

INVESTIGAÇÃO DE PRÁTICAS DE
DESENVOLVIMENTO E EVOLUÇÃO DE
SOFTWARE

MARKOS VIGGIATO DE ALMEIDA

INVESTIGAÇÃO DE PRÁTICAS DE
DESENVOLVIMENTO E EVOLUÇÃO DE
SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: EDUARDO FIGUEIREDO
COORIENTADOR: POOYAN JAMSHIDI

Belo Horizonte
Dezembro de 2018

MARKOS VIGGIATO DE ALMEIDA

**ON THE INVESTIGATION OF SOFTWARE
DEVELOPMENT AND EVOLUTION PRACTICES**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: EDUARDO FIGUEIREDO

CO-ADVISOR: POOYAN JAMSHIDI

Belo Horizonte

December 2018

© 2018, Markos Viggiato de Almeida.
Todos os direitos reservados.

Almeida, Markos Viggiato de

A447o On the Investigation of Software Development and
Evolution Practices / Markos Viggiato de Almeida. —
Belo Horizonte, 2018
xx, 60 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais — Departamento de Ciência da
Computação.

Orientador: Eduardo Magno Lages Figueiredo

Coorientador: Pooyan Jamshidi

1. Computação — Teses. 2. Engenharia de Software.
3. Algoritmo de mineração. 4. Domínio de software.
5. Plataforma de software I. Orientador.
II. Coorientador. III. Título.

CDU 519.6*32(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

On The Investigation of Software Development and Evolution Practices

MARKOS VIGGIATO DE ALMEIDA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Eduardo Figueiredo

PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

Pooyan Jamshidi

PROF. POOYAN JAMSHIDI - Coorientador
Departamento de Ciência da Computação e Engenharia - Universidade da Carolina do Sul

Marco Telio de Oliveira Valente

PROF. MARCO TELIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

Alessandro Fabricio Garcia

PROF. ALESSANDRO FABRICIO GARCIA
Departamento de Informática - PUC-Rio

Belo Horizonte, 17 de dezembro de 2018.

Acknowledgements

Many people have been extremely supportive to me and they are part of this achievement. I am very grateful to everyone who, directly or indirectly, helped me through this journey.

First of all, I thank God for giving me the strength necessary to persist and conclude this stage of my life. I am so grateful for everything and to everyone that was part of my life during this period.

I immensely thank my family for supporting me and understanding all the times I could not be with them. I specially thank my mother Margarete, my father João, and my sister Caroline.

I am so grateful to my advisor Eduardo Figueiredo for giving me the opportunity to be his student. I thank him very much for his patience and dedication during my studies. He is a person who I surely admire and respect.

I also thank my co-advisor Pooyan Jamshidi for all important contributions to my research. He surely played an important role during my Master studies.

I thank my lab mate Johnatan Oliveira for his contributions to my work. Johnatan has helped for a long time and surely is part of my achievement.

I thank professor Christian Kästner (Carnegie Mellon University) for his valuable contributions and ideas. With no doubt, Christian helped me with my research.

I am also very thankful to all members of the Software Engineering Laboratory (LabSoft). Everyone received me very kindly in the lab and has helped me through the difficult moments in my Master studies.

I thank all my friends who have supported me during this period of intense dedication to my Master studies and have completely understood my situation.

I would like to thank the panel members of my Master defense - Alessandro Garcia (PUC-Rio), Marco Tulio Valente (UFMG), and Pooyan Jamshidi (University of South Carolina).

Finally, I also thank CAPES and PPGCC-UFMG for the financial support.

Resumo

A engenharia de software é um campo diverso composto por diferentes plataformas e domínios, com uma grande variedade de pessoas envolvidas em todas as etapas do desenvolvimento. Plataforma de software refere-se à estrutura subjacente em que o software é construído (por exemplo, mobile) e domínios de software referem-se a sistemas desenvolvidos para segmentos específicos (por exemplo, para a área saúde). Sabe-se que diferentes plataformas podem ter diferentes práticas de evolução para atender suas necessidades específicas de mercado e devido a suas características intrínsecas. Por exemplo, os usuários do Android estão acostumados a rápidas correções de bugs, diferentemente de usuários de software desktop ou até mesmo de aplicações Web. Também é amplamente sabido que diferentes domínios podem ter diferentes políticas e valores, o que afeta a maneira como os desenvolvedores adotam as práticas de desenvolvimento. Apesar da relevância de compreender as práticas de desenvolvimento e evolução, pouco se sabe sobre como software mobile e não-mobile evoluem. Por exemplo, diferentes tipos de mudanças podem co-evoluir em uma plataforma e não em outras. Além disso, pouca pesquisa investigou quais e como as práticas de desenvolvimento são adotadas em diferentes domínios de software e se essas práticas são características intrínsecas dos domínios. Nesta Dissertação de Mestrado, propomos uma pesquisa de método misto visando compreender as diferenças e semelhanças de diferentes plataformas e domínios a partir de perspectivas quantitativas e qualitativas em diferentes granularidades. Primeiro, projetamos e conduzimos um estudo quantitativo no qual analisamos o histórico de commit de centenas de repositórios Java hospedados no GitHub para identificar como as alterações do código ocorrem. Mais especificamente, investigamos a frequência de commits e a co-evolução de três tipos de mudanças: alterações no código-fonte, alterações em arquivos de build e alterações em arquivos de testes. Para o último item, contamos a algoritmo de mineração Apriori para obter regras de associação relativas a tipos de alterações de código. Nossos resultados sugerem algumas diferenças relacionadas às plataformas (mobile e não-mobile). Por exemplo, em relação às frequências de commits, a plataforma móvel possui diferentes padrões de mudanças

ao longo do ano em comparação com sistemas desktop e Web. Também realizamos um estudo qualitativo no qual realizamos entrevistas semiestruturadas com desenvolvedores *cross-domain*, ou seja, desenvolvedores com experiência em mais de um domínio. Em seguida, desenvolvemos uma Web survey para confirmar ou não os resultados de domínios nos quais atingimos a saturação teórica de acordo com a Grounded Theory. Nosso objetivo é entender como as práticas de desenvolvimento são aplicadas a partir da perspectiva dos profissionais. Nossos resultados revelam que, na verdade, desenvolvedores de diferentes domínios aplicam práticas de desenvolvimento de diferentes maneiras. Por exemplo, domínios relacionados a finanças podem interromper práticas de integração contínua em períodos em que as transações financeiras aumentam, como no Natal e no Ano Novo.

Palavras-chave: Algoritmo de mineração Frequent Itemset, Alteração de Código, Plataforma de Software, Domínio de Software, Estudo de Entrevista..

Abstract

Software engineering is a diverse field composed of different platforms and domains with a large variety of people involved in all stages of the development. Software platform refers to the underlying structure where the software is built on (e.g., mobile) and software domains refer to systems developed for specific segments (e.g., health-care). It is known that different platforms may have different evolution practices to meet their specific market requirements and due to their intrinsic characteristic. For instance, Android users are used to fast bug fixes, differently from desktop or even Web applications. It is also widely known that different domains may have different policies and values, which impacts the manner how developers adopt development practices. Despite the relevance of comprehending development and evolution practices, little is known regarding how mobile and non-mobile software evolve. For example, different sorts of changes may co-evolve in one platform and not in others. Furthermore, little research has investigated which and how development practices developers follow in different software domains and whether these practices are intrinsic characteristics of the domains. In this Master dissertation, we propose a mixed-methods research aiming at understanding the differences and similarities of different platforms and domains from the quantitative and qualitative perspectives at different granularities. First, we designed and conducted a quantitative study in which we analyze the commit history of 363 Java repositories hosted in GitHub to identify how code changes occur. More specifically, we investigate the frequency of commits and the co-evolution of three types of changes: source code changes, build changes and test changes. For the last item, we rely on a Frequent Itemset mining algorithm (*Apriori*) to obtain association rules regarding sorts of code changes. Our results suggest some differences related to the platforms (mobile and non-mobile). For instance, regarding the frequency of commits, the mobile platform has different patterns of changes along the year in comparison to non-mobile systems. We also performed a qualitative study in which we conducted 19 semi-structured interviews with cross-domain developers from the industry, i.e., developers who have experience in more than one domain. Afterwards, we run a Web

survey to confirm or not the results from domains in which we reached the theoretical saturation according to the Grounded Theory. Our goal is to understand how development practices are applied from the perspective of practitioners. Our results reveal that in fact developers from different domains apply development practices in different fashions. For instance, financial-related domains may interrupt continuous integration practices in periods when financial transactions increase, such as in Christmas and New Year.

Keywords: Frequent Itemset Mining Algorithm, Code Change, Software Platform, Software Domain, Interview Study..

List of Figures

1.1	Proposed studies.	4
3.1	(a) Example of a transaction database with code change types; (b) Frequent types of code change (along with their support) and minimum support of 3.	19
3.2	Frequency of commits in a 2-year time period.	24
3.3	Distributions of response variable for mobile and non-mobile.	26
3.4	Correlations between predictor variables.	27
4.1	Our research methodology process.	35
4.2	Main adopted practices in domains. Banking domain is moderately regulated and interrupt continuous integration process in important commerce periods (e.g., Black Friday); e-commerce follows an user-centered development, focusing on non-functional requirements that provide a good user experience and also interrupt continuous integration process; and health-care is highly regulated, focuses on patient data privacy and security and requirements elicitation may be easier than in other domains.	39

List of Tables

2.1	Software domains and descriptions.	11
3.1	Aggregate statistics of the 363 repositories	20
3.2	Heuristics for identifying build and test files	23
3.3	Multiple linear regression coefficients for our two models.	26
3.4	Frequent types code changes in all commits.	28
4.1	Interviewees information.	36
4.2	Survey results with presented statements and Likert-scale agreement distribution.	40

Contents

Acknowledgements	ix
Resumo	xi
Abstract	xiii
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Motivation	2
1.2 Goal and Methodological Procedures	4
1.3 Results	6
1.4 Dissertation Outline	6
2 Background and Related Work	9
2.1 Software Domains	9
2.2 Development Practices in Software Domains	10
2.3 Software Evolution Practices	13
2.4 Final Remarks	14
3 Quantitative study	15
3.1 Goal and Research Questions	15
3.2 Mining Frequent Itemsets	16
3.3 Research Method	19
3.4 Results	23
3.4.1 Frequency of Commits	24
3.4.2 Frequent Code Change Types and Association Rules	28
3.5 Discussion	29

3.6	Threats to Validity	31
3.7	Final Remarks	31
4	Qualitative Study	33
4.1	Goal and Research Questions	33
4.2	Research Method	34
4.2.1	Interview Process	34
4.2.2	Validation	38
4.3	Results	39
4.3.1	Banking Domain	40
4.3.2	E-commerce Domain	42
4.3.3	Healthcare Domain	43
4.4	Discussion	45
4.4.1	Implications for Practice	46
4.4.2	Contrast with Current Beliefs	47
4.4.3	Results for Other Domains	47
4.5	Limitations and Threats to Validity	49
4.6	Final Remarks	50
5	Final Considerations	51
5.1	Conclusion	51
5.2	Contributions	53
5.3	Future Work	54
	Bibliography	55

Chapter 1

Introduction

Software engineering is a complex and diverse field, with a big variety of people applying different practices from the design and implementation phases all the way to the maintenance and evolution phases. Software developed in different platforms and domains have different practices due to their specific needs, as previously investigated by Murphy-Hill et al. [2014]:

In a larger sense, this work represents a step towards understanding software development not as a homogeneous whole, but instead as a rich tapestry of varying practices involving diverse people across diverse domains.

Software platform (e.g., mobile and desktop) refers to the underlying "structure" upon which software is built and it has specific characteristics [Zhang et al., 2018]. For instance, the mobile platform, differently from desktop and Web applications, is usually used to develop sensor-, gesture-, and event-driven applications and it has memory and power consumption constraints [Zhang et al., 2018]. Different software platforms present different characteristics and make use of practices in different ways. In this dissertation, we use *practice* as a general term to refer to the way software is developed and its characteristics as well, which includes not only software development phase, but also design, maintenance and evolution. For example, bug causes and bug fixing processes, for example, are different in desktop and Android. While in desktop most frequent high-severity bugs occur due to build issues, in Android, the cause of most problematic bugs is concurrency [Zhou et al., 2015].

Software domain refers to a category of systems developed to meet specific business segment requirements. Different software domains (e.g., aviation, e-commerce and healthcare) have different characteristics and, although some domains may be similar

with regard to the way they adopt practices, most domains apply practices in a specific way that is convenient to them [Murphy-Hill et al., 2014; Segura et al., 2014; Russo et al., 2017; Richardson et al., 2016]. For instance, software development within the game domain has substantial differences in comparison to software development in other domains, such as game developers rarely make use of automated testing practices [Murphy-Hill et al., 2014].

The diversity in software engineering has motivated several works from the research community and also industry-track works that investigated practices and other aspects in specific software platforms and domains [Zhou et al., 2015; Murphy-Hill et al., 2014; Richardson et al., 2016; Russo et al., 2017; Segura et al., 2014; Wright and Perry, 2012]. However, to the best of our knowledge, we still lack a more comprehensive understanding of how practices are adopted across different software platforms and domains, which allows us to identify similarities and differences among platforms and domains.

1.1 Motivation

Considering software engineering as a homogeneous whole may be problematic as the needs of systems from specific platforms or domains are ignored. Developers should have access to information regarding which and how practices are adopted, mainly those professionals who are looking for a new job. Shedding light on practices usage may strongly benefit developers as they can be aware of how that platform/domain works and behaves before getting into it. For instance, whether a professional is looking for a mobile developer position, a previous understanding of commonly used evolution (e.g., types of code changes usually made together) and development (e.g., how systems are tested) practices may prepare the developer for that job.

Revealing which and how practices are adopted may also support newcomers who intend to contribute to Open Source Software (OSS) projects, given their important role in the survival and long-term success of community-based OSS [Steinmacher et al., 2016]. Due to the quite independent and self-organized characteristics of working in open source projects [Steinmacher et al., 2016], newcomers should be provided with insights and technical support of how current contributors in fact work so that they can be prepared. Previous works have addressed this issue [Steinmacher et al., 2016], but the authors could not identify any significant improvement in supporting newcomers to overcome technical barriers.

For instance, understanding how changes are performed in a repository before sending pull requests or joining an OSS project may provide the newcomer with valuable information to support this initial phase of contribution and avoid rework or contribution rejection, which could demotivate the developer to keep contributing. As a concrete example, we can imagine that if developers from a specific open source repository usually change source code files together with configuration files, a developer who intends to submit a source code change via pull request should also provide a change in configuration files.

Another aspect of the importance of understanding practices' adoption is concerning the specific knowledge and behavior of the development team depending on the platform or domain of the company. For instance, developers behave differently when fixing bugs depending on the platform in which they are working. Few works have studied the differences in desktop and mobile platforms [Bhattacharya et al., 2013; Zhang et al., 2018]. Developers of desktop systems usually are not involved in reporting bugs. In addition, bug-fixing process takes a longer period of time compared to Android and iOS [Zhang et al., 2018; Bhattacharya et al., 2013; Breu et al., 2010]. On the other hand, bug fixers of mobile applications are more involved in reporting bugs to be discussed and the main causes of bugs are concurrency (in Android) and application logic (in iOS) [Zhang et al., 2018]. We believe companies should provide targeted training for their employees, not only software developers, but also training for people from other positions (e.g., software architect and technology leader). This training should focus on specific platforms' and domains' characteristics and needs, and how developers from those platforms usually work (behavior and practices adopted).

Software engineering education professionals also need a more comprehensive understanding of practices adopted mainly in different software domains. We believe new teaching approaches that consider the software domain should be developed. For instance, new specific undergraduate or graduate courses may be interesting. Interdisciplinary courses may also be a good idea, as Richardson et al. [2016] recently suggested an interdisciplinary course of software engineering for healthcare systems. Finally, software engineering education may take specific development practices into account as different domains have different development practices. For instance, a new branch of the software engineering course focused on game development may be suitable since this domain differs in many aspects from non-game software development, such as regarding different testing practices [Murphy-Hill et al., 2014].

1.2 Goal and Methodological Procedures

Given the limitations and restrictions of previous studies on evolution and development practices, in this dissertation we aim at providing a more thorough understanding of which and how practice are adopted in different software contexts. We focus on two sorts of practices: evolution practices and development practices. Regarding evolution practices, we adopt a more general mining-based approach and investigate how code changes are performed in two different platforms, namely mobile and non-mobile. Note that, for the mobile platform, we consider only Android applications given the large availability of Android repositories in GitHub. In addition, for non-mobile platforms, we consider both desktop and Web applications. In relation to development practices, we investigate how practices (e.g., testing practices and continuous integration/delivery practices) are applied across different software domains by means of an interview study.

In this dissertation, we propose a mixed-methods research to achieve our main goal. First, a quantitative study through which we address evolution practices usage in mobile and non-mobile software platforms by analyzing 363 repositories mined from GitHub. Second, a qualitative study to understand and reveal development practices usage in 13 software domains. Figure 1.1 presents an overview of the proposed studies and their main phases.

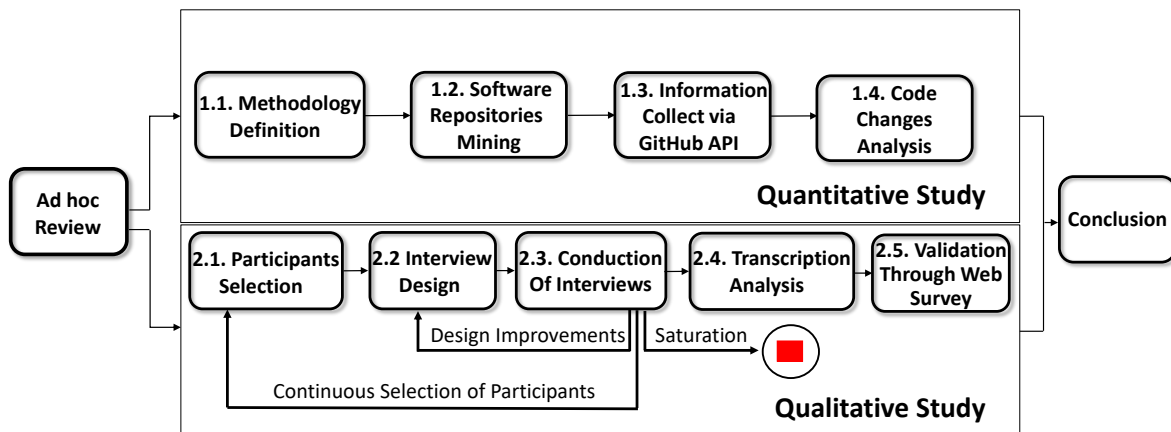


Figure 1.1. Proposed studies.

In the quantitative study (steps 1.1 to 1.4 in Figure 1.1), our goal is to investigate evolution practices within the context of software platforms with regard to code changes made to the systems using a Version Control System (VCS). In this study, we complement and expand previous works, identified in an ad hoc review, that investigated code changes [Levin and Yehudai, 2017; Macho et al., 2017]. Note that we focus

on Java projects due to some constraints, such as the need for build files, which are extensively used in Java projects.

As we can see in Figure 1.1, we started with the definition of the study scope, based on previous works identified in an ad hoc review. We then mine software repositories from GitHub, collect all necessary information via GitHub REST API and perform data analysis on code changes. More specifically, we investigate two groups of platforms: mobile (Android applications) and non-mobile (desktop and Web applications). We study two aspects of evolution by analyzing the commit history of the last 2 years of repositories' activity: (i) frequency of commits and (ii) co-evolution of different types of changes. For the frequency analysis, we compare both platforms and discuss how being mobile impacts the frequency. We make use of statistical modelling (multiple linear regression models) to provide explanation about frequency. Regarding the co-evolution of changes, we investigate three types of code changes: source code changes, build changes, and test changes. A frequent itemset mining algorithm (Apriori [Agrawal et al., 1996, 1993]) is used to find frequent types of code changes that occur together.

In the qualitative study (steps 2.1 to 2.5 in Figure 1.1), our goal is to deepen our analysis of practices in software development. In this study, we investigate development practices. We noticed a need for further investigation within this topic from previous works that suggested different domains follow different practices (or apply them in a different fashion) [Murphy-Hill et al., 2014; Richardson et al., 2016; Segura et al., 2014; Russo et al., 2017]. Given that software platforms are quite broad and coarse-grained, we concluded that development practices could not be categorized by the platform. That is, we believe development practices are not directly related to the software platform. Therefore, we decided to perform this study in a finer-grained level: software domains.

We investigate whether different software domains apply development practices in a different or similar way. As we can observe in Figure 1.1, we started with the selection of participants and the interview design. Then, we simultaneously conducted the interviews while transcribing the audio files. We conducted a total of 19 semi-structured interviews with cross-domain developers. That is, developers who have experience in at least two domains. For the transcription phase, we used Grounded Theory techniques, such as open coding [Stol et al., 2016; Strauss and Corbin, 1990; Glaser and Strauss, 2017]. Finally, we validated our results through a Web survey. We collected information about the use of development practices per domain, as obtained with the interviews. By analyzing the survey responses from 40 developers, we are able to check whether those practices are in fact adopted by developers from the target domains.

1.3 Results

Through the quantitative study, we can understand how evolution practices (with regard to code changes) are adopted in mobile and non-mobile platforms. For the frequency analysis, our findings show that non-mobile repositories have a higher number of commits per month compared to mobile. The trend graphs for both platforms are not similar, but both have a peculiar behavior in the holiday season (including Christmas and new year periods): the number of commits sharply decreases. Our regression models suggest that being mobile significantly impacts the number of commits in a negative direction when controlling for confound factors. The Cohen's f^2 measure is 0.19, indicating a medium effect size of the mobile variable in our models. We also diagnose our models, checking for multicollinearity. The variance inflation factors for all variables are below 3, which is a safe value. Regarding the co-evolution of different types of changes, the *Apriori* algorithm does not suggest that developers usually change source code files together with build or test files as the support values for frequent code changes are low. However, the association rules have high confidence, which indicates our rules are strong.

In the qualitative study, our interview findings suggest that some development practices are commonly adopted in different domains, while other practices are specific to a domain. For this study, we discuss the findings that were confirmed by the Web survey. Our results show that continuous integration practices are similarly adopted in the banking and e-commerce domains. Furthermore, regulatory-driven changes are common in the banking and healthcare domains, which must adapt their workflow to comply to regulatory demands. Our findings also indicate that requirements engineering practices are adopted in a unique way by the banking domain, involving the comprehension of complex financial operations. In addition, practices related to interoperability are more difficult in the healthcare domain in comparison to others, due to different standards used by health companies.

1.4 Dissertation Outline

This Master dissertation is organized in 5 chapters, as follows.

Chapter 2 provides background information and previous works related to this research. We briefly present important concepts regarding software domains and discuss past works related to development practices in software domains and also related to evolution practices.

Chapter 3 presents a study to investigate evolution practices in mobile and non-mobile platforms from a quantitative perspective through software repository mining in GitHub. We rely on statistical modelling to explain the frequency of commits and we use a frequent itemset mining algorithm to find co-occurrences of three types of code changes, namely source code changes, build changes, and test changes.

Chapter 4 presents a study to investigate and reveal how development practices are adopted across different software domains from a qualitative perspective. We perform an interview study using techniques from the Grounded Theory (e.g. open coding) and we run a Web survey to validate our results.

Chapter 5 concludes this Master dissertation, presenting the lessons learned from the quantitative and qualitative studies we performed, contributions of this work, and suggestions for future research.

Chapter 2

Background and Related Work

A thorough understanding of software evolution practices based on code changes in mobile and non-mobile platforms is relevant as insights can be given to companies and practitioners of each platform. In addition, the research community may define new research directions based on the comprehension of how mobile and non-mobile platforms are different. A more complete understanding may be achieved by performing a study on a finer-grained level to investigate how development practices are adopted in different software domains. In this chapter, we present information regarding the software domains we investigate in the qualitative study and discuss previous related works. Section 2.1 presents the target domains and a brief description of each one, Section 2.2 presents past studies related to development practices in software domains, Section 2.3 discusses previous works related to evolution practices with regard to code changes, and Section 2.4 concludes this chapter.

2.1 Software Domains

Software domain consists of systems that share specific characteristics, which allows us to identify and group those systems and differ them from other applications. Systems that belong to the same domain are designed for specific business segments (e.g. e-commerce, healthcare, aviation) and users. The domain may affect several aspects of software development [Mori et al., 2018; Linares-Vásquez et al., 2014; Russo et al., 2017; Richardson et al., 2016], such as the adopted development practices. Not considering software engineering as a uniform whole [Murphy-Hill et al., 2014] and understanding how the domain influences software development is important and may shed light on important issues and provide insights for solutions, such as creating interdisciplinary courses of software engineering and other areas [Richardson et al., 2016] and support

developers in a way they can keep themselves updated with the practices adopted in domains, specially those developers looking for a new job.

We selected a specific set of software domains for our study based on previous works [Russo et al., 2017; Linares-Vásquez et al., 2014; Richardson et al., 2016; Murphy-Hill et al., 2014; Mori et al., 2018]. That is, we selected domains which were already subject of research and, therefore, we believe they are well-known in the software engineering community and easily understandable by industry professionals. Our target domains encompass several types of systems, such as frameworks and tools. Furthermore, we believe it is feasible to find software developers who have worked in such domains through our participant search procedure. Initially, we selected the following 13 domains: accounting, aviation, banking, business, e-commerce, educational, games, healthcare, mining and metals, oil and gas, search engine, social network, and stock market. Table 2.1 presents the domains we investigate in this master thesis along with their descriptions.

2.2 Development Practices in Software Domains

In this section, we present previous work that investigated development practices in software domains. We also discuss how our research differs and complements past works.

Several studies have been proposed within the context of software development practices [Yost et al., 2016; Wright and Perry, 2012; Thongtanunam et al., 2015] and software domains [Segura et al., 2014; Russo et al., 2017; Fairbanks et al., 2006; Richardson et al., 2016]. For instance, Yost et al. [2016] used an online survey to collect data about software development practices and barriers in the field and the relationship to software quality. The authors received a total of fifty complete responses. The study concluded that there is evidence of certain problematic issues for developers and specific quality characteristics that seems to be affected by such issues. Stavnycha et al. [2015] analyzed the effects of nine development practices on the quality aspect correctness of software systems. The authors collected data from software developers around the world through an online survey. The study results indicated that four of the nine development practices show statistically significant effects on the correctness of released software, such as test coverage and code reviews. In addition, their results suggest that using development practices specifically focusing on improving software quality shows a positive effect on the level of correctness of released software. Wright and Perry [2012] reported the initial results of a study in which the authors interviewed

Table 2.1. Software domains and descriptions.

Domain	Description
Aviation	Embedded systems with legal regulations regarding safety and reliability to control several aspects of different types of aircrafts.
Banking	Systems that compose the core banking software and provide financial services to the company itself and to clients.
Business	Systems that implement validation, calculation, and law regulations of business requirements, such as pricing, and inventory management.
E-commerce	Systems in charge of supporting the transactions of products buying and selling, as well as providing services to consumers.
Educational	Systems used by students to manage their study life and by managers to administrate their schools.
Games	Entertainment games that can be played alone or in collaboration.
Healthcare	Systems that offer health-related services to people in general.
Mining and Metals	Systems to support the operation and management of industrial processes in mining companies.
Oil and Gas	Systems that support the planning, operation and optimization of oil and gas extraction processes.
Search Engine	Systems developed mainly to perform optimized searches on the internet.
Social Network	Systems that allow users to interact with each other and communicate about shared interests and hobbies.
Stock Market	Systems that provide stock-related services, such as stock-picking decisions, tracking of stock investments and stock market predictions.

4 practicing release engineers to understand the faults and failures of release practices, how companies recover from them and how to predict and avoid the failures in future. Their preliminary results indicate that a more thorough process analysis and efforts at process standardization are necessary. Unlike the works presented above, we do not focus only on finding the relationship of development practices and software quality aspects (such as [Yost et al., 2016; Stavnycha et al., 2015]) neither on one specific practice (such as [Wright and Perry, 2012]). Instead, in our qualitative study presented in Chapter 4, we adopt an exploratory approach to investigate several practices in different domains aiming at understanding the similarities and differences of practices adoption across domains.

Regarding studies within the context of specific software domains, Murphy-Hill et al. [2014] presented a study comparing game development to traditional software de-

velopment (other domains). The work indicated substantial differences between video game development and other software development segments, such as the rare use of automated tests in game development. Richardson et al. [2016] noticed that regulations and directives regarding medical device software were not being taken into account, and unregulated software was developed and used in healthcare organisations. This observation was a result of not trained software engineers, who lacked of knowledge in regulations of software solutions for healthcare. The authors recommended that healthcare software systems should be developed by professional software engineers in interdisciplinary teams with healthcare professionals. Another study [Nytrö et al., 2009] developed methods for gathering detailed observational data about care and communication practice. The authors explain how this data can be used for iterative, demand-driven, requirements elicitation and to answer design questions. The proposed approach was a supplement to other Requirements Engineering methods. The study concluded that observation has a higher initial cost than other elicitation methods, but the information is reusable and may lead to more valid requirements and system functionality. As we can see above, several previous studies have focused on a single domain in their investigations, while in our qualitative study (Chapter 4) we initially have 12 target domains.

Segura et al. [2014] explored the applicability of some of the practices for variability management in software product lines to an e-commerce system. The authors used a feature model to represent the store input space and techniques for the automated analysis of feature models for the detection and repair of inconsistent and missing configuration settings. They also used test selection and prioritization techniques for the generation of a manageable and effective set of test cases. Their findings suggested that variability techniques could successfully address many of the challenges found when developing e-commerce systems. Russo et al. [2017] aimed at identifying some relevant concerns in the Italian banking IT sector, through an investigation of the opinions of several stakeholders. The authors identified 15 concerns, which were discussed in a framework inspired by the ISO 25010 standard. Furthermore, the study identified the emergence of a new meta quality dimension which impacts both on software quality and architectural description. Again, unlike the past works above, our exploratory study focus on 12 software domains instead of only one. Furthermore, although some previous studies performed interviews, in our qualitative study we conducted 19 semi-structured interviews with cross-domains developers, that is, developers who have experience in at least two domains. To the best of our knowledge, this Master thesis is the first research to conduct a study with cross-domain professionals.

2.3 Software Evolution Practices

In this section, we present previous work that investigated evolution practices with regard to code changes made to the systems. We also discuss how our research differs and complements past works.

Several works have investigated different types of code changes and performed commit history analysis with many different goals. For instance, Levin and Yehudai [2017] conducted a study using 61 popular open source projects to investigate the co-evolution of test maintenance and code maintenance based on code changes. The authors reported the relationships they have established between test maintenance, production code maintenance, and semantic changes (e.g, statement added, method removed, etc.) through commit analysis. Their findings reveal that developers perform code fixes without performing complementary test maintenance in the same commit (e.g., update an existing test or add a new one). Macho et al. [2017] investigated changes in the build configurations. The authors presented an approach to extract detailed build changes from Maven build files (*pom.xml*) and classify them into 95 change types. The authors presented two studies using the build changes extracted from 30 open source Java projects to study the frequency and time of build changes. Their results showed that the top 10 most frequent change types account 73% of the build changes and build changes usually occur around releases. Unlike the previous researches discussed above, in this Master thesis we expand the analysis of code changes. While Levin and Yehudai [2017] analyzed the co-evolution of only test changes and source code changes, we included build changes in our co-evolution analysis, as presented in chapter 3. In addition, Macho et al. [2017] performed a study on build changes relying only in the Apache Maven build automation tool (*pom.xml* files). As detailed also in chapter 3, we included two other build automation tools: Ant (*build.xml*) and Graddle (*build.gradle*). Furthermore, we compare the co-evolution of changes in mobile applications against non-mobile applications, as we believe the mobile platform has a different behavior during software evolution. For instance, we expect source code changes to occur together with build changes more frequently in mobile when compared to non-mobile systems.

Kirinuki et al. [2014] investigated commits that presented tangled changes, which hinders analyzing code repositories as most mining software repository approaches are designed with the assumption that every commit includes only changes for a single task. The authors proposed a technique to warn developers that they may be committing tangled changes. Based on the proposed technique, the developer is notified about how the tangled changes can be split into a set of untangled changes. Faragó et al.

[2015] investigated whether modifications performed on frequently changing code have worse effect on software maintainability than those affecting less frequently modified code. The authors calculated cumulative code churn values and maintainability changes for every version control commit operation of three open-source and one proprietary software system. Their findings indicated that modifying high-churn code is more likely to decrease the overall maintainability of a software system, which can increase the number of defects. As we can note, the past works discussed above focused on different aspects related to code changes, as they investigated tangled changes and changes made to high-churn code. As explained in the quantitative study in chapter 3, we perform a broader analysis of code changes, investigating the frequency of changes and the factors that explain it, the scattering (similar to tangled changes) and deepness, and also the co-evolution of different types of changes.

2.4 Final Remarks

In this chapter, we presented important information that is necessary to better understand this work. We first discussed the software domains we investigate in our qualitative study along with a short description of each one and how they are important for the software engineering research community and industry. In addition, we presented previous works related development practices in software domains and to code changes. Unlike all previous works discussed in this chapter, in this Master thesis we aim at providing a better and more complete understanding of practices related to software development and evolution from quantitative and qualitative perspectives. In chapter 3, we analyze the commit history of popular open source software to understand the differences and similarities in the evolution of mobile and non-mobile software. Furthermore, in chapter 4, we expand our research by interviewing cross-domain developers with the aim of identifying development practices that are similar across domains and practices that are particular to specific software domains. To the best of our knowledge, this is the first exploratory research to investigate software development and evolution practices using a mixed-method of research.

In the next chapter, we present the quantitative study we designed and performed to understand software evolution practices with regard to code changes made to mobile and non-mobile system. We investigate the frequency of commits and we rely on linear regressions to provide explanation about what may impact the number of commits. We also study the scattering and deepness of code changes and the co-evolution of three different types of changes: source code changes, test changes and build changes.

Chapter 3

Quantitative study

In this chapter, we present a quantitative study aiming at understanding evolution practices in mobile and non-mobile platforms to provide a more thorough comprehension of which evolution practices are adopted in different platforms (mobile and desktop/Web). In our study, we investigate the evolution with regard to code changes made to the systems. We believe the mobile platform has different evolution patterns compared to non-mobile platforms, such as desktop and Web applications. We argue our results can have practical implications, such as supporting newcomers who desire to join or submit pull requests to open source repositories. We believe a previous understanding of how changes are performed in the target repository may significantly increase the chance of the newcomer being successful in contributing to the repository. In our study, the analyses are performed on a dataset composed of 363 popular open source systems from GitHub, being 181 Android applications (referred as mobile platform) and 182 desktop and Web applications (referred as non-mobile platform). We investigate the frequency of commits and whether being mobile significantly impacts the frequency and co-evolution of three different sorts of changes: source code changes, build changes, and test changes. Section 3.1 presents our main goal and the research questions we designed. Section 3.2 details the mining algorithm we used to identify the co-evolution of code changes. Section 3.3 presents the research method. Section 3.4 reports the results, which are discussed in Section 3.5. We discuss the threats to validity in Section 3.6 and conclude this chapter in Section 3.7.

3.1 Goal and Research Questions

Our goal in this study is to understand the evolution practices adopted in mobile and non-mobile platforms since mobile platform has different requirements and character-

istics and evolves differently from other platforms [Basole and Karla, 2011; Zhou et al., 2015]. We investigate evolution practices with regard to code changes made to systems by commits, which means that whenever we mention code changes we are referring to the evolution of the systems. In this study, we narrowed the mobile platform to Android applications since they are largely present in GitHub and we are able to find several repositories. In addition, we consider desktop and Web applications as non-mobile applications as we intend to compare mobile platform against other platforms. That is, we do not aim at comparing all platforms in the lowest granularity, but only mobile with other platforms which are not mobile.

More specifically, through the quantitative study, we investigate two different aspects regarding code evolution. First, we check whether the frequency of commits is similar or different in mobile and non-mobile applications. We also make use of statistical modelling (multiple linear regression models) to provide explanation about possible factors that may influence the number of commits per month (i.e., the frequency) in each platform and whether the fact that the repository is mobile impacts the frequency when controlling for confounds. Second, we use a frequent itemset mining algorithm, called Apriori [Borgelt, 2012; Han et al., 2000], to analyze the co-evolution of three types of changes: source code file changes, build file changes and test file changes. Furthermore, we generate association rules based on the frequent code change types. To achieve our goal and guide us on investigating the two evolution aspects we elicited above, we defined the following research questions:

- *RQ1: How frequent are code changes in mobile and non mobile platforms?*
- *RQ2: How is the co-evolution of source code changes, build changes and test changes in mobile and non mobile platforms?*

3.2 Mining Frequent Itemsets

To answer the second research question, we apply a frequent itemset mining algorithm to find co-occurrence of different types of code changes. In this section, we present some background information and concepts regarding this topic as we believe they are necessary for a complete understanding of our work. We also detail how we mapped this problem to our context.

Huge efforts of research have been dedicated to the subject of finding frequent itemsets and deriving associaton rules [Borgelt, 2012; Molderez et al., 2017]. These tasks have been investigated for a long period of time by the *data mining* and *knowledge discovery in databases* research areas [Agrawal et al., 1996, 1993]. Frequent itemset

mining is a data analysis method and has been initially developed for the market basket analysis, aiming at finding common products that are usually bought together. That method has been useful for companies (e.g., supermarkets and online shops) to recommend products for clients based on their pattern of purchase [Borgelt, 2012], which allows to obtain association rules. However, finding frequent itemsets can be applied in many more contexts, such as finding patterns and regularities of categorical variables in a large dataset [Borgelt, 2012; Molderez et al., 2017]. Several sophisticated and efficient algorithms have been proposed to mine frequent itemsets and find association rules, such as Apriori [Agrawal et al., 1996, 1993], Eclat [Zaki et al., 1997; Zaki and Gouda, 2003; Schmidt-Thieme, 2004], and FP-Growth (Frequent Pattern Growth) [Han et al., 2000; Grahne and Zhu, 2003; Rácz, 2004].

Before mapping the frequent itemset mining to the context of our study, we briefly present the basic terminology and definitions used within the frequent itemset mining and association rules areas [Borgelt, 2012]. The goal of mining frequent itemsets is to find sets of items that frequently occur together across different transactions. A set of items $I = \{i_1, i_2, \dots, i_n\}$ is called *item base*. Any subset T of I is called *itemset* and a *transaction* t is represented by $t = \langle \text{tid}, T \rangle$, defined by a transaction unique identifier *tid*, and an itemset. Let us say the item represents a product in a supermarket. Therefore, an itemset corresponds to a set of products and a transaction corresponds to a specific purchase done by a customer. Different customers may buy the same set of products (i.e., different transactions may have the same itemset). The goal is to find the number of times each set of products was bought (i.e., the number of times each itemset occurred in all transactions). The *cover* $K_T(B) = \{k \in \{1, \dots, m\} \mid B \subseteq t_k\}$ of an itemset $B \subseteq I$ represents all transactions the itemset B is contained in. The *support* $s_T(B)$ is defined as the number of these transactions and, therefore, we have that $s_T(B) = |K_T(B)|$. When applying frequent itemset mining algorithms, one must specify a desired *minimum support* $s_{min} \in N$. Given this minimum support, an itemset B is called frequent (in all transactions T) iff $s_T(B) \geq s_{min}$. Finally, frequent itemset mining aims at finding all itemsets $B \subseteq I$ that are frequent in all transactions (or database) T .

After obtaining the frequent itemsets, we are able to generate association rules. Basically, a frequent itemset is split into two disjoint subsets, in which one is the antecedent of a rule (on the left side) and the other one is the consequent (on the right side of a rule) [Borgelt, 2012]. To assess how strong a rule is, we rely on a metric called *confidence*, defined as: $c_T(X \rightarrow Y) = s_T(X \cup Y) / s_T(X)$, in which $s_T(X)$ is the support of the itemset X . Intuitively, the confidence represents the conditional probability. That is, the probability of $X \cap Y$, given X . *Lift* is another metric commonly used within

the context of association rules. It is popularly used to rank rules[Borgelt, 2012] and measures how much the relative frequency of Y is increased when we consider only transactions that contain X .

Mapping Frequent Itemsets To Frequent Code Change Types. To find co-occurrences of code changes across commits in GitHub repositories, we map the problem of mining frequent itemset to finding frequent co-occurrences of code changes. Finding which types of code changes occur together allows us to understand the co-evolution of changes in a system. In the context of our study, we represent an item by a type of code change we investigate here. Therefore, we have three items: **source code**, **build** and **test**. We have a total of 8 (2^3) possible itemsets (that is, subsets of items), which are: $\langle \rangle$, $\langle \text{sourcecode} \rangle$, $\langle \text{build} \rangle$, $\langle \text{test} \rangle$, $\langle \text{sourcecode}, \text{build} \rangle$, $\langle \text{sourcecode}, \text{test} \rangle$, $\langle \text{build}, \text{test} \rangle$, and $\langle \text{sourcecode}, \text{build}, \text{test} \rangle$. Finally, a transaction is represented by a commit. For instance, when committing, a developer may commit only to a source code file, or to a source code and a build file. Therefore, we are able to find co-occurrences of different types of files reached by all commits. Note that, although commits may reach several sorts of files, we focus on three types of files in this study: source code files, build files and test files.

Figure 3.1 presents an example of commits (transactions) and frequent code change types (frequent itemsets). For instance, in Figure 3.1(a), commit 0 has only a build change, while commits 1 and 2 have simultaneously two types of changes: source code and build, and source and test, respectively. In Figure 3.1(b), we can see the types of code changes that occur together with a minimum support of 3, that is, types of changes that appear at least in 3 commits. For instance, we can observe that, when we consider 1 item, all three types of changes occur individually more than 3 times: source code, build and test. In addition, when considering 2 items, only the set composed of source code and test appears together at least 3 times across all commits.

When performing the data analysis, we follow the next two steps in order to find the types of frequent code changes and association rules:

1. Find all frequent code changes types in the commit dataset, i.e. types of code changes with support greater or equal to the minimum support.
2. For each frequent code change C_2 found, generate all association rules $C_2 \Rightarrow \{C_1, C_3\}$, where C_1 , C_2 , and C_3 represent different types of code changes, and report those rules with confidence greater or equal to the predefined minimum confidence.

<p>a)</p> <p>Commits</p> <p>0: {build}</p> <p>1: {sourcecode,build}</p> <p>2: {sourcecode,test}</p> <p>3: {sourcecode}</p> <p>4: {sourcecode}</p> <p>5: {sourcecode,build,test}</p> <p>6: {sourcecode,test}</p> <p>7: {test}</p> <p>8: {sourcecode}</p> <p>9: {sourcecode}</p>	<p>b)</p> <p>Frequent types of changes (with support) (Minimum support: $s_{min} = 3$)</p> <table border="1"> <thead> <tr> <th>0 items</th> <th>1 item</th> <th>2 items</th> <th>3 items</th> </tr> </thead> <tbody> <tr> <td>$\emptyset = 10$</td> <td>{sourcecode} : 8 {build} : 3 {test} : 4</td> <td>{sourcecode,test}: 3</td> <td>--</td> </tr> </tbody> </table>	0 items	1 item	2 items	3 items	$\emptyset = 10$	{sourcecode} : 8 {build} : 3 {test} : 4	{sourcecode,test}: 3	--
0 items	1 item	2 items	3 items						
$\emptyset = 10$	{sourcecode} : 8 {build} : 3 {test} : 4	{sourcecode,test}: 3	--						

Figure 3.1. (a) Example of a transaction database with code change types; (b) Frequent types of code change (along with their support) and minimum support of 3.

3.3 Research Method

We designed a quantitative study to answer the proposed research questions aiming at achieving our main goal. In this study, we rely on statistical modelling to understand the frequency of commit activity in repositories from mobile and non-mobile platforms. We build linear regression models to explain the frequency of commits in both platforms. We also make use of a frequent itemset algorithm to analyze the co-evolution of code changes made to three types of files: source code, build, and test files. The quantitative study is composed of three main phases: (1) software repository mining, (2) data collection via GitHub REST API, and (3) data analysis. Next, we detail each phase of the study.

Phase 1 - Software Repository Mining. We initially selected the 1000 most popular Java repositories in GitHub based on their number of stars. In this study, we consider number of stars a reliable proxy to the repository popularity, as previous studies already investigated this topic [Borges et al., 2016a,b]. We focus on Java systems due to constraints in our data analysis phase. For instance, we analyze the evolution of changes in build files, including files from the Apache Maven, which is a build automation used primarily for Java projects. The set of 1000 repositories includes different types of systems, such as tools, libraries and software systems designed for the Android platform. Repositories were retrieved from GitHub between July and August 2018.

Aiming at retrieving the most relevant repositories, we designed a filtering process

with two criteria. First, we keep only repositories with more than 1000 source lines of code (SLOC) as we believe systems with a lower number of lines may represent only toy samples. The second criterion is the number of commits in the last two years (time period of analysis). Since commits are our main source of information, we need repositories with a reasonable number of commits. Thus, we defined a threshold of 24 as the minimum number of commits repositories must have had within the last two years. This criterion also helps us to ensure repositories are being actively developed. From the 1000 repositories initially mined, 363 remained after the filtering process. We automatically classified these systems as mobile or non-mobile by locally cloning the repository and checking whether the *AndroidManifest.xml* file exists or not. In case the file exists, the repository was classified as mobile. Our final dataset contains 181 mobile systems and 182 non-mobile systems (desktop and Web applications). Next, Table 3.3 presents aggregate statistics regarding the 363 repositories that compose our dataset. We see the number of stars, source lines of code (SLOC), number of contributors, number of pull requests, and number of issues. In general, non-mobile systems have higher values for all items compared to mobile systems. As we can observe, the systems in our dataset are relevant as indicated the mean number of starts, which is above 6,000 for both platforms. In fact, the minimum number of starts confirms all systems are relevant according to this criteria (stars). Furthermore, regarding source lines of code (SLOC), we can see that systems in non-mobile platforms are larger than systems in mobile, with 152319.4 SLOC and 40706.21 SLOC, respectively.

Table 3.1. Aggregate statistics of the 363 repositories

		Mean	St. Dev.	Min	Median	Max
Mobile	Stars	6308.32	4573.52	2451	4710	24975
	SLOC	40706.21	191940.8	1003	7807	2367689
	Contribut.	43.73	64.65	1	21	351
	Pull Req.	9.54	16.69	0	3	84
	Issues	125.98	193.72	0	65	1640
Non- Mobile	Stars	6490.28	6426.73	2443	4548	41653
	SLOC	152319.4	295851.9	1418	48158.5	2729887
	Contribut.	96.69	94.05	1	64	400
	Pull Req.	30.06	62.73	0	9	521
	Issues	231.83	304.37	0	120	1730

Phase 2 - Data Collection. In this phase, we developed a script in R language to access and make requests to the GitHub REST API¹. In general, we collected data to be used in our regression models and in the code change co-evolution analysis. We

¹<https://developer.github.com/v3/>

collected the following data at repository-level to be used in our regression models and also useful for dataset characterization: number of contributors, number of pull requests, and number of issues. We added the number of source lines of code information obtained by a shell script, which analyzed the local cloned repository. Furthermore, we collected data at commit-level to be used by the mining algorithm to perform the code change co-evolution analysis. For each commit, we collected its date and whether it changed source code files, build files and/or test files. We identify changed files directly from the files list obtained through a request to the API. In addition, we retrieved additional information which may be useful for future studies within the context of our quantitative work. Next, we can see the additional data collected: number of changed files, added lines of code, deleted lines of code, and total changed lines of code (added lines + deleted lines).

Phase 3 - Data Analysis. The last phase of our study corresponds to the analysis of collected data in the previous step. We have two main parts of the data analysis. First, we use statistical modelling to address the first research question regarding the frequency of commits. Second, we apply a frequent itemset mining algorithm to check whether different types of code changes co-occur in commits made to the systems. Next, we detail how we proceed when building linear regression models and applying the mining algorithm.

Statistical Modelling. To provide evidence on whether being mobile influences the frequency of commits, we developed two multiple linear regression models, one with only the control variables and a full model with the indicator variable in addition to the control ones. The models were developed using the function `lm` in R. By controlling for confound factors in the multiple regression, we evaluate whether the difference in the frequency of commits (in mobile platform when compared to non-mobile platforms) can be attributed to the fact of a repository being mobile or not. Our hypothesis is that mobile systems have a higher frequency of commits since users from that platform (in our case, Android users) expect fast bug fixes and rapid availability of new features [Oliveira et al., 2018; Banerjee and Roychoudhury, 2016].

In our models, the response (dependent) variable is the number of commits per month, **nCommMonth**, which corresponds to the frequency of commit in a monthly-basis. We consider the following repository-level independent variables: size of the system in terms of number of source lines of code - **sloc** (we expect that larger systems have more commits), number of contributors - **nCont** (we believe a higher number of contributors mean the repository has more commits), number of pull requests - **nPR** (we expect that many pull requests can increase the number of commits, which will merge pull requests to the system), and number of issues - **nIssues** (a higher number

of issues will likely increase the number of commits, for instance, to fix bugs indicated by issues). Finally, we have an indicator (experimental) variable, **isMobile**, which is a binary variable that indicates whether a repository is mobile (1) or not (0), that is, if it has the *AndroidManifest.xml* file. Before building our models, we log-transformed variables aiming at stabilizing their variance and reduce heteroscedasticity [Zhang et al., 2018; Cohen et al., 2014]. We proceed in the following steps to build robust multiple linear regression models.

Step 1: Distribution comparison. We hypothesize that the number of commits per month (i.e., the frequency) is higher in the mobile platform than in non-mobile. To compare the distribution of our raw data regarding number of commits per month for both groups (mobile and non-mobile), we adopt the non-parametric Wilcoxon Signed-Rank Test. This test is suitable for our case since our data cannot be assumed to be normally distributed [Lowry, 2014]. We also report the Cliff’s delta to indicate the size of the difference of distributions.

Step 2: Additional explanatory power. Mobile systems may (apparently) impact the number of commits per month, but underlying confound factors might actually be leading the response of our models. Here we explore whether the fact of the repository being mobile add information to explain the frequency of commits. We build two successive regression models. We start with a model that contains only the control variables. After controlling for the confound variables, we include the experimental (indicator) variable in our model. We then use Cohen’s f^2 measure to gauge the effect size of the indicator variable. We consider model coefficients important if they are statistically significant at a 0.05 level.

Step 3: Multicollinearity diagnosis. To tackle possible problems related to multicollinearity [Farrar and Glauber, 1967] in our regression analysis, we diagnose our models, checking for multicollinearity. More specifically, we verify whether the variance inflation factor (VIF) [Allison, 1999] is below 3, which is a safe, conservative value that allows us to statistically confirm that our models do not suffer from multicollinearity [Zhang et al., 2018; Trockman et al., 2018].

Mining frequent itemsets and association rules. To find co-occurrences of different code changes file types along the last 2 years period, we apply Apriori, a frequent itemset mining algorithm [Borgelt, 2012; Pasquier et al., 1999; Agrawal et al., 1996, 1993]. As this mining algorithm is order-insensitive [Molderez et al., 2017], we believe it is suitable for our study since we do not need ordered data. In this step, we rely on the **arules** package in R. In our study, we analyze whether there are co-occurrences of source code changes, build changes, and test changes. To use the algorithm, we must convert our raw data (json) to the format the algorithm requires (transaction class).

Table 3.2. Heuristics for identifying build and test files

Heuristics		
	Begins with	Ends with
Test	<i>Test</i> OR <i>test</i>	<i>Test</i> OR <i>test</i> OR <i>Tests</i> OR <i>tests</i> OR <i>TestCase</i> OR <i>testCase</i>
Build		<i>pom.xml</i> OR <i>build.xml</i> OR <i>build.gradle</i>

After obtaining the co-occurrences of code changes, we are able to find association rules using the *Apriori* algorithm [Agrawal et al., 1996, 1993]. Therefore, based on a change the developer performs, we can suggest other types of changes according to the learned association rules. Using *Apriori* algorithm requires the specification of a support value, as explained in Section 3.2.

Table 3.2 presents an overview of the heuristics used to identify build and test changes. To identify build changes, we look for files with names as recommended by the build automation tools we use. For Apache Maven² build files, we search for *pom.xml*; for Apache Ant³ files, we look for *build.xml*; finally, for Graddle⁴ files, we search for *build.gradle*. Regarding changes on test files, we adapt an heuristic adopted by previous works [Zaidman et al., 2011; Levin and Yehudai, 2017]. We classify a change as a test change if the name of the class (in which the change was performed) begins with the word "Test" or ends with the word "Test", or "Tests", or "TestCase". We also consider a test change if the modified class is contained in a directory with the word "Test", "Tests", or "TestCase". Note that all situations in which the word is lower case are considered in the same way.

3.4 Results

In this section, we report the results obtained from the analysis of code change frequency, including the statistical analysis and the mining algorithm. We analyzed a total of 465,500 commits from 363 repositories hosted in GitHub.

²<https://maven.apache.org/>

³<https://ant.apache.org/>

⁴<https://gradle.org/>

3.4.1 Frequency of Commits

Figure 3.2 presents the frequency of commits (average number of commits per month per repository) in mobile and non-mobile platforms along a 2-year period. By inspecting this figure, we can see how the number of commits vary along the months for both platforms from a temporal perspective. We double checked the data to confirm the sharp drop in the end of the plot, which may be caused due to often discontinuation of mobile applications. Note that we divided the absolute number of commits per number of systems in each platform. Although we have one more system in the non-mobile group, the different numbers of systems do not mislead our interpretation of the average number of commits per system.

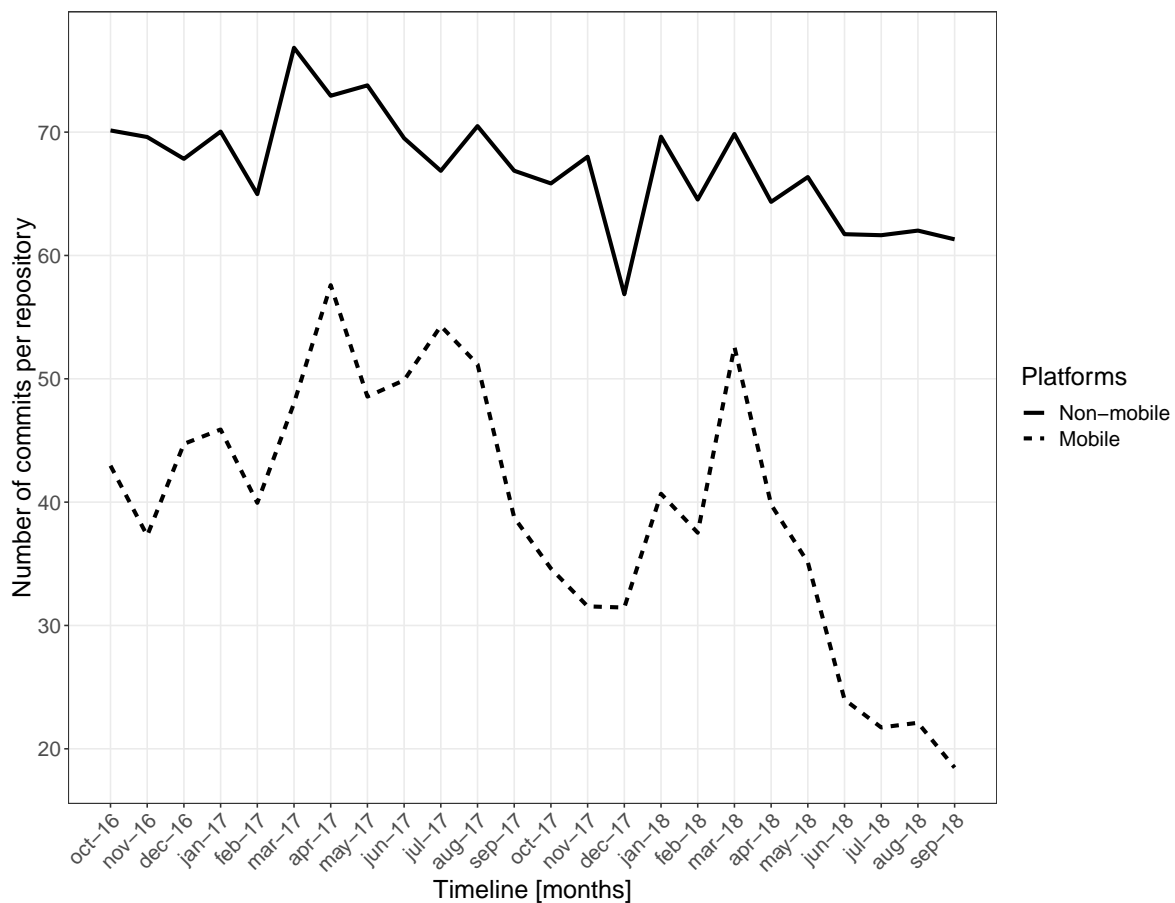


Figure 3.2. Frequency of commits in a 2-year time period.

We can observe in Figure 3.2 that the number of commits is always (in every month) higher in non-mobile platform compared to mobile. For the mobile platform, we can note a regular pattern in some periods of the year, specially in the holiday season, including Christmas and new year periods. For instance, in the period from

nov-16 to apr-17, the number of commits increased about 54%. The curve behaves similarly in the period from nov-17 to mar-18, with an increase of approximately 67%. This may suggest there are some factors influencing this behavior and contributions to OSS projects in that period of the year. Regarding those periods in non-mobile platforms, the increase in the number of commits was much smaller, with 4.8% for nov-16 to apr-17 and 2.7% for nov-17 to mar-18. However, we can observe that non-mobile platforms had a very low average number of commits in December 2017 (56 commits), which suggests that holiday season may influence the work activity in non-mobile projects.

This kind of temporal figure helps us to see the general picture of the situation, and how both platforms behave along the last 2 years. However, we still lack an explanation regarding what factors are really impacting the frequency. We developed multiple linear regression models to understand the impact of the platforms in the frequency when controlling for confound variables. Next, we analyze the results obtained by our models according to the methods and steps we proposed (Section 3.3).

Distribution comparison. Figure 3.3 presents the boxplots corresponding to the distribution of commits per month (response variable) for both platforms: mobile and non-mobile. By observing this figure, we can state that the two distributions are different. In fact, the median number of commits per month for mobile is approximately 63, while for non-mobile is 84.5. In addition, we obtained a Cliff’s Delta of -0.2181 (small), with a 95% confidence interval, indicating a small but statistically significant difference.

Additional explanatory power. Table 3.3 presents our model coefficients along with their p-values. From the model with only control variables, we can observe that most coefficients are in the positive direction (positive *T value*), as expected.

For instance, we expect that more contributors result in more commits per month. The same is valid for number of pull requests and number of issues. The unexpected results occurs for *sloc* coefficient, as its signal is negative. By inspecting the significance, apart from the intercept coefficient, all coefficients are not significant. We checked the correlation of the control variables with the response variable and in fact they are not highly correlated. The highest correlation value occurs for number of contributors and number of commits per month (pearson coefficient of 0.1992).

To gauge the effect of the indicator variable (*isMobile*), we build a successive regression model including the binary variable we defined to indicate whether a repository is mobile or not. In Table 3.3, we can see the coefficients of the full model. In fact, the indicator variable has a statistically significant impact on the frequency of commits in repositories (p-value $<2e-16$). The indicator variable also increased the explanatory

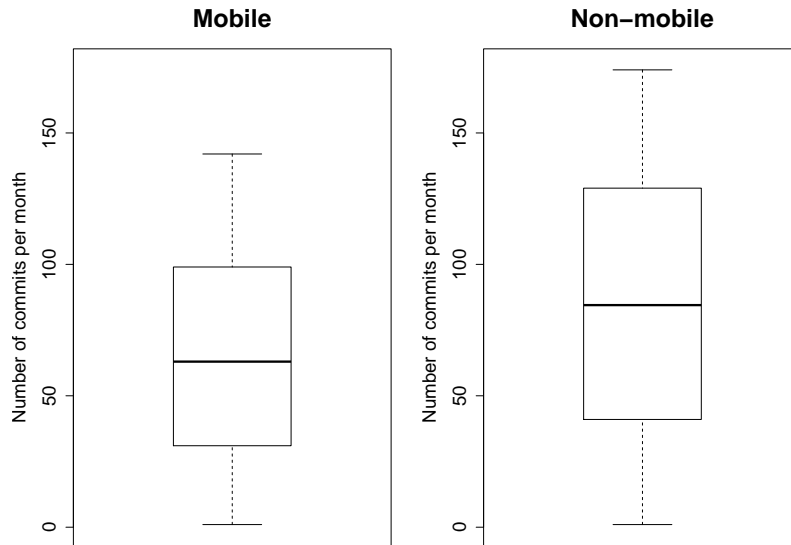


Figure 3.3. Distributions of response variable for mobile and non-mobile.

Table 3.3. Multiple linear regression coefficients for our two models.

	Variable	T value	P value (significance)
Control variables only	(Intercept)	61.02	<2e-16 ***
	nCont	1.283	0.2
	sloc	-0.224	0.823
	nPR	0.235	0.814
	nIssues	1.233	0.218
Full model, including indicator variable	(Intercept)	66.194	<2e-16 ***
	isMobile	-15.246	<2e-16 ***
	nCont	-1.19	0.235
	sloc	-1.525	0.128
	nPR	-0.117	0.907
	nIssues	0.751	0.453

***p <0.001, **p <0.01, *p <0.05

power of the model, as suggested by a proportional change in R^2 of 1,900% (from 0.02 to 0.4). We adopted the Cohen's f^2 measure to estimate the effect size of the indicator variable. Cohen's f^2 measure can be obtained by the following equation:

$$\frac{R^2_{contr+ind} - R^2_{contr}}{1 - R^2_{contr+ind}},$$

where the R^2 subscripts indicate the variables used in the model. We computed the R^2 for both models (controls and indicator, and only controls) and obtained a Cohen's

f^2 of **0.19**. The following thresholds are suggested⁵ to indicate the effect size: 0.02 (small), 0.15 (medium), and 0.35 (large) [Champely et al., 2018]. We can therefore conclude that the effect of being a mobile repository on the frequency of commits when controlling for confound variables is **medium**.

Multicollinearity diagnosis. We diagnose our models, checking for multicollinearity. Having highly correlated regressors in our models may inflate the variance. We first check the correlation between the predictors and then we get the variance inflation factor (VIF) for each predictor. Figure 3.4 presents a matrix-style image with the correlation between all pairs of predictors in our models. As we can see, predictors are not highly correlated (the highest correlation is 0.618 between number of pull requests and number of issues). Regarding the variance inflation factor, all variables have VIF values below 3, which is a safe value and indicate that our models do not suffer from multicollinearity. Next we can see the values for each regressor: nCont (1.6882), sloc (1.2904), nPR (1.5726), and nIssues (1.3440).

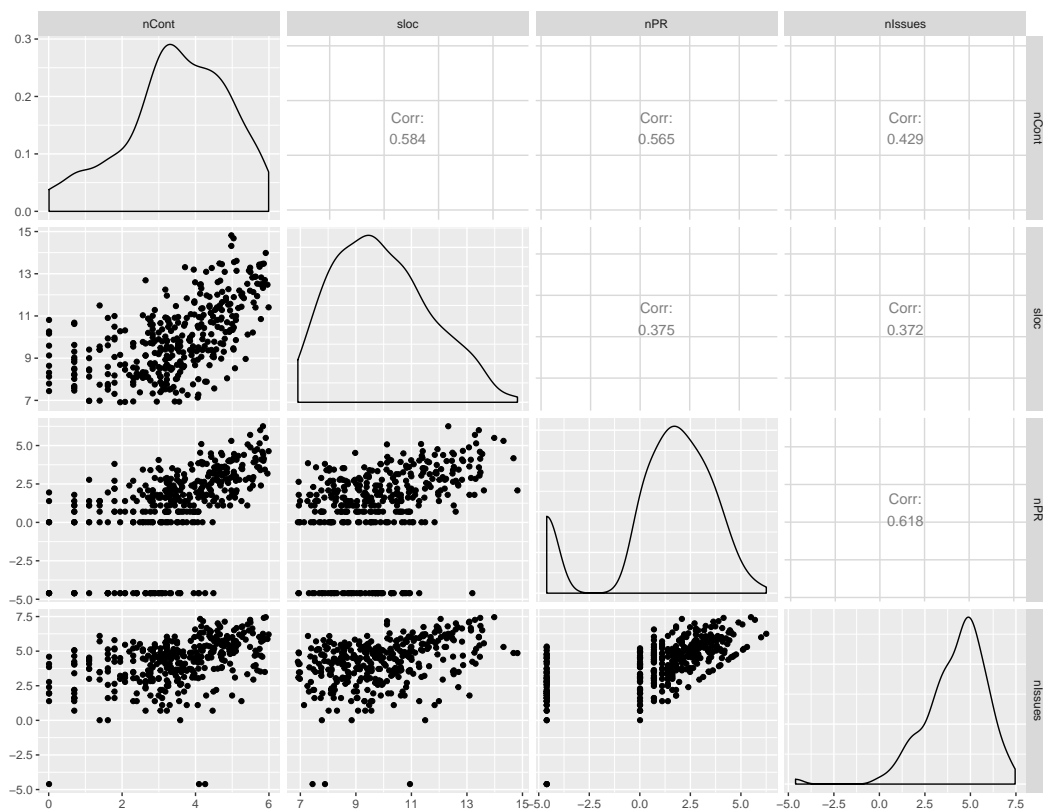


Figure 3.4. Correlations between predictor variables.

⁵https://en.wikipedia.org/wiki/Effect_size

Table 3.4. Frequent types code changes in all commits.

	Frequent change types	Support	Absolute count
Mobile	{build}	0.07081202	12166
	{test}	0.07848341	13484
	{source_code}	0.67038014	115176
	{source_code,test}	0.05435169	9338
Non-mobile	{test}	0.09728867	28573
	{build}	0.11002986	32315
	{source_code}	0.70661541	207528
	{source_code,test}	0.05106353	14997
	{source_code,build}	0.05217012	15322

3.4.2 Frequent Code Change Types and Association Rules

Regarding the types of frequent code changes, we analyzed the commit history using the *Apriori* mining algorithm to find types of code changes that frequently occur together and their association rules. We set a minimum support value of 0.05. Our minimum support must be a low value given the characteristics of our dataset, in which commits are much more likely to change a single type of file. More specifically, commits usually change only source code files (67% of mobile changes are source code changes and 70% of non-mobile changes are source code changes). Given these characteristics, we may expect that *support* metric values are low. Furthermore, when analyzing the association rules, we focus on the *confidence* and *lift* metrics to check the strength of the rules instead of *support*. Table 3.4 presents the code changes that occur together along with their support and absolute count values.

We can note that all types of changes, when considered individually, appear in the results returned by the algorithm. However, we focus only on types of changes that occur together with other types, that is, we analyze results where at least two types of changes appear. As we can observe, in mobile systems, the types of code changes that occur together (i.e., in the same commit) with a support greater than the minimum support is source code and test changes. This co-occurrence happened with a support of 0.054 (in 5.4% of all commits). The low support indicates that developers do not usually perform changes in source code and test files simultaneously, as we already discussed. Surprisingly, mobile developers do not usually change source code files together with build files as expected [Macho et al., 2017]. We performed some tests and found that source code changes occur together with build changes only with a support of 0.03. Regarding non-mobile platforms, we can see two co-occurrences of

types of changes. First, source code changes occur together with test changes with a support of 0.051, also a low support as for mobile. Second, source code changes also co-occur with build changes with a similar support, which is 0.052.

We also obtained association rules by applying the *Apriori* algorithm. For the association rules, we rely on default values of minimum support (0.001) and minimum confidence (0.8) defined by the `arules` package in R. We found the following rule for the mobile platform:

$$\{build, test\} \Rightarrow \{sourcecode\}$$

The association rule above has a support value of 0.00623, confidence of 0.9177, lift of 1.3689, and absolute count of 1070. This rule indicates that developers commonly perform changes in source code files given they changed build and test files. The low value of support shows that both sides of the rule (build and test changes, and source code changes) do not occur very frequently in commits. The high confidence value indicates that, given a scenario in which build and test changes are made (previous condition), a source code change is much likely to be necessary. The rule presented below was obtained for non-mobile platforms:

$$\{build, test\} \Rightarrow \{sourcecode\}$$

The association rule above has a support value of 0.01337, confidence of 0.8597, lift of 1.2166, and absolute count of 3927. Although the association rule for non-mobile is the same of mobile, we obtained slightly different values of support, confidence and lift. For instance, the support is 2.15 times higher in non-mobile than in mobile, which indicates that both sides of the rules (build and test changes, and source code changes) occur more frequently in commits of non-mobile applications. However, the confidence is lower than in mobile. This indicates that, although a source code change is likely to be necessary given that build and test changes were made, the strength of this statement (i.e., of the rule) is smaller compared to mobile platform.

3.5 Discussion

In this section, we discuss the obtained results to answer the research questions of our study.

RQ1: How frequent are code changes in mobile and non-mobile platforms?

We observed code changes (through commits made to the systems) are more frequent in non-mobile platforms when compared to mobile. By analyzing the trend graph over the past two years, we note that non-mobile repositories have more commits per month in all 24 months compared to mobile repositories. Furthermore, we can observe a regular pattern of change in the mobile platform, with a possible seasonal behavior. Our multiple linear regression models indicate that being mobile significantly impacts the frequency of commits when controlling for the following confound variables: number of contributors, size of system (in SLOC), number of pull requests, and number of issues.

Answering RQ1: Code changes are more frequent in non-mobile platforms compared to the mobile platform. Furthermore, being mobile significantly impacts (in the negative direction) the frequency of commits when controlling for confound variables.

RQ2: How is the co-evolution of source code changes, build changes and test changes in mobile and non-mobile platforms?

We obtained the most frequent co-occurrences of the target types of changes investigated in this study: source code changes, build changes, and test changes. By analyzing the results of the mining algorithm, we observed that source code changes evolve together with test changes in the mobile platform despite the low frequency of co-occurrence. When analyzing the co-evolution for non-mobile platforms, we observed that source code changes and test changes also evolve together, but source code changes evolve together with build changes as well, what, surprisingly, is not the case of mobile systems. Both co-occurrences also presented low values for the *support* metric, as in the mobile platform. Changing build files together with other changes in the system is essential for keeping build configurations synchronized with the rest of the system [Machó et al., 2017] and, therefore, developers should put efforts and be aware of the need of evolving build files together with the source code. Regarding the association rules, we found no surprising rule. We obtained the same rule for mobile and non-mobile platforms despite some slightly different values for support, confidence, and lift metrics. Both rules suggest that developers are likely to change source code files given they changed build and test files.

Answering RQ2: In the mobile platform, source code changes occur together with test changes, while in non-mobile platforms, source code changes occur together with both build and test changes. Both co-occurrences presented low frequency across all commits, which indicates that build and test changes do not co-evolve with source code changes. The association rules show that developers should likely change a source code file given they changed build and test files.

3.6 Threats to Validity

The quantitative study presented in this chapter has some limitations that could potentially threaten our results, as we explain next. First, the number of repositories in each group (mobile and non-mobile) may not be representative of the entire platform. However, to mitigate this threat, we selected very relevant projects among the top-1000 Java repositories. The dataset aggregate statistics presented in Table 3.3 shows the relevance of our projects.

Second, our study is restricted to Java repositories hosted on GitHub as we analyze changes made to build files, which are very common in Java projects. Therefore, our results may not generalize to other languages. Third, we analyze build changes related to build files from the following build automation tools: Apache Maven, Ant, and Graddle. Other tools may be included in the analysis and change the results. However, the selected build tools are largely used in Java projects and, therefore, we believe our analysis is reliable to some extent. Finally, despite trying to capture as many confounds as possible, we may not have considered all possible confounds in our regression models, which may affect our results and conclusions.

3.7 Final Remarks

This chapter presented a quantitative study aiming at understanding evolution practices in mobile and non-mobile platforms, that is, we seek to reveal how repositories from different platforms evolve. In Section 3.1, we presented our main goal and the proposed research questions. In Section 3.2, we detailed how frequent itemset mining algorithms work and how we mapped the problem of mining frequent itemset to our context of finding types of code change that frequently occur together. We presented our research method and steps in Section 3.3. The results were presented in Section 3.4

and discussed in Section 3.5. Finally, we discussed the limitations and threats to validity of our study in Section 3.6.

By analyzing the frequency of commits from a temporal perspective, we observed that non-mobile repositories have higher frequency of commits. Our statistical analysis through multiple linear regression models revealed that being mobile significantly impacts the frequency of commits (in a negative direction). By applying the Apriori mining algorithm to find types of code changes that frequently occur together, we observed that in mobile platform source code changes occur together with test changes with a low frequency. In non-mobile platforms, we found two groups of changes that occur together: (i) source code changes and test changes and (ii) source code changes and build changes. Both groups also have a low frequency of occurrence.

In the next chapter, we deepen our investigation on practices adopted in different contexts of software development by studying the adoption of development practices, such as testing practices. We found more suitable to investigate development practices in a finer-grained level rather than software platform, which led us to perform our analysis on software domains (e.g., e-commerce and healthcare). Therefore, we propose a qualitative study to reveal how development practices are adopted across different software domains.

Chapter 4

Qualitative Study

In this chapter, we present a qualitative study aiming to better understand which and how development practices are adopted across different software domains as we believe they have specific ways of applying development practices. In this study, we conducted 19 semi-structured interviews with cross-domain developers, i.e., developers who have worked in at least two different software domains. Afterwards, we run a Web survey to confirm or not the results obtained from the interviews. Section 4.1 presents the main goal and the research questions. In Section 4.2, we detail the methodology of the study. We present the results in Section 4.3 and discuss them in Section 4.4. In Section 4.5, we present the limitations and threats to validity. Finally, Section 4.6 concludes this chapter.

4.1 Goal and Research Questions

Our goal in this study is to understand how development practices vary in different software domains and whether there are specificities in their use, i.e., we aim to verify whether developers from different domains can adapt development practices to their specific context. We hypothesize that some domains may have similarities in the use of development practices and we also believe some domains may adopt practices in such a specific way that makes them very different from the others. However, works so far have not focused on the differences of domains regarding development practices. In addition, previous studies have not investigated the adoption of development practices based on the perception of cross-domain developers, as we do here. In this study, we adopt an exploratory and inductive research [Stol et al., 2016; Strauss and Corbin, 1990; Glaser and Strauss, 2017; Wohlin et al., 2012] to seek for differences and similarities of several practices across 13 domains. To guide our study, we defined the following

research questions:

- *RQ1: Which development practices are similar across domains?*
- *RQ2: Which development practices are specific to a domain?*
- *RQ3: Which factors may impact the adoption of development practices in different software domains?*

4.2 Research Method

We conduct a qualitative study to help us better understand how software development practices are used in different software domains. We follow an inductive research strategy, using a grounded, iterative approach to let development practice patterns of usage emerge from the interviews [Kitchenham et al., 2002; Wohlin et al., 2012]. This means we do not have previous categories to classify the use of development practices in different domains. To achieve our goal, we conducted semi-structured interviews with software professionals from industry with experience in multiple domains. As outline in Figure 4.1 and detailed later, the research methodology is composed of five stages: (i) participants selection in LinkedIn; (ii) interview design; (iii) conduct of interviews; (iv) transcription analysis; and (v) validation through a Web survey. The last stage was executed to confirm (or not) the main findings for domains in which we reached saturation. We noticed that 19 interviews were sufficient to gather interesting information regarding the adoption of development practices in different domains and to reach the saturation in three domains. In fact, previous interview studies performed a similar number of interviews, such as Murphy-Hill et al. [2014] (14 interviews), Stacey and Nandhakumar [2009] (20 interviews), Burger-Helmchen and Cohendet [2011] (8 interviews), and Dagenais and Robillard [2010] (22 interviews). We stopped conducting interviews in the following domains as new interviews were not bringing new information: banking (with 6 interviews), e-commerce (with 8 interviews), and healthcare (with 5 interviews). Therefore, in this study, we focus on presenting results from the aforementioned domains and we briefly indicate interesting findings from domains in which we have not yet reached the saturation, namely: oil and gas and social networks.

4.2.1 Interview Process

Our interview process is iterative and we use the open coding technique from grounded theory [Stol et al., 2016; Strauss and Corbin, 1990; Charmaz and Belgrave, 2007; Glaser

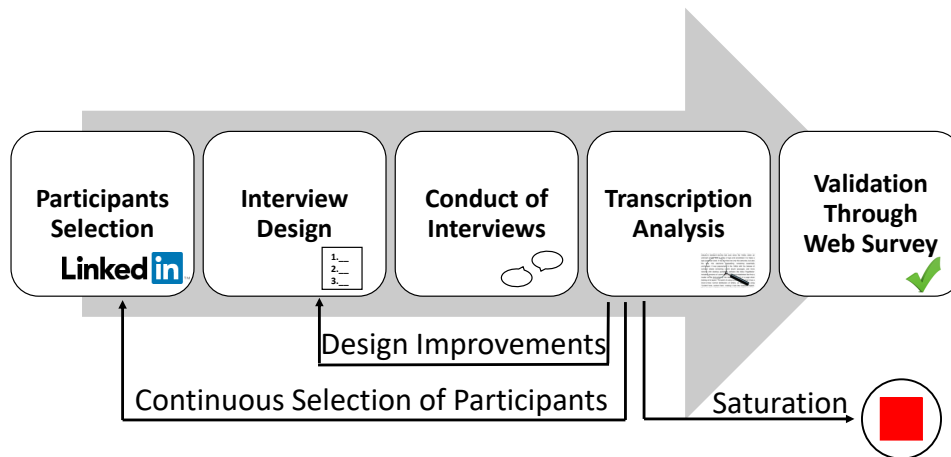


Figure 4.1. Our research methodology process.

and Strauss, 2017]. The interview phases are simultaneous, i.e., the stages overlap. For instance, while conducting interviews with some participants, we may also continuously select additional participants and iteratively build the interview script according to the previous interviews. Next, we describe each stage in detail.

Participants selection. We propose an innovative method to select the interview participants, which is an important contribution of our work. We selected only cross-domain developers, i.e., developers who have worked in more than one software domain. This selection criterion makes sure the developer has experienced more than one domain and, therefore, can confidently state the differences in the development practices' adoption. Table 4.1 presents information regarding the domains to which participants belong and years of experience with software development. We anonymously identify each participant by using the letter P followed by an identifier number (e.g., P1, P2, and so on until P19). On average, the interviewees have 11.7 years of professional experience and most participants hold at least one postgraduate degree, including master and doctorate. Most of the interviewees currently work or have worked as developers for large multinational companies, with thousands of employees and whose services and products reach millions of users, such as *Facebook*, *Google*, *Macy's*, *General Electric*, and *Petrobras*. In addition, the participants workplaces are distributed around the world, such as participants who are currently working in the United States, Canada and Brazil. Some participants have experience in three or even four domains and for such cases we decided to do the interview with respect to the domains in which developers have the most experience.

To check that participants were in fact cross-domain, we carefully and manually inspected their LinkedIn accounts and we selected only developers who have worked

Table 4.1. Interviewees information.

Participant	Experience (Years)	Domain 1	Domain 2
P1	20	Banking	Healthcare
P2	9	Accounting	E-commerce
P3	8	E-commerce	Social Network
P4	10	E-commerce	Education
P5	12	Healthcare	Oil and Gas
P6	10	E-commerce	Search Engine
P7	16	Banking	E-commerce
P8	9	Education	Healthcare
P9	11	Accounting	E-commerce
P10	25	Banking	Mining and Metals
P11	7,5	Games	Mining and Metals
P12	16	Banking	Games
P13	5	Aviation	Healthcare
P14	17	Banking	Stock Market
P15	7	Healthcare	Stock Market
P16	15	Business	Stock Market
P17	8	Accounting	Education
P18	8	Banking	E-commerce
P19	10	Accounting	E-commerce

in companies or projects within the targeted domains. In addition, developers should have at least 5 years of professional experience in total and 1 year of work within each domain. By following these criteria, we believe participants' statements are more confident regarding similarities and differences in adopted practices, which also brings more confidence to our results. We started with an opportunistic selection through a search in our LinkedIn contact lists. After that, we implemented an algorithm to automatically look for software developers from each domain by performing text analysis on LinkedIn profiles. The algorithm returns the developers' name and LinkedIn account, which were manually validated by the author and two collaborators: a Ph.D. student and a software engineering researcher. This double-check procedure helps to ensure that all participants meet the defined selection criteria. We contacted developers by email (when available anywhere online, such as on GitHub) or by the *LinkedIn InMail* functionality. The process of selecting cross-domain developers with experience in at least two domains for the interviews was very difficult. Finding cross-domain developers is even harder in some specific domains (e.g., aviation), as developers from these domains are highly specialized and usually do not have experience in other domain of our interest. We sent 62 emails to cross-domain developers we identified and validated in LinkedIn, and we received confirmations from 24 developers. However, 5 developers

declined later due to concerns regarding their companies' private information, even though we made it clear all the process would be anonymized and we were trying to understand general practices adopted. Thus, we interviewed 19 developers (response rate of 31%).

Interview design. To guide us during the interviews, we iteratively developed an interview script, which is composed of three main sections: background of the participant, general questions regarding differences in use of the software development practices, and specific questions regarding a set of practices, such as software testing and DevOps practices. Through the first section, we are interested in participants academic and professional background, such as the bachelor's degree, the highest academic degree, and years of experience. In the second section of the interview, we asked general questions regarding differences in the two software domains. In this part, we are interested in getting the participant's perception about the development practices in different domains without biasing our specific questions. Finally, in the last section, we asked specific questions about some development practices. In this section of the interview, we focus on the topics not mentioned by the interviewee in the second section in order to cover a broader set of all development practices. Our questions cover the following practices: *releasing practices* (e.g., regarding the deadlines of product releasing), *quality assurance* (which included test practices), *code review practices*, *continuous integration and delivery*, *version control practices*, and *practices related to the software architecture*, such as whether the team is aware and discusses architectural impacts caused by changes in the system [Paixao et al., 2017].

Conduct of the interviews. After the usual consent process with each participant, we start the interview, planned to last no more than 40 minutes. We observed that time frame was sufficient to do a concise interview, since we could collect all information we needed. We also recorded all interviews with the consent of the participants. Most interviews were conducted through a conversation on Skype. However, when the participant was not available due to agenda incompatibility, we sent out the interview by email, doing a follow-up whenever necessary (e.g., to better understand some responses).

Interview transcription analysis. The last stage of the interview process is the transcription and analysis of the interview to extract all relevant information. Here we used the open coding technique. We carefully analyzed the transcriptions and came up with the most relevant and groundbreaking topics stated by the interviewees, which were discussed afterwards by the author and two collaborators: a Ph.D. student and a software engineering researcher.

4.2.2 Validation

To check whether practices mentioned by interviewees are in fact broadly adopted by developers from each domain, we designed an online survey. It is important to note that we validated adopted practices only for software domains in which we reached the saturation, which occurred for the following domains: banking (6 interviewees), e-commerce (8 interviewees), and healthcare (5 interviewees). The survey is composed of two main sections: background (common to all surveys) and questions regarding a software domain (specific to the survey of each domain). Through the first section, we intended to collect information related to the participants background such as education, software development experience and development experience within the specific domain. The second section contains concise and objective statements that present characteristics and adopted practices within the domains, as indicated by the interviewees. In this section, the survey participant is asked to indicate the agreement with the statement through a Likert-type scale.

To find participants for the survey, we first mined repositories related to the target domains from GitHub in order to collect the names and emails from top-committers. We used specific search strings to make sure the repositories belong to the domains of our interest. To retrieve repositories from the banking domain, we used *bank*, and *banking* strings; for the e-commerce domain, we used *e-commerce*, *e-commerce* and *electronic commerce* strings; finally, for the healthcare domain, the following strings were used: *healthcare*, and *health*. Then, the repositories were manually validated to certify they are in fact software systems and they really belong to the domain. Finally, we automatically collected the name and email of top-committers (number of commits greater than 100) from repositories so that we could send the survey by email. We believe developers with more than 100 commits have sufficient knowledge within the domain and therefore are capable to answer our survey. With this procedure, we do not aim to find an extensive list of systems and committers. Instead, our goal is to find a representative number of developers with a good knowledge in the domain (top-committers) to answer our survey. We had to discard a large number of committers since they did not meet our criteria of 100 commits and possibly were not capable of confidently answering the survey.

In addition to this strategy, we searched for additional participants in LinkedIn since GitHub does not contain many popular repositories that meet our criteria (belonging to specific domains and developers with more than 100 commits). We looked for developers from the three domains within LinkedIn and sent the survey by email. After sending 329 emails, we received 40 complete responses from participants world-

wide (response rate slightly above 12%), being 14 for banking, 14 for e-commerce and 12 for healthcare.

4.3 Results

In this section, we report results observed in the interviews and validated in the survey. Note that we present interview quotes that are supported by at least three interviewees from different companies. In parallel, we report the percentages of agreement (or disagreement) of survey participants regarding each practice reported by interviewees. Figure 4.2 shows a summary of our main results obtained from the interviews, which are discussed in Sections 4.3.1, 4.3.2, and 4.3.3. Rectangles indicate domains and ellipses indicate practices or characteristics. Arrows indicate which domains are related to each practice or characteristic.

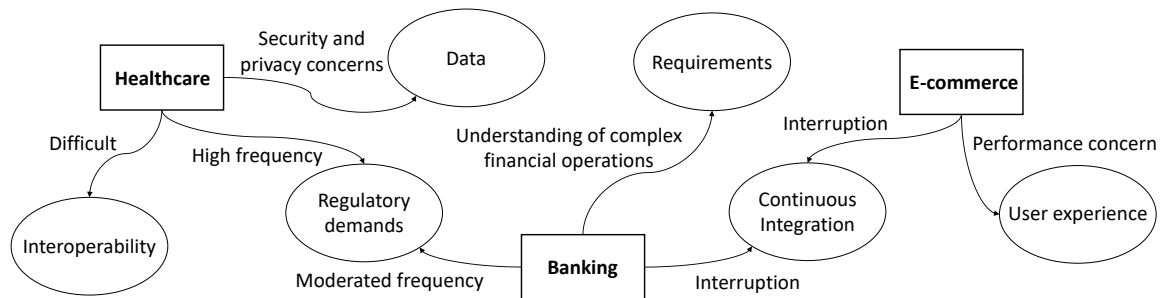












Figure 4.2. Main adopted practices in domains. Banking domain is moderately regulated and interrupt continuous integration process in important commerce periods (e.g., Black Friday); e-commerce follows an user-centered development, focusing on non-functional requirements that provide a good user experience and also interrupt continuous integration process; and healthcare is highly regulated, focuses on patient data privacy and security and requirements elicitation may be easier than in other domains.

Table 4.2 presents an overview of the survey results for the statements regarding characteristics and adopted practices in each domain. The first column shows the statements presented to participants. Each statement is identified by a unique label. For instance, we use *S1.B* to identify the first statement of the banking survey. The second column presents the Likert distribution of the participants agreement regarding each statement. The Likert-type scale varies from *completely disagree* (score 1, graphically in the left) to *completely agree* (score 5, graphically in the right).

Table 4.2. Survey results with presented statements and Likert-scale agreement distribution.

Banking	
S1.B - Code changes are less frequently released in periods of the year when large financial transactions are performed.	
S2.B - The banking segment is moderately regulated, many times requiring changes in the system to fulfill regulatory demands.	
S3.B - Requirements elicitation is hard because it involves the understanding of complex financial operations.	
E-commerce	
S1.E - Code changes are often not released in periods of high amount of sales, such as in Black Friday and Christmas.	
S2.E - This segment focuses on user-centered non-functional requirements, such as usability, security and performance.	
S3.E - Code is pushed into production with less frequency compared to other software segments.	
Healthcare	
S1.H - The healthcare segment is highly regulated, with frequent legal demands.	
S2.H - Requirements elicitation is relatively easier compared to other segments	
S3.H - Interoperability of systems from different workplaces is usually difficult.	
S4.H - Privacy, reliability and security of patient data are major concerns in healthcare software.	

4.3.1 Banking Domain

Continuous integration interruption. Interviewees from the banking domain often mentioned they are more careful with dates when financial transactions increase, mainly in the end of the year and beginning of each month. As a contrast to other domains, banking developers pointed out that in such periods they do not release large code changes to the servers, interrupting the continuous integration process in order to avoid inserting bugs in the systems during critical periods. They also stated that the priority is to fix bugs. Participant P10 said the following quote (supported by P1, P7, and P12):

Most of the banks have a freeze period about 30 days before the new year, when just emergency software updates are allowed.

Besides Black Friday and new years period, participants also mentioned that development is modified prior to salary payments, when the traffic is usually high. Below, a quote from P18 with support of P1 and P7 (note that salaries are paid in the beginning of the month in the participant's country, which may not be the case for other locations):

We usually do not release large code changes to the server in the first days of the month, just before salary payments.

According to the banking survey, 58.3% of participants agree with this practice (scores 4 and 5), while only 16.6% disagree (scores 1 and 2). In addition, 25% of participants were neutral (score 3). The survey responses indicate that developers usually adopt this practice in banking development.

Moderated regulatory demands. Developers also highlighted that banking systems are regulated by legal demands that come from the government, which is not common in other software domains. Hence, one common practice is to change the code to comply to a regulatory demand, such as stated by P10 and supported by P1 and P4:

Most of the time was used to enhance an existing feature, add a new one or comply to a regulatory change.

According to the banking survey, 83.3% of participants agree with this practice (scores 4 and 5), while only 16.7% are neutral (score 3). No respondents disagrees. The responses indicate that in fact the banking domain requires specific changes (besides usual ones) to comply to regulatory demands.

Overly complex requirements. Another characteristic of the banking domain is regarding requirements engineering practices. Developers from banking often said that understanding what stakeholders really want may be difficult due to the context where the system will operate, many times requiring the understanding of financial terms. Participant P10 mentioned (supported by P14 and P18):

...it is hard to understand and put everything together because it involves abstract, complex, and structured financial operations.

According to the banking survey, 83.4% of participants agree with this practice (scores 4 and 5), while only 8.3% are neutral (score 3) and 8.3% partially disagree (score 2). The responses strongly indicate the high complexity of requirements elicitation in the banking domain.

4.3.2 E-commerce Domain

Continuous integration interruption. Similarly to the banking domain, e-commerce developers also adopt practices of interrupting continuous integration, according to our interviewees. However, in this case, software development is oriented to commerce important dates, when the amount of sales increase. In such periods, participants mentioned that the priority is to fix bugs and give the best experience for users. Therefore, developers may change their usual continuous integration practices (i.e., they stop sending large changes to the servers) aiming at focusing on the most important tasks, such as bug fixing. A quote from P6 supported by P2, P4, P7 and P9:

We have code freezes a few weeks prior to Christmas holidays seasons, when only critical or major bug fixes could be introduced. A week prior to the holidays absolutely no code was checked in unless critical to the business.

According to the e-commerce survey, 83.4% of participants agree with this practice (scores 4 and 5), while only 16.6% disagree (scores 1 and 2). The high agreement percentage suggests this practice is in fact widely adopted by e-commerce developers.

Focus on user experience. According to e-commerce interviewees, developers give a special attention to specific user-centered non-functional requirements, mainly performance, usability and security. According to them, the user-focused development aims at providing the best user experience as possible, since a low performance system may prevent the user from concluding a purchase. E-commerce development practices include stress tests to guarantee the system will provide a good experience for users. P3 mentioned (supported by P6, P18 and P19):

Performance is critical for user experience. We have stress-test environments where the numbers are pushed to limits (visits, users, transactions, and many other metrics that could be extrapolated).

According to the e-commerce survey, 50% of participants partially agree with this practice (score 4), while 50% are neutral (score 3). This may indicate that focusing on the user-experience is a generic characteristic, being important in other domains as well.

Less frequent continuous delivery. Finally, interviewees also mentioned that continuous delivery is less frequent in e-commerce development in comparison to other software domains, since code changes are extensively tested before being put into production. This happens to make sure no bug would be inserted into the system, which

could cause a bad experience for the user and reduce the number of visitors of the online store. P6 stated (supported by P2 and P4):

We had less frequent pushes to production in e-commerce domain due to extensive code change tests.

According to the e-commerce survey, 41.6% of participants disagree with this practice (scores 1 and 2), while 25% are neutral (score 3) and 33.3% partially agree (score 4). The high percentage of disagreement and neutrality may indicate that this practice is not commonly adopted in the e-commerce domain and it may only reflect interviewees experience within their companies. Looking at this results, we believe that less frequent continuous delivery is strictly related to companies policies and culture, as it may reflect the personal experience of interviewees who mentioned that.

4.3.3 Healthcare Domain

Frequent regulatory demands. The healthcare domain is a well-established and largely known software domain within both academia and industry. This domain has peculiarities that differ it from the others, such as the regulations that health systems usually must follow [Richardson et al., 2016; Roed and Ellingsen, 2011]. In fact, interviewees from the healthcare domain corroborate with this belief. For instance, they mentioned that when a legal demand arrives, the developer team needs to focus on implementing this new demand, giving it the highest priority. P5 stated (supported by P1, P8 and P13):

Health domain is more regulated and oriented by legal demands, which come with a preestablished date.

According to the healthcare survey, 70% of respondents agree (scores 4 and 5) with this characteristic. More specifically, 60% completely agree with it, indicating that in fact healthcare software is highly regulated. Scores 1, 2, and 3 received 10% of responses each.

Clearer requirements. Regarding the requirements engineering practices, it is common believed that this phase of the software development is really difficult and complex [Yost et al., 2016]. However, participants from the healthcare domain contradicted this belief, claiming that requirements elicitation in healthcare domain is not as difficult as in other domains, such as Oil and Gas (pointed out by P5) and banking (stated by P10). As they said, despite the common lack of time of health professionals, the requirements in this domain are clearer due to the (usually) higher qualification of

health professionals (e.g., medical doctors). Therefore, such professionals can easily understand and keep a conversation with IT professionals, making the requirements elicitation relatively easier and clearer, as P5 mentioned (supported by P1 and P13):

Requirements are clearer in healthcare due to the higher qualification of health professionals (medical doctors).

According to the healthcare survey, 50% of respondents disagree (scores 1 and 2) with this characteristic. In addition, 10% are neutral (score 3) and 40% agree (scores 4 and 5). This agreement distribution indicates that requirements elicitation in healthcare domain may be strictly dependent on the personal experience of developers and the health companies for which they have worked. Therefore, it may reflect a characteristic of the companies' policies and culture, instead of an intrinsic characteristic of the healthcare domain itself. Furthermore, we believe that age and maturity of companies strongly influence requirement engineering practices, as older companies may have acquired experience with requirements elicitation, making it easier as indicated by interviewees.

Difficult interoperability. Interviewees from healthcare also mentioned the difficult they usually face regarding interoperability practices of systems from different companies. For instance, even though there are some standards, hospitals may have surprisingly different information patterns, which difficult the communication among them, as P5 mentioned (supported by P1 and P8):

Although there are standards, hospitals, for example, rarely switch information because they have different information formats.

According to the healthcare survey, 70% of respondents agree (scores 4 and 5) with this practice. In addition, 30% are neutral (score 3). The responses indicate that in fact interoperability is a challenge in the healthcare domain.

Data security and privacy concerns. Healthcare participants often mentioned the importance of reliability, privacy and security regarding patient data. The whole development process is concerned with the patient data, always trying to keep them reliable in order to avoid possible serious consequences. For instance, participant P5 stated (supported by P1, P13 and P15):

If I switch patient data, I can give wrong diagnoses and prescribe wrong medications.

According to the healthcare survey, 70% of respondents agree (scores 4 and 5) with this practice, while 30% are neutral (score 3). The responses suggest that developers in fact consider data security and data privacy major concerns in the healthcare software development process.

4.4 Discussion

In this section, we discuss the results obtained from the interviews and from the survey. It is important to note that we answer the research questions based on practices and characteristics from domains in which there was agreement between the interviewees and the survey participants. This discussion gives more confidence to our conclusions as a broader and more diverse set of developers agree with that practice.

RQ1: Which development practices are similar across domains?

We noticed that both banking and e-commerce domains share a common practice of interrupting the continuous integration process in periods of the year when the amount of sales increase, such as Black Friday and Christmas. Furthermore, regulatory demands are common in the banking and healthcare domains, usually requiring efforts from the development team to implement changes into the system to comply to regulatory requirements.

Answering RQ1: We found two similarities of practices across domains. First, continuous integration practices are adopted in a similar way in the banking and e-commerce domains, which suggests that other financial-related domains may also follow this practice. Second, regulatory-driven changes are common in the banking and healthcare domains, which must adapt their workflow to comply to regulatory demands.

RQ2: Which development practices are specific to a domain?

Requirements elicitation in the banking domain is different from the other domains we investigated, since an understanding of complex financial operations is necessary to precisely capture requirements needs. The healthcare domain is different from other domains regarding interoperability. For instance, many health companies may have different information standards, which may hinder information switching between companies. Other domains (e.g., mining and metals, banking, and oil and gas) have widely used standards that ease information switching whenever necessary.

Answering RQ2: We found two main practices specific to domain. First, requirements engineering practices are adopted in a unique way by the banking domain, involving the comprehension of complex financial operations. Second, practices related to interoperability are more difficult in the healthcare domain in comparison to others, due to different standards used by health companies.

RQ3: Which factors may impact the adoption of development practices in different software domains?

Through the third research question, we are interested in capturing the main factors that can influence which development practices the companies adopt. Based on our interpretations of the interviews, we noticed that the company's policy and culture play an important role when deciding about the software development process. Many times, the software engineering team is required to follow specific practices due to the company way of work. For instance, as we already discussed, we identified that less frequent continuous delivery practices in e-commerce and requirements engineering practices in healthcare resulted from companies' policies and culture. Furthermore, the age and maturity also have a strong impact on adopted practices. We realized that companies may change or adapt practices throughout the years, also as a result of the emergence of new technologies and development processes.

Answering RQ3: In addition to the software domain, the companies' policy and culture are important factors that guide the development process, therefore impacting the adopted practices. Moreover, age and maturity also may influence the practices' adoption and their way of use.

4.4.1 Implications for Practice

In this section, we elaborate on the three main practical implications our results can have based on the joint analysis of the interview findings and the survey responses. First, companies should provide targeted training for their employees, not only software developers, but also training for people from other positions (e.g., software architect and technology leader). The training should focus on specific domains' characteristics and how development practices are adopted within the company's domain.

Second, professionals should update themselves regarding which and how practices are adopted in domains, specially if they are looking for a new job. Understanding

how the companies of interest apply development practice may increase the chances of success in a job position application. For instance, developers who work (or intend to work) with banking software should understand (at least basic) financial operations as this may strongly aid the requirements elicitation.

Third, software engineering education professionals should consider specificities of different software domains. We believe new teaching approaches that consider the domain should be investigated. For instance, new specific undergraduate or graduate courses may be interesting. Interdisciplinary courses may also be a good idea, as Richardson et al. [2016] recently suggested an interdisciplinary course of software engineering for healthcare systems.

4.4.2 Contrast with Current Beliefs

Continuous integration may not always be continuous in some domains. This practice has emerged recently aiming at automating the compilation, building and testing of code, with weekly and even daily integration [Vasilescu et al., 2015; Ståhl and Bosch, 2014; Elbaum et al., 2014] and some studies have investigated continuous integration flexibility, costs and benefits [Hilton et al., 2017, 2016]. Most developers keep adopting this practice based on how everyone uses, but little research has investigated whether there are differences in continuous integration usage. Surprisingly, we identified that developers from banking and e-commerce (i.e., financial domains) usually interrupt continuous integration in critical commerce periods, such as Black Friday, aiming at avoiding inserting subtle bugs in the systems, which would be catastrophic for the company. We did not identify this practice in the other domains we investigated at all, suggesting it possibly is exclusive from financial domains.

4.4.3 Results for Other Domains

In this section, we present other interesting findings from the interviews in domains in which we did not reach the saturation. Therefore, these results provide insights regarding some domains and we emphasize the need for further investigation focusing on these specific domains.

Releasing practices flexibility in Social Network and Search Engine domains.

Interviewees from social network and search engine domains often mentioned the flexibility they usually have regarding many aspects, such as the release deadlines. We may expect that software development has extremely strict deadlines of releasing a prod-

uct, as indicated by interviewees from banking and e-commerce domains. However, this seems not to be the rule for social network domain, as participant P3 said:

...developers prioritize product and technology excellences. There is less pressure for the deadline itself.

Participant P6 reported how developers are assigned to the projects. We may expect developers are told what they need to develop and they just do it. However, a common practice in social network systems is that developers have the freedom to choose the project and the feature they work on, as P6 mentioned:

I have complete freedom to choose what kind of project I'm going to work on, what I want to do.

Although domains usually have a dedicated testing team, such as in banking (stated by P1) and e-commerce (stated by P7), interviewee P3 pointed out, as a contrast to other domains, that tests are performed by the developers themselves in the social network domain. More specifically, the developer who implemented a feature, for example, is responsible for testing it. This code-owner based approach has been adopted only recently in systems with modern architectures, such as microservices [Jamshidi et al., 2018; Prechelt et al., 2016]. Therefore, the adoption of this practice may be a result of architectural decisions in this domain, as quoted from P3:

...there is no test team. The developer is responsible for creating all integration, Web-driven, and unit tests.

Finally, we concluded that social network and search engine domains are quite peculiar, presenting unexpected management practices (decisions about the projects in which developers work and deadline policy) and test practices.

Autonomous fault-recovery in Oil and Gas domain. One participant from the oil and gas domain pointed out that this domain must take into account the need for an autonomous fault-recovery module, which is present during the entire development process, from the requirements until the delivery and operation. In addition, the software system must be extremely robust, given the environmental conditions of operations (e.g., an oil platform in the middle of the ocean). One of the reasons behind these needs is that the systems remain physically inaccessible for a long period of time, since professionals do not have continuously access to the location where the software is deployed, which is uncommon in other domains (e.g., healthcare as pointed out by P13). Remote connections may also be difficult given the location of the system. A quote from P5:

Oil and Gas requires more robust and autonomous solutions since the system is hard to reach for a long period of time.

Furthermore, the interviewee from the oil and gas domain mentioned that software systems from this domain must follow a set of well-defined standards from the industry. In contrast to other domains, such as healthcare, in which interchangeability is hard due to the particularity of each health company, oil and gas systems from different companies can easily communicate with each other. As stated by the participant, the adoption of these standards impacts code components dependencies and, therefore, the system's architecture.

4.5 Limitations and Threats to Validity

The qualitative study presented in this chapter has some limitations that could potentially threaten our results, as we explain next. For instance, one may point a company from a domain we investigated and may say the company does not adopt the practices as we presented. However, our findings are based on interview participants' perceptions and their experience, and therefore our results may not generalize to all companies, as each one can adopt development practices based on its own culture and policy. Note that, in this study, we focus on large companies, such as *Facebook*, *Google*, *Petrobras*, and *Macy's*. Therefore, our results may not hold for small companies possibly with informal software engineering processes. This kind of limitation is characteristic of qualitative studies, as previously studies [Begel and Zimmermann, 2014; Lo et al., 2015]. However, Flyvbjerg [2006] demonstrated that even individual cases (i.e., studies limited to one company) contributed to discoveries in several fields, such as physics and social sciences. Therefore, even within a limited context of a few companies and participants, we believe our results can impact how companies from the studied software domains can adopt development practices.

Another limitation of our study is related to our methodology for finding cross-domain developers. We rely on a semi-automated search for interview participants, manually validating LinkedIn profiles returned by an algorithm we implemented. However, we may have misclassified developers as cross-domain (e.g., assigning a domain in which the developer has never worked). This may have caused a reduction in the response rate for the interview since there would be wrong information regarding the domains in which the developers we contacted have worked. To mitigate such issue, we have performed a double check for each participant before contacting them.

Finally, one may point that our interview results are based only on participants personal experience. However, we selected practitioners with a diverse background. This scenario composed of several large companies and different work locations bring more generalization to our results since we believe that biases (e.g., from a specific sort of company or a specific location) are attenuated. In addition, the Web survey collected responses from developers worldwide with different backgrounds, which supports our interview results regarding adopted practices within domains.

4.6 Final Remarks

This chapter reported a qualitative study aiming at understanding which and how development practices are adopted across different software domains. In Section 4.1, we presented our main goal and the research questions. Section 4.2 presented the adopted research method. Sections 4.3 and 4.4 presented the results we observed from the interviews and validated by the Web survey. In Section 4.5, we discussed the threats and limitations of our study.

By analyzing the data collected from the interview together with the survey data, we observed that two practices are commonly adopted across different domains: first, continuous integration practices are similarly interrupted in the banking and e-commerce domains, suggesting this is a characteristic of financial-related domains; second, regulatory-driven changes are common in the banking and healthcare domains. Furthermore, we also noticed two practices specific to domains: first, requirements engineering practices are adopted in an unique way by the banking domain; in addition, practices related to interoperability are more difficult in the healthcare domain in comparison to others. Finally, we also observed that the company's policy and culture strongly influence the adoption of development practices. In the next chapter, we present the final considerations, in which we conclude this dissertation, present the lessons learned from both quantitative and qualitative studies, highlight the main contributions of this work and discuss insights for future works.

Chapter 5

Final Considerations

Understanding which practices are adopted (and how they are adopted) by developers when developing software is important as insights can be provided about how professionals are in fact working. Revealing how practices are adopted may have several benefits, such as supporting newcomers in OSS projects and supporting companies that intend to provide training for their employees. We also must take into consideration the context in which software is being developed. For instance, practices may differ when we compare different software platforms (e.g., mobile and desktop). As previously investigated by Murphy-Hill et al. [2014], software engineering should not be considered as a uniform whole given its diversity, with different practices being applied in several domains and involving different people in the process of software development. In this chapter, we present the final considerations regarding this dissertation. We first conclude our work by summarizing our motivation, goal, methodological procedures and results achieved. Then, we discuss the main contributions of this dissertation. Finally, we give directions for future work.

5.1 Conclusion

We identified a research gap regarding the adoption of development and evolution practices. More specifically, we noticed that we still lacked empirical evidence about how evolution practices are adopted in different software platforms. Although some past works have investigated the differences between platforms [Zhou et al., 2015; Basole and Karla, 2011], such as Android, iOS, and desktop, and others have studied software evolution through code changes [Macho et al., 2017; Levin and Yehudai, 2017; Molderez et al., 2017], no study has investigated differences and similarities between software platforms with regard to evolution practices (concerning code changes). Furthermore,

we still lacked a more comprehensive, exploratory study regarding development practices adoption in software domains. Past works have studied specific characteristics of single domains, such as e-commerce [Segura et al., 2014], healthcare [Richardson et al., 2016], and banking [Russo et al., 2017], but no study has compared the adoption of development practices across several domains.

To fill these research gaps, we proposed a mixed-methods research. First, we conducted a quantitative study aiming at understanding how evolution practices are adopted in different software platforms. In this study, we focused on mobile and non-mobile platforms. For mobile, we considered Android applications, and for non-mobile, we considered desktop and Web applications. We mined 363 repositories from GitHub and analyzed a total of 465,500 commits. Our analysis on the frequency of commits indicated that non-mobile repositories have a higher number of commits per month compared to mobile repositories. We also built multiple linear regression models to explain whether being mobile impacts on the frequency of commits while controlling for confounds. Our models suggested that being mobile significantly impacts the frequency of commits in a negative direction (lower frequency than non-mobile). In addition, we highlight that developers aiming at contributing to mobile repositories should be aware that source code file changes usually require test file changes.

Second, we performed a qualitative, exploratory study aiming at understanding and revealing the similarities and differences of software domains regarding development practices. We conducted 19 semi-structured interviews with cross-domain developers; that is, developers who have experience in at least two domains. For the transcription phase, we used a Grounded Theory technique called open coding. We also run a Web survey to validate the interview findings in domains in which we reach the theoretical saturation. Our results suggest that two practices are commonly adopted across different domains: interruption of continuous integration practices (in the banking and e-commerce domains), and regulatory-driven changes (in the banking and healthcare domains). Furthermore, we also noticed two practices specific to domains: requirements engineering practices are adopted in an unique way by the banking domain; also, practices related to interoperability are more difficult in the healthcare domain in comparison to others. Finally, we also observed that the company's policy and culture may influence the adoption of development practices.

We also identified that some results from the quantitative study are in accordance with results from the qualitative work. According to our quantitative analysis, the frequency of commits is always higher in non-mobile repositories compared to mobile. We did not clearly identify a similar pattern when comparing both platforms (that is, the curves do not necessarily follow the same pattern). However, we can observe a peculiar

behavior in the holiday season (including Christmas and new year period) for both platforms. For instance, between October 2017 and January 2018, there is a sharp decrease in the number of commits. This finding is somewhat in accordance with the interview results. Most interviewees from financial domains (e-commerce and banking) mentioned they usually interrupt continuous integration/delivery in the holiday season to avoid inserting subtle bugs. This results calls for a further investigation on repositories from the financial domain hosted on GitHub. From these results, we can learn that this practice of interrupting continuous integration/delivery, largely adopted in proprietary software from financial domains, may also be adopted in OSS projects as well.

5.2 Contributions

We believe this dissertation have important contributions to the software engineering research community and industry. Next, we present our main contributions.

- A statistical modelling specifically to investigate the frequency of commits in GitHub repositories. We provided multiple linear regression models to explain frequency of commits in mobile and non-mobile repositories while controlling for confound factors. Our models can be applied to any set of repositories hosted on GitHub.
- A mapping of the problem of mining frequent itemset across transactions in a database to finding frequent types code changes across commits.
- A novel method for selecting interview participants. We provided a procedure to find cross-domain developer, that is, developers who have worked in more than one software domain. We believe our method can be adapted to other contexts, such as cross-platform developers.
- An interview script for that can be used to conduct semi-structured interviews about development practices.
- A concise Web survey with the main development practices adopted in different software domains. We provided a customized survey for each domain.

5.3 Future Work

With this dissertation, we could identify several research directions for future works that can complement and expand our work. In the quantitative study, at this moment, we consider three automation build tools. As a next step, we propose to include other build automation tools so that more build files can be identified in the analysis. We also highlight the need for further investigation regarding the frequency of commits, such as performing a time-series analysis on the trend graph of frequency. This kind of analysis may help researchers and practitioners to understand and explain the behavior of mobile and non-mobile repositories along a period of time. Still related to the quantitative study, it may be interesting a more detailed comparison of different platforms using a finer-grained level, such as comparing Android, iOS, desktop, and Web. Finally, we intend to investigate whether changes on different types of files evolve together with different semantic code changes, such as *method added* and *statement removed*.

Regarding future work for the qualitative study, it is important to complement the work by conducting more interviews with a focus on domains in which we did not reach the saturation, but interesting information was collected, such as for social networks and oil and gas domains. In addition, highly specialized domains that potentially have interesting practices need focused studies as well, such as aviation. Finally, it would be interesting to investigate whether exists a close relationship between software domains and platforms.

Bibliography

- Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207--216. ACM.
- Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A. I., et al. (1996). Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307--328.
- Allison, P. D. (1999). *Multiple regression: A primer*. Pine Forge Press.
- Banerjee, A. and Roychoudhury, A. (2016). Automated re-factoring of android apps to enhance energy-efficiency. In *Mobile Software Engineering and Systems (MOBILE-Soft), 2016 IEEE/ACM International Conference on*, pages 139--150. IEEE.
- Basole, R. C. and Karla, J. (2011). On the evolution of mobile platform ecosystem structure and strategy. *Business & Information Systems Engineering*, 3(5):313.
- Begel, A. and Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In *36th Int'l Conference on Software Engineering*.
- Bhattacharya, P., Ulanova, L., Neamtiu, I., and Koduru, S. C. (2013). An empirical analysis of bug reports and bug fixing in open source android apps. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 133--143. IEEE.
- Borgelt, C. (2012). Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6):437--456.
- Borges, H., Hora, A., and Valente, M. T. (2016a). Predicting the popularity of github repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, page 9. ACM.

- Borges, H., Hora, A., and Valente, M. T. (2016b). Understanding the factors that impact the popularity of github repositories. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 334--344. IEEE.
- Breu, S., Premraj, R., Sillito, J., and Zimmermann, T. (2010). Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301--310. ACM.
- Burger-Helmchen, T. and Cohendet, P. (2011). User communities and social software in the video game industry. *Long Range Planning*, 44(5-6):317--343.
- Champely, S., Ekstrom, C., Dalgaard, P., Gill, J., Weibelzahl, S., Anandkumar, A., Ford, C., Volcic, R., De Rosario, H., and De Rosario, M. H. (2018). Package ‘pwr’.
- Charmaz, K. and Belgrave, L. L. (2007). Grounded theory. *The Blackwell encyclopedia of sociology*.
- Cohen, P., West, S. G., and Aiken, L. S. (2014). *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press.
- Dagenais, B. and Robillard, M. P. (2010). Creating and evolving developer documentation: understanding the decisions of open source contributors. In *18th Int’l Symposium on Foundations of Software Engineering*.
- Elbaum, S., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *22nd Int’l Symposium on Foundations of Software Engineering*.
- Fairbanks, G., Bierhoff, K., and D’Souza, D. (2006). Software architecture at a large financial firm. In *21st symposium on Object-oriented programming systems, languages, and applications*.
- Faragó, C., Hegedűs, P., and Ferenc, R. (2015). Cumulative code churn: Impact on maintainability. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 141--150. IEEE.
- Farrar, D. E. and Glauber, R. R. (1967). Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics*, pages 92--107.
- Flyvbjerg, B. (2006). Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219--245.

- Glaser, B. G. and Strauss, A. L. (2017). *Discovery of grounded theory: Strategies for qualitative research*. Routledge.
- Grahne, G. and Zhu, J. (2003). Efficiently using prefix-trees in mining frequent item-sets. In *FIMI*, volume 90.
- Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1--12. ACM.
- Hilton, M., Nelson, N., Tunnell, T., Marinov, D., and Dig, D. (2017). Trade-offs in continuous integration: assurance, security, and flexibility. In *11th Joint Meeting on Foundations of Software Engineering*.
- Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *31st Int'l Conference on Automated Software Engineering*.
- Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.
- Kirinuki, H., Higo, Y., Hotta, K., and Kusumoto, S. (2014). Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 262--265. ACM.
- Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721--734.
- Levin, S. and Yehudai, A. (2017). The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 35--46. IEEE.
- Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., and Guéhéneuc, Y.-G. (2014). Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *22nd Int'l Conference on Program Comprehension*.
- Lo, D., Nagappan, N., and Zimmermann, T. (2015). How practitioners perceive the relevance of software engineering research. In *10th Joint Meeting on Foundations of Software Engineering*, pages 415--425.

- Lowry, R. (2014). Concepts and applications of inferential statistics.
- Macho, C., McIntosh, S., and Pinzger, M. (2017). Extracting build changes with build-diff. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 368--378. IEEE.
- Molderez, T., Stevens, R., and De Roover, C. (2017). Mining change histories for unknown systematic edits. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 248--256. IEEE.
- Mori, A., Vale, G., Vigiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., and Kastner, C. (2018). Evaluating domain-specific metric thresholds: an empirical study. In *Int'l Conference on Technical Debt*.
- Murphy-Hill, E., Zimmermann, T., and Nagappan, N. (2014). Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *36th Int'l Conference on Software Engineering*.
- Nytro, O., Sorby, I. D., and Karpati, P. (2009). Query-based requirements engineering for health care information systems: Examples and prospects. In *31st ICSE Workshop on Software Engineering in Health Care*.
- Oliveira, J., Vigiato, M., Santos, M., Figueiredo, E., and Marques-Neto, H. (2018). An empirical study on the impact of android code smells on resource usage. In *International Conference on Software Engineering & Knowledge Engineering (SEKE)*.
- Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C., and Harman, M. (2017). Are developers aware of the architectural impact of their changes? In *32nd Int'l Conference on Automated Software Engineering*.
- Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *International Conference on Database Theory*, pages 398--416. Springer.
- Prechelt, L., Schmeisky, H., and Zieris, F. (2016). Quality experience: a grounded theory of successful agile projects without dedicated testers. In *38th Int'l Conference on Software Engineering*.
- Rácz, B. (2004). nonordfp: An fp-growth variation without rebuilding the fp-tree. In *FIMI*.

- Richardson, I., Reid, L., and O’Leary, P. (2016). Healthcare systems quality: development and use. In *Int’l Workshop on Software Engineering in Healthcare Systems*.
- Roed, K. and Ellingsen, G. (2011). Users as heterogeneous engineers-the challenge of designing sustainable information systems in health care. In *44th Hawaii Int’l Conference on System Sciences*.
- Russo, D., Ciancarini, P., Falasconi, T., and Tomasi, M. (2017). Software quality concerns in the italian bank sector: the emergence of a meta-quality dimension. In *39th Int’l Conference on Software Engineering: Software Engineering in Practice Track*.
- Schmidt-Thieme, L. (2004). Algorithmic features of eclat. In *FIMI*.
- Segura, S., Sánchez, A. B., and Ruiz-Cortés, A. (2014). Automated variability analysis and testing of an e-commerce site.: an experience report. In *29th Int’l Conference on Automated software engineering*.
- Stacey, P. and Nandhakumar, J. (2009). A temporal perspective of the computer game development process. *Information Systems Journal*, 19(5):479–497.
- Ståhl, D. and Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59.
- Stavnycha, M., Yin, H., and Römer, T. (2015). A large-scale survey on the effects of selected development practices on software correctness. In *2015 International Conference on Software and System Process*, pages 117–121.
- Steinmacher, I., Conte, T. U., Treude, C., and Gerosa, M. A. (2016). Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering*, pages 273–284. ACM.
- Stol, K. J., Ralph, P., and Fitzgerald, B. (2016). Grounded theory in software engineering research: A critical review and guidelines. In *38th Int’l Conference on Software Engineering*.
- Strauss, A. and Corbin, J. M. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications, Inc.
- Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. (2015). Investigating code review practices in defective files: An empirical study of the qt system. In *12th Working Conference on Mining Software Repositories*.

- Trockman, A., Zhou, S., Kästner, C., and Vasilescu, B. (2018). Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*, pages 511–522. ACM.
- Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in github. In *10th Joint Meeting on Foundations of Software Engineering*.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.
- Wright, H. K. and Perry, D. E. (2012). Release engineering practices and pitfalls. In *34th Int'l Conference on Software Engineering*.
- Yost, B., Coblenz, M., Myers, B., Sunshine, J., Aldrich, J., Weber, S., Patron, M., Heeren, M., Krueger, S., and Pfaff, M. (2016). Software development practices, barriers in the field and the relationship to software quality. In *10th Int'l Symposium on Empirical Software Engineering and Measurement*.
- Zaidman, A., Van Rompaey, B., van Deursen, A., and Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364.
- Zaki, M. J. and Gouda, K. (2003). Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM.
- Zaki, M. J., Parthasarathy, S., Ogihara, M., Li, W., et al. (1997). New algorithms for fast discovery of association rules. In *KDD*, volume 97, pages 283–286.
- Zhang, Y., Yu, Y., Wang, H., Vasilescu, B., and Filkov, V. (2018). Within-ecosystem issue linking: a large-scale study of rails. In *Proceedings of the 7th International Workshop on Software Mining*, pages 12–19. ACM.
- Zhou, B., Neamtiu, I., and Gupta, R. (2015). A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 7. ACM.