their psychological factors. Based on their scope of effects, these categories are organized into three contextual levels; the market level, organizational level, and individual level (see Figure 14).

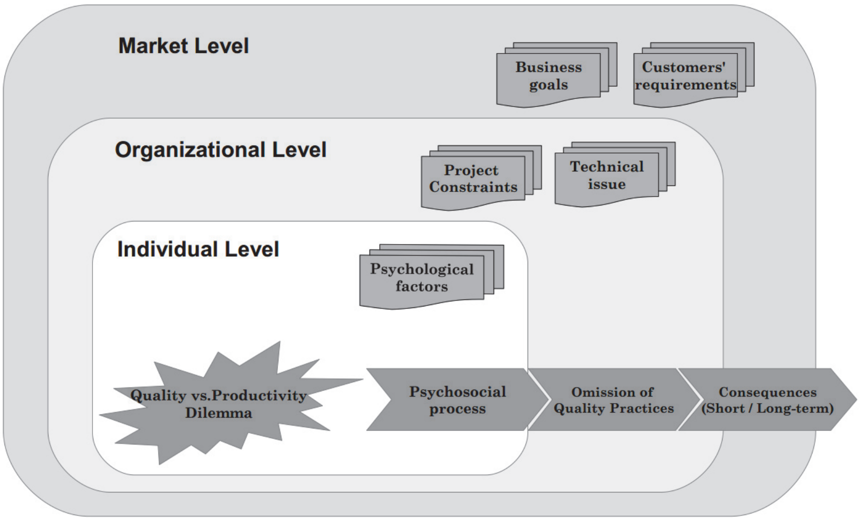


FIGURE 14 The context of the psycho-social process of omitting quality practices

As shown in Figure 14, in the market level, business goals and customers' requirements are two main factors influencing the extent to which quality software practices are followed. In the organizational level, in contrast, the decision to ignore quality practices is influenced by project constraints and technical issues. Finally, in the individual level, psychological factors affect individuals' decisions regarding the omission of quality practices.

These different levels of context, together with the psycho-social process that entails omission instantiations and their consequences, are presented in the following sections.

Market level

Based on our analysis, software development approaches and the extent to which they are followed by organizations are influenced by the market environments that firms are active in. Therefore, two main factors that influence these approaches are firms' business goals and customers' requirements in such market environments.

From a business perspective, increasing sales (Ahonen & Junttila 2003, Codabux & Williams 2013) and extending market share (Lim, Taksande & Seaman 2012, Lindgren et al. 2008) play important roles in increasing firms' revenues. Alternately, reducing time to market (Potdar & Shihab 2014, McConnell 1996, Codabux & Williams 2013, Baskerville & Pries-Heje 2004) and development costs (Ahonen & Junttila 2003, Potdar & Shihab 2014, Shah, Harrold & Sinha 2014) enables software companies to increase their profits or might even be critical for a firm's survival in highly competitive markets. Achieving such business goals in the short term might increase companies' eagerness to speed up their development processes and rapidly deliver new products or novel functional features to the market. By this, not only are firms able to reach the market before their competitors, but they also might be able to increase their revenue and, in some cases, collect external funding (Lim, Taksande & Seaman 2012) to further develop and improve their products. Such strategies are especially vital for small companies active in highly competitive and turbulent market environments with fierce competition. Therefore, and as a result of companies' strategic decisions, development teams might decide to ignore certain quality software development practices. As an example, Ahonen and Junttila (2003) report that while preparing project offers for clients, it is tempting for sales people to reduce unnecessary costs and delays by implementing feasibility studies without proper technical and managerial knowledge. This is because sales people might consider involvement of technical people in this process as an additional cost and delay.

Another factor that influences firms' strategic decisions to ignore certain quality practices is customers' requirements in markets. Customers' needs and requirements are often vague and fuzzy, especially in the initial stages of software projects (Lim, Taksande & Seaman 2012, Baskerville & Pries-Heje 2004). Lack of adequate understanding of requirements among software stakeholders in general, and customers in particular, often leads to rework as developers make design assumptions that later need to be changed (Lim, Taksande & Seaman 2012). Therefore, especially during the early stages of projects, development teams might decide to follow "quick-and-dirty" practices to quickly deliver mockups or even prototypes with minimal functionality to collect feedback from customers (Lim, Taksande & Seaman 2012) and improve the requirements.

Alternately, stakeholders gain a better understanding of customers' needs over time, and therefore initial requirements need to be changed. To satisfy customers, firms often try to increase their response to be able to accommodate such requirement changes (Martini, Bosch & Chaudron 2014, Baskerville & Pries-Heje 2004, Holvitie, Leppanen & Hyrynsalmi 2014, Lindgren et al. 2008). Thus, development teams might decide to ignore certain quality practices to aid in being responsive to customers' needs. For example, as reported by Baskerville and Pries-Heje (2004), Internet-time development is characterized by fuzzy requirements and market pressures. In such an environment, customers appreciate the fast delivery of changing requirements, and they do not even expect high-quality products. As a result of such "negotiable quality" (Baskerville & Pries-Heje 2004), the omission of quality practices becomes acceptable to satisfy customers' needs and expectations.

Our interpretation is that these external market-level factors have a determining role in the omission of quality practices. This is because, depending on the market environment, organizations follow different strategies to increase their sales (and consequently profit) and to satisfy their customers. Such underlying factors motivate firms to speed up negotiation and development processes and to extend their market share by delivering their products to market faster.

Organizational level

Based on our analysis, in the organizational level, project constraints and technical issues are the main factors influencing firms' decisions regarding the omission of quality practices.

Project constraints point to the lack of resources necessary for performing quality software development practices. Such resources include time, budget, skilled workforce, official development guidelines and procedures, and control mechanisms in the company. In software projects, unreliable cost and effort estimation and schedule errors (McConnell 1996, Elssamadisy & Schalliol 2002) often lead to a lack of necessary resources to perform each development phase. A lack of sufficient time (Lim, Taksande & Seaman 2012, Ahonen & Junttila 2003, Martini, Bosch & Chaudron 2014, Murugesan 1994), financial resources (Ahonen & Junttila 2003, Seth, Taipale & Smolander 2014, Holvitie, Leppanen & Hyrynsalmi 2014), or skilled human resources (Codabux & Williams 2013, Holvitie, Leppanen & Hyrynsalmi 2014, Lindgren et al. 2008) are among the main reasons that often force development teams to ignore quality software development practices. To deal with scarce resources, developers are often encouraged to skip those development tasks and activities that, from their perspective, are considered unnecessary (McConnell 2007, Potdar & Shihab 2014, Fleming 1999). Additionally, when there is a split of budget and resources between different development phases, for example, between implementation and testing or development and maintenance, software professionals might become motivated to skip certain tasks and practices during the development phase and postpone them to the maintenance phase (Martini, Bosch & Chaudron 2014).

While lack of time, financial, and human resources restrict developments teams' abilities to follow quality practices, the lack of clear software development guidelines (Martini, Bosch & Chaudron 2014, Codabux & Williams 2013, Seth, Taipale & Smolander 2014, Bayer & Muthig 2006) and inadequate inspection and quality control mechanisms (Austin 2001, Samalikova et al. 2011) facilitate the omission of quality practices (Martini, Bosch & Chaudron 2014, Murugesan 1994, Codabux & Williams 2013, Seth, Taipale & Smolander 2014, Austin 2001). In the absence of proper requirements identification and analysis practices, not only is it very difficult for developers to identify and explicitly document software requirements, but cost and effort estimation also becomes unreliable. For example, according to Martini, et al. (2014) and Lim, et al. (2012), inadequate requirements specification and a lack of clear architectural documentation might be misinterpreted by developers in subsequent development stages and eventually lead to the implementation of incorrect functionalities that must be fixed in the future.

Alternately, because software development processes are invisible to non-developer stakeholders (Austin 2001), this might provide developers with an opportunity to consciously ignore certain quality practices without managers or customers being aware of it (Lim, Taksande & Seaman 2012, Austin 2001). In such situations, the lack of adequate inspection and quality control mechanisms facilitates the omission of quality practices. For example, Ahonen and Junttila (2003) report that the lack of formal inspections causes obvious mistakes to be retained in documents, as experienced people seem to think that an obvious mistake must be there for a reason.

In the organizational level, technical issues are, in some situations, the underlying reasons for the ignoring of quality practices. Such issues include technology evolution (Martini, Bosch & Chaudron 2014, Codabux & Williams 2013, Baskerville & Pries-Heje 2004), the use of legacy code, and the use of third-party software (Martini, Bosch & Chaudron 2014). The software field is a fast changing environment due to rapid technological improvements. With such technological evolution, it is possible for software and hardware to become obsolete over time (Martini, Bosch & Chaudron 2014), and therefore it might not be beneficial for firms to invest too many resources in improving the quality of their software products. This creates a constant need to replace old software and hardware components with new ones. If legacy code or third-party software (Martini, Bosch & Chaudron 2014), for example, is used, any potential architectural debt underlying these components will be transferred to the new software (Martini, Bosch & Chaudron 2014). This means that, if refactoring is not performed and the debt is not paid back, software complexity grows, and future development of the software becomes problematic (Martini, Bosch & Chaudron 2014). In some situations, software developers might be forced to perform temporary workarounds to address such structural issues and complexities. For example, Murugesan (1994) suggests that, due to the complexity of systems, software testing and evaluation become more challenging, and as a result, it is more likely for developers to ignore quality practices.

As discussed in this section, different organizational-level factors, under the influence of the market environment, might force or even motivate development teams to ignore quality practices. However, such institutional-level constraints or motivators cannot be seen as sufficient for the omission of quality practices because the decisions to ignore such practices are made and implemented by individuals. In the next section, we discuss the psychological factors underlying the omission of quality practices.

Individual level

At the individual level, managers and developers engage with the decision-making processes regarding the omission of quality practices. This decision-making is a psycho-social process because it is influenced by individuals' psychological characteristics and thought processes, as well as the characteristics of the development context (i.e., the market-level and organizational-level factors). Here, the psychological factors relate to attitudes, beliefs, and cognitive tendencies that may incline managers and developers to omit quality practices.

For example, as suggested by previous studies, the lack of commitment to firms' software procedures (Samalikova et al. 2011, Fleming 1999, Bayer & Muthig 2006) and lack of appreciation for quality control and testing activities (Murugesan 1994, Shah, Harrold & Sinha 2014, Seth, Taipale & Smolander 2014) might decrease developers' motivations to perform quality practices (Shah, Harrold & Sinha 2014, Bayer & Muthig 2006). In the previous section, we explained that the lack of clear procedures might facilitate the omission of quality practices. However, it must be noted that, even if there are proper software development procedures available in firms, the lack of commitment to these guidelines from managers might simply lead to the neglect of those procedures by development teams (Ahonen & Junttila 2003, Murugesan 1994). In such situations, development teams might prefer to concentrate on producing 'real software' (Fleming 1999) rather than planning (Lim, Taksande & Seaman 2012, Murugesan 1994, Fleming 1999), and as a result, they decide to jump directly to coding and skip project planning activities and other important steps, such as requirements analysis and architectural design. It seems that such negative attitudes towards quality practices are more common in the case of performing quality control activities. Software testing has often received so-called 'second-rate' consideration from stakeholders, which leads to the undermining of testing activities, and therefore it is common for quality control and testing activities to be ignored (Murugesan 1994).

Underestimation of the importance of quality practices by managers alongside cognitive characteristics of individuals, such as confirmation bias (Çalikli & Bener 2013) and risk-taking (Seth, Taipale & Smolander 2014), might lead to developers' decisions to ignore quality practices. For example, Seth (2014) reports that testers are not always involved in project planning because managers overly trust development teams' abilities to produce high-quality software and may take the risk of deciding to skip certain important tests (Seth, Taipale & Smolander 2014). Çalikli and Bener (2013) suggest that, under the influence of confirmation bias, developers might skip certain tests that could possibly break their code and reveal its underlying defects. Such decisions are more likely to be taken if developers do not have any fear of being caught and being punished by organizations (Wang & Zhang 2010, Austin 2001). Because such questionable practices are not easily observable, developers do not feel any fear of facing punishment, and as a result, they might decide to ignore quality practices. The result from an empirical study by Wang and Zhang (2010) supports this finding, as they show that implementing penalty policies could lead to the avoidance of intentional technical debt (Wang & Zhang 2010). This means that the existence of penalty policies may prevent intentional omission of quality practices.

Based on these exemplary studies, we suggest that psychological factors affect the decision-making processes regarding the omission of quality practices, whether pro or against. For example, cognitive tendencies, such as risk-taking or cognitive bias, may positively affect the emergence of omission behavior, and the fear of penalties may work against omission behavior.

The psycho-social process and consequences of omission behavior

During the review and synthesis, we have tried to identify mechanisms or processes through which software professionals decide to neglect quality practices. The majority of the studies deemed that developers simply decide to neglect quality practices, either under schedule and management pressure or based on some personal motives. However, none of the primary studies provide any in-depth explanation regarding the psycho-social mechanisms underlying the omission of quality practices. While this psycho-social process is still unknown, managers and developers go through it by producing omission instantiations on a daily basis in every software development project.

The psycho-social process is likely to be affected by the factors we identified and discussed in this review. However, while this psycho-social process and its underlying mechanisms are undiscovered, the current literature reports a variety of possible factors that may affect this process. Our current understanding is that the identified factors may play different positive or negative roles in the omission of quality practices depending on the context or situation. Despite the fact that a group of studies suggest that the omission of quality practices is associated with a lack of technical skill (Holvitie, Leppanen & Hyrynsalmi 2014, Lindgren et al. 2008, Codabux & Williams 2013), Potdar and Shihab (2014) found in their empirical study that a higher amount of technical debt is produced by developers who are more experienced. These findings show that, if the level of individuals' skills is a factor influencing the psycho-social process, it may have a negative or positive role, meaning that, in the case of skills, both a lack of skill and skillfulness may positively affect the omission behavior.

Based on our analysis, we believe that the values of both productivity and quality seem to play an important role in the psycho-social process of the omission of quality practices. In the software industry, higher productivity is often associated with faster delivery of more features (Fleming 1999, Lindgren et al. 2008). However, concentrating on producing more and doing so faster might increase the number of software defects, which require extra effort and rework to be fixed and consequently decrease both the quality of the software and the long-term productivity (Lindgren et al. 2008). Therefore, from a technical perspective, producing high-quality software might be seen as the key to higher productivity (Fleming 1999). It seems that a typical scenario of the omission of quality practices relates to the conflict between these contradictory concerns of developers and managers. Developers are mainly concerned with performing quality work because they have to work with the code and face its issues on a daily basis (Lim, Taksande & Seaman 2012). Managers, in contrast, experience the pressures of business demands and therefore are concerned with getting work done quickly and with the available resources. Such conflicts might trigger the psycho-social processes that lead to the omission of quality practices.

Regarding its outcome, the omission of quality practices can have different short-term and long-term consequences. In the short term, ignoring quality practices might enable firms to speed up software delivery to capture market share and obtain early feedback with which to improve the software (Lim, Taksande & Seaman 2012). Alternately, the consequences of the omission of quality practices might occur only after the completion of projects and have long-term effects on the organizational level as well as with respect to the firm's position within the market environment. For example, because the omission of quality practices increases software defects (Lim, Taksande & Seaman 2012), organizations may spend extra time and resources to solve these defects, while facing too many issues makes customers unhappy (Lim, Taksande & Seaman 2012) and eventually might decrease the firm's market share.

As discussed earlier, the omission of quality practices might be unavoidable in certain business contexts, and therefore, firms need to find the best possible compromises. Therefore, considering both the short-term and long-term consequences of ignoring quality practices plays a key role in the psycho-social process of the omission of quality practices.

3.5 Research Agenda

Based on the synthesis of the literature review presented in the previous section, we suggest a research agenda that attempts to fill the identified gaps in the current knowledge by addressing a number of main research questions discussed in the following paragraphs.

Future research could approach these questions in several steps. First, instantiations of the omission of quality practices and their nature must be precisely revealed. Second, future research needs to focus on investigating the suggested psycho-social process and its underlying mechanisms that explain the reasons and the processes of decision-making regarding the omission of quality practices. Following this, the consequences of the omission of quality practices, both short- and long-term, must be studied in more detail. Finally, this interpretive knowledge must be used to develop means and solutions to address such questionable behaviors in practice. These suggested steps are briefly described in the following sections.

3.5.1 Research Area 1: What are the instantiations of the omission of quality practices and their nature?

Our literature review suggested that the omission of quality practices may occur in the context of skipping stages of software development, for example, design and testing phases, or certain tasks or activities might be ignored, such as the skipping of unit tests. However, there is no comprehensive description of omission instantiations, no exact timings with respect to the stages of software development, and no analysis of professionals' key roles in such omission instantiations (decision-makers and implementers, for example).

Furthermore, it is important to analyze the nature of omission instantiations with respect to dimensions, such as voluntariness. A comprehensive description is needed to motivate future research that attempts to explain the omission of quality practices (e.g., reasons) and target interventions that aim to prevent omission instantiations to correct both tasks and stages of software development.

3.5.2 Research Area 2: What is the psycho-social process of making decisions regarding the omission of quality practices?

Previous studies mainly suggest that the omission of quality practices occurs as a result of strategic organizational decisions under the influence of different market-level, organizational-level, and human factors. However, there are no studies that have specifically examined the individual and psychological underpinnings of such questionable behaviors, and for that reason, it is not clear why developers decide to omit appropriate software development practices when they know that they could improve the overall quality of the software. This shortcoming becomes even more meaningful when considering that software developers have a tendency to develop high-quality and bug-free software (Austin 2001, Yang, Hu & Jia 2008, McConnell 1996). Therefore, to gain a better understanding of this phenomenon, it is necessary to further investigate both the psychological and social processes through which developers decide to ignore quality practices.

Future research needs to identify psychological processes in developers' cognition and social processes that occur as developers interact with other entities in the development context. Therefore, we call this process psycho-social because the omission of quality practices occurs in a social context but is implemented by single individuals.

3.5.3 Research Area 3: What are the consequences of the omission of quality practices?

In our literature review, we identified several short-term and long-term consequences of the omission of quality practices that mainly concern organizations. In the short term, the omission of quality practices might lead to a reduction of development costs and delivery times or even higher levels of customer satisfaction. Alternately, if quality practices are ignored, it might increase software complexity and decrease software quality in the long term and, as a result, lead to user dissatisfaction, escalating maintenance costs, and financial loss. While such consequences have major importance in today's competitive business environment, the identified primary studies do not provide any clear explanation of how the omission of quality practices might affect software development stakeholders and society beyond such financial factors. Therefore, further research is needed to investigate how the omission of quality practices might affect developers' perceptions of "quality," which eventually influences moral standards and ethics in the software development community and society as well. Furthermore, it is necessary to understand how the omission of quality practices might affect customers' expectations and users' experiences and how such influences might affect the role of information technology in society.

3.5.4 Research Area 4: How to consider omissions of quality practices

Our current wisdom is that the omission of quality practices is ultimately an unwanted behavior that is caused by organizational constraints and incorrect understandings of success, productivity, and quality within the software industry. Therefore, there is a need to develop novel means and solutions to prevent omission instantiations. Such means may be guidelines, methods, or programs that aim to improve quality culture among organizations to consider omission behaviors. Intervention research that aims to change the attitudes or the underlying values of software development, for example, may be used to develop such means and solutions. The previous steps of this research agenda provide knowledge as an input for such intervention studies.

3.6 Conclusions

In recent years, software has become an integrated part of everyday life. Despite the significant amount of resources that have been spent in software development projects, problems in software are widely reported in the research and practice literature. Recent literature hints that software deficiencies might be the result of ignoring proper software development practices. We refer to such 'quick-and-dirty' shortcuts as "omissions of quality software development practices." In such situations, to achieve some short-term gains, a software professional purposefully opts to not follow a proper software development practice, and instead, the developer or the manager chooses to follow a questionable practice that might compromise the software quality.

In this paper, our goal was twofold: first, to discover the state of research on the omission of quality practices and to understand the extent to which this phenomenon has been investigated previously; and second, to determine the root causes underlying the omission of quality practices as suggested by previous studies. To reach these goals we conducted a systematic literature review and produced a synthesis of our findings. We identified five categories of factors underlying the omission of quality practices. Each of these categories, which are originated from different levels of context, affects the omission of quality practices.

In the market level, specific characteristics of the business environment, such as highly competitive and turbulent markets, put development teams under pressure or encourage them to gain competitive advantages through the omission of quality practices. In the organizational level, different factors, including available resources and technical obstacles, might create conditions under which developers decide to omit quality software development practices. Finally, in the individual level, human factors, such as attitudes and cognitive tendencies, under the influence of market- and organizational-level factors might push managers and developers to neglect quality practices.

Based on the analysis of primary studies, we hypothesize that the deliberation of omission behavior concerns the contradiction between quality and productivity and that there is a psycho-social process pertaining to omission instantiations. The current literature does not consider the omission of quality practices adequately with respect to why and how software developers make the decision to omit a quality practice and how to address this phenomenon in practice. Therefore, we have proposed a research agenda with four research areas.

The first research area concerns the determination of instantiations of the omission of quality practices, their timing with respect to stages of software development, tasks they relate to, and the nature of those instantiations. This information is needed to motivate further study to explain omission behavior and to target the interventions that aim to prevent omission instantiations to correct stages and tasks of software development. Alternately, the second research area concerns revealing the psychosocial process of decision-making regarding omission behavior. It is necessary to investigate the psychological processes in developers' cognition and social processes that occur as developers interact with other entities in the development context.

The third research area concerns the consequences of the omission of quality practices. Further research is needed to investigate how such omission practices might affect developers' perceptions of "quality practices" and, consequently, moral standards and ethics within the software development community and society as well. Furthermore, it is necessary to understand how the omission of quality practices might affect customers' expectations and users' experiences and how such influences might affect the role of information technology in society.

Finally, the fourth area concerns possible solutions for considering the omission of quality practices. Our current wisdom is that the omission of quality practices is undesirable behavior and that there is therefore a need to develop novel means (e.g., guidelines, methods, culture) to prevent omission instantiations.

To summarize, the goal of this research agenda is to provide computing scholars research avenues for developing knowledge on the omission of quality practices. New knowledge is needed to develop preventive or developmental means for the practice of software development to consider the identified issue.

4 STUDY 2 - SEEKING TECHNICAL DEBT IN CRITICAL SOFTWARE DEVELOPMENT PROJECTS: AN EXPLORATORY FIELD STUDY[6]

4.1 Abstract

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

4.2 Introduction

Despite all the financial and human resources that have been spent on large software and system development projects, the majority of these projects continue to fail or face severe challenges (The Standish Group International, Inc. 2009). Although different technical and human problems might be the potential

[6] Ghanbari, H. (2016). Reproduced with kind permission by IEEE Computer Society.

cause for software vulnerabilities and failure, previous research has shown that software defects are the main cause of most software vulnerabilities (Fonseca & Vieira 2008, Wijayasekara et al. 2012).

Even though identifying and fixing software deficiencies such as bugs, missing requirements or flaws in software design (Notander, Höst & Runeson 2013) has major importance in increasing the quality and reliability of software products, some of these defects might stay hidden; even if they are identified, they may not be fixed rapidly (Wijayasekara et al. 2012). In addition, fixing software deficiencies at the later stages of projects becomes more expensive and time consuming (Banker, Davis & Slaughter 1998, Van Emden & Moonen 2002). Thus such deficiencies must be avoided in the first place, especially in critical systems where software failure might cause devastating financial and infrastructural consequences, or human life loss or injuries (Wijayasekara et al. 2012, Sommerville 2015).

In response to these problems, the software community has been mainly attempting to identify new software development tools and methods. Over time, a large number of software development methods were designed to manage complexity in software projects (Avison & Fitzgerald 2003, Iivari & Maansaari 1998). However, it is widely reported in previous studies that these software development methods are rarely followed in their entirety but are customized (Boehm & Turner 2003, Conboy & Fitzgerald 2010, Baskerville & Pries-Heje 2004). A group of scholars explain this customization mainly in terms of quality compromising trade-offs for minimizing development costs and delivery times (Vartiainen & Siponen 2012, Codabux & Williams 2013, McConnell 2007, Ahonen & Junttila 2003). In such situations, developers are often forced or motivated to cut back on software development processes or to postpone certain activities (Tom, Aurum & Vidgen 2013, Austin 2001). The metaphor of technical debt (Cunningham 1992) has been increasingly used by scholars and practitioners to point out such quality compromising shortcuts (McConnell 2007, Tom, Aurum & Vidgen 2013, Kruchten, Nord & Ozkaya 2012, Brown et al. 2010, Lim, Taksande & Seaman 2012).

While minimizing development time and costs might play a key role in highly competitive markets (Eberlein & Leite, Julio, Cesar, Sampaio, do Prado 2002, Sommerville 2005), quality of software often has a higher priority in developing critical systems (Sommerville 2015). Since technical debt has a negative impact on software quality, it must be avoided when developing critical systems. However, previous studies showed that technical debt does not always occur because of bad design and development decisions but also might be due to environmental factors that cannot be controlled by development teams (Tom, Aurum & Vidgen 2013, Brown et al. 2010). Therefore, it is important for development teams to identify sources of technical debt in their context and to properly manage it in order to maintain software quality (Kruchten, Nord & Ozkaya 2012).

While in recent years technical debt has received considerable attention from the agile community (Kruchten, Nord & Ozkaya 2012, Brown et al. 2010, Bavani 2012, Holvitie, Leppanen & Hyrynsalmi 2014), to the best of our knowledge, there are no studies that explore this phenomenon in critical software projects. As it is suggested by (Brown et al. 2010), there is a need for empirical studies to investigate the potential sources of technical debt across development contexts. Therefore, we performed an exploratory field study to gain a better understanding of the nature of technical debt and its potential sources in critical domains.

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

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

4.3 Related work

In today's highly competitive business environment, development teams are under constant pressure to produce high-quality software in a shorter time and with minimum amount of costs (Eberlein & Leite, Julio, Cesar, Sampaio, do Prado 2002, Sommerville 2005). However, since producing high-quality software is usually associated with higher costs and delivery times (Sommerville 2015), maximizing both software development productivity and software quality simultaneously becomes challenging. Software development productivity is often measured based on the number of lines of new code produced per person-day (Cusumano et al. 2003, MacCormack et al. 2003) while software quality is measured based on the number and frequency of defects identified in the software products (Cusumano et al. 2003, MacCormack et al. 2003, Kan 2002).

Sometimes firms have to make trade-offs between long-term software quality and short-term productivity (McConnell 2007, Lim, Taksande & Seaman 2012, Potdar & Shihab 2014). In such situations often quality practices are neglected to deal with the urgent demands imposed by the business environment (Vartiainen & Siponen 2012, Ahonen & Junttila 2003). For example, according to (Ahonen & Junttila 2003), planned tests are often neglected or postponed due to insufficient amount of time. A group of studies use the metaphor of technical debt (Cunningham 1992) to explain such quality compromising trade-offs (Tom, Aurum & Vidgen 2013, Brown et al. 2010, Lim, Taksande & Seaman 2012).

The term technical debt has been originally introduced by Cunningham (1992) to point out poorly written code. However, in last two decades the metaphor has been used in a variety of ways to explain flaws and imperfections in documentation, design, coding and testing activities (Tom, Aurum & Vidgen 2013, Kruchten, Nord & Ozkaya 2012, Brown et al. 2010, Lim, Taksande & Seaman 2012). In this study, following (McConnell 2007, Tom, Aurum & Vidgen 2013, Lim, Taksande & Seaman 2012) we use the definition of technical debt as conscious decisions to cut back on software development processes in order to minimize development costs and delivery times. The most common demands reported in the literature to be the cause of technical debt are time pressure and insufficient amount of budget and human resources (McConnell 2007, Brown et al. 2010, Nan & Harter 2009). However, some studies show that technical debt could be the result of environmental factors which are out of control of development teams (Tom, Aurum & Vidgen 2013, Brown et al. 2010).

Even though it may sometimes be necessary for organizations to take on technical debt, such decisions lead to higher levels of software deficiency and complexity (Brown et al. 2010). This is even more problematic in bigger projects where a larger number of developers simultaneously develop different parts of the system, and the increased complexity makes it difficult—and sometimes even impossible—for them to develop and maintain the software under development (Banker, Davis & Slaughter 1998, Van Emden & Moonen 2002).

While development costs and delivery time are important aspects of every software project, in developing critical systems factors such as software reliability and maintainability are of major importance. Due to high failure costs and consequences a set of expensive and trusted software development methods must be utilized for developing critical systems (Sommerville 2015). Often in such projects, developers follow plan-driven and document-centric software processes to show compliance with certain standards (Notander, Höst & Runeson 2013). Despite using such strictly defined and heavily planned processes (Notander, Höst & Runeson 2013), development teams might sometimes ignore predefined processes and given standards due to the occurrence of unexpected issues within business environments and organizations. As a result, the occurrence of technical debt becomes unavoidable even in critical software projects. For that reason development teams must be aware of and properly manage technical debt in order to maintain quality of software (Kruchten, Nord & Ozkaya 2012, Brown et al. 2010).

4.4 Research method

In this research, our aim is to gain an understanding about the nature of technical debt and its potential sources in critical domains. As it is suggested by (Runeson & Höst 2009), conducting exploratory studies is a suitable method for gaining such knowledge and generating insights about the phenomenon under study. Therefore, we decided to conduct a two-stage exploratory field study to build an empirically grounded understanding about the nature of technical debt in critical software projects.

4.4.1 Data collection

Following the qualitative interview guidelines suggested by (Myers & Newman 2007), two rounds of face-to-face, semi-structured interviews with international software engineers were conducted (see Table 11).

TABLE 11 A summary of the interviewees characteristics

	Interviewee	Position	Years of Experience	Domain
Stage 1	Interviewee 1	Software Engineer	8	Commerce
	Interviewee 2	Software Engineer	3	Healthcare
	Interviewee 3	Software Engineer/ Process Manager	9	Automotive
	Interviewee 4	Software Engineer	11	Automotive
	Interviewee 5	Software Engineer	14	Automotive
Stage 2	Interviewee 6	Software Engineer/ Business Manager	22	Aerospace
	Interviewee 7	Software Engineer/Team Leader	7	Aerospace
	Interviewee 8	Software/System Engineer	6.5	Aerospace
	Interviewee 9	Software Engineer/ Project Manager	6	Aerospace
	Interviewee 10	Software Engineer/ Team Leader	8.5	Aerospace
	Interviewee 11	Software Engineer	20	Aerospace
	Interviewee 12	Software Engineer/ Team Leader	8	Aerospace

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

4.4.2 Stage1: Preliminary interviews

In Stage 1, we interviewed five international software developers from the automotive, healthcare, and financial sectors between July and November 2013. The aim of these interviews was to investigate whether technical debt might occur in such critical domains. Since the results from these interviews provided some initial evidence that technical debt occurs even in developing critical systems, we decided to further our investigation during the second stage.

4.4.3 Stage 2: Case study

In Stage 2, we conducted a single-case case study (Runeson & Höst 2009) in a company called Beta[7] that is active in the aerospace domain. During this stage we interviewed seven international software engineers from Beta in January

[7] Please note that Beta is a pseudonym.

2014. Due to the diversity of projects in Beta, we decided to interview software experts with a variety of work experience levels from different teams. All of these interviewees hold a university degree in Software Engineering or relevant fields.

In addition to the interview data, supplementary data sources such as firm's official procedures, project documents and public information available on their website were used to analyze different aspects of development processes. Since a large amount of data was collected, we used a tool called NVivo[8] for proper data analysis and management.

4.4.4 Case description

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

Due to the criticality of the aerospace domain, the companies active in this field, including Beta, are expected to comply with certain standards and regulations such as ECSS-E-ST-40C (European Space Agency 2009) and ECSS-Q-ST-80C (European Cooperation for Space Standardization 2013). To comply with these standards a V-model (Boehm 1984) is proposed by authorities to be followed (see Figure 15).

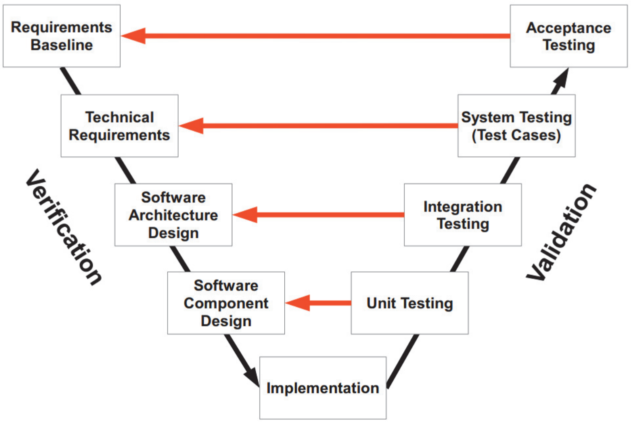


FIGURE 15 An overview of the V-model used in Beta

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

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

At the moment, a wide variety of tools are utilized in the case company to support its teams' day-to-day activities. However, the extent to which Beta teams utilize these tools highly depends on their projects. Some teams mainly use simple tools and technologies, while in other teams more complex tool chains and technologies are used in order to deal with the complexity and criticality of products under development. In addition to common Integrated Development Environments, when needed, Beta teams use configuration management systems, code repositories, and other tools for reporting and tracking bugs and source code documentation.

4.4.5 Data analysis

To build an understanding that is empirically grounded in the experience of professionals involved in software processes, a systematic data analysis process was conducted by following the techniques suggested by the Grounded Theory method (Glaser & Strauss 1967, Glaser 1978, Glaser & Holton 2004, Glaser 1992). Using NVivo, we first performed a line-by-line *open coding* (Glaser 1978, Glaser 1992, Glaser & Holton 2004)to closely examine fractures of data and to form categories of codes (Urquhart, Lehmann & Myers 2010).

During the next stage, *selective coding* (Glaser 1978, Glaser 1992, Glaser & Holton 2004), we grouped these categories of codes into four higher levels of abstraction (i.e., concepts) called *Ambiguity of Requirement*, *Diversity of Projects*, *Inadequate Knowledge Management*, and *Resource Constraints*. These categories represent the challenging aspects of software processes from our interviewees' perspective, which might lead to the occurrence of technical debt in critical software projects.

Finally, during the last stage of data analysis, *theoretical coding* (Glaser 1992, Glaser & Holton 2004, Glaser 1978), a theoretical model was formed by indicating the relations between the identified selective codes. This theoretical model and the relations between these theoretical concepts are explained in the next section. It is worth noting that a constant comparison through iterative data collection and analysis enabled us to enrich the emerging theoretical concepts by identifying shortcomings in the collected data and to address them by collecting more data.

4.5 Results

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

4.5.1 Ambiguity of Requirements

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

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

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

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

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

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

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

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

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

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

4.5.2 Diversity of Projects

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

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

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

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

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

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

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

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

Our data show that in Beta developers often learn to use those practices and tools that are used within their teams. Therefore, in case they switch from one team to another, they need to spend some time to familiarize with the tools and practices utilized within the new team. From our data we realized that if developers consider software development processes in the new team to be outdated or time-consuming, they might underestimate the value of these processes.

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

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

4.5.3 Inadequate Knowledge Management

Both technical and product knowledge are among the most essential resources for performing software development activities. It is suggested in previous studies that knowledge documented in a software company or held by its employees is one of the key competitive assets of that firm (Notander, Höst & Runeson 2013). In Beta, extensive documentation is often required by regulations and, for that reason, development teams might spend a considerable amount of time and effort to fulfill this requirement. The following quotation indicates an example of such extensive documentation.

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

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

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

The majority of the employees in Beta has similar levels of education and basic knowledge of the aerospace industry. Still, if individuals move from one team to another, the information from previous projects stored on the server can be a key source of information for them to familiarize themselves with the overall responsibilities of the new team. Thus, the lack of a well-structured and updated source of information forces them to spend extra time and effort to gain the necessary knowledge for performing their tasks. Due to such shortcomings, it is likely that technical debt occurs during knowledge-creation and management activities.

4.5.4 Resource Constraints

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

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

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

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

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

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

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

4.5.5 Theoretical model

In previous sections, we discussed the four main factors that make software processes challenging in critical domains, particularly in the case of Beta. As a result, development teams might not be able to perform planned development activities as recommended by given standards, which might eventually lead to the occurrence of technical debt. These four categories include ambiguity of requirements, diversity of projects, inadequate knowledge management, and resource constraints. Figure 16 shows our theoretical model, which is formed by indicating the relations between these identified selective codes.

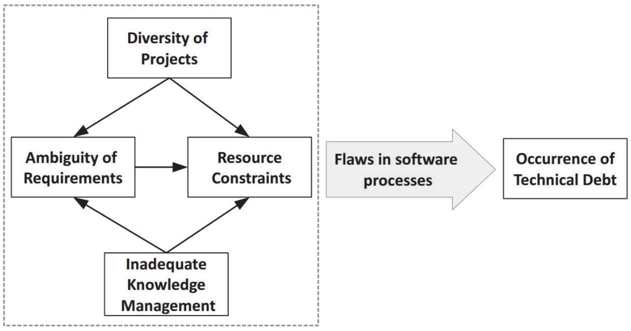

Diversity of Projects
Ambiguity of Requirements
Resource Constraints
Inadequate Knowledge Management
Flaws in software processes
Occurrence of Technical Debt

FIGURE 16 The occurrence of technical debt in Beta

As shown in Figure 16, the diversity of projects influences both the ambiguity of requirements and resource constraints. Depending on the project, stakeholders' requirements might be vague or well defined. On the other hand, the amount of resources allocated to projects highly depends on the type of the project. Inadequate knowledge management worsens the ambiguity of requirements due to the fact that performing proper requirements engineering and specification becomes challenging. Inadequate knowledge management alongside with ambiguity of requirements makes it difficult for development teams to perform a precise cost and effort estimation and, as a result, evaluation of necessary development resources becomes difficult. This might lead to a lack of necessary resources later in the project.

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

4.6 Discussion

Depending on the development context, there might be different constraints and regulations that force development teams to concentrate more on certain aspects of software development processes. Often in critical domains, quality of the systems and compliance with certain standards has a higher importance for stakeholders, due to which plan-driven processes and expensive techniques are followed by software development teams (Notander, Höst & Runeson 2013, Sommerville 2015). However, our data collected from several projects indicate that even in critical domains, pressure caused by different business and organizational sources makes it challenging for developers to follow plan-driven processes. In order to deal with such challenges, development teams might decide to ignore or postpone certain software development activities (Notander, Höst & Runeson 2013, Ahonen & Junttila 2003). As a result of this minimization, technical debt might occur in such critical projects.

Ambiguity of requirements is one of the key factors that make software projects challenging in critical domains. Requirement change is reported by previous studies to be one of the biggest challenges of software projects (Sommerville 2005). Even in critical domains that are considered to be more stable, software requirements might change over the course of a project, especially if these requirements are not clearly defined and specified at the beginning of the project. Such deficiencies in requirements specification might lead to the occurrence of requirements debt (Tom, Aurum & Vidgen 2013, Brown et al. 2010, Lim, Taksande & Seaman 2012).

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

The availability of necessary resources is another factor that directly affects the way software processes are followed. Often cost and effort estimation is challenging in software projects and it is almost impossible to clearly specify the necessary development resources at the beginning of projects. Our results show that due to a lack of necessary resources, development teams sometimes have to omit certain steps of software processes. Resource constraints are widely reported by previous studies to cause technical debt (McConnell 2007, Tom, Aurum & Vidgen 2013, Brown et al. 2010, Lim, Taksande & Seaman 2012, Yang, Hu & Jia 2008).

Finally, inadequate knowledge management is another issue that makes software projects challenging. Technical and product knowledge is one of the key elements integrated into every software development process (Notander, Höst & Runeson 2013). Thus any potential obstacle in proper knowledge creation and management might cause severe problems for individuals while performing their activities and, as a result, lead to the occurrence of technical debt. This kind of debt has been reported by previous studies as knowledge distribution and documentation debt (Tom, Aurum & Vidgen 2013, Brown et al. 2010, Lim, Taksande & Seaman 2012).

Our analysis shows that the occurrence of technical debt becomes unavoidable even in critical projects. Therefore, it is critical for development teams to properly identify and effectively manage debt (Kruchten, Nord & Ozkaya 2012, Brown et al. 2010).

It is suggested by previous studies that using agile practices assist development teams to reduce and manage technical debt (Codabux & Williams 2013, Brown et al. 2010, Holvitie, Leppanen & Hyrynsalmi 2014, McCaffery, Pikkarainen & Richardson 2008). On the other hand a group of studies (Notander, Höst & Runeson 2013, McCaffery, Pikkarainen & Richardson 2008, Silva & Cunha 2006), suggests that following a combination of plan-driven and agile methods in critical projects not only allows teams to perform their tasks in a cost-effective manner but also to comply with the different quality levels requested by customers or regulatory authorities. Following these studies, we suggest that integrating agile practices into common, plan-driven software processes used in critical domains enables development teams to tackle technical debt.

Following practices such as small releases, burndown charts, daily meetings, test-driven development, and continuous testing might assist development teams to avoid technical debt to accumulate in their projects. Using burndown charts and daily meetings help developers to monitor their progress and to identify potential obstacles in performing their tasks (Codabux & Williams 2013, McCaffery, Pikkarainen & Richardson 2008, Silva & Cunha 2006). By this they will be able to better estimate the cost and effort necessary for performing their future tasks. In addition following test driven development and continuous testing (Brown et al. 2010, Holvitie, Leppanen & Hyrynsalmi 2014) enables developers to identify defects and problems in small releases (Codabux & Williams 2013, McCaffery, Pikkarainen & Richardson 2008). As a result, teams are able to deal with their problems as soon as possible by renegotiating or even changing their initial plans as needed (Silva & Cunha 2006). Especially in companies like Beta that have small, collocated teams, communication and collaboration between developers becomes easier and, as a result, teams are more flexible to follow iterative methods.

On the other hand, conducting review meetings and retrospectives and preparing technical debt backlogs (Kruchten, Nord & Ozkaya 2012) enables development teams to properly communicate, trace and manage their technical debt. In addition, organizing company-wide review meetings enables individuals from different teams and departments to communicate their problems and to identify possible solutions to deal with them (Notander, Höst & Runeson

2013, Codabux & Williams 2013). Using such meetings, as also suggested by a number of previous studies (Codabux & Williams 2013, Holvitie, Leppanen & Hyrynsalmi 2014, McCaffery, Pikkarainen & Richardson 2008), enables teams to be engaged in more frequent and reciprocal information exchange and, as a result, better communicate and manage technical debt. In addition, the general awareness of teams regarding the potential sources of technical debt increases. One of the most important benefits of such awareness is to avoid spending resources on overlapping attempts for identifying solutions that have already been identified by other teams (Codabux & Williams 2013).

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

4.7 Conclusions

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

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

5 STUDY 3 - WHY SOFTWARE DEVELOPMENT METHODS ARE CUSTOMIZED IN PRACTICE - A THEORY OF SOFTWARE DEVELOPMENT BALANCE[9]

5.1 Abstract

Over the last four decades, software development has been one of the mainstream topics in the Software Engineering (SE) and Information Systems (IS) disciplines. Thousands of methods have been put forward offering prescriptions for software development processes. The goal of these methods is to produce high-quality software in a systematic manner. However, it is widely known that these methods are rarely followed as prescribed; rather developers often modify or ignore different steps and practices recommended by given methods. While a group of previous studies suggests that maximizing the flexibility and leanness of software development processes is the key driver of such customizations, another group argues that the inadequacy of these methods to fulfill stakeholders' expectations is the main reason that they are ignored in practice. However, to the best of our knowledge, there are no theory-based and empirically grounded explanations elucidating why and under what conditions software development methods are customized in practice. As a first step in overcoming this gap in the research, we conducted a longitudinal field study, using a Grounded Theory methodology, and built a process theory. This theory explains the mechanisms through which software development methods are customized in order to maintain balance between contrasting and sometimes contradictory contextual forces associated with software development.

[9] Ghanbari, H. and Siponen, M. (Under review at IEEE Transactions on Software Engineering).

5.2 Introduction

Software development continues to be a popular topic in the Software Engineering (SE) and Information Systems (IS) disciplines and at the same time receives a considerable attention from practitioners. Software development is a highly dynamic and complex phenomenon (McLeod & Doolin 2012) consisting of a set of interrelated steps and activities (Bourque & Fairley 2014) which its purpose is to ensure that user requirements are transformed into working software (Sabherwal & Robey 1993, Slaughter et al. 2006, Truex, Baskerville & Travis 2000). To better manage these complex processes, a large number of software development methods have been put forward by the SE and IS communities. These methods often recommend a set of predefined steps, activities, and best practices for developing and maintaining software products (Iivari 1991, Iivari, Hirschheim & Klein 1998b). While it is argued that the development process to be followed is an important factor that may contribute to a project's success (Iivari 1991, Baskerville & Pries-Heje 2004), it has been widely reported by previous studies that these methods are not followed as prescribed (Truex, Baskerville & Travis 2000, Baskerville & Pries-Heje 2004, Avison & Fitzgerald 2003, Coleman & O'Connor 2007, Conboy & Fitzgerald 2010, Fitzgerald 1998, Iivari & Maansaari 1998, Huisman & Iivari 2006, Kiely & Fitzgerald 2002, Ralph 2016, Fitzgerald, Hartnett & Conboy 2006); rather, software development teams modify or skip certain steps or practices that are recommended by these methods (Highsmith & Cockburn 2001, Sommerville 2005).

From an organizational perspective, maximizing the leanness of software development processes is one of the main reasons for customizing methods (Baskerville & Pries-Heje 2004, Fitzgerald, Hartnett & Conboy 2006, Boehm 2002, Baskerville et al. 2001, Lim, Taksande & Seaman 2012, Lindgren et al. 2008), while the lack of a universal method that is suitable for all types of software projects (Truex, Baskerville & Travis 2000, Iivari 1991, Brinkkemper 1996, Henderson-Sellers & Serour 2005) is another important reason forcing development teams to modify and even combine different software development methods and practices in each project (Conboy & Fitzgerald 2010, Boehm 2002, Henderson-Sellers & Serour 2005, Leppanen 2006, Iivari & Iivari 2011).

While the selection of software development methods is an organizational decision, developers are the ones who must apply these methods in practice (Khalifa & Verner 2000). Despite the important role of individuals in implementing software development methods, to the best of our knowledge, there are no empirically grounded theories explaining how and under what conditions developers may decide to modify software development methods or even ignore best practices recommended by the software development community. In this study, we aim to contribute to resolving this issue by proposing an empirically grounded theory. To this end, we investigate the following research questions:

- *RQ 1: Under what conditions software development methods are customized in practice?*
- *RQ 2: Through what causal mechanisms software development methods are customized in practice?*

Drawing from the experience of software professionals active in different contexts, we built a process theory (Markus & Robey 1988, Van de Ven, A. H. & Poole 2005, Van de Ven, A. H. 1992, Ralph 2015a) to explain the causal mechanisms underpinning the customization of software development methods.

Our proposed theory contributes to both research and practice. Our theory contributes to the research by explaining how the mechanisms of method customization is initiated by the unique characteristics of software and how it progresses through a series of complex interactions among software stakeholders under the influence of contextual forces. This theory indicates how developers, depending on the development context, decide to modify or ignore certain software development practices or activities to deal with such complex and inconsistent contextual settings. Until now, the majority of research on software development, especially in the SE field, has focused on identifying and improving novel methods and practices, while lacking firm theoretical foundations (Ralph 2016, Zhang & Budgen 2012). Therefore, there is a need to propose novel SE theories (Sjøberg et al. 2008, Wohlin, Šmite & Moe 2015, Johnson, Ekstedt & Jacobson 2012) that explain why these software development methods are widely ignored in practice. To that end, we propose a novel theory called the *Theory of Software Development Balance* to explain the customization of software development methods as mechanisms—at both individual- and organizational-levels—for maintaining balance between stakeholders' multi-concerns and contradictory contextual forces over time.

Our theory also contributes to practice by indicating how developers choose different strategies to balance inconsistent situations within software projects that are caused by contrasting contextual forces and stakeholders' concerns. This explanation enables organizations to better understand developers' attitudes toward firms' business strategies and software development procedures and how these procedures are followed in the face of contextual inconsistencies. Utilizing such insight may enable software firms to improve their development approaches and to form integrated teams that are suitable for a given context.

The rest of this paper is structured as follows. Section 4.3 reviews the prior related research, followed by a description of the research methodology in Section 4.5. Section 4.6 represents our theory, and Section 4.7 discusses our results in light of the related literature and reports the implications and limitations of our study. Finally, Section 4.8 concludes the paper.

5.3 Research Background

Software development is a dynamic (McLeod & Doolin 2012) and knowledge intensive (Wohlin, Šmite & Moe 2015) phenomenon consisting of a set of interrelated steps and processes such as requirements engineering, software construction, evaluation, and maintenance (Bourque & Fairley 2014). During each of these steps, a variety of activities are performed, and as a result of constant interaction and collaboration between software stakeholders, different types of software artifacts (e.g., requirement documents, architectural designs, code, and test scripts) are produced.

Early computer systems were developed by individual programmers without following any development methods (Avison & Fitzgerald 2003, MacCormack et al. 2003). Over time, as software projects failed or were faced with severe challenges, organizations realized a need for more systematic and formalized approaches to manage complexity of software projects and to improve the quality of software processes (Fitzgerald 1998, Avison & Fitzgerald 2003). Consequently, during the 1960s, systems development methods emerged for the purpose of managing software processes (Avison & Fitzgerald 2003). Since that time, thousands of methods have been put forward (Conboy & Fitzgerald 2010, Huisman & Iivari 2006) to assist developers in improving the software development processes (Iivari & Maansaari 1998, Iivari, Hirschheim & Klein 2004). To that end, these methods often provide guidelines for performing a set of generally accepted practices and steps for developing and maintaining software products (Iivari 1991, Iivari, Hirschheim & Klein 1998b), the assumption being that software development is an orderly, progressive and systematic process (Truex, Baskerville & Travis 2000). In the following sections we briefly discuss different viewpoints regarding the application of software development methods in practice.

5.3.1 Software Development Methods

Using software development methods is traditionally seen as valuable (Fitzgerald 1998), since it facilitates management's control over software projects and enables firms to improve their software development processes (Bourque & Fairley 2014, Fitzgerald 1998, Vavpotič & Vasilecas 2012). Advocates of software development methods suggest that greater use of software development methods, alongside with skilled professionals and the appropriate resources, leads to higher levels of productivity, product quality, budgetary and schedule adherence, and customer satisfaction (Bourque & Fairley 2014, Coleman & O'Connor 2007, Fitzgerald 1998, Vavpotič & Vasilecas 2012, Carmel & Becker 1995, Cugola & Ghezzi 1998, Shirado et al. 1996). Especially since software development practices have long-term consequences that are difficult and costly to reverse (Banker, Davis & Slaughter 1998), designing more effective and efficient methods has received a great deal of attention from the software community (Leppanen 2006, MacCormack et al. 2003, Banker, Davis & Slaughter 1998, Lyytinen 1989).

Despite the attention that has been given to suggesting and improving software development methods, according to the literature, these methods are rarely used effectively (Iivari & Maansaari 1998). In general, there are three viewpoints regarding the use of methods in software and system development processes (see Figure 17).

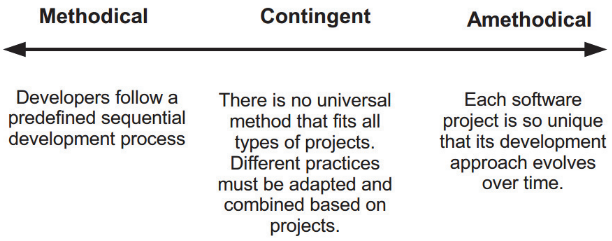

Methodical
Contingent
Amethodical
Developers follow a predefined sequential development process
There is no universal method that fits all types of projects. Different practices must be adapted and combined based on projects.
Each software project is so unique that its development approach evolves over time.

FIGURE 17 Different viewpoints about the use of software development methods

The first view considers software development as a systematic (Wynekoop & Russo 1997) and highly rationalistic process (Hirschheim & Newman 1991) that can be completed by following a method in a step-by-step fashion. Truex et al. (2000) argue that such a viewpoint, where developers must follow a predefined sequential development process, has dominated the majority of publications and textbooks on software and information system development (Truex, Baskerville & Travis 2000). The wide range of available methods can be classified as traditional life-cycle methods such as *waterfall* (Royce 1970), iterative-incremental methods like *spiral* (Boehm 1988), and lightweight methods such as *agile methods* (Fowler & Highsmith 2001, Abrahamsson et al. 2003).

The second perspective regarding the application of methods suggests that, like other production processes, software development and its outcomes are subject to uncertainty and risk (Iivari & Lyytinen 1998), and almost every software project has its unique characteristics, including individual goals, the technological and business environment, and the organizational context (Truex, Baskerville & Travis 2000, Conboy & Fitzgerald 2010, Brinkkemper 1996). Therefore, software development cannot be conceived of as a systematic and merely technical process but as a highly complex socio-technical process (McLeod & Doolin 2012). This complex process is comprised of various social, technical, and institutional entities (Orlikowski & Iacono 2001), and therefore is influenced by different social and environmental factors, including but not limited to social interactions (Kiely & Fitzgerald 2002, Hirschheim & Newman 1991), contradictory stakeholder values, and chance occurrence (Truex, Baskerville & Travis 2000). From this perspective, since there is no single universally applicable method that fits all types of projects (Iivari 1991, Brinkkemper 1996) developers often prefer to adapt and combine different practices based on the project characteristics instead of following one specific method (Baskerville & Pries-Heje 2004, Conboy & Fitzgerald 2010, Boehm 2002, Iivari & Iivari 2011, Wynekoop & Russo 1997). Two famous approaches stemming from this viewpoint are *contingent system development* (Brinkkemper 1996), and *method engineering* (Brinkkemper 1996, Kumar & Welke 1992).

Finally, *amethodical software development* (Truex, Baskerville & Travis 2000) is the third perspective which can be viewed as an extreme case of the second perspective. Advocates of the amethodical perspective suggest that each software development project is so unique and unpredictable that its development approach evolves over time as "an outcome of myriad development activities that emerge more or less independently" (Truex, Baskerville & Travis 2000, p.63). Therefore, this development approach cannot be viewed as a predefined and controllable sequence of steps and activities (Truex, Baskerville & Travis 2000, Baskerville & Pries-Heje 2004). However, it is mentioned in (Truex, Baskerville & Travis 2000) that while amethodical development rejects a predefined sequence and structure, it "does not imply anarchy or chaos"(Truex, Baskerville & Travis 2000, p.54).

Despite their differences, available software development methods share some core development activities, such as requirements identification, implementation, and testing (Shirado et al. 1996). Even if development teams do not use any specific method or explicit process, these core activities are necessary for developing software (Zhang & Lyytinen 2001). Therefore, it can be said that all software development projects clearly apply some sort of structure (Shirado et al. 1996) or process (Hohmann 1997), although this process may not be defined in the same way that the methods have been defined. What remains in dispute is how these activities should be performed and documented.

5.3.2 Customizing Software Development Methods

A number of previous studies explain method customization from a business perspective and as a result of organizational-level decisions for increasing the leanness of software development processes (Baskerville & Pries-Heje 2004, Fitzgerald, Hartnett & Conboy 2006, Sommerville 2005, Boehm 2002, Baskerville et al. 2001, Lim, Taksande & Seaman 2012, Lindgren et al. 2008). The metaphor of *technical debt* (Cunningham 1992) has been employed by the SE community to point out to such organizational level trade-offs (Tom, Aurum & Vidgen 2013). These studies suggest that certain elements of software methods are ignored because of organizational level trade-offs often made to increase productivity by reducing development costs and delivery times (Lim, Taksande & Seaman 2012, Lindgren et al. 2008, Tom, Aurum & Vidgen 2013). These studies assume that developers, as a result of such organizational trade-offs, take shortcuts and ignore best practices.

Another group of studies argue that while software methods fail to deliver the expected benefits, such as increasing productivity (Avison & Fitzgerald 2003), they are usually complex and costly to use, since following methods often slows down development processes (Fitzgerald, Hartnett & Conboy 2006) and requires expensive tools and high technical skills (Avison & Fitzgerald 2003). Additionally, because organizations lack the necessary knowledge for choosing appropriate methods, they end up choosing methods that do not suit their needs (Vavpotič & Vasilecas 2012). As a result, software development methods are ignored, since development teams consider such expensive and complex methods to be ineffective (Avison & Fitzgerald 2003, Fitzgerald 1998), especially in smaller projects with scarce resources (Fitzgerald 1998, Giardino et al. 2013). Additionally, if development teams consider a project to be simple or if the product is not being developed for a major client, it is likely that development methods are not followed to the letter, especially if the developers are more experienced (Fitzgerald, Hartnett & Conboy 2006).

Finally, software development is a dynamic phenomenon and almost always unexpected circumstances arise during software projects that are not reflected in the initial plans (Kiely & Fitzgerald 2002, Cugola & Ghezzi 1998). Therefore, development teams may decide to modify software processes to cope with such unexpected situations (Fitzgerald 1998, Kiely & Fitzgerald 2002, Cugola & Ghezzi 1998). Although it is widely acknowledged that developers frequently customize software development methods, we find no explanation in the literature clarifying why and under what conditions developers purposefully decide to follow or neglect different software development practices.

While the behaviors of developers are influenced by the organizational environment, software developers are human beings, and their actions are also influenced by their motivations and intentions. Therefore, they should not be viewed as obedient and subservient entities who blindly follow organizational-level decisions. In this study, therefore, we argue that investigating the rationale behind customizing software development methods is not possible unless both the behavior of developers and their development context are taken into consideration.

5.4 Research Methodology

In this study, we needed to employ a suitable research method to precisely identify and analyze the mechanisms underlying the customization of software methods while being able to explore contextual factors affecting these mechanisms. The Grounded Theory Method (GTM) is a suitable methodology for generating “conceptual theory that accounts for a pattern of behavior which is relevant and problematic for those involved” (Glaser 1978, p.93). Such a theory is generated through the systematic collection and analysis of empirical data based on the experiences of humans and the contextual factors associated with a phenomenon (Hoda, Noble & Marshall 2013, Glaser & Strauss 1967). In recent years, the GTM has been used increasingly in the fields of SE and IS (Giardino et al. 2013, Hoda, Noble & Marshall 2013, Birks et al. 2013, Urquhart & Fernandez 2013). It has been suggested that this interest is due to the effectiveness of the GTM to generate context-based and process-oriented explanations about socio-technical phenomena (Urquhart, Lehmann & Myers 2010, Myers 1997), such as software and information systems development.

Therefore, we decided to use the GTM (Glaser 1978, Glaser & Strauss 1967, Glaser & Holton 2004, Glaser 1992) to develop a process theory (Mohr 1982) that is grounded in the experience of software development professionals. Process theories (Mohr 1982) are suitable means for providing explanations of how emergent entities (e.g., software development processes) change and how events occur (e.g., ignoring certain software development activities and practices) over time (Van de Ven, A. H. & Poole 2005, Van de Ven, A. H. 1992, Ralph 2015a). With regard to "causality" (Markus & Robey 1988), what is necessary in process theories is that the precursor produces the outcome through a dynamic causal mechanism (Beach & Pedersen 2013). Therefore, a process theory requires a satisfactory causal explanation (Markus & Robey 1988) that reveals how a causal process composed of several necessary, albeit insufficient, interacting parts contributes to producing the outcome.

5.4.1 Data Collection

We collected different types of qualitative data (i.e., interviews, field notes, project reports, emails, and software procedures) during three years of field work in an industrially led research and development project. Primarily relying on qualitative interviews and following the guidelines suggested by (Myers & Newman 2007), we conducted 17 semi-structured interviews with international software professionals from six European companies. The interviewees held a variety of positions and had different levels of work experience, with an average of 11 years. Table 12 shows a summary of the interviewees and their development domain.

TABLE 12 A summary of the interviewees and their development contexts

Round	Interviewee	Position	Years of Experience	Domain	Type(s) of Development Projects
1st	Interviewee1	Software engineer	8	Commerce	Cloud-based solutions, web and mobile applications
	Interviewee2	Software designer	3	Healthcare	Web-based solutions and mobile applications
	Interviewee3	Team leader	6	Media	Enterprise software and web-based market analysis solutions
	Interviewee4	Software tester	9,5	Telecom	Mobile applications, web applications, enterprise and project management systems for energy sector
2nd	Interviewee5	Project manager	9	Automotive	Embedded systems including microchips and sensors interfaces
	Interviewee6	Project manager	22	Aerospace	Simulators, monitoring and control software

Round	Interviewee	Position	Years of Experience	Domain	Type(s) of Development Projects
3rd	Interviewee7	Team leader	7	Aerospace	Web-based knowledge management systems, semantic-based interfaces, and search engines
	Interviewee8	Software developer	6.5	Aerospace	Embedded flight systems
	Interviewee9	Software engineer	6	Aerospace	Software solutions in R&D projects
	Interviewee10	Team leader	8.5	Aerospace	Embedded software and systems, flight dynamics systems
	Interviewee11	Software developer	20	Aerospace	Software for defense and space industries
	Interviewee12	Team leader	11	Aerospace	Robotics R&D projects
	Interviewee13	Software engineer	8	Automotive	Embedded systems
	Interviewee14	Software engineer	14	Automotive	Embedded software
4th	Interviewee15	Software engineer	10	Aerospace	Avionics and embedded software
	Interviewee16	System engineer	8	Aerospace	Avionics and embedded software
	Interviewee17	Software engineer	21.5	Aerospace	Avionics systems

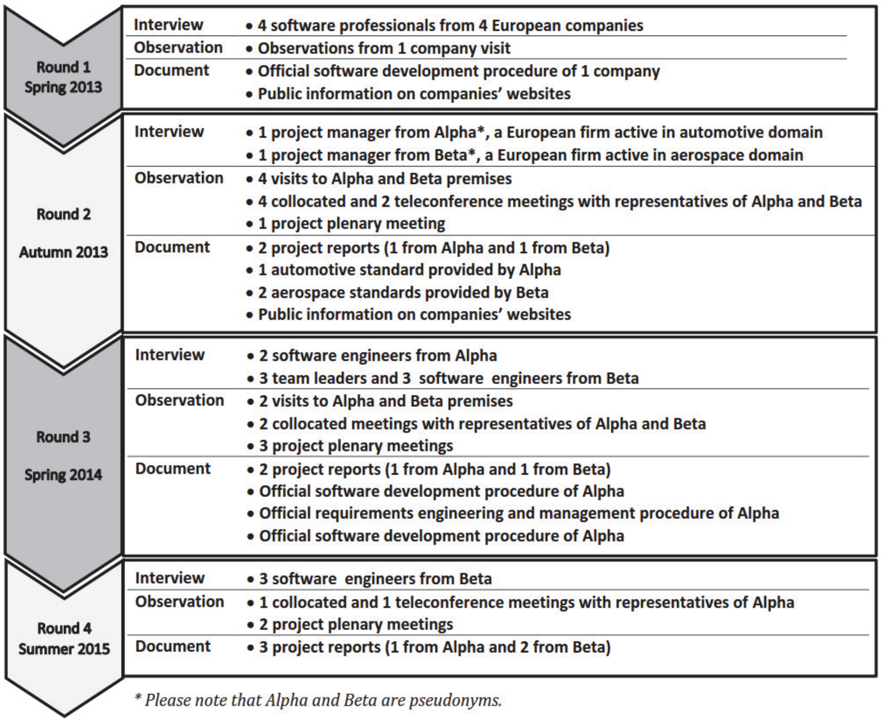
An interview protocol consisting of several high-level questions regarding the research problem was prepared during the first round of interviews and improved in each round of data collection. More-detailed questions were improvised during the interviews based on the answers provided and the terminologies used by the interviewees (Myers & Newman 2007). All of the interviews, which lasted from one to two hours, were recorded and transcribed. It must be noted that we asked for permission to record the interviews and assured the interviewees in written form that their identities and answers would be treated anonymously and would be accessible only to the research team.

We complemented the data collected from interviews with supplementary data sources, including software development procedures, standards, and project reports provided by the companies or publicly available on their websites. Additionally, because of our close collaboration with two companies (i.e., Alpha and Beta), from which we collected data in the 2nd, 3rd, and 4th rounds of interviews, we were able to closely observe and better understand their organizational and business environments. In particular, we prepared field notes during our four visits to these two companies and during regular face-to-face and teleconference meetings with company representatives throughout the project. By using these supplementary data sources, we were attempting to identify contextual factors or attributes that could be associated with developers' behavior during software development processes as well as instances of method customization.

We tried to make our data collection more productive by conducting the interviews in an informal, quiet, and comfortable environment (Myers & Newman 2007). Additionally, because of previous collaborations with some of the interviewees or their companies, we were able to maintain trust (Myers & Newman 2007) between the research team and the interviewees. Furthermore, because the interviewers had several years of work experience in the software industry, they were able to act as "cultural insiders" (Coleman & O'Connor 2007, Fitzgerald 1998) and were therefore able to overcome barriers in terms of communication and professional knowledge during the field work. Professional and scientific experience in software development also provided us with the necessary theoretical sensitivity (Glaser 1978, Glaser & Strauss 1967, Glaser & Holton 2004), which is the "ability to have theoretical insight into" the research problem to "conceptualize and formulate a theory as it emerges from the data" (Glaser & Strauss 1967, p.46). Due to the large amount of data collected during this study, we used the *QSR Nvivo* tool to properly store and manage the collected data and to perform data analysis.

5.4.2 Theoretical Sampling

A theoretical sampling strategy (Glaser & Strauss 1967, Glaser & Holton 2004) was followed during data collection to further our understanding of the nature of emerging theoretical concepts and to enrich these concepts (Birks et al. 2013). In particular, these data were collected in four rounds between 2013 and 2015 (see Figure 18).



Round 1 Spring 2013	Interview	• 4 software professionals from 4 European companies
	Observation	• Observations from 1 company visit
	Document	• Official software development procedure of 1 company • Public information on companies' websites
Round 2 Autumn 2013	Interview	• 1 project manager from Alpha*, a European firm active in automotive domain • 1 project manager from Beta*, a European firm active in aerospace domain
	Observation	• 4 visits to Alpha and Beta premises • 4 collocated and 2 teleconference meetings with representatives of Alpha and Beta • 1 project plenary meeting
	Document	• 2 project reports (1 from Alpha and 1 from Beta) • 1 automotive standard provided by Alpha • 2 aerospace standards provided by Beta • Public information on companies' websites
Round 3 Spring 2014	Interview	• 2 software engineers from Alpha • 3 team leaders and 3 software engineers from Beta
	Observation	• 2 visits to Alpha and Beta premises • 2 collocated meetings with representatives of Alpha and Beta • 3 project plenary meetings
	Document	• 2 project reports (1 from Alpha and 1 from Beta) • Official software development procedure of Alpha • Official requirements engineering and management procedure of Alpha • Official software development procedure of Alpha
Round 4 Summer 2015	Interview	• 3 software engineers from Beta
	Observation	• 1 collocated and 1 teleconference meetings with representatives of Alpha • 2 project plenary meetings
	Document	• 3 project reports (1 from Alpha and 2 from Beta)

** Please note that Alpha and Beta are pseudonyms.*

FIGURE 18 Different types of qualitative data collected in four rounds

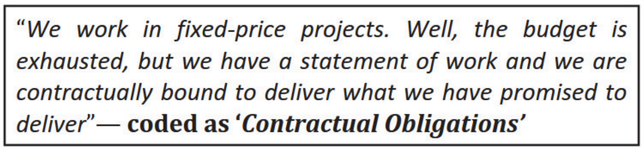
During the first round of data collection, we conducted four interviews with software professionals active in the e-commerce, healthcare, media, and telecommunications industries. Based on the analysis of these preliminary interviews, the characteristics of the software under development and contrasting stakeholders' concerns emerged as the key drivers influencing developers' decisions in customizing methods. Therefore, by focusing on these areas, we decided to conduct more interviews with developers active in diverse contexts. Thus, we conducted two interviews with project managers from two companies in the automotive and aerospace domains (i.e., Alpha and Beta) because the systems developed in these domains are critical and their failure might have devastating consequences in terms of finance, infrastructure, or human life (Sommerville 2015). After this round of data collection, we realized that even in critical domains, development teams must often accommodate contradictory concerns (e.g., increasing software quality while decreasing development time and costs); therefore, there is a difference in terms of how rigorously the methods are followed in different projects. Thus, we decided to extend the data collection within these two companies and to conduct an additional eight interviews during the 3rd round. To examine the differences between development contexts, we collected data from four teams (i.e., one team from Alpha and three teams from Beta) that were active in a variety of projects. Using data collected from these interviewees, we were able to enrich our theoretical concepts. However, we realized that further data collection could be beneficial to gain a better understanding of how quality control and assurance activities may influence the customization of methods. Therefore, we conducted three more interviews (i.e., 4th round) in Beta in which we mainly focused on quality management mechanisms and practices. After the 4th round of data collection, we were confident that theoretical saturation had been reached (Glaser & Strauss 1967, Glaser & Holton 2004).

As explained in this section, using a constant comparison technique (Glaser & Holton 2004, Glaser & Strauss 1967) through iterative data collection and analysis enabled us to enrich the emerging theoretical concepts by identifying and addressing shortcomings in the collected data.

5.4.3 Data Analysis

Next, we conducted a systematic data analysis process performing *open coding*, *selective coding*, and *theoretical coding* (Glaser 1978, Glaser & Holton 2004, Glaser 1992, Urquhart, Lehmann & Myers 2010). Additionally, during the data collection and analysis phase, the first author prepared a considerable number of memos to capture his thoughts and insights regarding the conceptual categories and their interrelationships (Glaser & Strauss 1967).

In the first stage of analysis, namely, open coding (Glaser 1978, Glaser & Holton 2004, Glaser 1992), data collected during the interviews were broken down into small pieces, and conceptual labels (i.e., code) were given to them. The purpose was to identify conceptually similar fractures of data that indicated instances where software development practices were ignored as well as the contextual factors that lead to such actions. Figure 19 shows an example of open codes produced during data analysis.



"We work in fixed-price projects. Well, the budget is exhausted, but we have a statement of work and we are contractually bound to deliver what we have promised to deliver"— **coded as '*Contractual Obligations*'**

FIGURE 19 An example of open codes produced from interview transcripts

The codes identified during the open coding stage were constantly compared with each other and assigned to different categories based on their similarities and relations. We tried to saturate these emergent preliminary conceptual categories by identifying all the relevant indicators of these categories from our data.

We moved to the second stage of data analysis, selective coding (Glaser 1978, Glaser & Holton 2004, Glaser 1992), as soon as we identified our core category called *Balancing Contradictory Contextual Forces*. During the selective coding stage, the preliminary conceptual constructs were further developed and refined through further data collection and coding only for those categories that were sufficiently related and unified around the core category (Glaser 1978, Glaser & Holton 2004). These categories include *Requirements Evolution*, *Maintaining Structural Balance*, *Maintaining Social Balance*, and *Loose Quality Management*. These conceptual constructs are the necessary parts (i.e., causal forces) of a process that leads to the customization of software development methods. An overall view of these necessary causal forces is shown in Figure 20.

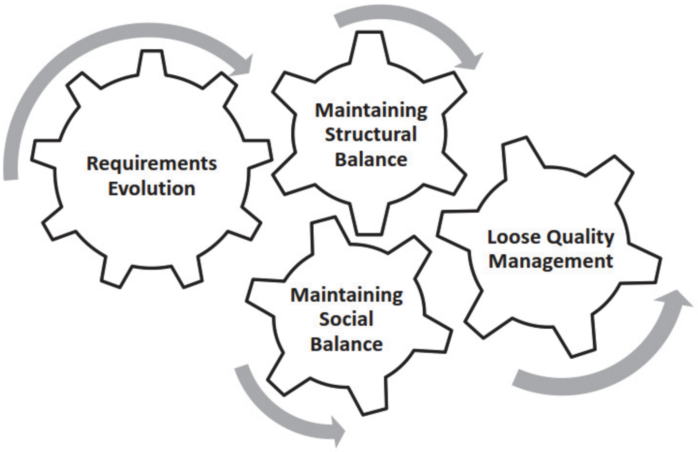


FIGURE 20 An overview of the causal forces contributing to the customization of software development methods

It is important to note that for software development practices to be ignored, all the necessary causal forces must function together. However, we still need to conceptualize how these causal forces are related and how they interact with each other.

Finally, during the third stage of data analysis, theoretical coding (Glaser 1978, Glaser & Holton 2004, Glaser 1992), our grounded theory was formed by conceptualizing the interrelationships between the necessary parts of the causal mechanism through which software development practices are ignored.

In this process theory, we propose that software development is influenced by the unique characteristics of software and the contrasting social and structural forces imposed by software stakeholders and market environments. Therefore, the software development context is stable as long as harmony exists between these contextual forces, and change is unavoidable whenever the power between these opposing forces falls out of balance. Development teams often decide to customize software processes to resolve such inconsistencies and to maintain balance between contradictory contextual forces. In the next section, we will discuss our theory in detail.

5.5 The mechanisms underlying the customization of methods

Despite differences in their domains and work experience, all of the interviewees indicated that software development methods are useful and should be utilized in practice. However, they suggested that depending on the development context, the characteristics of the software under development, and the needs of the development teams, these methods must be customized.

> *If you go by the book, sometimes things become much bureaucratic that there is no point of doing it. But of course you should always keep these main points in mind […] but sometimes it doesn't matter if you do something before or after." — Interviewee 3*

From our data, we identified a wide range of instances where software development methods and practices were ignored. Such instances include, but are not limited to, ignoring design and coding guidelines, skipping documentation and quality control activities, abandoning recommended tools, performing workarounds, and postponing external quality assurance activities. During the data analysis, we found that such changes in the software development processes are caused by different factors, such as the complexity of the software under development, the evolution of the requirements, technology advancement, and contrasting and sometimes contradictory stakeholders' expectations and market forces.

Software development can be seen as a dynamic socio-technical system consisting of different stakeholders in which the relations between entities are formed around a set of common goals and are structured according to organizational strategies and agreements between internal (e.g., stakeholders) and external parties (e.g., regulatory authorities). According to our data, the dispositional attributes of these entities, their structural arrangement, and their interactions lead to instability in the software development context, and as a result, changes in the software development process are unavoidable. Software stakeholders may have inconsistent personal goals and contradictory opinions about how these goals must be achieved. As a result, the perceptions of software stakeholders regarding the success of the project vary. While some stakeholders might consider faster delivery of functional software a success, others might measure success based on the quality of the software. Because of these varying goals and perceptions, which are sometimes contradictory, software stakeholders might have contrasting perspectives regarding the necessity of following software development processes. On the other hand, software is evolving and being developing rather than being produced in the same way as other tangible products are produced. Therefore, software development processes are dynamic and often subject to change; as a result, software development steps and their outcomes cannot be universally predefined.

Over time, software requirements evolve, and therefore, software firms have to identify possible solutions to address these changes while fulfilling their contractual obligations to deliver the software within an agreed-upon time and budget. Additionally, from a business perspective, managers are motivated to reduce development costs and delivery times in order to increase profit and market share. In either of these situations, firms may decide to maximize the leanness of software development processes by ignoring or minimizing the extent of certain software development activities.

In response to unexpected issues and evolving requirements as well as organizational-level decisions that are made to maintain strategic balance, developers are forced to perform additional tasks and activities that might not be planned. In such situations, developers might decide to perform workarounds by which they have to modify or ignore certain software development practices. However, several interviewees mentioned that they were not comfortable with performing such quality-compromising workarounds. Therefore, it can be said that due to contradictions between contextual forces and developers' personal concerns, they experience inconsistent social situations. Our data show that developers try to resolve such inconsistent situations either by identifying alternative solutions to perform their tasks while avoiding quality-compromising practices or by justifying such compromises for themselves.

Although maintaining balance between contradictory structural and social concerns is a key driver for customizing methods, a lack of proper quality control and assurance mechanisms makes it possible for development teams to ignore software development practices. Controlling the quality of software products and processes is a very challenging task, often requiring a sufficient amount of resources and technical skills. Therefore, in cases where proper quality management mechanisms are not in place or not rigorously followed, it is very difficult for software stakeholders to identify compromises in software development processes. Figure 21 represents our suggested theory.

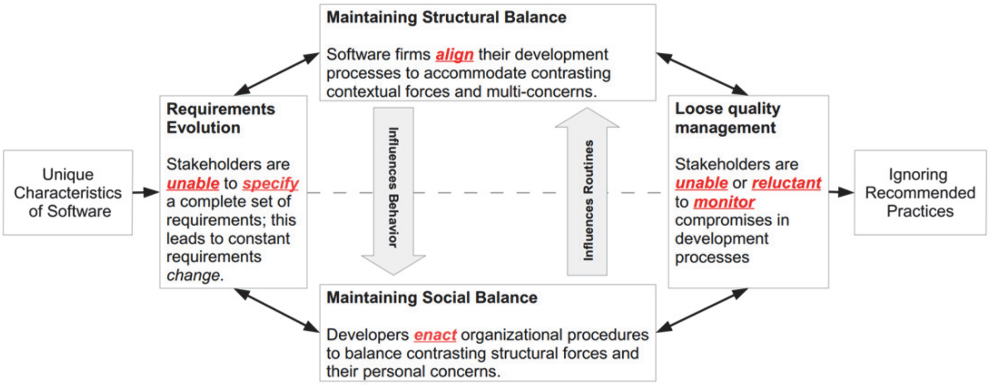

Maintaining Structural Balance
Software firms align their development processes to accommodate contrasting contextual forces and multi-concerns.
Requirements Evolution
Stakeholders are unable to specify a complete set of requirements; this leads to constant requirements change.
Loose quality management
Stakeholders are unable or reluctant to monitor compromises in development processes
Unique Characteristics of Software
Ignoring Recommended Practices
Influences Behavior
Influences Routines
Maintaining Social Balance
Developers enact organizational procedures to balance contrasting structural forces and their personal concerns.

FIGURE 21 The causal mechanisms underlying the customization of methods

In the following sections, we explain each part of the causal mechanisms through which software development processes are modified or ignored.

5.5.1 Unique Characteristics of Software

Software artifacts are different from other kinds of engineering products due to their unique characteristics, such as *intangibility, upgradability*, and *unpredictability*. We believe that ignoring software development methods becomes possible in the first place because of such unique characteristics of software artifacts.

Intangibility is one of the main characteristics making software a unique product. It is therefore difficult to measure and evaluate software in the same way as other traditional engineering artifacts, which are physical and tangible. Estimation of the necessary resources for developing software is therefore extremely difficult and, in some cases, even impossible. As a result, software estimation is highly performed based on individuals' subjective experiences, so these estimates are often imprecise or inaccurate. For example, one of our interviewees, a senior project manager, mentioned that:

> *"I've never seen anyone, not even a friend or whoever, in my professional network who has ever managed to associate a cost in Euros or in manpower to [an] individual requirement. It's always wild guesses; wrong guesses." – Interviewee 5*

Since the budget and resources are often agreed upon at the beginning of projects, such inaccurate estimates lead to scarce resources during the projects. This is even more problematic in projects where schedules and resources cannot be extended. As a result, firms may face financial losses if their initial estimates are not accurate.

On the other hand, intangibility makes software evaluation challenging. Therefore, it is difficult to identify software deficiencies, while the consequences of potential changes in software processes might not be immediately apparent. Customers and end-users often receive the final software products, and they are mainly able to interact with the software through its user interface. Therefore, it is not possible, or even necessary, for them to observe different elements of the software or how these components are developed. Thus, it was observed that evaluating the quality of software development processes is nearly impossible for non-technical stakeholders.

> *"Everyone can say something about GUI but maybe not about the performance because they don't know for instance, CPU or RAM consumption" – Interviewee 11*

Even in the case of professional stakeholders, it is usually the final products delivered to customers that are evaluated, not the actual processes through which those software products were developed. One exception may be certain safety-critical projects in which a considerable amount of resources are spent and every development phase and its deliverables are audited by regulatory authorities. Thus, identifying deviations during the development process is extremely difficult, expensive, and, in some cases, even impossible.

Another characteristic that makes software unique is its upgradability. Upgrading software products, during and after delivery, is often easier and cheaper than other engineering artifacts. Thus, software development teams are able to modify and update software products even after delivery. Such opportunity might motivate firms to postpone the delivery of certain features or the implementation of certain activities (e.g., quality control) to the later stages of the projects or even, in many cases, after the software is delivered. For example, development teams may decide to deliver defective software rather than fail to meet a deadline and solve the problems in future releases.

> *"Sometimes, you have to release that feature. So you deliver the feature and you buy some time. Your customer is busy with that feature for a couple of days, and then you go and test it and you release an update to fix those bugs." – Interviewee 4*

Especially with the emergence of web- and cloud-based technologies, upgrading the software has become easier and less apparent to end-users. However, in other engineering artifacts, due to their physical nature, the delivery of defective products and fixing post-delivery defects might be very expensive or even impossible. This also applies to certain critical systems in which software is embedded inside the hardware and upgrading becomes extremely difficult.

Unpredictability is another characteristic that differentiates software from other engineering artifacts. Since the behavior of the software is unpredictable, it is difficult for development teams to predict how complex the software will be at the end of the projects and how it will behave when it is ready.

> *"You deliver a very complex system and then bugs start popping up from everywhere, then you have to dig in and pinpoint where the bugs are coming from, there you lose time." – Interviewee 6*

This becomes even more challenging when there are a variety of software modules and subsystems being developed in parallel and in many cases by different teams, not to mention that at the end of the projects, these modules must be integrated.

> *"The typical thing [is that] everything is working fine when you develop individual components; then you put them all together and then the system is too slow suddenly." – Interviewee 10*

Therefore, at the time of integration, developers must fix any unforeseen issues, often within a limited time frame. Therefore, they may perform quick fixes simply to make the software function and possibly perform refactoring in the future to solve those problems.

The above-mentioned characteristics demonstrate that software is a unique product, and software development is a highly complex, unpredictable, and dynamic process that cannot be expected to adhere to a fixed, predefined plan.

5.5.2 Requirements Evolution

Understanding customers' expectations and specifying software requirements at the beginning of the projects is often very challenging. At the beginning of a project, when the software under development is not mature, development teams might not have sufficient product knowledge to prepare a complete, clear, and precise set of software requirements. Thus, as mentioned by the majority of our interviewees, the initial set of software requirements must be updated constantly to accommodate unforeseen issues and needs.

> *"Everything is very dynamic over three or four years of the project, and it changes so much that everything [that] you would have consolidated in the first six or nine months will, almost certainly, iterate very quickly." – Interviewee 12*

During the later stages of the project, when the software is being developed, the requirements become clearer, making changes to the initial requirements inevitable. Requirements evolve over time as software developers face unexpected events or issues with the artifact under development. Additionally, it is common for customers to ask for new requirements during a project. To accommodate these changes, developers need to perform extra work and sometimes switch between tasks in response to high-priority requests.

> *"It happens that we are at the middle of coding, and then the requirements specifier says that the requirements have to be changed, and then everything must change accordingly." – Interviewee 1*

Because the project costs and timetable are agreed upon based on initial estimates, changes in the requirements must be accommodated in contracts; otherwise the changes will lead to project constraints, particularly during the later stages of software development. In some projects, negotiations and re-planning might be possible, but in many projects with fixed budgets and schedules, changing the contractual terms is not possible.

> *"Once this deadline is there, it's really fixed and you cannot move it anymore." – Interviewee 6*

Therefore, as requirements evolve, the initial plans need to be changed, and development teams need to be able to accommodate these changes within their budget and schedule.

> *"Usually, we don't get more resources, so it's mostly a discussion of which features get the highest priority, and some of them get dropped." – Interviewee 14*

As can be seen from the above excerpt, one option is to cut corners and ignore those features or activities that can be postponed to later phases or that can be considered unnecessary for releasing the software.

5.5.3 Maintaining Structural Balance

While requirements evolution makes software development dynamic, the contrasting and sometimes contradictory concerns of stakeholders and contextual forces make it highly complex. With the complex and dynamic nature of software development in mind, it might be easier to understand the contradictory forces software firms are faced with. For instance, software firms are often faced with the dilemma of either increasing productivity or improving quality. While software companies are willing to reduce development times and costs and increase sales and revenue, they must also be able to accommodate extra workloads and costs associated with the evolution of requirements. On the other hand, because the quality of the software is important for increasing customer satisfaction and retaining market share, or because it may be demanded by regulatory authorities (Notander, Höst & Runeson 2013), software development teams must spend more time and effort improving the quality of the software. Therefore, software firms must be able to balance such contradictory concerns and other contextual forces such as resource constraints and competitive market environments.

Based on their business domain and market environment, organizations might choose different strategies to maintain balance that influences their software development approaches. For instance, in fast-moving environments, the focus is on cost-effective and quick delivery of new and important features, and therefore, firms will follow leaner processes. On the other hand, in critical domains, where the quality and reliability of software products have significant importance (Sommerville 2015), it is more likely that firms will utilize plan-driven approaches to perform extensive development and quality assurance activities (Boehm 2002).

Despite the importance of the development domain with respect to the application of different methods, the objectives and characteristics of a certain project also affect the extent to which methods are followed. Our data show that even companies active in critical domains are sometimes faced with resource constraints or market demand to reduce development costs and time. On the other hand, a lack of quality in developing non-critical software, such as hedonic mobile applications, might cause user dissatisfaction and eventually lead to a loss of customers.

> *"We might be able to stay within budget and time by dropping, for example, testing, but if some problem comes up, it'll have the potential of bringing your reliability down in the customer's eyes. If you have a malfunctioning mobile app, you might immediately lose your customers, and they might go to your competitors." – Interviewee 1*

To deal with such contrasting forces, organizations often combine different software development and quality control practices or even the principles of different software development approaches. Some organizations might prepare clear in-house software procedures, while others prefer to apply available software development approaches in response to their needs and market demands. For instance, Alpha is active in the automotive domain, and therefore, they have to comply with a set of standards that are required by the regulatory authorities. On the other hand, this firm is subject to market pressure to reduce development time and costs. Therefore, it is common for software development teams to work with scarce resources. In response to such contradictory forces, the firm has decided to build an in-house software procedure based on automotive standards while utilizing agile principles and practices to bring flexibility to plan-driven processes that are recommended by the automotive standards.

> *"In the software group, we try to break the waterfall a little bit into small waterfalls and make it more iterative and [follow] some test-driven design as well." – Interviewee 6*

Our data reveal that some of the organizations, especially the more-mature ones, may go even further than combining elements from different approaches by strategically restructuring their teams while at the same time creating procedures to clearly indicate the development approaches and boundaries within which those approaches must be applied. Thus, these organizations have been able to create flexible structural units (e.g., teams or projects) within which software developers are able to autonomously enact procedures while complying with structural rules and principles. In other words, developers have enough freedom to utilize their creativity and skills to perform recommended practices as needed as opposed to abandoning discipline altogether or slavishly following procedures like robots. For instance, Beta, which is active in the aerospace domain, has to comply with aerospace standards and regulations. However, the company is also active in a wide range of innovative research and development projects within the aerospace domain; these projects often have a fixed budget and high rates of change. Therefore, in this type of project, teams must utilize flexible and cost-effective development approaches. To accommodate such contrasting forces, the company is divided into several development teams with various missions, and an in-house software procedure is prepared based on the aerospace standards to guide all these teams. However, based on their needs and project characteristics, teams have the ability to customize this method. Thus, while the key principles of the aerospace standards and the organizational strategies are taken into account, teams have the autonomy to use different technologies while following different versions of the firm's official procedure.

As demonstrated in this section, performing or ignoring software development practices can be seen as a strategic decision to maintain balance between contradictory contextual forces imposed by stakeholders' concerns, market environments, domain-specific regulations, organizational structures, and project characteristics. If an organization has no clear strategy to maintain such balance, their software procedures are most likely not flexible enough to accommodate such contradictory concerns. As a result, when faced with unexpected contextual inconsistencies, developers might be forced to take shortcuts and perform workarounds on the fly, which may be difficult to trace and fix.

5.5.4 Maintaining Social Balance

In addition to structural inconsistencies, software developers experience social inconsistencies. Previous studies show that software developers like to produce high-quality software (Lim, Taksande & Seaman 2012, Yang, Hu & Jia 2008) they can be proud of (Katz 2005, Peters 2014). Additionally, developers are the ones who deal with software artifacts on a daily basis. Thus, if there are deficiencies in software artifacts, not only are developers responsible for these defects, but they must also spend additional time and effort to fix them. Therefore, it can be expected that developers prefer to perform their tasks properly in order to produce high-quality software and avoid future difficulties.

> *"In my opinion, it's always best to keep good quality in what we do and skip a part of the work instead of doing everything but with bad quality." – Interviewee 7*

However, in practice, developers experience situations in which they are forced to deviate from the original plans and ignore recommended best practices. Although individual developers have preferences on how to perform practices and tasks, sometimes they cannot act in accordance with their personal desires, as they belong to their structural unit (i.e., team or firm) and their decisions are influenced by constraints and opportunities within the unit.

> *"Me, as an engineer, I also would like to strive for excellence, but on the other side, we have to also look at the budget." – Interviewee 8*

In cases where contextual factors force developers to ignore certain tasks or practices they prefer to perform, they may face the dilemma of performing or ignoring recommended practices. In other words, they must make trade-offs between acting based on their personal desires or based on structural forces.

> *"I prefer to follow what is defined in the project […] for instance, all the unit tests and integration tests. I prefer, but sometimes it's not about preferences." – Interviewee 11*

When faced with such socially inconsistent situations, individuals have a tendency to resolve the tension by maintaining balance between contradictory forces (Heider 1946, Heider 1958, Heider 1967, Cartwright & Harary 1956). According to our data, depending on the situation, developers choose different strategies for maintaining social balance within their context. They may simply ignore best practices or, if this is not personally acceptable to them, they may try to identify an alternative solution to accommodate both concerns. Some of our interviewees mentioned that in some cases, when they are under pressure to deliver product within a limited time frame, despite having the opportunity to cut corners, they might decide to stay at work longer or ask their teammates for help to perform their task as they desire.

Similar to any other decision making processes, developers search within their behavioral space (i.e., implicit and explicit knowledge) to identify available alternatives to perform their tasks. Naturally, developers might choose a solution that was learned previously, suggested by their colleagues, or recommended in organizational archives or even on the internet. Since developers belong to organizational units, the solution to be selected greatly depends on other stakeholders' viewpoints and organizational routines. If other teammates, especially superior ones, do not consider certain practices important, or if ignoring them is common within a certain context, developers might choose to ignore these practices even if they know it is a questionable behavior.

> *"Our manager was saying just let it go, and we will improve the code and fix the bugs during the next release. We were saying it shouldn't be like this [because] the code will be messy. Unfortunately, these next days never came; we were never going back to fix the problem, except that the mess was coming up." – Interviewee 2*

If such "let it go" and "fix it later" attitudes become common within a given context, ignoring practices becomes legitimate from the unit's perspective, and it becomes acceptable for individuals as well.

Our data indicate that if developers are apprehensive about ignoring practices, they may justify their actions by, for example, blaming external forces or diffusing the responsibility for their actions to others. For instance, in some cases when managers suggest skipping certain activities to speed up the delivery of certain features, developers blame managers for not being able to perform their job well. Alternatively, as can be seen from the following excerpt, in some cases where developers are under pressure to perform a task in an unachievable timeframe, they try to delegate the responsibility of making decisions onto their teammates.

> *"I put my effort, but sometimes it's not achievable. Before coming closer to deadline, I talk to our manager, and he finds a way; he should find a way; his responsibility is to manage this kind of problems." – Interviewee 11*

Our data also reveal that in maintaining social balance, developers with higher levels of experience and decision making power rely more on their personal experience and intentions, while junior developers' decisions are more influenced by organizational guidelines and norms and the potential consequences of their actions. In other words, the more experienced the developers become, the greater is the chance that they will make decisions based on their personal judgment as opposed to organizational guidelines. This can be seen from the following statement made by a senior project manager:

> *"The rules that are imposed to me were my best practices five years ago; these are obsolete for me; that for me is ridiculous; it would be a regression for me to comply with certain rules of the company." – Interviewee 5*

Consequently, in the case of more-experienced developers, if they have a tendency to ignore methods and recommended best practices, there is a higher chance that they will decide to do so.

5.5.5 Interaction between Structural and Social Balance

The mutual reinforcement of maintaining structural balance and social balance is a central driver of the customization of software development methods. The behavior of developers is shaped by organizational strategies and routines, and their perception of the benefits of following software development methods will be adversely affected if doing so is viewed as non-critical in their context. In such a situation, if from an organizational perspective ignoring certain practices or activities is viewed as beneficial for solving contextual inconsistencies (i.e., structural balance), it is likely that developers will ignore these practices in order to maintain social balance.

On the other hand, due to constant social interaction among team members they may form a common understanding of the development context as well as similar perceptions of the necessity of following methods over time. If team members, especially more-senior ones, behave in certain ways, junior developers will most likely view their behavior as valuable and imitate it. In particular, the behavior of superiors influences the norms of the team because senior members often have the responsibility to provide guidance or to make critical decisions.

> *"My time is more dedicated to other things, but I can still help new guys [with] the selection of the proper way, among the different palette of possibilities and explain [to] them why this or that is a good idea or a better idea." – Interviewee 5*

Additionally, due to close collaboration with other stakeholders, developers might form stronger social ties with other stakeholders, and their concerns and actions might become more similar overtime. As a result, identifying commonly acceptable solutions that accommodate multiple concerns becomes easier. For instance, close collaboration between stakeholders through iterative planning and review meetings in agile projects not only enables developers to respond to requirements changes by re-prioritizing backlogs, but it also makes it possible for them to receive quick feedback about their actions and plan future steps accordingly. Such information will become part of the organizational behavioral space and be accessible to developers in the future. Enacting such elements of experience may lead to changes in the norms and routines within the software development context. Additionally, since more-senior members are often involved in developing software development guidelines, they influence organizational norms and official guidelines based on their personal experience and preferences for certain methods and best practices.

5.5.6 Loose Quality Management

In software development, like any other production process, deviations from plans become possible in the absence of proper control mechanisms. However, a lack of proper knowledge and control mechanisms makes it more difficult for software stakeholders to identify deviations in software development processes as well as their potential consequences. In software development, such control is maintained through quality management, which consists of quality assurance and quality control activities (Bourque & Fairley 2014).

While quality control is performed to assess the level of quality built into software artifacts, quality assurance is needed to evaluate the extent to which quality practices and recommendations have been followed in developing these artifacts (Bourque & Fairley 2014). In some domains, such as automotive and aerospace, there are clear regulations and standards providing the quality assurance recommendations that firms are expected to comply with. However, in other domains, there may be no clear definition of quality practices or no external demands to comply with quality practices. Since in software projects quality control is mainly achieved by performing a variety of tests, if there are no proper quality assurance practices, it becomes possible for development teams to ignore quality control activities.

> *"We would like to have a full review of our documentation and the code before we release. Currently, this is not always the case, mainly due to time constraints." – Interviewee 6*

If developers intentionally ignore recommended practices for personal reasons, since their actions and their consequences cannot be easily observed by other stakeholders, they have no fear of exposure. This becomes more problematic if firms consider stakeholders' inability to identify software defects or a lack of proper quality assurance mechanisms as an advantage to strategically ignore certain practices, making it is easier for software developers to neglect recommended practices.

> *"There are coding standards that we are supposed to comply with. I'm saying 'supposed to' because who really verifies? From my own knowledge nobody; Now, if we talk about more critical software, not only the coding standard [is] defined but [also] verified." – Interviewee 5*

However, if developers know that their questionable practices could be easily observed by other stakeholders, they would most likely avoid ignoring recommended practices, as they do not want to be blamed for inferior work and failure. It is apparent that such avoidance is especially vital in critical domains.

> *"If you leave some defect and you don't report it or you pass some test without making it work properly, anyway it 'll come back later to you because there is an audit trial, and they can say why you didn't do this." – Interviewee 3*

As discussed earlier, regardless of the context, in every domain there might still be certain rules and constraints that restrict developers' ability to perform or ignore recommended practices. Thus, in some domains, ignoring best practices might be more difficult and may have more severe consequences; as a result, software developers may pay more attention to quality practices.

5.5.7 Ignoring Software Development Practices

Ignoring certain steps or practices in software development processes becomes possible when the abovementioned parts of the causal process function together. It is important to remember that these parts are necessary but not sufficient (Markus & Robey 1988, Beach & Pedersen 2013) for the outcome to occur.

Our data show that in safety-critical projects within Alpha and Beta, customers are often highly knowledgeable domain experts who are able to clearly explain and provide details about their needs and expectations as well as the expected methods and standards to be followed. Therefore, in these projects, development teams might be able to specify a clear and complete set of requirements and necessary resources from the outset. Since the requirements are largely fixed and software development teams are faced with less resource constraints or external market pressure, they are able to better follow development activities as planned. Additionally, proper quality management mechanisms are utilized in such projects, and therefore, any unintentional or intentional deviations from the original plans can be identified and fixed before software delivery. Therefore, in such safety-critical settings, the possibility that best practices will be ignored is farfetched.

An opposite case can be projects in software startups. Often software startups are active in highly competitive and turbulent market environments where customers' requirements are vague, resources are often scarce, competition is fierce, and there are no external pressures to comply with quality standards. In such an environment, ignoring best practices is highly probable and may even be vital for the survival of the company.

5.6 Discussion

In the previous section, we explained our process theory, which is grounded in empirical data collected from different development contexts. In this section, we first discuss how our findings stack up against previous studies as well as our contributions to theory and research. Then, we outline the implications for practice.

5.6.1 New Theoretical Contributions and Related Work

The main theoretical contribution of this study is a novel theory of Software Development Balance. While the identification of novel software development methods has been the focus of previous research, there has been a lack of empirically grounded theories explaining why these methods have been ignored in practice.

Our study is the first to provide a mechanistic explanation of the causal forces leading to the customization of software development methods in practice. This process, which is initiated by the unique characteristics of software, indicates how software development teams customize methods for balancing contrasting and sometimes contradictory forces and concerns within their context.

In our grounded theory, we propose that the evolution of requirements is one of the main causal forces contributing to the customization of software development processes. Previous studies have indicated that software projects are subject to constant requirements changes due to turbulence in the business environment (Mathiassen et al. 2007, Highsmith & Cockburn 2001), ambiguous customer requirements (Highsmith & Cockburn 2001, Mathiassen et al. 2007, Ghanbari, Similä & Markkula 2015), and a lack of understanding of system functionalities (Highsmith & Cockburn 2001, Ghanbari, Similä & Markkula 2015).

In addition to requirements evolution, software firms must often deal with contrasting and contradictory concerns either caused by differences between stakeholders' needs and viewpoints (Truex, Baskerville & Travis 2000, Tsumaki

& Tamai 2005) or external forces such as market demands (Baskerville & Pries-Heje 2004, Lindgren et al. 2008). To accommodate these contradictory forces in a cost-effective way, organizations must follow more-flexible and leaner software development approaches (Highsmith & Cockburn 2001, Baskerville et al. 2003). A group of studies explains how software firms might increase the leanness of software development methods by ignoring certain development practices (Baskerville & Pries-Heje 2004, Fitzgerald, Hartnett & Conboy 2006, Boehm 2002, Baskerville et al. 2001, Lim, Taksande & Seaman 2012, Lindgren et al. 2008, Tom, Aurum & Vidgen 2013). Most of the previous studies assume that method customization is the result of such organizational-level decisions made for gaining short-term benefits such as faster delivery times and lower development costs (Baskerville & Pries-Heje 2004, Baskerville et al. 2001, Fitzgerald, Hartnett & Conboy 2006, Boehm 2002, Lim, Taksande & Seaman 2012, Lindgren et al. 2008, Cunningham 1992, Tom, Aurum & Vidgen 2013).

In this study, however, we suggest that software methods should not be customized randomly but strategically and by considering the organizational structure and contextual characteristics of software firms. This is because certain software development practices and activities are "indispensable" (Zhang & Lyytinen 2001), and ignoring them will compromise the quality of software artifacts. For instance, using agile methods has been widely misinterpreted as performing development activities without heeding any rules (Conboy & Fitzgerald 2010). However, advocates of agile methods dismiss such claims and state that using agile methods requires greater discipline (Conboy & Fitzgerald 2010, Schwaber & Beedle 2001, Beck & Boehm 2003, Dyba 2000), and since using agile methods "is not an excuse for unilateral behavior" (Beck & Boehm 2003, p.44), agile teams should not "abandon discipline altogether" (Dyba 2000, p.83).

The concept of ambidexterity has been used in the field of organizational sciences to explain such strategic approaches for dealing with conflicting demands (Gupta, Smith & Shalley 2006, O Reilly & Tushman 2004, Tushman & O'Reilly 1996, Gibson & Birkinshaw 2004). Recently, several studies have suggested that using ambidextrous strategies might be suitable for addressing contradictory demands in software development projects (Katz 2005, Dyba 2000, Ramesh, Mohan & Cao 2012, Lee, DeLone & Espinosa 2006, Vinekar, Slinkman & Nerur 2006). Based on such a perspective, combining agile and plan-driven methods can be seen as a strategic solution for responding to requirements changes while producing high-quality software (Boehm & Turner 2003, McCaffery, Pikkarainen & Richardson 2008, Ghanbari 2016). However, our study is unique in considering the role of developers in implementing such organizational-level ambidextrous strategies, an issue that has been widely overlooked by previous studies.

Our study extends the previous literature by proposing that organizational-level decisions to customize software processes often force developers to take quality-compromising shortcuts, which might not be appreciated from a technical perspective (Lim, Taksande & Seaman 2012, Lindgren et al. 2008, Katz 2005, Austin 2001). Drawing upon the concept of balance from the field of psychology (Heider 1946, Heider 1967, Heider 1958, Cartwright & Harary 1956), we argue that if developers are forced by structural arrangements to ignore certain practices, they might experience social inconsistencies that would motivate them to engage in mental or physical actions to resolve such tensions.

Our study is the first to suggest that in such inconsistent situations, developers will choose different strategies based on their personal experience and organizational routines in order to perform tasks while at the same time balancing personal values with organizational demands. We also showed that developers with higher levels of experience and decision making power rely more on their personal judgment while ignoring methods. Therefore, if senior developers are personally motivated to ignore recommended best practices, their decision and action is less influenced by structural constraints. This may explain the findings of previous studies (Fitzgerald 1998, Potdar & Shihab 2014) suggesting that more- experienced developers tend to ignore methods to a greater extent.

Finally, in this study, we suggest that ignoring software development methods becomes possible due to improper quality management mechanisms. This observation is in line with results reported by previous studies (Baskerville & Pries-Heje 2004, Austin 2001, Wang & Zhang 2010, Ahonen & Junttila 2003), which suggest that lack of attention to quality control and assurance activities is one of the main reasons software developers engage in quality-compromising workarounds and shortcuts.

Our findings highlight the importance of individual-level preferences and decisions as well as organizational-level decisions in balancing contrasting and sometimes contradictory contextual forces. Future field studies are needed to reveal and provide in-depth explanations of processes through which structural and social balance is maintained across development contexts. In doing so, our proposed theory can be used as a lens for observing and analyzing software development processes in different contexts so that future researchers will understand how software development methods are customized under the influence of different social and organizational settings. Consequently, future research will be able to articulate how software firms must maintain balance between contradictory context-specific forces by clarifying the conditions under which and the boundaries within which software development approaches must be followed.

5.6.2 Implications for Practice

Our theory emphasizes the key role of software developers in implementing organizational strategies and development approaches. Organizations should both define clear strategies for balancing contrasting contextual forces within software projects and understand how developers react to such organizational-level strategies. In doing so, organizations must identify and consider the behavioral patterns shown by software developers in certain inconsistent situations to better understand the socio-structural factors influencing developers' decisions to follow or ignore organizational procedures.

Such understanding enables firms to strategically define development approaches and recommend a set of consistent software development and quality management practices to be used within their contextual boundaries. Thus, firms are able to maintain balance between inconsistent contextual concerns as well as promote a professional culture in which software professionals are encouraged to utilize their creativity to achieve high-quality work within the contextual boundaries.

Finally, by taking into account that developers tend to restructure their social arrangement, organizations can form integrated teams suitable for different development contexts based on developers' personality, contextual concerns, and the collective skills of the team.

5.6.3 Limitations and Future Work

It must be noted that our study has some limitations that might affect the validity of the results. First, we primarily relied on interviews with software professionals. As a result, our results are influenced by the personal opinions and perceptions of interviewees, which of course may be different from reality. This is because interviewees may not have felt comfortable admitting that they or their companies ignore recommended best practices and potentially compromise software quality. We tried to compensate for this threat by collecting supplementary data such as project reports, software development procedures, and field notes that could support the interviewees' statements. Additionally, we tried to ensure our interviewees that any information provided by them would be treated confidentially and would be accessed only by researchers.

Another potential limitation of interviewing software professionals is that the outcomes could have been different if we had interviewed business personnel. Since we were aiming at capturing how software development methods are customized across domains, software professionals, who are responsible for implementing these methods on a daily basis, can provide a better perspective on the phenomenon than high-level managers. One might also argue that high-level managers are too far from the operational decisions of how to apply software development methods, and therefore, are not good candidates for this study. Team leaders, however, could be more-relevant informants. Therefore, we also interviewed team leaders and department managers, as they are more involved in organizational-level decision making and not solely in the technical parts of software development.

Finally, as mentioned earlier, software development is an extremely complex phenomenon that is affected by a variety of social, technical, and organizational factors. Therefore, it is very difficult to identify and consider every factor that influences software development processes and their outcomes. Therefore, in this study, we attempted to identify a set of necessary conditions and to provide a satisfactory mechanistic explanation of how software development methods are ignored in practice.

To address these limitations and to improve the generalizability of our results, further research is necessary. Although our substantive grounded theory provides an explanation of how and under what conditions software development methods are customized in practice, future studies are necessary to further improve our proposed theory. As suggested in (Glaser & Strauss 1967), since theory evolves as "an ever-developing entity, not as a perfected product", the theory presented in this study can be improved by further scientific investigation as well as practical application. In particular, there is a need for future empirical studies to conduct in-depth investigations on the mechanisms through which structural and social balance are maintained within different development contexts.

5.7 Conclusions

In this study, we used GTM in a longitudinal field study to build an empirically grounded theory. Our process theory reveals the causal mechanisms underlying the customization of software development methods, indicating how and under what conditions certain software development activities and practices are ignored. In particular, this theory explains how the process of software development customization is initiated by the unique characteristics of software, and how it progresses through the interactions among several causal forces influenced by inconsistent socio-structural settings.

In this study, we propose that development teams, depending on their context, decide to customize software development processes to maintain balance between contrasting or even contradictory forces imposed by stakeholders' concerns and contextual determinants. Therefore, we suggest that, if needed, software development methods should be customized strategically and according to the organizational structures of software firms and their development contexts.

6 SUMMARY

In this doctoral research, we attempted to advance the SE and IS research and practice by providing novel mechanism-based understandings of the customization of software development methods.

In doing so, we first identified and reported the shortcomings in the literature regarding this phenomenon. We then proposed a set of directions for future research, in four areas, to address these identified gaps. The first research area calls for in-depth empirical studies to investigate the contextual factors and conditions under which the customization of software development methods takes place. The second research area that requires further research is the psychological processes through which software professionals decide to perform such questionable practices. The third research area needs to study the long-term consequences of such questionable practices. Finally, the fourth research area must identify and suggest different interventions and solutions that could enable the software community to overcome the omission of quality practices.

In this doctoral research, we have attempted to contribute to the first and second research areas by conducting longitudinal field studies. In Study 2, we conducted a case study to investigate the customization of software development methods in a company active in a critical domain. The results of this case study show that customization of software development methods is not limited to highly turbulent and competitive business domains. Rather, the decision to ignore certain software development activities and practices is influenced by contextual factors including business-, organizational-, and project-level settings. In Study 2, we proposed a theoretical model that shows how the customization of software development methods is initiated by several challenges within the development context.

In Study 3, we conducted a field study across different development contexts (i.e., different domains and different organizational settings) and proposed a novel process theory to indicate the causal mechanisms underlying the customization of software development methods. In this theory, called the *Theory of Software Development Balance*, we indicate how methods are customized as a result of organizational- and individual-level mechanisms to maintain balance between contrasting or even contradictory forces imposed by stakeholders' concerns and contextual factors.

From an academic perspective, the proposed theory represents a deep theoretical understanding of developers' behavior in ignoring recommended practices and the consequences of such decisions. The proposed process theory also describes how software developers make quality-compromising decisions in order to deal with inconsistent forces in their work environments. This understanding assists researchers in analyzing the effects of different contextual determinants on developers' tendency towards following software development practices. In doing so, scholars are able to build new theories or to extend existing theories to explain the differences between software development processes across organizational and business environments.

Since "nothing is as practical as a good theory" (Lewin 1945), our suggested theory assists software practitioners in understanding the processes through which and the conditions under which developers make quality-compromising decisions and skip recommended practices and quality rules. Therefore, our proposed theory enables managers to identify and promote a set of suitable software development practices and quality control mechanisms based on their organizational settings and the characteristics of their development teams. At the same time, developers are able to better understand the reasons behind organizational-level trade-offs and to choose the best alternative for dealing with such trade-offs.

Finally, according to our suggested theory, we recommend that if it is necessary for a software firm to customize methods, they should do so strategically and by considering organizational and social structures within their context.

YHTEENVETO (FINNISH SUMMARY)

Viimeisen neljän vuosikymmenen aikana ohjelmistokehitys on ollut yksi ohjelmistotuotannon ja tietojenkäsittelytieteiden tärkeimmistä aiheista. Tuhansia menetelmiä, jotka tarjoavat ohjeita ohjelmistokehityksen prosesseihin, on ehdotettu. Näiden ohjelmistokehitysmenetelmien tavoite on systemaattinen korkealaatuisten ohjelmistojen tuottaminen. Kuitenkin on laajalti tunnettua, että näitä menetelmiä seurataan harvoin. Pikemminkin ohjelmistokehittäjät usein muuttavat tai ohittavat erilaisia vaiheita, käytäntöjä ja laatusääntöjä, joita suositellaan ohjelmistokehityksen menetelmissä. Joukko aiempia tutkimuksia viittaa siihen, että joustavuuden maksimoiminen ja kehitykseen käytetyn ajan ja resurssien vähentäminen ohjelmistokehityksen prosesseissa ovat keskeisiä tekijöitä näiden muokkauksien takana. Toinen ryhmä väittää, että näiden menetelmien riittämättömyys sidosryhmien odotusten täyttämisessä, on pääsyy siihen, että niitä räätälöidään käytännön työssä. Kuitenkaan tiedossa ei ole teorialähtöisiä ja empiirisesti todennettuja selityksiä valaisemaan syitä metodien räätälöinnin taustalla. Väitöskirjassa on yritetty ottaa ensimmäinen askel tämän tutkimusaukon täyttämiseen.

Tutkimuksen ensimmäisessä vaiheessa toteutettiin kattava kirjallisuuskatsaus määrittämään puutteita ohjelmistokehitysmenetelmien räätälöinnin tutkimuksessa ja selventämään tarpeita tutkimusaukkojen täyttämiselle. Toisessa vaiheessa kolmivuotisen kenttätutkimuksen kautta pyrittiin täyttämään joitain tunnistettuja puutteita. Keräämällä dataa useista ohjelmistoprojekteista eri teollisuuden aloilla ja käyttämällä Grounded Theory -tutkimusmenetelmää rakennettiin prosessiteoria nimeltä *Theory of Software Development Balance*. Teoria selittää ohjelmistokehitysmetodien räätälöinnin mekanismeja. Näiden mekanismien avulla toimintaympäristön vastakkaiset voimat pyritään säilyttämään tasapainossa.

REFERENCES

Abrahamsson, P., Warsta, J., Siponen, M. T. & Ronkainen, J. 2003. New Directions on Agile Methods: A Comparative Analysis. Proceedings of 25th International Conference on Software Engineering, 2003. IEEE, 244.

Agrawal, M. & Chari, K. 2007. Software effort, quality, and cycle time: A study of CMM level 5 projects. IEEE Transactions on Software Engineering 33 (3), 145-156.

Ahonen, J., J. & Junttila, T. 2003. A case study on quality-affecting problems in software engineering projects. IEEE International Conference on Software: Science, Technology and Engineering, 2003. SwSTE'03. , 145.

Alemzadeh, H., Iyer, R. K., Kalbarczyk, Z. & Raman, J. 2013. Analysis of safety-critical computer failures in medical devices. Security & Privacy, IEEE 11 (4), 14-26.

Austin, R., D. 2001. The effects of time pressure on quality in software development: An agency model. Information Systems Journal 12 (2), 195-207.

Avison, D. & Fitzgerald, G. 2003. Where Now for Development Methodologies? Communications of the ACM 46 (1), 79-82.

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

Bargh, J. A., Chen, M. & Burrows, L. 1996. Automaticity of social behavior: Direct effects of trait construct and stereotype activation on action. Journal of personality and social psychology 71 (2), 230.

Baskerville, R., Levine, L., Pries-Heje, J. & Slaughter, S. A. 2001. How Internet software companies negotiate quality. IEEE Computer 34 (5), 51-57.

Baskerville, R. & Pries-Heje, J. 2004. Short cycle time systems development. Information Systems Journal 14 (3), 237-264.

Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J. & Slaughter, S. A. 2003. Is Internet-Speed Software Development Different ? IEEE Software , 70-77.

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

Bayer, J. & Muthig, D. 2006. A view-based approach for improving software documentation practices. 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006. ECBS 2006. IEEE, 10 pp.

Beach, D. & Pedersen, R. B. 2013. Process-tracing methods: Foundations and guidelines. University of Michigan Press.

Beck, K. & Boehm, B. 2003. Agility through discipline: A debate. Computer (6), 44-46.

Beck, K. 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional.

Birks, D. F., Fernandez, W., Levina, N. & Nasirin, S. 2013. Grounded theory method in information systems research: its nature, diversity and opportunities. European Journal of Information Systems 22 (1), 1-8.

Black, P., E. 2012. Static Analyzers: Seat Belts for Your Code. IEEE Security and Privacy 10 (3), 48-52.

Boehm, B. W. 1984. Verifying and validating software requirements and design specifications. IEEE Software 1 (1), 75.

Boehm, B. 2002. Get Ready for Agile Methods, with Care. IEEE Computer 35 (1), 64-69.

Boehm, B. 1988. A Spiral Model of Software Development and Enhancement. IEEE Computer 21 (5), 61-72.

Boehm, B. & Turner, R. 2003. Observations on balancing discipline and agility. Proceedings of the Agile Development Conference, 2003.ADC 2003 , 32-39.

Bourque, P. & Fairley, R. E. 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press.

Brinkkemper, S. 1996. Method engineering: engineering of information systems development methods and tools. Information and Software Technology 38 (4), 275-280.

Brooks, F., P. 1995. The mythical man-month: Essays on software engineering. (Anniversary Edition edition) Addison-Wesley.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I. & Others 2010. Managing technical debt in software-reliant systems. Proceedings of the FSE/SDP workshop on Future of software engineering research. , 47.

Burton-Jones, A., Mclean, E. R. & Monod, E. 2011. On Approaches To Building Theories : Process, Variance and Systems. Sauder School of Business, UBC.

Çalikli, G. & Bener, A. B. 2013. Influence of confirmation biases of developers on software quality: an empirical study. Software Quality Journal 21 (2), 377-416.

Carmel, E. & Becker, S. 1995. A process model for packaged software development. IEEE Transactions on Engineering Management 42 (1), 50-61.

Cartwright, D. & Harary, F. 1956. Structural balance: a generalization of Heider's theory. Psychological review 63 (5), 277.

Codabux, Z. & Williams, B. 2013. Managing technical debt: An industrial case study. Proceedings of the 4th International Workshop on Managing Technical Debt. IEEE Press, 8.

Coleman, G. & O'Connor, R. 2007. Using grounded theory to understand software process improvement: A study of Irish software product companies. Information and Software Technology 49 (6), 654-667.

Conboy, K., Fitzgerald, G. & Mathiassen, L. 2012. Qualitative methods research in information systems: motivations, themes, and contributions. European Journal of Information Systems 21 (2), 113-118.

Conboy, K. & Fitzgerald, B. 2010. Method and developer characteristics for effective agile method tailoring. ACM Transactions on Software Engineering and Methodology 20 (1), 1-30.

Cugola, G. & Ghezzi, C. 1998. Software Processes: a Retrospective and a Path to the Future. Software Process: Improvement and Practice 4 (3), 101-123.

Cunningham, W. 1992. The WyCash portfolio management system, Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum). British Columbia, Canada , 29-30.

Cusumano, M., MacCormack, A., Kemerer, C., F. & Crandall, B. 2003. Software development worldwide: The state of the practice. IEEE Software 20 (6), 28-34.

Dalcher, D. 1999. Disaster in London. The LAS case study. Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop on. IEEE, 41.

Dyba, T. 2000. Improvisation in small software organizations. IEEE Software 17 (5), 82.

Eberlein, A. & Leite, Julio, Cesar, Sampaio, do Prado 2002. Agile Requirements Definition : A View from Requirements Engineering. Proceedings of the International Workshop on Time-Constrained Requirements Engineering (TCRE’02). , 4.

Eklund, A., Nichols, T. E. & Knutsson, H. 2016. Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. Proceedings of the National Academy of Sciences , 201602413.

Elssamadisy, A. & Schalliol, G. 2002. Recognizing and responding to bad smells in extreme programming. Proceedings of the 24th International conference on Software Engineering. ACM, 617.

European Cooperation for Space Standardization 2013. Space engineering: software. ECSS-E-ST-40C. Noordwijk, The Netherlands.

European Space Agency 2009. ECSS-E-ST-40C, SPACE ENGINEERING: SOFTWARE. ESA Requirements and Standards Division.

Fitzgerald, B. 1998. An empirical investigation into the adoption of systems development methodologies. Information & Management 34 (6), 317-328.

Fitzgerald, B., Hartnett, G. & Conboy, K. 2006. Customising agile methods to software practices at Intel Shannon. European Journal of Information Systems 15 (2), 200-213.

Fleming, R. 1999. A fresh perspective on old problems [software industry]. Software, IEEE 16 (1), 106-113.

Fonseca, J. & Vieira, M. 2008. Mapping Software Faults with Web Security Vulnerabilities. IFIP International Conference on Dependable Systems and Networks. IEEE, 257.

Fowler, M. & Highsmith, J. 2001. The agile manifesto. Software Development 9 (8), 28-35.

Fraser, S. & Mancl, D. 2008. No silver bullet: Software engineering reloaded. IEEE Software 25 (1), 91-94.

Garousi, V. & Fernandes, J. M. 2016. Highly-cited papers in software engineering: The top-100. Information and Software Technology 71, 108-128.

Ghanbari, H. 2016. Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study. 2016 49th Hawaii International Conference on System Sciences (HICSS). IEEE, 5407.

Ghanbari, H., Similä, J. & Markkula, J. 2015. Utilizing online serious games to facilitate distributed requirements elicitation. Journal of Systems and Software 109, 32-49.

Giardino, C., Paternoster, N., Unterkalmsteiner, M., Gorschek, T. & Abrahamsson, P. 2016. Software development in startup companies: The greenfield startup model. IEEE Transactions on Software Engineering 42 (6), 585-604.

Gibson, C. B. & Birkinshaw, J. 2004. The antecedents, consequences, and mediating role of organizational ambidexterity. Academy of management Journal 47 (2), 209-226.

Gibson, V., R. & Senn, J., A. 1989. System structure and software maintenance performance. Communications of the ACM 32 (3), 347-358.

Glaser, B. G. 1992. Basics of grounded theory: Emergence vs. forcing. Mill Valley, CA: Sociology Press.

Glaser, B. G. 1978. Theoretical sensitivity: Advances in the methodology of grounded theory. Sociology Press.

Glaser, B. G. & Holton, J. 2004. Remodeling grounded theory. Forum Qualitative Sozialforschung/Forum: Qualitative Social Research.

Glaser, B. G. & Strauss, A. 1967. The discovery grounded theory: strategies for qualitative inquiry. Chicago, U.S.: Aldine Transactions.

Gregor, S. & Jones, D. 2007. The Anatomy of a Design Theory. Journal of the Association for Information Systems 8 (5).

Gregor, S. 2006. The nature of theory in information systems. Mis Quarterly 30 (3), 611-642.

Guba, E. G. & Lincoln, Y. S. 1994. Competing paradigms in qualitative research. Handbook of qualitative research 2 (163-194), 105.

Gupta, A. K., Smith, K. G. & Shalley, C. E. 2006. The interplay between exploration and exploitation. Academy of management journal 49 (4), 693-706.

Heider, F. 1967. Attitudes and cognitive organization. Readings in attitude theory and measurement , 39-41.

Heider, F. 1958. The psychology of interpersonal relations. New Jersey: Lawrence Erlbaum Associates, Inc.

Heider, F. 1946. Attitudes and cognitive organization. The Journal of psychology 21 (1), 107-112.

Henderson-Sellers, B. & Serour, M. K. 2005. Creating a Dual-Agility Method : The Value of Method Engineering. Journal of Database Management 16 (4), 1-23.

Highsmith, J. & Cockburn, A. 2001. Agile Software Development : The Business of Innovation. IEEE Computer (September), 120-122.

Hirschheim, R., Klein, H. K. & Lyytinen, K. 1995. Information systems development and data modeling: conceptual and philosophical foundations. Cambridge University Press.

Hirschheim, R. & Newman, M. 1991. Symbolism and Information Systems Development: Myth, Metaphor and Magic. Information Systems Research 2 (1), 29-62.

Hoda, R., Noble, J. & Marshall, S. 2013. Self-organizing roles on agile software development teams. IEEE Transactions on Software Engineering 39 (3), 422-444.

Hohmann, L. 1997. In methods we trust? Computer 30 (10), 119-121.

Holvitie, J., Leppanen, V. & Hyrynsalmi, S. 2014. Technical Debt and the Effect of Agile Software Development Practices on It-An Industry Practitioner Survey. Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE, 35.

Huisman, M. & Iivari, J. 2006. Deployment of systems development methodologies: Perceptual congruence between IS managers and systems developers. Information & Management 43 (1), 29-49.

Hummon, N. P. & Doreian, P. 2003. Some dynamics of social balance processes: bringing Heider back into balance theory. Social Networks 25 (1), 17-49.

Iivari, J. 1990. Hierarchical spiral model for information system and software development. Part 1: theoretical background. Information and Software Technology 32 (6), 386-399.

Iivari, J., Hirschheim, R. & Klein, H. K. 1998a. A paradigmatic analysis contrasting information systems development approaches and methodologies. Information Systems Research 9 (2), 164-193.

Iivari, J. 1991. A paradigmatic analysis of contemporary schools of IS development: The framework and literature analysis. European Journal of Information Systems 1, 249-272.

Iivari, J., Hirschheim, R. & Klein, H. K. 2004. Toward a distinctive body of knowledge for Information Systems experts: coding ISD process knowledge in two IS journals. Information Systems Journal 14, 313-342.

Iivari, J., Hirschheim, R. & Klein, H. K. 1998b. Information Systems Development Approaches and Methodologies. Information Systems Research 9 (2), 164-193.

Iivari, J. & Iivari, N. 2011. The relationship between organizational culture and the deployment of agile methods. Information and Software Technology 53 (5), 509-520.

Iivari, J. & Lyytinen, K. 1998. Research on Information Systems Development in Scandinavia-Unity in Plurality. Scandinavian Journal of Information Systems 10 (1\&2), 135-186.

Iivari, J. & Maansaari, J. 1998. The usage of systems development methods : are we stuck to old practices ? Information and Software Technology 40 (January), 501-510.

Johnson, P., Ekstedt, M. & Jacobson, I. 2012. Where's the theory for software engineering? IEEE Software (5), 96.

Judy, K. H. 2009. Agile principles and ethical conduct. 42nd Hawaii International Conference on System Sciences, 2009. HICSS'09. IEEE, 1.

Kan, S. H. 2002. Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc.

Katz, R. 2005. Motivating technical professionals today. Research-Technology Management 48 (6), 19-27.

Khalifa, M. & Verner, J. M. 2000. Drivers for software development method usage. IEEE Transactions on Engineering Management 47 (3), 360-369.

Khanafiah, D. & Situngkir, H. 2004. Social balance theory. arXiv preprint nlin/0405041 .

Kiely, G. & Fitzgerald, B. 2002. An Investigation of the Information Systems Development Environment: The Nature of Development Life Cycles and The Use of Methods. Eighth Americas Conference on Information Systems, 1289.

Kitchenham, B. & Charters, S. 2007. Guidelines for performing systematic literature reviews in software engineering.

Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O. P., Turner, M., Niazi, M. & Linkman, S. 2010. Systematic literature reviews in software engineering–a tertiary study. Information and Software Technology 52 (8), 792-805.

Klein, H. K. & Myers, M. D. 1999. A set of principles for conducting and evaluating interpretive field studies in information systems. MIS quarterly , 67-93.

Kruchten, P., Nord, R. L. & Ozkaya, I. 2012. Technical debt: from metaphor to theory and practice. IEEE Software (6), 18-21.

Kumar, K. & Welke, R. J. 1992. Methodology Engineering R: a proposal for situation-specific methodology construction. Challenges and strategies for research in systems development. John Wiley & Sons, Inc., 257.

Lann, G. L. 1997. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on. IEEE, 339.

Lee, G., DeLone, W. & Espinosa, J. A. 2006. Ambidextrous coping strategies in globally distributed software development projects. Communications of the ACM 49 (10), 35-40.

Leppanen, M. 2006. Conceptual evaluation of methods for engineering situational ISD methods. Software Process: Improvement and Practice 11 (5), 539-555.

Leveson, N., G. & Turner, C., S. 1993. An investigation of the Therac-25 accidents. IEEE Computer 26 (7), 18-41.

Lewin, K. 1945. The research center for group dynamics at Massachusetts Institute of Technology. Sociometry 8 (2), 126-136.

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.

Linberg, K., R. 1999. Software developer perceptions about software project failure: a case study. Journal of Systems and Software 42 (9), 177-192.

Lindgren, M., Wall, A., Land, R. & Norstrom, C. 2008. A method for balancing short-and long-term investments: quality vs. features. 34th Euromicro Conference Software Engineering and Advanced Applications, 2008. SEAA'08. IEEE, 175.

Lyytinen, K. 1989. New challenges of systems development: a vision of the 90's. ACM SIGMIS Database 20 (3), 1-12.

Lyytinen, K. 1987. A taxonomic perspective of information systems development: theoretical constructs and recommendations. Critical issues in information systems research. John Wiley & Sons, Inc., 3.

MacCormack, A., Kemerer, C., F., Cusumano, M. & Crandall, B. 2003. Trade-offs between productivity and quality in selecting software development practices. IEEE Software 20 (5), 78-85.

Mancl, D., Fraser, S., D. & Opdyke, W., F. 2007. No silver bullet: a retrospective on the essence and accidents of software engineering. Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications. , 758.

Markus, L. M. & Robey, D. 1988. Information Technology and Organizational Change : Causal Structure in Theory and Research. Management Science 34 (5), 583-598.

Martin, R., Cecil 2003. Agile software development: principles, patterns, and practices. Upper Saddle River, NJ.: Prentice Hall PTR.

Martini, A., Bosch, J. & Chaudron, M. 2014. Architecture technical debt: understanding causes and a qualitative model. 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2014. IEEE, 85.

Mathiassen, L., Saarinen, T., Tuunanen, T. & Rossi, M. 2007. A Contigency Model for Requirements Development. Journal of the Association for Information Systems 8 (11), 569-597.

Maxwell, J. A. 2004. Using qualitative methods for causal explanation. Field methods 16 (3), 243-264.

McCaffery, F., Pikkarainen, M. & Richardson, I. 2008. Ahaa--agile, hybrid assessment method for automotive, safety critical smes. ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08. IEEE, 551.

McConnell, S. 2007. Technical Debt. Available in: http://www.construx.com/10x_Software_Development/Technical_Debt/.

McConnell, S. 1996. Avoiding classic mistakes. IEEE Software 13 (5), 112-112.

McLeod, L. & Doolin, B. 2012. Information systems development as situated socio-technical change: a process approach. European Journal of Information Systems 21 (2), 176-191.

Mohr, L. B. 1982. Explaining organizational behavior. Jossey-Bass San Francisco.

Murugesan, S. 1994. Attitude towards testing: a key contributor to software quality. Proceedings of the First International Conference on Software Testing, Reliability and Quality Assurance, 1994. IEEE, 111.

Myers, M. D. 1997. Qualitative research in information systems. Management Information Systems Quarterly 21 (2), 241-242.

Myers, M. D. & Newman, M. 2007. The qualitative interview in IS research: Examining the craft. Information and Organization 17 (1), 2-26.

Nan, N. & Harter, D., E. 2009. Impact of budget and schedule pressure on software development cycle time and effort. IEEE Transactions on Software Engineering 35 (5), 624-637.

Newcomb, T. M. 1961. The acquaintance process. New York: Holt, Rinehart & Winston.

Notander, J. P., Höst, M. & Runeson, P. 2013. Challenges in flexible safety-critical software development–an industrial qualitative survey. In Product-Focused Software Process Improvement. Springer, 283-297.

O Reilly, C. A. & Tushman, M. L. 2004. The ambidextrous organization. Harvard business review 82 (4), 74-83.

Okoli, C. & Schabram, K. 2010. A Guide to Conducting a Systematic Literature Review of Information Systems Research. All Sprouts Content , Paper 348.

Orlikowski, W. J. & Baroudi, J. J. 1991. Studying information technology in organizations: Research approaches and assumptions. Information systems research 2 (1), 1-28.

Orlikowski, W. & Iacono, S. 2001. Research Commentary: Desperately Seeking the IT in IT Research--A Call to Theorizing the IT Artifact. Information Systems Research 12 (2), 121-134.

Peters, L. 2014. Technical Debt: The Ultimate Antipattern - The Biggest Costs May Be hidden, Widespread, and Long Term. Managing Technical Debt (MTD), 2014 Sixth International Workshop on. , 8.

Petersen, K., Feldt, R., Mujtaba, S. & Mattsson, M. 2008. Systematic mapping studies in software engineering. 12th International Conference on Evaluation and Assessment in Software Engineering. sn.

Potdar, A. & Shihab, E. 2014. An Exploratory Study on Self-Admitted Technical Debt. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014. IEEE, 91.

Poth, A. & Sunyaev, A. 2014. Effective Quality Management: Risk-and Value-based Software Quality Management. IEEE Software 31 (6), 79-85.

Ralph, P. 2016. Software engineering process theory: A multi-method comparison of Sensemaking–Coevolution–Implementation Theory and Function–Behavior–Structure Theory. Information and Software Technology 70, 232-250.

Ralph, P. 2015a. Developing and evaluating software engineering process theories. Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 20.

Ralph, P. & Wand, Y. 2008. A teleological process theory of software development. Proceedings of JAIS Theory Development Workshop 8 (23).

Ralph, P. 2015b. The Sensemaking-Coevolution-Implementation Theory of software design. Science of Computer Programming 101, 21-41.

Ramesh, B., Mohan, K. & Cao, L. 2012. Ambidexterity in agile distributed development: an empirical investigation. Information Systems Research 23 (2), 323-339.

Rothman, K. J. 2012. Epidemiology: an introduction. New York: Oxford University Press.

Rowe, F. 2014. What literature review is not: diversity, boundaries and recommendations. European Journal of Information Systems 23 (3), 241-255.

Royce, W. W. 1970. Managing the Development of Large Software Systems. IEEE WESCON. , 1.

Runeson, P. & Höst, M. 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering 14 (2), 131-164.

Sabherwal, R. & Robey, D. 1993. An Empirical Taxonomy of Implementation Processes Based on Sequences of Events in Information System Development. Organization Science 4 (4), 548-576.

Samalikova, J., Kusters, R., Trienekens, J., Weijters, T. & Siemons, P. 2011. Toward objective software process information: experiences from a case study. Software Quality Journal 19 (1), 101-120.

Sarker, S., Lau, F. & Sahay, S. 2000. Using an adapted grounded theory approach for inductive theory building about virtual team development. ACM SiGMIS Database 32 (1), 38-56.

Schwaber, K. & Beedle, M. 2001. Agile Software Development with Scrum. Prentice Hall PTR.

Seth, F. P., Taipale, O. & Smolander, K. 2014. Organizational and customer related challenges of software testing: An empirical study in 11 software companies. Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on. IEEE, 1.

Shah, H., Harrold, M. J. & Sinha, S. 2014. Global software testing under deadline pressure: Vendor-side experiences. Information and Software Technology 56 (1), 6-19.

Shirado, W., Straka, W., Arkwright, T., Levay, M. & Lundholm, D. 1996. Software process in a mixed R&D environment. Proceedings of Aerospace Applications Conference, 1996. IEEE, 315.

Silva, J. G. A. & Cunha, P. R. d. 2006. Reconciling the irreconcilable? A software development approach that combines Agile with Formal. Proceedings of the 39th Annual Hawaii International Conference on System Sciences, 2006. HICSS'06. IEEE, 216b.

Sjøberg, D. I., Dybå, T., Anda, B. C. & Hannay, J. E. 2008. Building theories in software engineering. In Guide to advanced empirical software engineering. Springer, 312-336.

Slaughter, S. A., Levine, L., Ramesh, B., Pries-Heje, J. & Baskerville, R. 2006. Aligning Software Processes with Strategy. MIS Quarterly 30 (4), 891-918.

Sommerville, I. 2015. Software engineering. (10th edition) Pearson.

Sommerville, I. 2011. Software engineering. (9th edition) Boston: Addison-Wesley.

Sommerville, I. 2005. Integrated Requirements Engineering: A Tutorial. IEEE Software 22 (1), 16-23.

Stol, K., Ralph, P. & Fitzgerald, B. 2016. Grounded theory in software engineering research: a critical review and guidelines. Proceedings of the 38th International Conference on Software Engineering. ACM, 120.

Tassey, G. 2002. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project , 7007 (011).

Thanh, N. C. & Thanh, T. 2015. The interconnection between interpretivist paradigm and qualitative methods in Education. American Journal of Educational Science 1 (2), 24-27.

The Standish Group International, Inc. 2009. The Chaos Summary 2009.

Tighy, G. 2012. Evaluation of software engineering management best practices in the Western Cape. Software Engineering Colloquium (SE), 2012 4th. IEEE, 21.

Tom, E., Aurum, A. & Vidgen, R. 2013. An exploration of technical debt. Journal of Systems and Software 86 (6), 1498-1516.

Truex, D., Baskerville, R. & Travis, J. 2000. Amethodical systems development: the deferred meaning of systems development methods. Accounting, Management and Information Technologies 10 (1), 53-79.

Tsumaki, T. & Tamai, T. 2005. A Framework for Matching Requirements Engineering Techniques to Project Characteristics and Situation Changes. , 44-58.

Tushman, M. L. & O'Reilly, C. A. 1996. The ambidextrous organizations: managing evolutionary and revolutionary change. California management review 38 (4), 8-30.

Urquhart, C. & Fernandez, W. 2013. Using grounded theory method in information systems: the researcher as blank slate and other myths. Journal of Information Technology 28 (3), 224-236.

Urquhart, C., Lehmann, H. & Myers, M. D. 2010. Putting the 'theory'back into grounded theory: guidelines for grounded theory studies in information systems. Information Systems Journal 20 (4), 357-381.

Van de Ven, A. H. 1992. Suggestions for studying strategy process: a research note. Strategic Management Journal 13 (5), 169-188.

Van de Ven, A. H. & Poole, M. S. 2005. Alternative approaches for studying organizational change. Organization Studies 26 (9), 1377-1404.

Van Emden, E. & Moonen, L. 2002. Java quality assurance by detecting code smells. Ninth Working Conference on Reverse Engineering, 2002. Proceedings. , 97.

Vartiainen, T. & Siponen, M. T. 2012. What Makes Information System Developers Produce Defective Information Systems For Their Clients? Proceedings of Pacific Asia Conference of Information Systems (PACIS).

Vartiainen, T., Siponen, M. T. & Moody, G. 2011. Gray-Area Phenomenon In Information Systems Development: A Call For Research. PACIS. , 198.

Vavpotič, D. & Vasilecas, O. 2012. Selecting a methodology for business information systems development: decision model and tool support. Computer Science and Information Systems 9 (1), 135-164.

Vinekar, V., Slinkman, C. W. & Nerur, S. 2006. Can agile and traditional systems development approaches coexist? An ambidextrous view. Information Systems Management 23 (3), 31-42.

Wang, Y. & Zhang, M. 2010. Penalty policies in professional software development practice: a multi-method field study. Proceedings of the 32nd

ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 39.

Wijayasekara, D., Manic, M., Wright, J., L. & McQueen, M. 2012. Mining bug databases for unidentified software vulnerabilities. 5th International Conference on Human System Interactions (HSI), 2012. , 89.

Wohlin, C., Šmite, D. & Moe, N. B. 2015. A general theory of software engineering: Balancing human, social and organizational capitals. Journal of Systems and Software 109, 229-242.

Woodside, A. G. & Chebat, J. 2001. Updating Heider's balance theory in consumer behavior: A Jewish couple buys a German car and additional buying–consuming transformation stories. Psychology & Marketing 18 (5), 475-495.

Wynekoop, J. L. & Russo, N. L. 1997. Studying system development methodologies: an examination of research methods. Information Systems Journal 7 (1), 47-65.

Wynekoop, J. L. & Russo, N. L. 1995. System development methodologies: unanswered questions. Journal of Information Technology 10, 65-73.

Yang, B., Hu, H. & Jia, L. 2008. A study of uncertainty in software cost and its impact on optimal software release time. IEEE Transactions on Software Engineering 34 (6), 813-825.

Zhang, C. & Budgen, D. 2012. What do we know about the effectiveness of software design patterns? IEEE Transactions on Software Engineering 38 (5), 1213-1231.

Zhang, Z. & Lyytinen, K. 2001. A framework for component reuse in a metamodelling-based software development. Requirements Engineering 6 (2), 116-131.