UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Mívian Marques Ferreira

A Hybrid Approach to Change Impact Analysis in Object-oriented Systems

Belo Horizonte
2023

Mívian Marques Ferreira

A Hybrid Approach to Change Impact Analysis in Object-oriented Systems

Dissertation proposal 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 Doctor in Computer Science.

Advisor: Mariza Andrade da Silva Bigonha
Co-Advisor: Kecia Aline Marques Ferreira

Belo Horizonte
2023

Resumo

A modificação de artefatos de software para inserção novas funcionalidades ou correção de erros é algo inerente ao ciclo de vida de software. Somente por meio dela o software poderá continuar a atender às necessidades de seus usuários. No entanto, realizar uma modificação de software pode ser uma tarefa desafiadora. Ao alterar um artefato de software, o desenvolvedor deve analisar o impacto que essa modificação terá em outros artefatos e, se necessário, modificá-los para que o comportamento do software permaneça consistente. A esse processo damos o nome de Análise de Impacto de Modificação (*Change Impact Analysis* - CIA). Alguns métodos para CIA, particularmente para nível de classe, têm sido propostos. No entanto, esses métodos não são práticos para serem aplicados no dia-a-dia dos desenvolvedor, uma vez que são muito complexos. Portanto, analisar o impacto que a modificação de uma classe tem em um sistema de software ainda é um desafio a ser superado. Sendo assim, o objetivo desta tese de doutorado é definir um novo método para CIA em nível de classe para software orientado por objetos. Para fundamentar nossa proposta, realizamos uma série de estudos. (i) Investigamos o estado da prática de desenvolvimento, buscando identificar o nível de conhecimento e aplicação dos conceitos de manutenção de software por parte dos desenvolvedores. (ii) Investigamos o estado da arte por meio de um mapeamento sistemático da literatura (*Systematic Mapping Review* - SMR) sobre CIA. Entre os resultados do SMR, identificamos o uso emergente de dados históricos de *commits*. No entanto, tais abordagens apresentam vieses significativos porque não consideram as características dos *commits*. (iii) Realizamos um estudo empírico para caracterizar *commits* em sistemas *open-source* desenvolvidos em Java. Com base nos resultados desse estudo empírico, propusemos uma nova heurística baseada em *commits* para CIA, visando superar as principais fragilidades das abordagens baseadas em *commits* propostas anteriormente na literatura. Em seguida, comparamos a heurística proposta com outra abordagem baseada em *commits* para CIA previamente proposta na literatura. Baseamos essa análise em dados de 237.366 *commits* de 38 sistemas de software *open-source* em Java. Essa análise mostrou que o uso exclusivo de *commits* para análise de CIA não é uma abordagem precisa. Assim, levantamos a hipótese de que os dados extraídos do grafo de dependência de um software também devem ser considerados na análise de impacto de modificação. Portanto, propomos o uso de grafo de dependência ponderado para estimar o impacto da modificação de uma classe em sistemas orientados a objetos. Os pesos aplicados no gráfico de dependências serão definidos tanto com base nos dados históricos de *commits* quanto nas características estáticas do software.

Palavras-chave: manuntenção de software, análise de impacto de modificação, *commits*

Abstract

The modification of software artifacts to insert new functionalities or correct errors is inherent to the software life cycle. Only through it can a software system continue to meet the user's needs. However, making a software modification may be a challenging task. When changing a software artifact, a developer must analyze the impact this modification will have on other artifacts and, if necessary, modify them so that the software system remains consistent. This analysis is called Change Impact Analysis (CIA). Some methods for change impact analysis, particularly for class-level, have been proposed over the years. However, these methods fail to be impractical for the day-to-day developer since they are very complex methods. Therefore, analyzing the impact that modifying a class has on a software system is still a challenge to overcome. Hence, this Ph.D. dissertation aims to define a new class-level CIA method for object-oriented software. To base our proposal, first, we conducted a series of studies. (i) We investigated the state of the development practice, seeking to identify developers' level of knowledge and application of software maintenance concepts. (ii) We investigated the state-of-art through a systematic mapping review (SMR) on change impact analysis. Among the SMR results, we identified the emerging use of historical commit data in the CIA. However, such approaches present significant biases because they do not consider the commits' characteristics. (iii) We conducted an empirical study to characterize commits in open-source systems developed in Java. Based on the results of this empirical study, we proposed a new commit-based heuristic for the CIA, aiming to overcome the main fragilities of the commit-based approaches previously proposed in the literature. Then, we compared the proposed heuristic with another commit-based approach for CIA previously proposed in the literature. We based this analysis on data from 237,366 commits from 38 Java open-source software systems. This analysis showed that the exclusive use of commits to the CIA is not an accurate approach. Therefore, we hypothesize that the data extracted from the software dependency graph should be also considered in CIA. Hence, we propose to use a weighted dependency graph to estimate the impact of modifying a class in object-oriented systems. The weights applied in the dependency graph will be defined both in historical commits data and the software system's static characteristics.

Keywords: software maintenance, change impact analysis, commits

List of Figures

2.1 Ph.D. dissertation workflow. 18

4.1 Distribution of participants' academic background. 32
4.2 Distribution of participants' professional experience. 33
4.3 Distribution of participants' companies size by the number of employees. . . . 33
4.4 Percentage of participants who declare to be familiar with refactoring, software metrics, bad smell, and change impact analysis. 34
4.5 Percentage of participants who declared to apply the refactoring, software metrics, bad smell, and change impact analysis. 35
4.6 Tools most used by practitioners to collect metrics. 37
4.7 Tools most used by practitioners to perform refactoring. 38
4.8 Tools most used by practitioners to perform change impact analysis. 39
4.9 How practitioners perform change and impact analysis. 39
4.10 Developers' main challenges when performing software maintenance. 41

5.1 Framework for CIA studies characteristics. 53
5.2 Number of published papers per year. 57
5.3 Number of papers published by venues. 58

6.1 Percentage of commits by category. 82
6.2 Distribution of files modified in commits. 86
6.3 Distribution of files modified in commits - upper outer fence outliers. 86
6.4 Distribution of Java files modified in commits. 87
6.5 Distribution of Java files modified in commits - Upper outer fence outliers. . . 87
6.6 Alluxio distribution of files modified in commits grouped by category. 88
6.7 JDK distribution of files modified in commits grouped by category. 89

7.1 Heuristic steps performed to co-change detection. 96
7.2 Data extraction flow 99
7.3 Precision of the Proposed Heuristic (PH) and the Commit Heuristic (CH). . . 102
7.4 Precision of the proposed heuristic (PH) and the commit heuristic (CH) according to the commits value range. 104
7.5 Correlation between classes co-change and classes distance. 105

8.1 The data sources to be used in the proposed model. 109

List of Tables

4.1 Questions and response options of the questionnaire. 28
4.2 p-values of Fisher's test for the association between familiarity and participants' background 36
4.3 p-values of Fisher's test for the association between practical application and participants' background 36

5.1 The electronic databases used in the SMR. 49
5.2 List of the inclusion and exclusion criteria. 50
5.3 Number of documents after applying each stage of the selection process. . . . 51
5.4 Nature of the works. 59
5.5 Scientific methods applied by the works. 60
5.6 Change Impact Analysis approaches applied by the works. 60
5.7 Data source used to perform change impact analysis. (Part I) 61
5.8 Data source used to perform change impact analysis. (Part II) 62
5.9 Type of source code analysis applied by the works. 63
5.10 Techniques applied by the works. (Part I) 64
5.11 Techniques applied by the works. (Part II) 65
5.12 Types of graphs used to perform change impact analysis when the applied technique is graph analysis. 66
5.13 Analyzed elements used to perform change impact analysis (Part I). 67
5.14 Analyzed elements used to perform change impact analysis (Part II). 68
5.15 Programming languages supported by the CIA proposals. 69
5.16 Evaluation methods applied in the studies (Part I). 70
5.17 Evaluation methods applied in the studies (Part II). 71
5.18 Evaluation metrics applied in the studies. 73

6.1 Dataset systems sorted by number of commits. 78
6.2 Primary keywords used to identify the activity category of commits. 80
6.3 Categorization Results: Precision and Recall 80
6.4 Percentage of co-occurrences of commits categories. 83
6.5 Percentiles of number of files per commit, where, Q1 = 1st quartile, Q3 = 3rd quartile. 85

7.1 Description of the analyzed systems and their respective number of commits and star rate on Github. 98

7.2 The precision of each analyzed system. 103
7.3 Correlation values between the number of co-changes and the distance between classes. The abbreviation ID means insufficient data. 106

8.1 Dataset - the systems in the large category 112
8.2 Dataset - the systems in the medium category. 113
8.3 Dataset - the systems in the small category. 114

9.1 Schedule of activities to be developed. 116

Contents

1 Introduction **11**

1.1 Aim . . . 12

1.2 Publications . . . 12

1.3 Proposal Organization . . . 13

2 Dissertation Method **15**

3 Related Work **19**

3.1 State of the Practice . . . 19

3.2 State of the Art . . . 20

3.3 Commits Characterization . . . 22

3.4 Heuristics for Co-change . . . 23

4 State of the Practice **26**

4.1 Study Design . . . 26

4.1.1 Questionnaire Construction . . . 26

4.1.1.1 Validation of the Questionnaire . . . 28

4.1.2 Participants Selection . . . 29

4.1.3 Research Questions . . . 30

4.2 Participants Characterization . . . 32

4.3 Results . . . 34

4.4 Discussion . . . 42

4.5 Threats to Validity . . . 45

4.6 Final Remarks . . . 45

5 Systematic Mapping Review **47**

5.1 Study Design . . . 47

5.1.1 Why a Systematic Mapping Review? . . . 48

5.1.2 Planning . . . 48

5.1.2.1 Research Questions . . . 48

5.1.2.2 Electronic Databases . . . 49

5.1.2.3 Search String . . . 49

5.1.2.4 Inclusion and Exclusion Criteria . . . 50

5.1.3 Execution Phase . . . 50

5.1.3.1 Search Process . . . 50
5.1.3.2 Selection of the Studies . . . 51
5.2 Framework for CIA Studies Characteristics . . . 52
5.2.1 Method . . . 52
5.2.2 Change Impact Analysis Approaches . . . 54
5.2.3 Data Source . . . 54
5.2.4 Technique . . . 55
5.2.5 Analyzed Elements . . . 55
5.2.6 Evaluation Method . . . 55
5.2.7 Supported Language . . . 56
5.3 CIA Studies Characteristics . . . 56
5.3.1 Change Impact Analysis thru the Time . . . 56
5.3.2 Change Impact Analysis Publication Venues . . . 57
5.4 Results . . . 58
5.4.1 Models and Tools . . . 58
5.4.2 Methods and Tools Characteristics . . . 59
5.4.2.1 Method . . . 59
5.4.2.2 Change Impact Analysis Approaches . . . 60
5.4.2.3 Data Source . . . 61
5.4.2.4 Technique . . . 63
5.4.2.5 Analyzed Elements . . . 65
5.4.2.6 Supported Languages . . . 68
5.4.3 Methods and Metrics Evaluation . . . 69
5.5 Discussion . . . 71
5.6 Final Remarks . . . 74

6 Commits Characterization **76**
6.1 Study Design . . . 76
6.1.1 Dataset . . . 76
6.1.2 Data Extraction . . . 77
6.1.3 Commits Categories . . . 78
6.1.4 Research Questions . . . 81
6.2 Results . . . 82
6.3 Discussion . . . 89
6.4 Threats to Validity . . . 92
6.5 Final Remarks . . . 93

7 A Heuristic for Co-change **94**
7.1 The Heuristic . . . 94
7.2 Study Design . . . 95

7.2.1 Evaluation Approach . . . 96
7.2.2 Dataset . . . 97
7.2.3 Data Processing . . . 98
7.2.3.1 Cloning the Repositories . . . 98
7.2.3.2 Extracting the Data from Repositories . . . 99
7.2.3.3 Applying the Proposed Heuristic . . . 100
7.2.3.4 Getting Classes Dependencies . . . 100
7.2.3.5 Building the Graphs . . . 100
7.2.3.6 Comparing Co-change and Dependency Graph . . . 101
7.3 Results . . . 101
7.4 Discussion . . . 106
7.5 Threats to Validity . . . 107
7.6 Final Remarks . . . 108

8 The Proposed Change Impact Model 109
8.1 Proposal Description . . . 109
8.2 Evaluation Method . . . 111

9 Conclusion 115
9.1 Next Steps . . . 115

Bibliography 117

Chapter 1

Introduction

Software maintenance is the most expensive activity in the software life cycle; about 50% of the effort related to software development is used in the execution of this task [120]. To serve the purpose it was developed, the software system must evolve, adapting to the users' needs [98]. This adaptation is made through the insertion of modifications that seek to insert new functionalities into the system, correct errors, improve computational performance, and improve usability.

Among the various artifacts used in developing a software system, the most commonly used is the source code. The punctual modification in a part of the system's source code may generate a chain reaction, causing other parts to be modified. When inserting a modification in the system, it is up to the developer to identify which different parts of the system were impacted and must be altered so that the expected behavior remains and new errors are not inserted in the system. We call this process Change Impact Analysis (CIA).

When performing CIA, the developer needs to understand the scope of the modification performed and have a broad command over the system's structure. They need to understand the relationships between system components, and these relationships are only sometimes explicit - this could cause the CIA to introduce a high cost and time development of the system. In addition, software evolution makes its structure increasingly complex, which increases the possibility that CIAs performed manually miss essential changes to the system. These unfinished modifications can make the software no longer present satisfactory results for the user and lose its development purpose.

Over the years, researchers have developed methods that seek to reduce the challenges encountered by developers in performing CIA in various software artifacts [167, 141, 76, 29, 45, 2, 135, 166, 123]. These methods present a range of data sources and techniques for change analysis. Given the importance of object orientation, many specific change impact analysis methods have been proposed in recent years for this paradigm [85, 111, 127, 5]. These methods are diverse concerning the technique employed and the artifacts analyzed. Kchaou et al. [85] used information retrieval and the structural dependencies derived from UML models to identify the change impact analysis. Based on traceability between requirements, architectural models, and source code, Nemr and

Elzanfaly [127] constructed a flow control graph to develop a framework for CIA detection. Ferreira et al. [111] developed a stochastic model to estimate the number of classes changed during the modification process. Agrawal and Singh [5] proposed a model to identify structures that changed together using source code metrics and the system version history.

Despite the important advances of research in change impact analysis, the complexity of the application of the proposed methods and the lack of support tools makes difficult the use of these methods in the daily life of developers. Hence, developers still deal with the challenges of analyzing change impact in a rudimentary way, having as main support tools for browsing the source code and the prior knowledge they have about software systems. Therefore, effective methods for change impact analysis are a practical need for software engineering.

1.1 Aim

Given this scenario, this Ph.D. dissertation aims to define, implement and evaluate a new method for change impact analysis. Our proposal is based both on change history and source code static analysis since we consider that these approaches are complementary and, then, may lead to better results.

To achieve this general aim, we define the following specific aims:

1. Investigate the state of the practice on software maintenance, specifically to identify how the practitioners deal with the problem of change impact analysis.
2. Investigate state of the art on change impact analysis.
3. Define a method for change history analysis.
4. Define a method for change impact analysis by applying change history and static analysis.

1.2 Publications

The following publications are outcomes of the Ph.D. dissertation in progress.

Published papers:

- Bruno L. Sousa, Mívian M. Ferreira, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. *Software Engineering Evolution: The History Told by ICSE*, (short-paper). In Proceedings of the 33rd Brazilian Symposium on Software Engineering (SBES), 2019, pp. 17–21.
- Mívian M. Ferreira, Bruno L. Sousa, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. 2021. *The Software Engineering Observatory Portal.* In Proceedings of the 18th International Conference on Scientometrics & Informetrics (ISSI), 2021, pp. 407-412.
- Mívian M. Ferreira, Mariza A. S. Bigonha, and Kecia A. M. Ferreira, *On The Gap Between Software Maintenance Theory and Practitioners' Approaches.* In Proceedings of the 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), 2021, pp. 41-48.
- Mívian M. Ferreira, Diego S. Gonçalves, Kecia A. M. Ferreira, and Mariza Bigonha. *Inside Commits: An Empirical Study on Commits in Open-Source Software,* (short-paper). In Proceedings of the 35th Brazilian Symposium on Software Engineering (SBES), 2021, pp. 11–15.
- Mívian M. Ferreira, Diego S. Gonçalves, Mariza A. S. Bigonha, and Kecia Ferreira. *Characterizing Commits in Open-Source Software.* In Proceedings of the XXI Brazilian Symposium on Software Quality (SBQS), 2022, pp. 1–10.

Submitted papers:

- Mívian M. Ferreira, Diego S. Gonçalves, Yuri B. Carvalhais, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. *Detecting Co-Change Using Categorized Commit Data.* pp. 1–9. Submitted to an international conference.
- Mívian M. Ferreira, Kecia A. M. Ferreira, and Mariza Bigonha. *What Changed? A Systematic Mapping on Change Impact Analysis.* pp. 1–20. Submitted to an international journal.

1.3 Proposal Organization

We organized the remaining of this Ph.D. dissertation proposal as follows. Chapter 2 describes the workflow we applied for developing this Ph.D. dissertation proposal. Chapter 3 discusses the main related works. Chapter 4 presents the study we performed

to investigate the state of the practice of maintenance concepts and techniques, including change impact analysis. Chapter 5 brings the systematic literature mapping about change impact analysis. Chapter 6 presents the study we performed on commit characterization. Chapter 7 presents the heuristic we propose to co-change detection in object-oriented software systems. Chapter 8 describes the proposal of the hybrid method to change impact analysis in object-oriented software systems. Chapter 9 brings the conclusions of this Ph.D. dissertation proposal and describes the next steps.

Chapter 2

Dissertation Method

In this chapter, we describe the workflow executed for the development of the Ph.D. dissertation proposal. Figure 2.1 summarizes each workflow's stage objectives, applied methods, outcomes, and current status. The workflow's stages are described as follows.

General studies on Software Engineering. The first stage consisted of a study to understand how Software Engineering research has evolved and, specifically, to asses the general status of software maintenance research. We analyzed data from papers published at the two main venues of Software Engineering: the International Conference on Software Engineering (ICSE) and the IEEE Transactions on Software Engineering (TSE). From TSE, we retrieved data from papers published from 1975 to 2018, accomplishing 3,357 papers. From ICSE, we gathered data from 1988, the first year IEEE Xplore provides data on the conference, to 2018, accomplishing 3,300 papers. We identified the main topics investigated in Software Engineering and how the investigation of those topics has evolved. Besides, we constructed a web-based portal[1] providing visualization of the publications' data. The results brought a compilation of Software Engineering evolution that may be of value to the software community. We identified the main topics investigated by the research community and concluded that software maintenance is a major problem that has been investigated. As this stage is not specifically related to change impact analysis, we do not describe this study in this document. However, the study was published in two papers [148, 64].

State of the practice. After identifying, in the previous stage, that software maintenance is a major area of interest for researchers, we sought to identify how software maintenance has been conducted in practice. For this, we surveyed 112 practitioners from 92 companies and 12 countries. We concentrated on analyzing if and how practitioners understand and apply the following subjects: bad smells, refactoring, software metrics, and change impact analysis. This study's results show a large gap between research approaches and industry practice in those subjects, especially in

[1] https://mivianferreira.github.io/docs/TheSoftwareEngineeringObservatoryPortal

change impact analysis and software metrics. Chapter 4 presents the study on the state of the practice.

State of the art. Having analyzed the practical aspect of the CIA, the next stage of this Ph.D. dissertation proposal was to assess the state of the art of the subject. For this, we conducted a literature review through a systematic mapping review of the CIA. In this mapping, we collected data from methods and tools proposed in the literature since 1978. The search in the digital libraries found 2006 documents. After analyzing them, we identified 141 papers related to the CIA. Among other conclusions, we concluded that the CIA is a relevant subject that has been studied by academia; however, there is a lack of methods with practical application in the developers' daily work. Chapter 5 presents the systematic mapping review.

Dissertation Aim. Based on the results of the studies produced in the last two stages, we define the objective of the Ph.D. dissertation: define a new hybrid approach to CIA based on code change history and static code analysis. We defined a hybrid approach because we identified that previous main studies had applied code change history or static analysis. Despite being promising approaches, they present basic problems that inhibit their practical application. The change history analysis demands collecting and analyzing historical data of software systems, which may be highly costly. Moreover, change history analysis cannot be applied when the software system is in the early stages of its life. Static analysis, such as dependency analysis, is applicable since the early stages of software development; however, it may fail in identifying the dependencies that are indeed prone to change propagation. Therefore, we consider that historical analysis and static analysis are complementary. Hence, the central idea of this Ph.D. dissertation is to define an approach that explores the benefits of change history and static analysis. We focused our studies on object-oriented software systems, specifically Java-based software systems. The rationale for this decision is that Java is one of the most popular programming languages, both in practice and in research on software engineering. Besides, Java has been considered in most previous works on change impact analysis, which may aid comparison between the results of these works and our proposal.

Commits' characterization. Based on studies of state of the art, we defined that one of the software elements used in our approach would be the commit since commits' data provide information on how software systems are changed over time. Understanding this element's structure is necessary for this purpose. Therefore, we carried out a commit characterization study. In our analysis, we considered the following characteristics that may guide the definitions of our approach: categories

of activities performed in the commits, co-occurrences of activities in commits, the size of commits in the total number of files, the size of commits in the number of source-code files; the size of commits by category; and the time interval of commits performed by a contributor. The results showed that the commits have several factors that must be considered when analyzed in research and that ignoring them may lead to biased results. Chapter 6 presents the study we performed on commits' characterization.

Change history analysis. This Ph.D. dissertation proposes defining a change impact analysis based on change history and static analyses. We analyzed the change history using the detection of *co-change.* A co-change occurs between two artifacts that are changed together. We proposed a new commit-based heuristic for co-change analysis by considering the commits' characterization explored in the previous study. When analyzing the commit data, our heuristic discards commits that have more than ten files, group commits related to the same issue, and group commits of the same author registered in the time interval of eight hours. To evaluate the heuristic, we compare its results with a heuristic that considers "pure commit", i.e., without optimizations. We use the systems' graph dependency as an oracle of the actual dependency between the classes. Thus, the heuristics' detection of co-change between two classes, A and B, is considered valid if there was a path from A to B or vice versa in the dependency graph. We analyzed data from 32 open-source Java systems hosted on GitHub. The results indicate that our heuristic has better accuracy than heuristics that do not consider the commit's characteristics, leading to results nearer to the actual dependencies among the classes. Chapter 7 presents our heuristic and its evaluation.

The hybrid CIA proposal. This item is the last step in developing this Ph.D. dissertation. In this step, we will describe, propose and implement a new model for change impact analysis for Java systems. This model will be based on data from the system's change history and data extracted through static analysis of the source code. Chapter 8 describes the change impact analysis proposal.

Chapter 3 discusses the previous main related works to this Ph.D. dissertation.

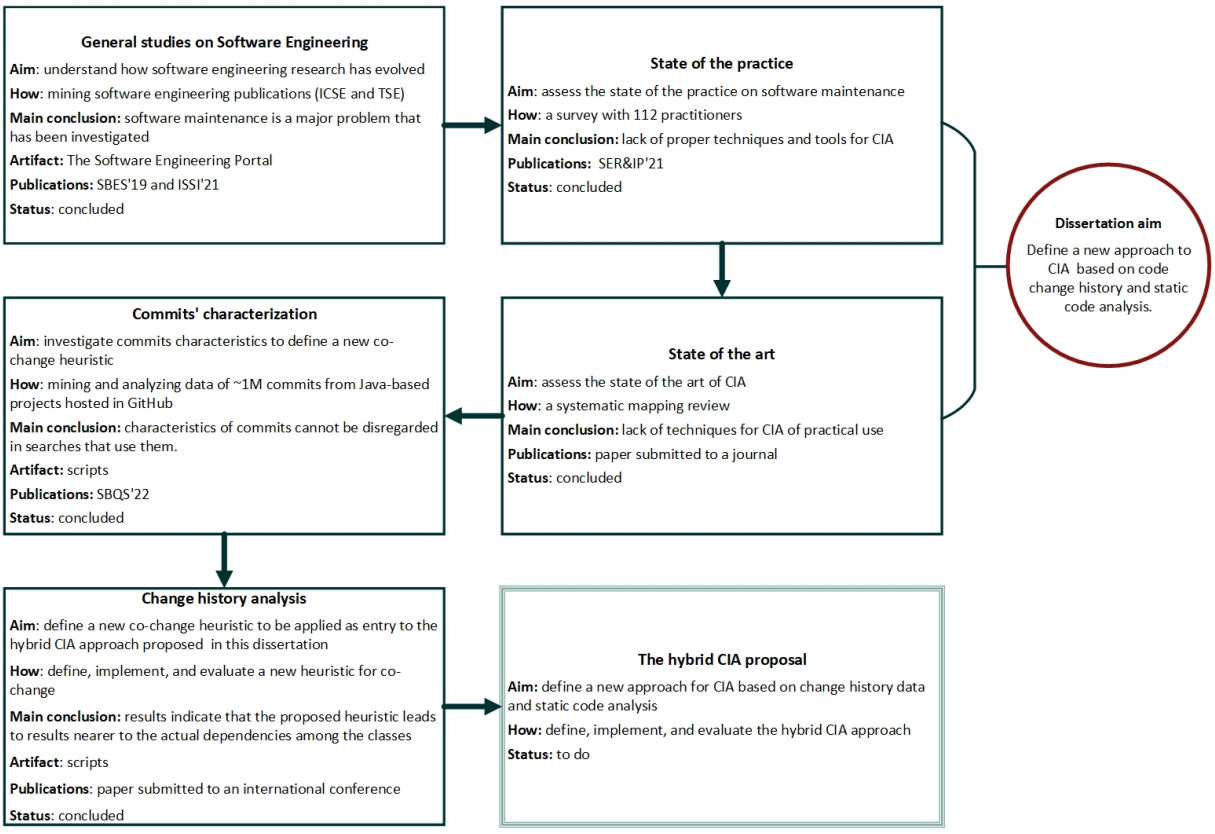

Figure 2.1: Ph.D. dissertation workflow.

Chapter 3

Related Work

This chapter discusses the previous main works related to this dissertation. We organized the related work according to the phases of this dissertation. First, we describe studies concerned with investigating the state of the practice of software maintenance. Secondly, we describe previous literature reviews on change impact analysis. In the sequel, we discuss works related to commit characterization. Finally, we discuss the main approaches for co-change detection related to the heuristic for co-change we propose.

3.1 State of the Practice

In this Ph.D. dissertation, we carried out a survey aiming to assess the state of the practice regarding a set of topics on software maintenance, including change impact analysis. The survey is presented in Chapter 4. Here, we discuss the related works investigated the software maintenance state of practice.

The interest of researchers in the effectiveness of knowledge transfer from academia to industry is not recent. One of the first articles on this subject, in the mid-'80s, Redwine and Riddle [137] indicated that the process of maturation technology, in which knowledge is transferred from theory to practice, may take 15 to 20 years to happen. Also, they stated that for this transition, there must be "a recognized need, a receptive target community and believable demonstrations of cost/benefit". Pfleeger [133] highlights the difficulties researchers face in evidencing the effectiveness of knowledge transfer from academia to industry. Diebold and Vetrò [59] investigated how this transfer occurs and the possible ways to measure it. The results show that, on average, technology transfer may happen faster in three years.

Another aspect studied of Software Engineering was presented by Lo et al.[109]. The authors surveyed software practitioners to understand how relevant some topics presented in publications of ICSE, ESEC/FSE, and TSE[1] were to these practitioners. Ac-

[1]ICSE: International Conference on Software Engineering; ESEC/FSE: Joint European Software

cording to the authors, practitioners consider that empirical studies are not realistic and present problems of generalization. Carver et al. [36] replicated the study of Lo et al. and applied the same methodology to the works presented in ESEM[2]. Besides, the authors suggested an orientation section for researchers based on the opinion of practitioners. Maintenance was one of the aspects analyzed. According to Carver et al., practitioners consider relevant studies that make the software more maintenance-friendly, specifically those that address legacy code, code maintenance, and technical debt.

According to Brodin and Benitti [32], over 70% of software developers work with maintenance. However, they point out that this is still a topic that academics have not researched. They studied, through a survey, whether practitioners in the industry use the subjects taught in faculties about software maintenance. By contrasting these two aspects, they identified that engineering, reverse engineering, software processes, and software measurement are widely considered by academia and rarely used in the industry. According to the authors, the topics that are little discussed in academia and widely used in industry are legacy systems, modification impact analysis, tools related to testing, and configuration management. In the results reported, Brodin and Benitti state that refactoring is the only topic widely studied by academia that practitioners extensively use. They consider that the software engineering courses should be restructured to better prepare practitioners for the industry.

In our work, we also applied surveys to comprehend practitioners' perceptions of software engineering while focusing on software maintenance practice. Our work differs from previous ones as we aim to identify how practitioners understand and apply concepts related to maintenance, specifically: software metrics, refactoring, bad smells, and change impact analysis. Moreover, we identified the most significant challenges practitioners face when performing software maintenance. Another difference between our study and the related ones is that we extended the vision of software practice since the participants of our survey work in 92 companies from 12 countries around the world.

3.2 State of the Art

To obtain a deep and broad knowledge of state of the art on change impact analysis, we performed a Systematic Mapping Review on this subject, presented in Chapter 5. This section discusses previous studies that performed a literature review on change impact

Engineering Conference and Symposium on the Foundations of Software Engineering; FSE: Joint Meeting on Foundations of Software Engineering

[2]ESEM: International Symposium on Empirical Software Engineering and Measurement

analysis.

In 2012, Li et al. [105] presented a systematic literature review on change impact analysis techniques based explicitly on source code analysis. The authors analyzed 30 articles published between 1997 and 2010. After analyzing the articles, the authors reported the mapping of 23 techniques for change impact analysis and performed their evaluation. To evaluate these techniques, the authors developed a framework for comparing CIA techniques. The framework considers approaches based on the following aspects: what is the impact set - the type of report presented to the user after applying the method; the type of analysis performed on the source code - static or dynamic; intermediate kind of representation, if any; supported programming paradigm; existence of a tool that implements the technique and if there was an empirical study to evaluate the method. Our work used this framework as a basis for structuring the characterization of the studies analyzed in this work.

De Luca et al. [55] present a literature review of works related to artifact traceability management in the context of change impact analysis. The authors also showed a framework for the characterization of the approaches found. They categorized each paper into three dimensions. The 1st dimension is the *Vertical* ability to track dependent artifacts within a model and the *Horizontal* ability to track artifacts across different models. The 2nd dimension is *Structural,* which considers the nature of the data used to derive traceability between the artifacts analyzed, derived from the analysis of artifacts related to syntax and semantics, versus *Knowledge-based,* which refers to the dependency between artifacts that cannot be obtained automatically. The 3rd dimension identifies the *Implicit Traceability* links retrieved during system execution. And the *Explicit Traceability* of statically retrieved links. Besides that, they present works related to the main challenges for traceability and connection evolution between the artifacts involved in a modification.

The study developed by AlSanad and Chkih [11] presents a review of works related to the impact of modifying requirements in software development projects. For this, the authors evaluated articles published between 1972 and 2014. The results were divided into a timeline according to the focus presented by each evaluated article. This timeline was divided into four periods. The initiation (1972 to 1999) was the period whose main objective was discovering resources related to requirement change. The second period described by AlSanad and Chkih was Analyzing the Requirement Change (2000 to 2003) time frame, where the research had as its primary focus the analysis and evaluation of the modification of requirements. Discovering the impact (2004 to 2009), the third period established by the authors, researchers were interested in finding the effects of modifying a condition during software development. And finally, the fourth period Reducing the Impact (2010 to 2014), was a period in which the research aimed to propose strategies to reduce the impact of modifying requirements during software development. After the

analyses, the authors indicated that there are still open questions in the area, such as the impact that changing conditions requirements can have on software quality; and the need to develop techniques and tools capable of helping to support the evolution of requirements specifications in large systems.

Dhamija and Sikka [57] conducted a systematic literature review to report the current status of studies, techniques, and tools related to change impact analysis. For this, 33 articles were analyzed and separated into three categories: CIA based on traceability, Dependency-based CIA, and Experimental CIA - used to group informal methods such as protocol review, knowledge, and personal judgment, that is, articles that do not fit into the first two categories. In addition, the study presented an analysis of the latest tools for the CIA. After analyzing the articles, Dhamija and Sikka indicated the following main findings of the survey: analyses based on dependency and traceability are the most used; techniques based on graph, dependency, method of execution, and the trace of execution are commonly used by authors who develop strategies for CIA. Finally, they identified a gap concerning CIA techniques that explore the issue of hidden dependencies.

Unlike our systematic mapping review, these works cover specific aspects of methods and tools related to CIA. They do not provide a comprehensive view of the state of the art of CIA. Besides, the most recent literature review [57] analyzed 33 studies, whereas our SMR found 141 studies.

3.3 Commits Characterization

In this Ph.D. dissertation, we aim to define a hybrid method to change impact analysis based on change history and static analysis. Defining a reasonable heuristic to detect co-change was necessary to perform the change history. Before defining the heuristic, we performed an empirical study to characterize commits, presented in Chapter 6. Here, we discuss the main related work on commit characterization.

We identified three previous works that characterize commits regarding the aspects we consider in our approach: commits' size and time interval of commits by author.

Hatorri and Lanza [78] did the first work before the GitHub advent. They sought to relate the activities performed by developers and the number of files modified in a commit. They investigated three activities: bug fixing, insertion of new features, and management tasks. The inclusion of a commit in one of these categories was performed by analyzing the commit message. The authors also ranked commits according to the number of files they modified. For this, they established four categories: tiny (1 to 5 files), small (6 to 25 files), medium (26 to 125 files), and large (up to 126 files). The

analysis results of nine open source systems identified that tiny and small commits are associated with bug fixes, and large commits are associated with management activities, such as source code review. They concluded that activities of inserting new features in the system are heterogeneously related to the size of commits.

Based on the work by Hatorri and Lanza [78], we also characterized Java system commits, however, considering GitHub repositories. The authors categorized the activities registered by a commit in Reengineering, Forward Engineering, Corrective Engineering, Merge, and Management. The results indicated that the activities most performed by developers are related to Reengineering, insertion of new features (Forward Engineering), and fault correction (Corrective Engineering). Furthermore, the reported results indicate that 30% of commits register more than one activity. Our previous study identified that the number of files modified in a commit follows a long-tail distribution - meaning that few commits change many files, and most of the commits modify few files - this value becomes between one and ten files. Also, the average time a developer performs a commit is eight hours. We used these characteristics in our heuristic.

3.4 Heuristics for Co-change

In Chapter 7, we proposed a heuristic for co-change detection that applies commits characteristics. This section discusses the main works related to our proposed heuristic.

The Software Engineering research community widely uses data mining of commits from software repositories. In recent years, several works have used these data to detect co-change between software artifacts.

Geipel and Schweitzer [67] analyzed whether the structural dependence of classes in a system is linked to co-change. They considered that the files involved in a commit are co-changed. The authors used a dependency matrix to represent the dependencies between modules. In their analysis, they evaluated data from 35 Java systems. The results indicated positive evidence that the changes propagate between the modules through structural dependence. Besides, modifications are not concentrated in a subset of dependencies, i.e., they are distributed highly unequally.

In their work, Modal et al. [126] sought to identify whether it is possible to detect software entities that are highly coupled by analyzing the history of software modifications. They made the investigation at the method level and presented a technique to detect MMCGs - *Method Appearing in Multiple Commit Groups*. Modal et al. developed a tool to identify and rank the methods coupled to each method that underwent co-change. Through an investigation considering data from seven software systems, the

authors identified that MMCGs detected by the technique are generally logically coupled to more methods and are more prone to change. The authors indicate that methods seen as MMCGs should have a higher priority in carrying out maintenance.

Oliveira and Gerosa [128] conducted an empirical study to identify the influence of structural dependence on change propagation. They based their study on a qualitative analysis of the relationship between structural dependencies and co-change occurrence based on commits. The dataset of the study had four Java systems. They concluded that the fact that two artifacts co-change in a commit is not linked to the structural dependence of these artifacts. However, the authors found that the rate at which artifacts co-changed is even more significant when one is structurally dependent on the other. In addition, the authors also observed several cases of co-change that could not be explained by structural dependency, thus indicating that relationships of another nature may have induced co-change.

In their work, Jaafar et al. [81] present a study on modification patterns based on data mining of software repositories and the time between these modifications. They proposed identifying two modification patterns: macro co-change and dephase co-change. *Macro co-change* occurs with a significant time interval, and *Dephase co-change* occurs in the same period. They used the K-nearest Neighbor machine learning technique to identify these patterns. To evaluate their proposed approach, the authors conducted two empirical studies: a quantitative and a qualitative study. In the quantitative research, the authors compared their proposed technique (Mococha) with UMLDiff and association rules. They evaluated seven systems, and the results showed that Mococha obtained better precision and recall. The authors used external information and static source code analysis in the qualitative study, and the results showed that the two co-change patterns exist.

Rolfsnes et al. [139] proposed an approach based on the evolutionary coupling to detect relationships between source code structures and point out possible co-change between system files. The authors established these relationships by analyzing the history of system modifications. The technique proposed by them indicates that for a given set X of files modified in a specific commit, a set Y will also be modified in the same commit. The authors analyzed six systems and compared their proposed technique's results with two others. The results demonstrate that the proposed approach presents the best performance among the evaluated scenarios.

Our approach is similar to these previous ones because we also consider historic commits' data. However, the main difference between our approach and the previous ones is that we consider commits' characteristics. Specifically, we consider: commits' size in the number of files, issue number, commit's author, and the time-frame between commits. Besides, in our analysis, we consider a higher number of software systems than the previous works. We plan to apply our heuristic to perform change history analysis and

use the results to define our proposal for a hybrid method to change impact analysis.

Chapters 4 to 7 present the studies we carried out in this Ph.D. dissertation. Chapter 8 describes our proposal for a hybrid approach to change impact analysis.

Chapter 4

State of the Practice

Boehm [30], in his mid 70's article "Software Engineering", defined Software Engineering as *"the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them"*. This classical 40 years old definition demonstrates the nature of the practical application of the area. Software Engineering research aims to offer solutions to real problems related to crafting and maintaining software systems. In this context, this chapter presents a study that investigates the applicability in real software development scenarios of concepts and techniques proposed in the literature regarding software maintenance.

We organized this chapter as follows. Section 4.1 presents the method applied to conduct the study. Section 4.2 characterizes the survey participants. Sections 4.3 and 4.4 present and discuss the results of this study. Section 4.6 presents the final remarks.

4.1 Study Design

This section describes the method used to survey practitioners. The following aspects are detailed: the construction and validation of the questionnaire, the participants' selection, and the applied method to analyze the data and answer the research questions.

4.1.1 Questionnaire Construction

To collect the information provided by the survey participants, we made a questionnaire based on the guideline of Kitchenham and Pfleeger [90]. The questionnaire comprises seven sections: Term of Consent, Participants' Characterization, Challenges to

performing software maintenance, Metrics, Refactoring, Bad smell, and Change Impact Analysis. In the sequel, we describe these sections. Table 4.1 exhibits the survey questions and their answer options.

Term of consent. In this segment, we introduce the purpose of the study to participants and request their endorsement to use the data collected. The term also presents the researchers' affiliation and the description of participants' anonymity assurance.

Participants' characterization. In this section, we collect data about the participants' professional lives: name and country of the company where they work; what position they hold in the company; their academic background; years of professional experience; programming languages that the participant currently use in their jobs, and the methodology they use in the software development process of the company.

Challenges to perform software maintenance. This section contains only one question in which we requested the participants to describe the main challenges and difficulties they face in performing maintenance activities.

Metrics. In this section, we asked the participants if they were familiar with metrics and if they considered them useful. Also, we asked if they use metrics to measure the code quality, which tools they use to do it, and which are the most common metrics they apply.

Refactoring. About refactoring, we asked if the participants were familiar with the term "refactoring" and if they usually perform code refactoring. We also asked them to name the types of refactoring techniques they apply and the tools used to perform such activity.

Bad smells. In the section, we asked if the participants were familiar with the term "bad smell" and if they used to verify the presence of bad smells in the software code. If so, we requested them to list which bad smells they use to search in software systems.

Change impact analysis. In the last section, we asked the participants: if they used to observe the need to change other pieces of code when they carried out a change in the code, if they were familiar with the terms "change impact analysis" or "dependency analysis"; if they use to search other pieces of code that need to be modified when they correct a bug or insert a new feature in the code, what kinds of techniques they use to perform such analysis; and, finally, if they use any tool to perform change impact analysis.

Subject		Question	Reply Options
Challenges to Perform Maintenance	1	Describe the main difficulties you face when performing maintenance on software.	Open Field
Software Metrics	2	Are you familiar with software metrics concept?	Yes or No
	3	What is your opinion about the use of software metrics to ensure the quality of the source code?	'Very important', 'Important', 'Little important', 'Unnecessary' or 'I don't have background to give an opinion.'
	4	Do you use software metrics to evaluate the quality of the source code at your work?	Yes or No
	5	If you use software metrics to evaluate the quality of the source code at your work, please name them.	Open Field
	6	If you use metrics to evaluate the quality of the source code at your work, which measurement tool(s) do you use?	Open Field
Refactoring	7	Are you familiar with the concept of refactoring ?	Yes or No
	8	Have you ever applied code refactoring at your work?	Yes or No
	9	If you have ever used code refactoring at your work, what kind (s) of refactoring did you use?	Open Field
	10	If you have ever used code refactoring at your work, have you used a tool for this?	Yes or No
	11	If you have ever used code refactoring at your work and have used a tool to do so, which tool (s) did you use?	Open Field
Bad Smell	12	Are you familiar with the concept of bad smell?	Yes or No
	13	When developing or maintaining a system at work, do you usually check bad smells in the source code?	Yes or No
	14	If you answered 'yes' to the previous question, what are the bad smells most commonly detected by you?	Open Field
Change Impact	15	Have you ever noticed whether a change performed in a software system by you had caused the need to make other changes not initially foreseen?	'Never', 'Few times', 'Oftentimes' or 'Always'
	16	Are you familiar with the term "Change Impact Analysis "?	Yes or No
	17	When correcting a bug (error or failure), performing a change or creating a new functionality in the system, do you usually analyze the impact of the change in the rest of software system?	Yes or No
	18	What kind of technique do you apply to analyze parts of the software that need to be modified?	'I explore the code manually and intuitively, not always with prior knowledge about it.', 'I explore the code manually guided by the prior knowledge I have about it.', 'I use a tool for this.', or 'I do not analyze all the parts that need to be modified, I make the modifications as I identify the problems.'
	19	If you use a tool to analyze which parts of the software need to be modified, please name them.	Open Field

Table 4.1: Questions and response options of the questionnaire.

4.1.1.1 Validation of the Questionnaire

Before sending the questionnaire to the participants, we made a pilot survey to test the questionnaire and identify its improvement needs. For this purpose, we sent the

questionnaire to two developers from different companies.

We identified the following main improvements needed from the first version. (1) Observing the answers, we notice that the questions we asked before the challenge section may influence the answer of the participants about the main challenges they face in software maintenance. This happened because we asked about refactoring, dependency analysis, bad smells, and metrics before the section 'Challenges to Perform Software Maintenance', which may induce the participants to include problems related to such subjects in their answers. We then decided to put the section 'Challenges to Perform Software Maintenance' as the first in the questionnaire. (2) We reformulated the answer options that involve ranges: number of employees and years of professional experience. (3) We divided some long questions into two or more to be clear and to facilitate the data analysis.

We sent the second version of the questionnaire with such improvements to less than ten developers in a second round. We made a previous analysis of the responses to ensure that the questionnaire was more precise and could gather the information we needed. Then, we sent the survey to the entire list of selected participants.

4.1.2 Participants Selection

The contact with practitioners was done in two ways: directly or indirectly. Some participants were contacted directly via email, LinkedIn, or Facebook and were chosen by convenience since the authors had their contact previously. The authors randomly selected other participants via LinkedIn and StackOverflow. To motivate the practitioner to answer the questionnaire, we sent a particular message to each participant explaining the survey's aim and inviting them to participate in the research. In this message, we asked them to forward the email to other colleagues, aiming to invite more participants indirectly.

We received 112 responses. We directly contacted 204 practitioners; out of this amount, 77 answered the questionnaire, achieving a response rate of 37.8%. The other 35 practitioners that answered the questionnaire were contacted indirectly.

As we aimed to have a global assessment of the perceptions and practices of practitioners worldwide, we contacted professionals from different countries. We also aimed to include participants from a large number of companies. The practitioners that answered the questionnaire were from 92 companies.

4.1.3 Research Questions

The research questions aim to elucidate if and how some of the main concepts and techniques proposed by the research community for software maintenance are applied in practice. This section describes how we analyzed the data to answer these research questions.

RQ1. Are developers familiar with software metrics, bad smells, refactoring, and change impact analysis?

RQ1 aims to investigate if the participants know the concepts of software metrics, refactoring, bad smells, and change impact analysis. To answer this question, we calculated the percentage of participants who answered 'yes' to the familiarity-related questions - which appears in Rows 2, 7, 12, and 16 of Table 4.1.

RQ2. Do practitioners apply software metrics, refactoring, bad smells, and change impact analysis in practice?

With this research question, we intend to identify if the participants apply these concepts in practice. To answer this research question, we calculated the percentage of participants who answered 'yes' to questions described in Rows 4, 8, 13, and 17 of Table 4.1.

RQ3. Is the background of practitioners associated with the familiarity and application of the concepts and techniques of software maintenance?

This research question aims to investigate if there are associations of participants' backgrounds with familiarity and application of metrics, refactoring, bad smell, and change impact analysis. Therefore, we verified the following:

- if familiarity with software metrics/refactoring/bad smell/change impact analysis is associated with the following categories of participants' background:
  - academic background;
  - years of professional experience;
  - company size.
- if the practical application of software metrics/refactoring/bad smell/change impact analysis is associated with those same categories of participants' backgrounds.

To test each of these associations, we applied Fisher's exact test, a statistical hypothesis test, to check if there is independence or any association between categorical

data regardless of the size or distribution of the sample. Therefore, we verified the hypotheses described in the sequel. Here, we are generically presenting the hypotheses so that they may be applied to all investigated associations.

- H_0*: the variables analyzed are independent.*
- H_a*: there is association between the variables analyzed.*

RQ4. Which are the most used tools by practitioners in software maintenance?

This research question aims to identify which tools are used in practice to support the activities related to software measurement, detection of bad smells, refactoring, and change impact analysis. The participants answered the questions associated with RQ4 in a text field. To analyze the data, we read all the answers and tabulated the tools described by the participants.

RQ5. How do practitioners perform change impact analysis?

This research question aims to investigate whether and how practitioners analyze the impact of changes they need to perform in software systems, such as fixing a bug, inserting new features, or any other type of maintenance. To answer this research question, we summarized and reported the practitioners' answers in Item 18 of Table 4.1.

RQ6. Which metrics, refactoring techniques, and bad smells do practitioners apply in their activities?

With RQ6, we aim to identify the most used software metrics, the bad smells most considered, and the refactoring techniques most applied by the participants. To answer this research question, we read all the answers to the questions described in Rows 5, 9, and 14 of Table 4.1 and summarized the data.

RQ7. What are the biggest challenges practitioners face when carrying out software maintenance? This research question aims to identify the most significant challenges developers face when performing software maintenance. For this, two authors tabulated e labeled, separately, the answers to the question shown in the first row of Table 4.1; then, comparing the labels, if they found different labels for the same answers, the final label is determined by a consensus between these two authors.

4.2 Participants Characterization

This section describes the characteristics of the participants. From such characterization, we observe that the sample is diverse in most aspects.

Academic Background. Participants with an undergraduate degree, specialization course, and master's degree correspond to the majority of the sample, 95.5%, as shown in Figure 4.1. Only 4.5% of participants are high school graduates, technicians, or Ph.D.

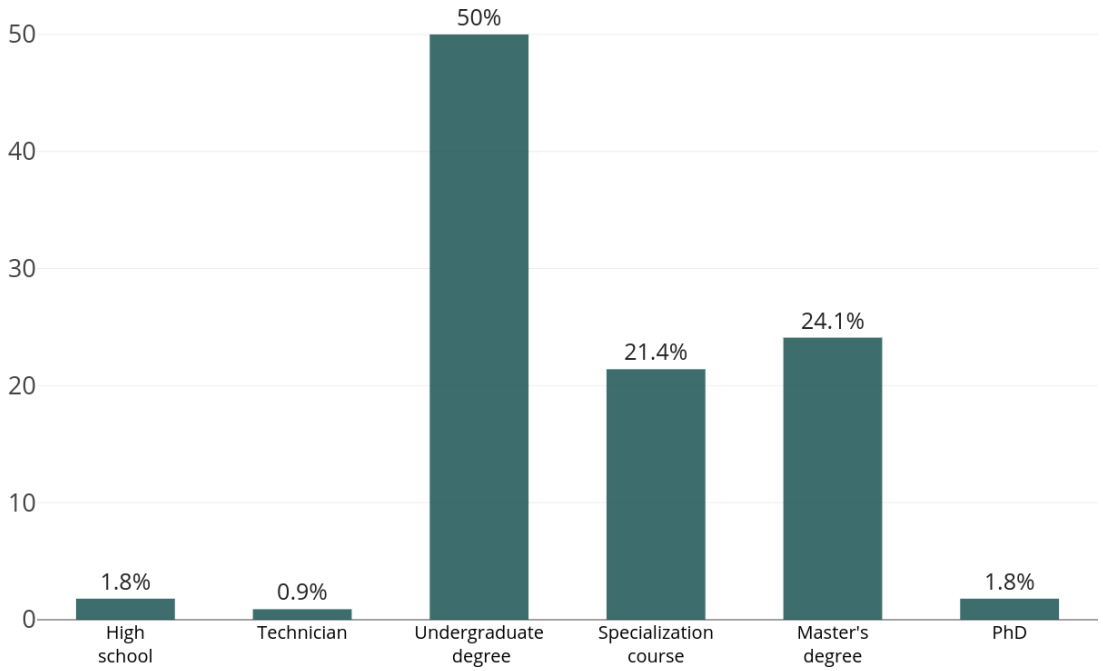

Figure 4.1: Distribution of participants' academic background.

Programming Languages. The programming languages used by the participants are Java, C#, C++, Python, JavaScript, PHP, Scala, Kotlin, TypeScript, ShellScript, Delphi, Swift, Objective C, Golang (Go), Groovy, Pearl, Dart, Ruby, Visual FoxPro, VB.NET, and ASP.NET.

Methodologies. Agile methodologies are the most widely used - Scrum and XP are the most cited. Only one participant informed the company uses a Waterfall process.

Professional Experience. Most participants, 63.4%, have between two and ten years of professional experience, and 25.9% of the participants are experienced professionals with more than ten years of career. Junior practitioners are 10.7% of the participants. Figure 4.2 shows the distribution of this data.

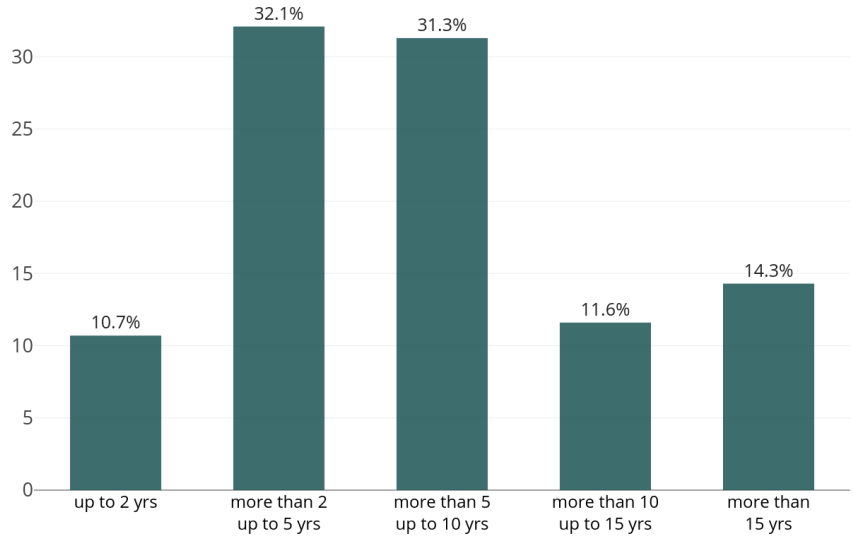

Figure 4.2: Distribution of participants' professional experience.

Companies Characterization. Among the participants, 69.7% work in medium-sized or large companies, and 30.3% work in micro and small companies. Figure 4.3 presents the distribution of participants according to their company size.

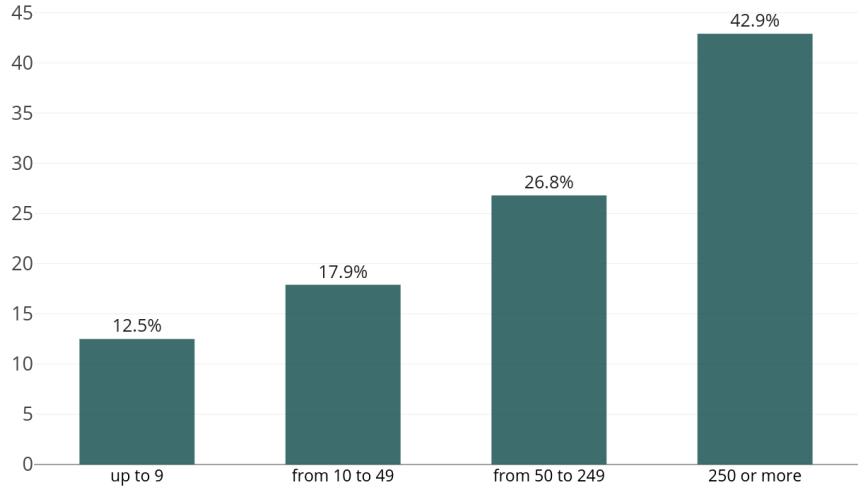

Figure 4.3: Distribution of participants' companies size by the number of employees.

Companies Sector. 71% of the companies are from the IT area. The other companies are in the following areas: trade, financial, banking, industry, marketing, health, education, and government.

4.3 Results

This section presents the data analysis and the answers to the research questions.

RQ1. Are developers familiar with software metrics, bad smells, refactoring, and change impact analysis?

To answer this question, we considered the number of participants who declared familiarity with the subjects investigated in this study. Figure 4.4 shows the percentage of participants who answered 'yes' to questions regarding familiarity with software metrics, refactoring, bad smells, and change impact analysis, respectively (see Rows 2, 7, 12, and 16 of Table 4.1).

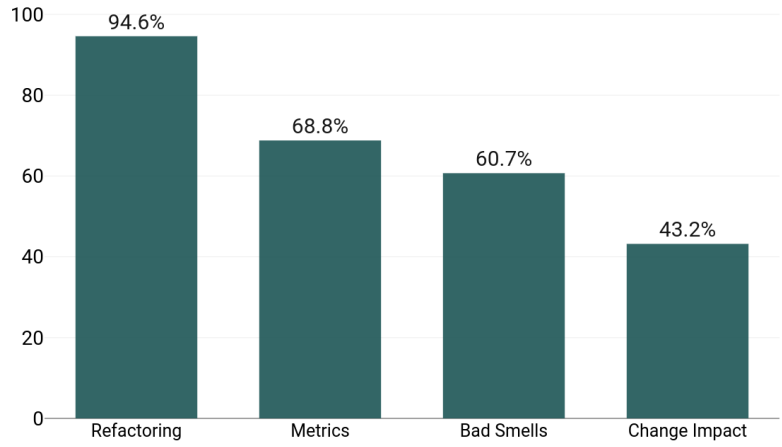

Figure 4.4: Percentage of participants who declare to be familiar with refactoring, software metrics, bad smell, and change impact analysis.

Refactoring is the most popular subject among participants; 94.6% of them claim to be familiar with this subject. *Metrics* is the second best-known subject; 68.8% of participants are familiar with it. *Bad Smell* is a term known by 60.7% of the participants. *Change Impact Analysis* is the least familiar term to the practitioners who answered the survey; however, a large part of the participants, 43.2%, stated they are familiar with this concept.

RQ2. Do practitioners apply software metrics, refactoring, bad smells, and change impact analysis in practice?

To answer RQ2, we calculated the number of participants that answered 'yes' to questions regarding the practical application of metrics, refactoring, bad smells, and

change impact analysis (Rows 4, 8, 13, and 17 of Table 4.1). Figure 4.5 shows the results.

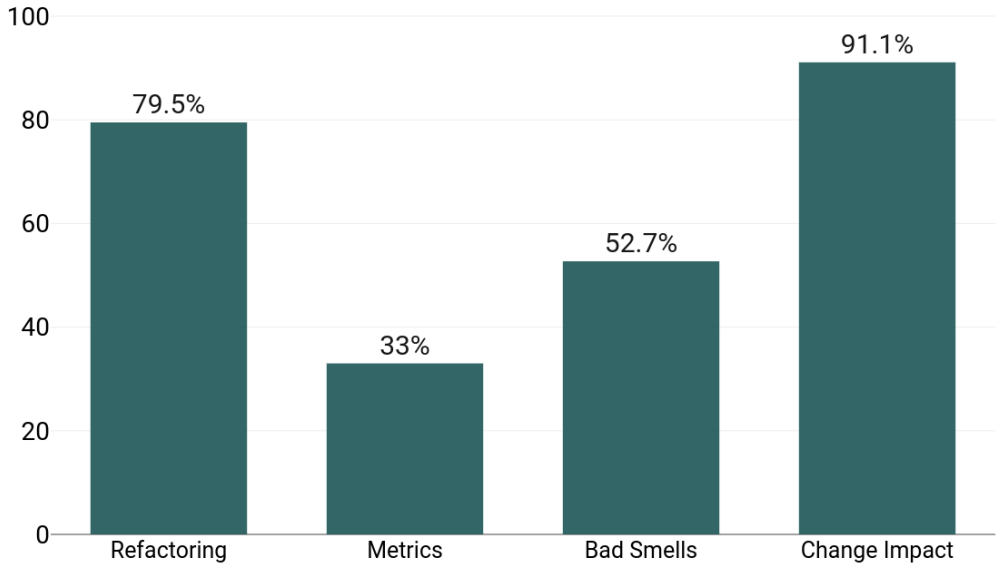

Figure 4.5: Percentage of participants who declared to apply the refactoring, software metrics, bad smell, and change impact analysis.

Although it is the least known concept among survey participants, change impact analysis is the most applied concept; 91.1% of the participants affirmed that they perform impact analysis when making modifications in software code. Refactoring is the second most commonly used concept in practice; 79.5% of participants perform code refactoring. Bad smell is the concept that has more balance between theoretical knowledge and practice: 60.7% declared to know the concept of bad smell, and 52.7% of the participants affirmed verifying the presence of bad smells in software code. Software metrics are the least applied concept in practice. Only 33% of participants claimed to use software metrics. However, 68.8% of the participants consider software metrics important or very important, contrasting with the 9.8% who believe that it is of little importance; 21.4% said that they do not have the background to manifest their opinion about such a subject.

RQ3. Is the background of practitioners associated with the familiarity and application of the concepts and techniques of software maintenance?

This research question investigates if academic background, professional experience (years), or company size are associated with familiarity and practical application of the concepts analyzed in this study. To answer this research question, we assembled 16 contingency tables and applied them to Fisher's exact test. The test was performed with the significance level of $\alpha = 0.05$ e $\alpha = 0.1$.

Familiarity	Participants' Background		
	Academic Background	Professional Experience	Company Size
Software Metrics	0.84	0.65	0.51
Refactoring	0.43	0.48	0.21
Bad Smell	0.18	**0.09**	**0.06**
Change Propagation Analysis	**0.07**	0.71	0.27

Table 4.2: p-values of Fisher's test for the association between familiarity and participants' background

Table 4.2 presents the p-values obtained as a result of Fisher's test to verify if there is an association between familiarity with software metrics, refactoring, bad smell, and change impact analysis. None of the associations' analyses resulted in a p-value less than $\alpha = 0.05$, which means one cannot reject the null hypothesis H_0 (The variables analyzed are independent). Thus, the tests show that familiarity with software metrics, refactoring, bad smell, and change impact analysis is not associated with the participants' academic background, years of professional experience, or company size.

Nevertheless, when performing the analysis considering $\alpha = 0.1$, one should reject the null hypothesis H_0 and accept the alternative hypothesis H_a. Therefore, with a 90% confidence interval, is possible to state that there is an association between academic background and familiarity with change impact analysis. In the same way, there is an association between professional experience and familiarity with bad smells and an association between company size and familiarity with bad smells.

Practical application	Participants' Background		
	Academic Background	Professional Experience	Company Size
Software metrics	0.99	0.99	0.95
Refactoring	0.33	**0.04**	0.62
Bad Smell	0.90	0.16	0.49
Change Propagation Analysis	0.42	0.25	0.32

Table 4.3: p-values of Fisher's test for the association between practical application and participants' background

The Fisher's test indicated a strong association between the application of refactoring and years of professional experience since the p-value for this association, presented in Table 4.3, is less than $\alpha = 0.05$, the level of significance used in this analysis. Therefore, with a 95% confidence interval, it is possible to affirm that the application of refactoring is associated with participants' professional experience. Among the participants who declared to perform refactoring, 56% have at least five years of experience. It is worth

noting that the p-value of the association between bad smell and professional experience is 0.16, not far from the $\alpha = 0.1$, which makes sense, as refactoring and bad smells, though different concepts are related to improving code structure: bad smell is the symptom of a structural problem, and refactoring aims to overcome such problems.

RQ4. Which are the most used tools by practitioners in software maintenance?

We asked participants whether they use tools to assist in collecting metrics, checking refactoring, and change impact analysis. In the sequel, we present the results for each subject covered in this study.

Software Metrics. Only 33% of participants affirmed to use of a software measurement tool. The participants pointed out 62 tools. Figure 4.6 shows the citation percentage of the most used tools. SonarQube is the most used tool, 32.8% of practitioners used this platform to collect metrics. 6.6% of the participants cited ESLint and Jira. 3.3% of practitioners pointed out CodeFactor, Excel, FindBugs, and NewRelic. Other tools were mentioned just once.

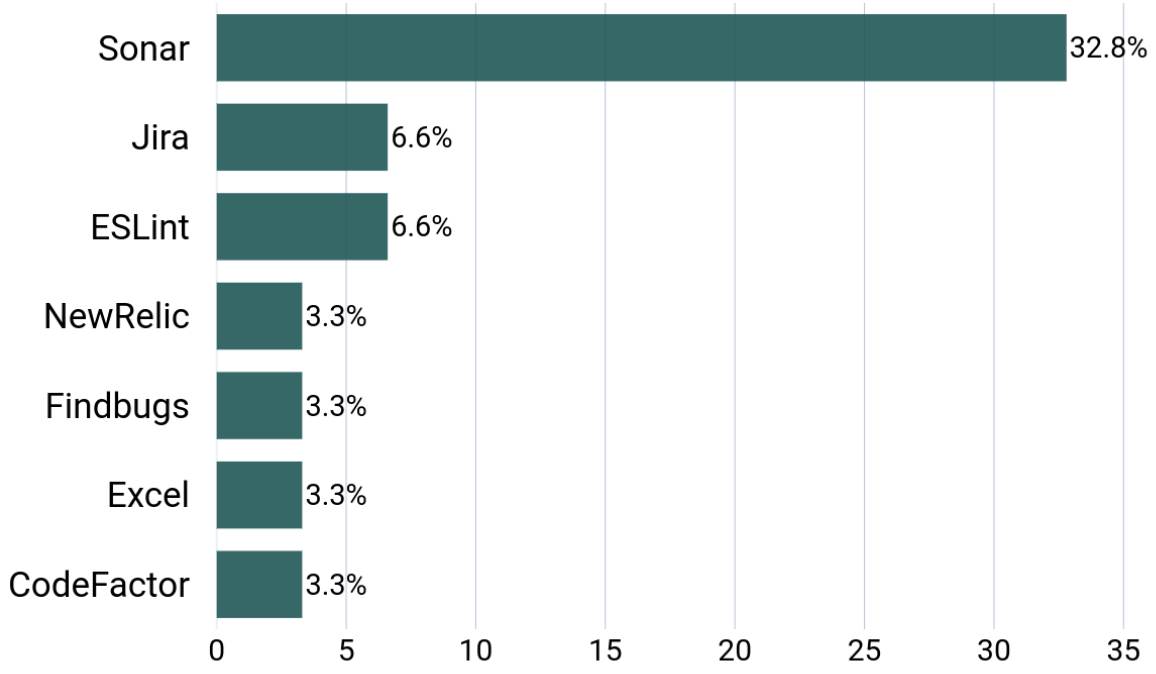

Figure 4.6: Tools most used by practitioners to collect metrics.

Refactoring. Only 36.6% of participants declared they use tools to perform refactoring. They mentioned 68 tools. Figure 4.7 shows the distribution of the citations of the tools. The most commonly used tools for refactoring are the Eclipse (19.1%), IDE Visual Studio

(14.7%), and IntelliJ IDEA (11.8%), or an extension of an IDE, such as ReSharper (2.9%), an extension of the Visual Studio platform. SonarQube is used by 5.9% of the participants to perform code refactoring. The other tools were mentioned just once.

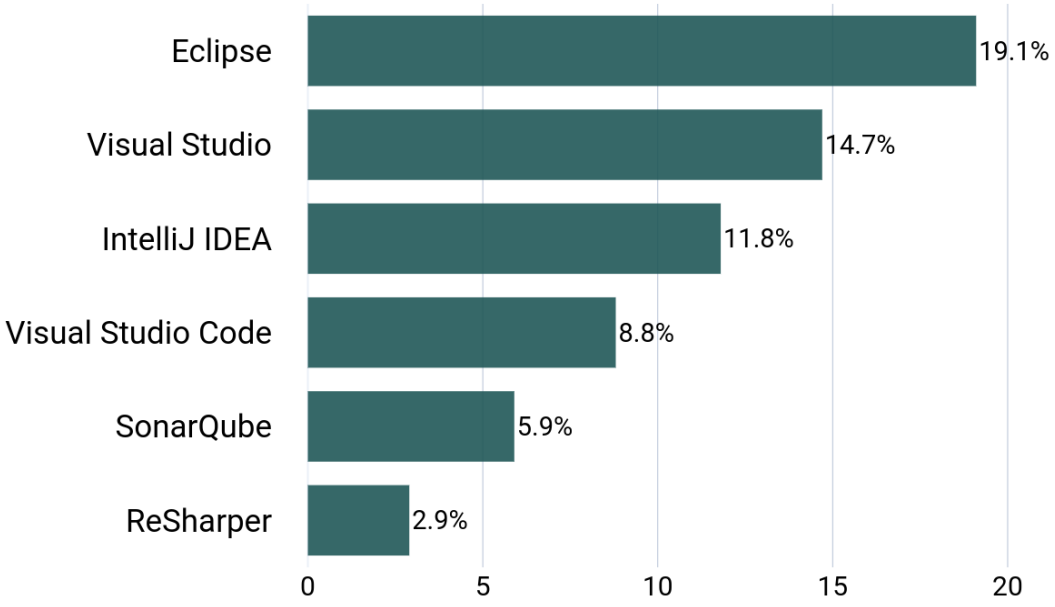

Figure 4.7: Tools most used by practitioners to perform refactoring.

Change Impact Analysis. The participants pointed out 13 tools they apply to perform this task. Figure 4.8 exhibits the percentage of the most used change impact analysis tool. The following IDEs are the most cited tool in this case: Eclipse, IntelliJ IDEA, and Visual Studio. It is essential to mention that 14.3% of the participants informed that "search and replace" is the tool they apply to perform change impact analysis, making it the second most cited tool.

RQ5. How do practitioners perform change impact analysis?

In this research question, we aim to investigate whether and how practitioners perform change impact analysis. Figure 4.9 shows the rank of participants' answers.

Most of the participants, 46.4%, informed that they analyze parts of the code that need to be modified by exploring the code manually and intuitively, guided by prior knowledge about the software source code. Another significant part of the participants, 32.1%, declared that they explore the code manually and intuitively, only sometimes based on prior knowledge about the system. The use of tools to assist in change impact analysis is pointed out by only 10.7% of the participants. A small part of the participants, 5.4%, declared that they do not analyze all the elements that need to be modified and make

the modifications as they identify the problems. Other ways of performing change impact analysis correspond to 3.6% of the responses.

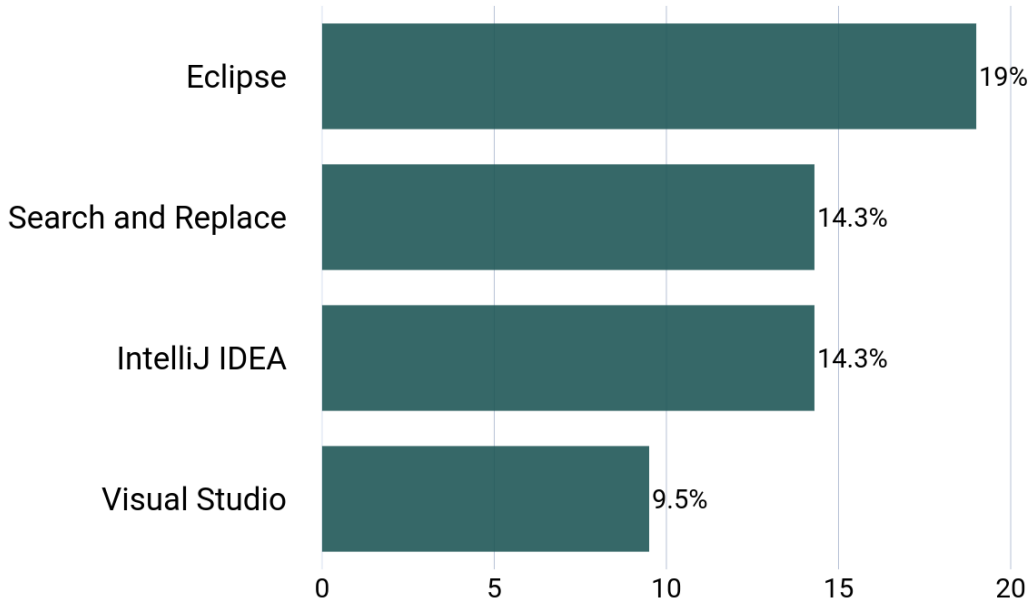

Figure 4.8: Tools most used by practitioners to perform change impact analysis.

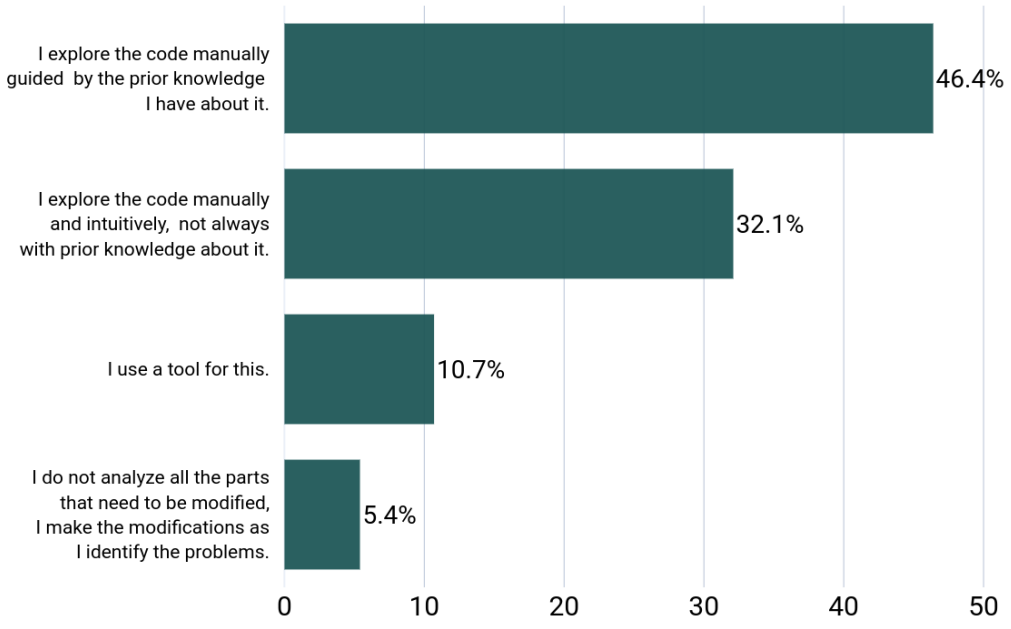

Figure 4.9: How practitioners perform change and impact analysis.

RQ6. Which metrics, refactoring techniques, and bad smells do practitioners apply in their activities?

We asked the participants to point out the software metrics they use the most, the common refactoring techniques performed by them, and the bad smells they use to find. We describe the results in the sequel.

Refactoring. The most common refactoring techniques applied by the participants are the following: *Extract Method* (21.43%), *Rename Method* (13.39%), and *Extract Class* (12.5%).

Metrics. As found in the analysis of the RQ1: *Are developers familiar with software metrics, bad smells, refactoring, and change impact analysis?*, software metrics is the second most well-known subject. However, just a few software metrics were pointed out by the participants. The most cited terms regarding software metrics were: *Number of Bugs* (9.9%); *Test Coverage* (8.91%); and *Cyclomatic Complexity* (7.92%). Many software metrics have been proposed in the literature in the last decades. The survey results show that the software metrics proposed in the literature have not been widely applied in the industry. In particular, it is noticed that practitioners do not mention the widely known software metrics in the academy, such as those proposed by Chidamber and Kemerer [49].

Bad Smell. The most cited bad smells were *Duplicated Code* (23.21%), *Long Method* (19.64%), and *Long Class* (9.82%), indicating that the main concerns of developers when analyzing code structure quality are: code duplication, method size, and class size.

RQ7. What are the biggest challenges practitioners face when carrying out software maintenance?

In an open text field, we asked the participants to write the main difficulties they face when performing maintenance on the software systems. The participants indicated several challenges in software maintenance, and we reported them in Figure 4.10.

Lack of documentation is the most cited challenge; 22.3% of the survey participants have this perception. In general, they described this problem as more critical due to the high turnover, which is very common in IT companies.

Lack of standard for software development is cited by 18.8% of participants and, then, is the second biggest challenge. The participants mentioned that the development patterns established by the companies are not always followed by developers, which makes the code difficult to understand and change.

Legacy system is cited by 18% of participants as a challenge in software maintenance. In this context, the participants refer to the legacy system as being "*long-standing codes in which several people have already worked.*".

Bad coding practices are cited by 17% of participants. According to the participants, this problem mainly involves low code readability, poorly structured functions, replicated methods in various parts of the code, and non-explanatory comments.

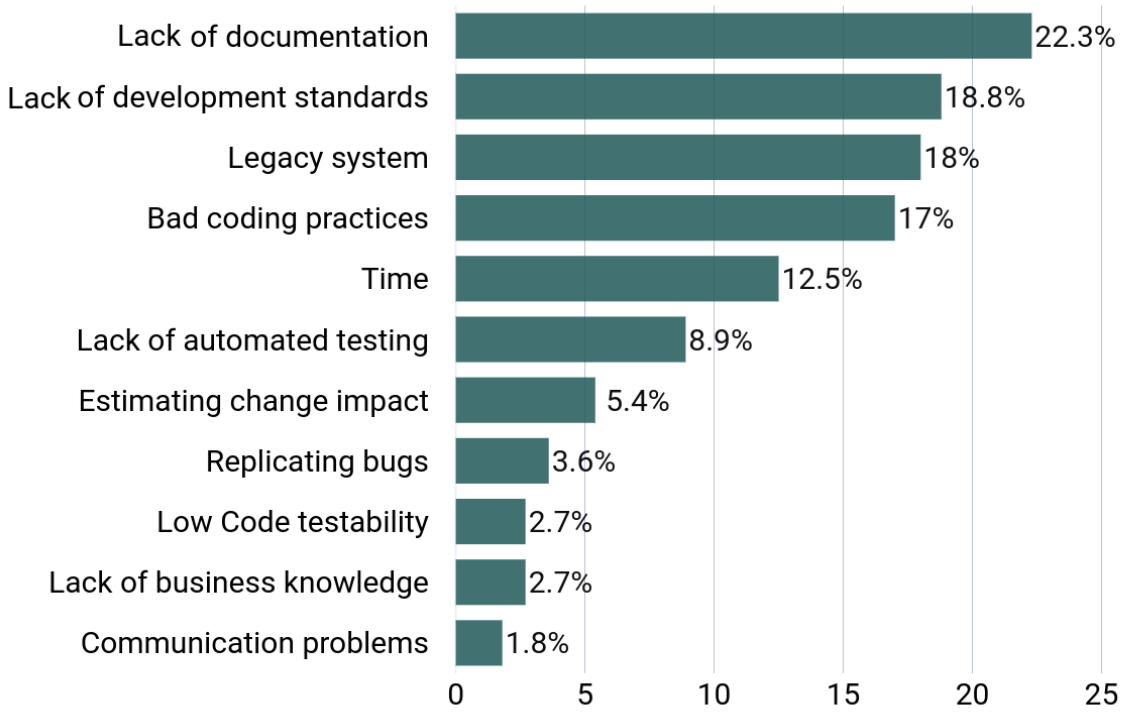

Figure 4.10: Developers' main challenges when performing software maintenance.

Time is also mentioned in participants' responses (12.5%). The main issues pointed out by participants are the short deadlines for performing maintenance-related activities and little time to understand the code and implement the change.

Lack of automated testing (8.9%), difficulty in **Replicating Bugs** (3.6%), and **Low Code Testability** (2.7%) are also challenges raised by the participants. Regarding automated testing, the participants described that the absence of unit, regression, and integration tests could negatively affect software code. The participants associate the difficulty in code testability with the maintenance of monolithic and poorly structured systems.

Estimating change impact is pointed out as a challenge by 5.4% of participants. According to them, not knowing how a change in the code will impact the software system raises the "fear" of side effects in the system.

Lack of business knowledge (2.7%) and **Communication problems** (1.8%) also appeared as challenges for software maintenance. Participants report that when the customer does not have a solid knowledge of the business, they demand more changes in the system. Practitioners also pointed out that communication problems with customers and coworkers negatively impact software maintenance.

Summary of Results:

- Refactoring is the best-known concept among practitioners. However, just a few refactoring techniques - the simpler ones - have been used in practice.
- The most applied refactoring techniques are *Extract Method*, *Rename Method*, and *Extract class*.
- Change impact analysis is the most applied technique. However, without proper tools.
- Manual inspection is commonly used to perform change impact analysis.
- IDE is the tool most used by participants to apply refactoring and change impact analysis.
- The most popular bad smells are *Duplicated Code*, *Long Method*, and *Long Class*.
- Sonar is the most commonly used tool for collecting software metrics.
- The most used metrics are the number of bugs, test coverage, and cyclomatic complexity.
- There is an association (considering $\alpha = 0.1$) between the familiarity with change impact analysis and participants' academic background, bad smells and professional experience, and bad smells and company size.
- The practical application of refactoring is associated with the participants' professional experience (considering $\alpha = 0.05$).
- The biggest challenges to performing maintenance on code are the following: lack of documentation, standard software development, legacy systems, and bad coding practices.

4.4 Discussion

Change impact analysis is not properly performed in practice. The research questions RQ1:*Are developers familiar with software metrics, bad smells, refactoring, and change impact analysis?* and RQ2:*Do practitioners apply software metrics, refactoring, bad smells, and change impact analysis in practice?* investigate if developers

know the concepts considered in this work and if they applied them in practice. The results concerning change impact analysis showed that participants know the term and its meaning, but not widely. On the other hand, it is the most applied concept. This result suggests that even not knowing the term used in the literature, the practitioner applies it. A possible cause is that change impact analysis is an intrinsic and indispensable activity in software maintenance. Most of the developers (78.5%) perform change impact analysis manually, guided or not by the previous knowledge they have about the code analyzed, as described in the results of RQ5. In addition, the results reveal that they do not use specific tools to support change impact analysis. The participants mentioned IDE to perform change impact analysis, which is not a specific tool for such activity. A worrying finding in this study is that participants reported they still use inadequate mechanisms such as "find and replace" to perform change impact analysis, as described in the results of RQ4. Additionally, participants pointed out change impact analysis as an important challenge in software maintenance. These results indicate the need for creating proper techniques to aid change impact analysis.

Refactoring is a popular concept. Different from other subjects, refactoring is a well-known concept in the industry: 94% of participants are familiar with the refactoring concept, 79.5% declared to apply the refactoring, but only 36% affirmed they use tools to perform refactoring. Nevertheless, the refactoring techniques performed in practice are simple and provided by IDE: *Extract Method*, *Rename Class*, and *Extract Class*. It is essential to notice that 93.7% of participants indicated that their companies use an agile methodology or a mix of agile methodologies. Among those methodologies, the participants indicated XP (eXtreme Programming), which has refactoring as the main practice. Therefore, the popularity of agile methodologies might be the cause of the popularity of refactoring. Moreover, agile methodologies emerged from the industry and not academia [26], which may explain its popularity. Bad smell is also prevalent, but not at the same level as refactoring, despite being very close concepts. A possible reason for that may be the lack of tools of practical use to detect bad smells. Therefore, it is essential to create techniques and tools to aid the application of more complex refactoring and the detection of bad smells.

Do companies apply the principles of agile methodologies? Analyzing the main challenges described by participants, it seems that the industry lives in a vicious circle: maintenance in legacy systems must be performed, but there is a lack of documentation about these systems. Therefore, with restricted deadlines, the developer tries to solve the problem in the way that is possible, but not always in the best way: they possibly abandon standards and good development practices, as well as generate complex and unreadable codes, in which another developer, at some point,

will need to do maintenance. This code will probably be undocumented, the software structure will be more complex, and the maintenance will become increasingly complex.

In the survey, 99.1% of respondents stated that the company where they work applies agile methods. However, the Agile Manifesto does not prescribe the total abandonment of documentation. Moreover, standards and good practices are highly recommended in agile methodologies. Therefore, further investigation is essential to assess the actual application of agile practices by organizations and to identify to which extent organizations still follow the premise of "responding to change over a plan" [26].

Software metrics are not yet widely applied in the industry. The results of this survey show that software metrics are considered important by most participants: 68.8 % declared they consider software measurement important or very important, and also 68,8% of participants affirm they know the concept of software metrics. However, a relevant part of the participants, 21,4%, reported they do not have the background to give their opinion about the importance of software metrics. Moreover, the survey also shows that software metrics are not widely applied in the industry yet: only 33% declared applying software metrics in practice and using tools for collecting software metrics.

These results are exciting because many software metrics have been proposed and evaluated in the literature. For instance, the work of Chidamber and Kemerer, known as CK metrics, was published 25 years ago and is widely known by software engineering researchers [49]. However, none of the participants mentioned these metrics. Therefore, this finding suggests that software metrics research might fail in proposing techniques and tools of practical use and/or making their proposals known to the industry.

Another point of discussion in this context is that, as stated by CMMI (Capability Maturity Model Integration), measurement and analysis are characteristics of organizations in Level 2 of maturity. Still, quantitative project management is a characteristic of highly mature organizations in Level 4 [153]. Therefore, another possible explanation for software metrics is not widely applied might be the low maturity of most organizations.

4.5 Threats to Validity

The survey data was based on the responses collected by a questionnaire, and therefore, it is susceptible to the participants' interpretation. To mitigate this issue and to ensure that the questions would be clear to the participants, providing accurate data for our analysis, we ran a pilot questionnaire. Based on the answers collected in this preliminary round, we constructed a final survey reducing or removing ambiguities and biases.

In this study, we present results based on the responses given by 112 participants. Aiming representativeness, we just considered exclusively participants that are professionals in software development and maintenance. Besides, the sample used in this study comprises participants from 92 companies and 12 countries, with a wide range of years of professional experience, with all kinds of academic backgrounds, and using many programming languages. Even though, as occurs in this kind of study, it is not possible to claim that the results can be generalized. Nevertheless, the results are of value both to academia and industry because it reveals how software maintenance has been practiced.

To identify the main challenges participants face when performing software maintenance, we asked them to describe them in an open text field. The answers to this question were manually categorized; therefore, they are subject to interpretation by those who performed this categorization. To mitigate this threat to validity, we standardize the labels used in the categorization, i.e., we identified the keywords mentioned in the participants' responses, such as documentation, legacy system, readability, and standard, among others. Besides, the label assigned to each answer was made separately by two authors of this study. After this, the classifications were compared to obtain the final classification of each answer. If there were any divergence in the categorization, it was analyzed by both authors to obtain a consensus.

4.6 Final Remarks

We surveyed software practitioners to investigate whether and how software maintenance techniques have been applied in practice. In particular, we investigated the usage of the following concepts and techniques: software metrics, refactoring, bad smells, and change analysis impact. For this purpose, we surveyed 112 software development practitioners from 92 companies and 12 countries. The results showed that change impact

analysis is the most applied technique among the ones considered in this work. However, there is a lack of proper tool support to perform change impact analysis. Although refactoring is widespread, only some refactoring techniques have been applied in practice. Moreover, refactoring is mainly provided by IDE. Bad smells and software metrics are the less known and applied concepts.

This study also revealed that participants considered the lack of system documentation, development patterns, and legacy software as the leading software maintenance challenges. The results indicate that software maintenance demands even more community effort to develop and provide proper tools and methods for software maintenance, especially in change impact analysis and software measurement.

This chapter described the study we performed to identify the state of the practice on software maintenance, including change impact analysis. Chapter 5 presents a systematic mapping review we carried out to identify the state-of-art on change impact analysis.

Chapter 5

Systematic Mapping Review

Change impact analysis (CIA) allows for identifying the pieces of the software system affected by an initial set of changes. Therefore, CIA is an imperative activity to ensure software quality since it aids in guaranteeing that new faults will not be inserted in the software system. Given the importance of this activity for software development and maintenance, several researchers have proposed, over the years, methods and tools to assist developers in performing CIA. Some works sought to present the CIA state-of-the-art through systematic reviews [55, 105, 11, 57]. However, they consider specific aspects of methods and tools and do not comprehensively analyze the works related to CIA. Therefore, we conducted a Systematic Mapping Review (SMR) on change impact analysis as part of this Ph.D. dissertation. The mapping aims to carry out a broad characterization of the methods and tools proposed for the CIA.

This chapter presents the SMR on CIA and is organized as follows. Section 5.1 describes the method we applied to perform the SMR. Section 5.2 presents the framework we defined to classify the CIA studies. Section 5.3 describes the characteristics of the studies analyzed in this SMR: the number of studies published by year and venues. Section 5.4 presents the results of the SMR. Section 5.5 discusses the results, and Section 5.6 brings the conclusions of the SMR

5.1 Study Design

This section describes the method we applied to this study. First, we justify why this study is characterized as a systematic mapping review. Then, we describe the planning phase, involving the research question statements, the digital libraries considered in the search, the search string definition, and the criteria to include and exclude the documents. Finally, we describe the SMR execution phase.

5.1.1 Why a Systematic Mapping Review?

According to Wohlin et al., [162], researchers can perform the aggregation of empirical studies through a Systematic Literature Review (SLR) or a Systematic Mapping Review (SMR) (a.k.a Mapping Study). In an SLR, researchers respond, similarly to empirical studies, to specific research questions on a given topic. This type of review presents a vertical deepening in a particular aspect of the analyzed theme and aims to aggregate the studies according to their research outcomes [89]. On the other hand, an SMR seeks to present an overview of state of the art related to the theme, i.e., it is based on broader research questions. According to Kitchenham et al. [89], "the categories used in a mapping study are usually based on publication information (authors' names, authors' affiliations, publication source, publication type, publication date, etc.) and or information about the research methods used". Since we aim to characterize various aspects of the proposed methods and tools for the CIA, we concluded that performing an SMR is more appropriate to our purposes. To perform the SMR, we followed the guidelines proposed by Kitchenham and Charters [88].

5.1.2 Planning

This section describes the planning phase's steps, which are the following:

- formulating the research questions
- selecting the search databases
- constructing the search string
- defining the criteria to select the studies.

5.1.2.1 Research Questions

This SMR aims to provide an analysis of the state-of-art on software change impact analysis. For this purpose, we define the following research questions:

RQ1. Which approaches and tools are proposed for CIA?

RQ2. Which are the characteristics of these approaches and tools?

RQ3. Which methods and metrics did the studies use in evaluating these approaches and tools?

5.1.2.2 Electronic Databases

We considered the main digital libraries and electronic databases of software engineering publications for collecting the studies. Table 5.1 exhibits the chosen databases, their respective websites, and the number of documents we retrieved from each database.

DataBase	Address	#Documents
ACM	dl.acm.org	87
Engineering Village (Compendex)	www.engineeringvillage.com	259
IEEE Xplore	ieeexplore.ieee.org	294
Science Direct	www.sciencedirect.com	300
Scopus	www.scopus.com	323
SpringerLink	link.springer.com	621
Web of Science	www.webofscience.com	122
Total		**2006**

Table 5.1: The electronic databases used in the SMR.

5.1.2.3 Search String

The search string was built to be as comprehensive as possible. For that, we conducted a pilot study with search strings composed of the most usual terms regarding change impact analysis. Then, we applied them in each database, evaluating the results and aiming to identify which search string has reached as many studies as possible in the literature. At the end of this process, we defined the following search string and retrieved 2,006 documents, as shown in Table 5.1:

Search String

("change impact" OR "change propagation" OR "modification impact" OR "modification propagation" OR "ripple effect" OR "co-change" OR "software modification") AND "software maintenance"

5.1.2.4 Inclusion and Exclusion Criteria

A search made in a digital library usually returns documents not related to the search string keywords. Therefore, establishing inclusion and exclusion criteria is necessary to guarantee that we will include and analyze only relevant studies related to change impact analysis. For this reason, we defined the inclusion and the exclusion criteria to select the primary studies, reported in Table 5.2.

Inclusion Criteria	Exclusion Criteria
Papers written in English	Conferences Proceedings
Papers available online	Round tables/Lectures
Papers about methods and tools related to CIA in software	Duplications

Table 5.2: List of the inclusion and exclusion criteria.

5.1.3 Execution Phase

The execution phase consisted of three steps: the search, the studies selection, and the studies' analysis. In the search process, we applied the search string in the selected databases to identify the primary studies. In selecting the studies, we used the exclusion and inclusion criteria. At the end of these two steps, we identified 141 papers presenting methods and tools related to change impact analysis. Then, we analyzed and summarized these studies to answer the research questions.

5.1.3.1 Search Process

With the search string described in Section 5.1.2.3, we retrieved a total of 2006 studies from the seven electronic databases, including duplicates. We searched for all studies published from 1978 to 2021 since the first year allowed by the databases is 1978, and we carried out the search process in December 2021.

5.1.3.2 Selection of the Studies

This section describes the five stages of the selection process, aiming to select relevant papers according to their contents. Table 5.3 shows the number of studies selected in each stage.

Stage 1: Grouping all the studies. As the first step of the selection step, we grouped all the documents retrieved in our search in the databases.

Stage 2: Removing duplicates and non-English documents. We removed all the duplicate files and any paper not written in English. As the final result of this step, we obtained 1.444 documents.

Stage 3: Inclusion and exclusion criteria. We applied the inclusion and exclusion criteria defined in Section 5.1.2.4, removing from our results: the conference proceedings, lecture notes, and round tables, thus ensuring that we considered only papers.

Stage 4: Reading titles and abstracts. We excluded papers whose titles and the content of the abstracts did not present elements related to methods or tools for change impact analysis in software systems.

Stage 5: Complete reading. We proceed with the reading of the papers. In our final set of papers, we only included those that present a well-established method or tool related to change impact analysis in software systems written in any programming language. As a final result, we selected 141 papers.

Stage	Description	#Documents
1	All files returned from all database	2006
2	Removing duplicate and non-English documents	1444
3	Application of inclusion/exclusion criteria	1082
4	Reading title and abstract	277
5	Complete reading	168
Final Result		**141**

Table 5.3: Number of documents after applying each stage of the selection process.

5.2 Framework for CIA Studies Characteristics

After reading the 141 papers selected in this systematic mapping, we use a framework proposed by Li et al. [105] as the basis for structuring the characterization of the studies analyzed in this work. Figure 5.1 exhibits this framework. We identified the primary characteristics of CIA studies, which are divided into other features: method, change impact analysis approach, data source, technique, analyzed elements, evaluation method, and supported language. We applied this framework to describe the results of this systematic mapping. This section presents the framework.

5.2.1 Method

As stated by Wohlin et al. [162], software engineering is a broad field in computer science that ranges from operational issues, such as databases and program languages, to human aspects. Therefore, researchers need to apply the correct method when developing their studies. We use the following four methods described by Wohlin et al. to classify the analyzed papers according to their software engineering research context:

Scientific Method: is used when the researchers aim to propose a model or theory based on the observation of the real world. The model/theory needs to be validated thru measurement and analysis.

Analytical Method: is based on the principle that researchers propose a formal theory or a set of axioms, develop them, derive the results, and compare them with observations from the real world.

Empirical Method: is based on statistical or qualitative methods to validate a proposed model. These methods are applied to study cases, and then measures and analyses are performed to validate the proposed model.

Engineering Method: is used when a previous solution is studied and, based on this, a new and better solution is proposed. Then, the proposed solution is developed, and measurement and analysis to validate it are performed.

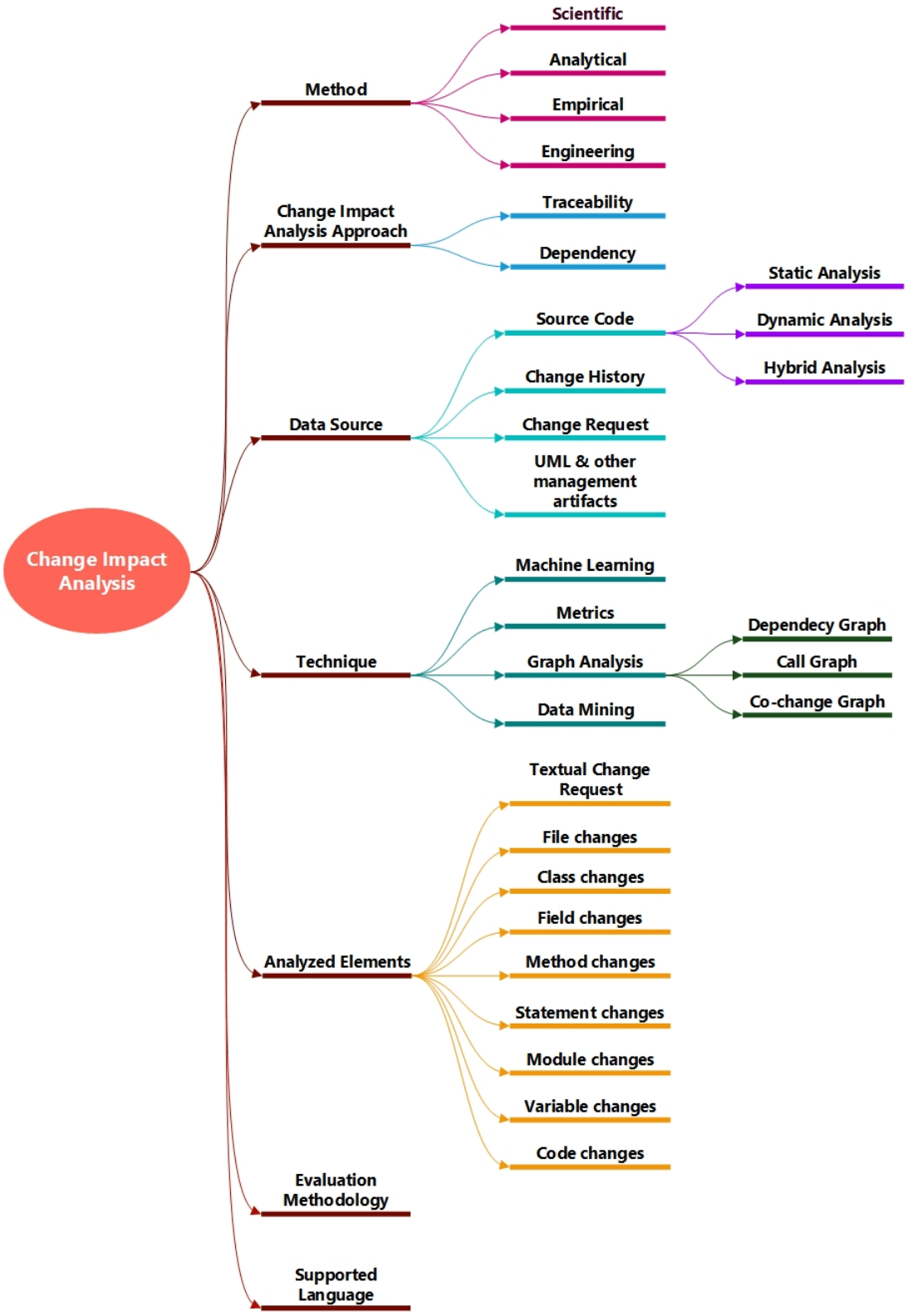

Figure 5.1: Framework for CIA studies characteristics.

5.2.2 Change Impact Analysis Approaches

We may categorize a CIA approach into traceability and dependency, described as follows.

Traceability: is applied in studies that analyze relationships among several software artifacts from different levels of abstraction. For instance, Lehnert et al. [99, 100] proposed an approach to change impact analysis that combines UML models, Java source code, and JUnit Tests. There is no pattern concerning the types of artifacts used by the researchers, i.e., we identified several types of artifacts considered in the studies.

Dependency: is applied in studies that analyze software artifacts on the same level of abstraction, e.g., studies that only use source code to predict change impact analysis as in Chaumun et al. [42] model. This approach is usually based on source code analysis.

5.2.3 Data Source

The data analyzed by a CIA approach are of four types, described as follows.

Source Code: in this study, we classified the data source as source code when the model and tools used elements related to the source code, i.e., source code snippets, the source code as a whole, the execution path, and the byte code of the software. There are three types of source code analysis: static, which considers only static information about the source code; dynamic, which considers only information about the execution of the code; and hybrid, which considers both types of information.

Change History: in software development, using versioning and code management tools is very common. Code versioning consists of storing the history of modifications made to system files and other information related to this modification, such as the modification date, author, and description, among additional metadata.

Change Request: in addition to file modifications, versioning, and code management, systems can also store the description of the reason for the modification, which we call a change request. Unlike the source code modification description - which only describes the modification performed, a change request contains a high-level

description of what should be added or modified in the system. The term 'bug report' is also considered a change request in this work.

UML and other Management Artifacts: in this study, we classified UML and Management Artifacts as a source of data related to the description or specification of the software architecture; design templates, domain; requirements specification; UML artifacts. Therefore, works labeled in this category have artifacts describing the system's structure to be analyzed as a data source.

5.2.4 Technique

The CIA approaches apply several techniques, among which four are more present in the evaluated works: graph analysis, data mining, machine learning, and metrics.

5.2.5 Analyzed Elements

The analyzed elements in a CIA approach represent the inputs used by the methods to detect change propagation, i.e., they refer to which type of modification is interpreted by the CIA method. As main analyzed elements, we identified the following categories: textual change requests, file changes, class changes, field changes, method changes, statement changes, module changes, variable changes, and code changes.

5.2.6 Evaluation Method

The studies on CIA have applied different methods to evaluate their results, such as empirical evaluation, case study, comparative analysis, and examples of usage. They have also used several metrics in the evaluation, such as accuracy, precision, recall, and correlation.

5.2.7 Supported Language

A CIA approach may be independent of a programming language when the CIA approach is for a level that does not involve programming or when the CIA approach is generic, i.e., it is defined in such a way it may be implemented for any programming language. However, a CIA may be defined considering a specific programming language.

5.3 CIA Studies Characteristics

In this section, we present the number of papers published by year and venue. We also describe the framework we defined to characterize CIA studies.

5.3.1 Change Impact Analysis thru the Time

The first conference paper we could retrieve about change impact analysis was published in 1978 by Yau et al.[167] at the Computer Software and Applications Conference (COMPSAC'78). In 1980, the authors of the first paper, Yau and Collofello [168], published in the IEEE Transactions on Software Engineering their first journal article about the same subject. Since then, the general interest in this topic has continued. In subsequent years, the scientific community has counted on at least one article per year on this subject.

Figure 5.2 exhibits the distribution of publications related to change impact analysis in software systems over time. The results of our research show that starting in 2006, a significant increase in the number of publications related to change impact analysis appeared. Between 1994 and 2021, the average of relevant publications related to the topic was five. The apex of publications on this topic occurred in 2009, with 12 publications in that year. Furthermore, the graph in Figure 5.2 shows that change impact analysis is a topic under researchers' discussion. However, despite the practical importance of change impact analysis, the number of publications on this topic is not high, which shows the need for further research.

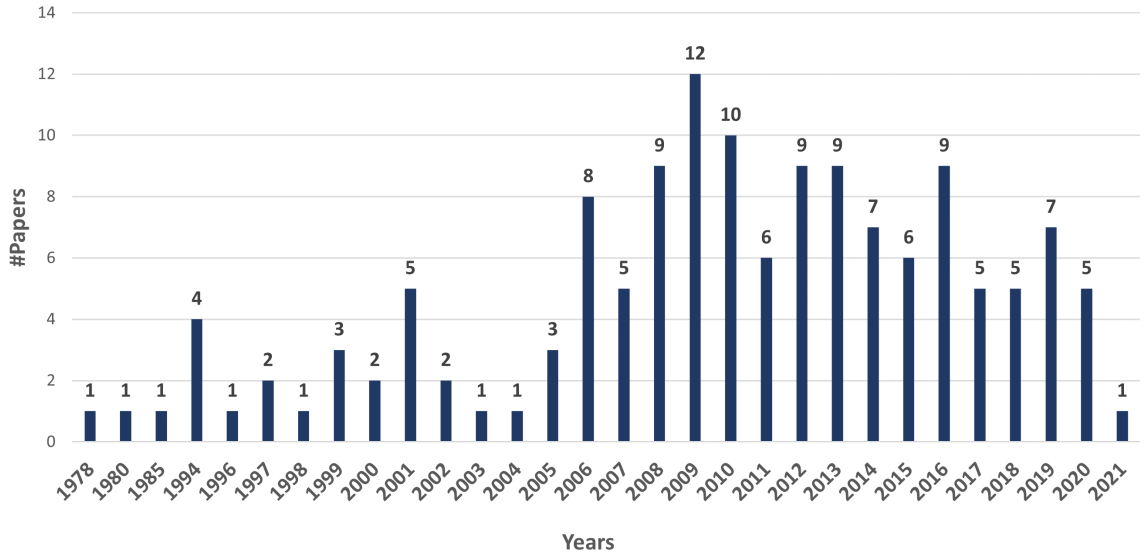

Figure 5.2: Number of published papers per year.

5.3.2 Change Impact Analysis Publication Venues

The publications analyzed in this systematic mapping were published by well-known and established vehicles in the academic scenario. We identified publications in 61 conferences and journals.

Analyzing which conferences and journals published the most studies on change impact analysis in software systems, we found that 20 venues have published 78 publications (55.3%). The other venues have only one publication each. Figure 5.3 shows the number of publications by venue. To improve the data visualization in the chart, we omitted the publication vehicles with only one publication. The changes in the conferences' names are worth noting over the years. The International Conference on Software Analysis, Evolution, and Reengineering (SANER) is the result of the merger of the European Conference on Software Maintenance and Reengineering (CSMR) with another venue called the Working Conference on Reverse Engineering (WCRE). Besides, the International Conference on Software Maintenance and Evolution (ICSME) was previously named as International Conference on Software Maintenance (ICSM).

Among the conferences, the one that stands out the most is ICSME (previously named ICSM), which published 22 papers (15.6% of the analyzed studies). Regarding the journals, the one with the higher number of publications on change impact analysis is the Journal of Systems and Software (JSS), with six articles related to change impact analysis.

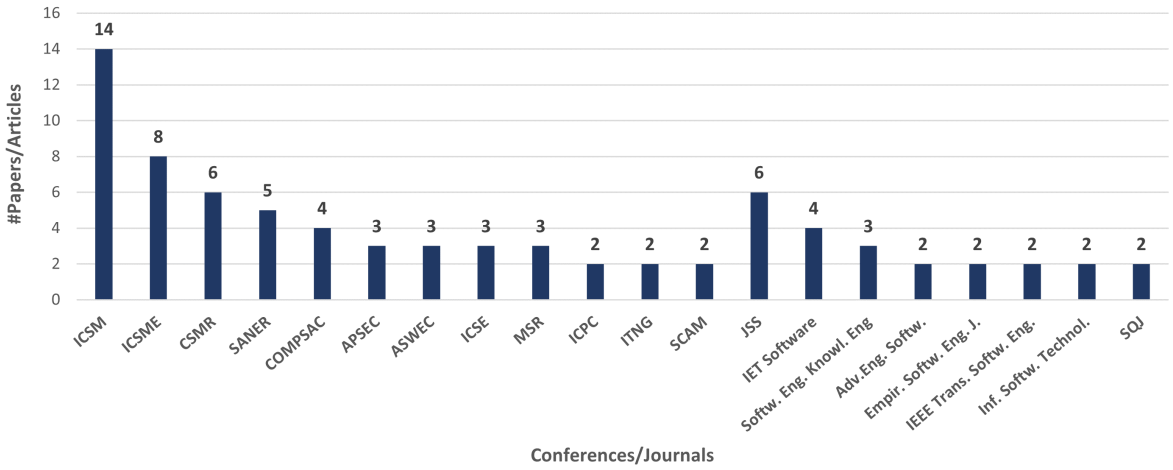

Figure 5.3: Number of papers published by venues.

5.4 Results

In this section, we answer the RQs thru the classification of the studies related to change impact analysis according to the framework we defined in Section 5.2.

5.4.1 Models and Tools

Seeking to answer the research question *RQ1 - Which approaches and tools are proposed for CIA?*, during our review, we found works of three natures: (i) works that present models for software propagation analysis (*model*); (ii) works that, in addition to a model, also present tools that researchers built to support the execution of the proposed model (*model and tool*), and (iii) works that present tools for previously proposed models (*tool*). Table 5.4 presents the results of this classification. Works presenting models are the most common among the analyzed data, 76%. Works presenting models and tools are the second category, representing 17.7% of the works. Works that present only tools correspond to 4,3% of the studies.

Work Nature	Papers
Model	[167], [168], [7], [156], [102], [42], [96], [43], [56], [141], [31], [76], [21], [54], [171],[33], [174], [118], [3], [154], [145], [50], [165], [121], [41], [68], [144], [8], [117], [66], [40], [138], [18], [107], [163], [2], [44], [135], [77], [39], [87], [108], [150], [47], [17], [116], [114], [75], [61], [131], [92], [84], [140], [132], [23], [69], [104], [35], [93], [151], [86], [103], [124], [83], [53], [106], [147], [149], [60], [125], [119], [19], [48], [13], [164], [62], [1], [129], [70], [139], [110], [130], [14], [158], [112], [24], [16], [159], [72], [85], [161], [127], [111], [9], [79], [74], [10], [12], [146], [115], [160], [122], [123], [5], [6], [101]
Model and Tool	[95], [82], [136], [73], [65], [97], [28], [91], [173], [155], [157], [152], [170], [172], [29], [22], [45], [80], [38], [134], [52], [166], [169], [20], [100], [143], [142], [51]
Tool	[27], [46], [71], [34], [4], [113]

Table 5.4: Nature of the works.

5.4.2 Methods and Tools Characteristics

In this section, we answer *RQ2 - Which are the characteristics of these approaches and tools?*, showing the results of the following characteristics of the analyzed studies: scientific method, change impact analysis approach, data source, technique, elements analyzed, and supported languages.

5.4.2.1 Method

The empirical method is the most commonly used by the studies: 114 out of 141 analyzed papers applied this method. The scientific method was applied in 21 studies, and only three studies applied the analytical method. None of the researched papers presented characteristics categorized as an engineering method. Table 5.5 reports the studies by the scientific method they applied.

Method	Papers
Empirical	[156], [95], [82], [102], [136], [73], [42], [27], [97], [43], [56], [141], [28], [46], [91], [173], [71], [76], [34], [155], [21], [33], [152], [121], [41], [170], [172], [68], [29], [144], [22], [117], [45], [66], [40], [138], [18], [107], [163], [2], [80], [44], [38], [135], [39], [150], [17], [134], [116], [114], [75], [52], [131], [92], [166], [140], [132], [23], [69], [104], [35], [169], [93], [20], [151], [86], [103], [124], [83], [100], [53], [106], [147], [149], [60], [125], [119], [19], [48], [13], [164], [62], [1], [143], [142], [139], [110], [130], [14], [158], [112], [24], [16], [159], [72], [85], [161], [111], [9], [79], [4], [74], [10], [12], [146], [115], [160], [51], [122], [113], [123], [5], [6], [101]
Scientific	[65], [96], [31], [157], [54], [171], [174], [118], [3], [154], [145], [50], [165], [8], [77], [87], [47], [61], [129], [70], [127]
Analytical	[7], [108], [84]

Table 5.5: Scientific methods applied by the works.

5.4.2.2 Change Impact Analysis Approaches

Regarding the relationship between the elements of change, we categorized the studies into two groups: those based on traceability between software entities and those based on dependency between them.

Table 5.6 presents the obtained results. We found that, among the returned works in our search, most researchers (69.7%) use dependency as an abstraction level when developing models and tools for change impact analysis. We found that traceability is an abstraction level in 30.3% of the analyzed works.

Abstraction Level	Papers
Dependency	[167], [168], [7], [95], [102], [73], [96], [27], [43], [141], [28], [46], [91], [173], [71], [76], [34], [155], [21], [171], [33], [174], [118], [154], [170], [172], [29], [144], [22], [8], [66], [138], [18], [163], [2], [135], [77], [39], [87], [150], [47], [17], [116], [114], [52], [61], [131], [92], [166], [140], [132], [23], [104], [35], [169], [93], [20], [151], [86], [103], [124], [83], [53], [106], [147], [149], [119], [19], [48], [13], [164], [62], [1], [129], [70],[143], [139], [110], [130], [14], [159], [72], [85], [161], [111], [9], [79], [4], [74], [10], [12], [146], [115], [51], [122], [123], [6]
Rastreability	[156], [82], [136], [65], [42], [97], [56], [31], [157], [54], [3], [152], [145], [50], [165], [121], [41], [68], [117], [45], [40], [107], [80], [44], [38], [108], [134], [75], [84], [69], [100], [60], [125], [142], [158], [112], [24], [16], [127], [160], [113], [5], [101]

Table 5.6: Change Impact Analysis approaches applied by the works.

5.4.2.3 Data Source

The data source is the basis for developing an approach for change impact analysis. During our investigation, we identified 50 types of data sources used in works on change impact analysis. We reported the results in Tables 5.7 and 5.8.

Source code is the most used data source - present in 56.33% of the papers, and the change history is the second most used data source (30.3%). It is worth noting that the works frequently use two or more data sources - 29 out of 50 methods and tools (58%). Besides UML diagrams and change requests presented in Section 5.2.3, other examples of data sources applied in studies on change impact analysis are: architectural descriptions, design models and information, requirements descriptions, and bug reports.

Data Source	Papers
Architecture Description	[77]
Architecture Description and Design Model	[61]
Architecture Description and Change History	[9]
Change History	[174], [93], [124], [147], [119], [48], [139], [161], [146], [123], [6], [39], [17], [170], [144], [2], [23], [1], [129], [130], [83]
Change History and Source Code	[122], [68], [95], [82], [5], [76], [157], [121], [125], [164], [16], [112]
Change History and UML	[163], [75]
Change History, Change Request and Source Code	[69], [60]
Change History, Source Code and Design Information	[101]
Change History, Source Code and Bug Report	[38]
Change History, Source Code, Architecture Design and Byte Code	[108]
Design Documents, Requirements Description, and Software Components	[50]
Design Information	[54]
Domain Model	[18], [19]
Feature Model	[132]
Formal Architecture Specification	[173]
History Modification Metadata	[160]
Requirements Specification	[7], [47], [171]
Requirements Specification, and Design Elements	[80]
Requirements Specification and UML Diagrams	[44]
Software Artifacts	[73], [65], [96]
Requirements Artifacts and Execution Path	[84]

Table 5.7: Data source used to perform change impact analysis. (Part I)

Data Source	Papers
Software Documentation	[156]
Software Life-cycle objects	[31]
Source Code	[14], [159], [167], [168], [102], [136], [42], [27], [43], [141], [28], [46], [91], [71], [34], [155], [21], [33], [118], [172], [29], [22], [117], [138], [135], [150], [114], [52], [166], [140], [104], [20], [151], [103], [53], [149], [13], [62], [70], [143], [72], [74], [12], [115], [51], [87]
Source Code (Bytecode)	[111], [35]
Source Code, and AST	[79]
Source Code, and Business Process	[40],[165], [41]
Source Code, and Call History	[131]
Source Code, and Configuration Artifacts	[169]
Source Code, and Database Schema	[56]
Source Code, and Software Documentation	
Source Code, and Software Release	[110]
Source Code, Requirements Artifacts, and Test cases	[142]
Source Code, Requirement Specification, and Architectural Models	[127]
Source Code, Requirement Specification and System Design	[24]
Source Code, UML Diagrams, Requirements Specification, and System Database	[45]
Source Code, UML Diagrams, and JUnit tests	[100]
System Artifacts ,and Metrics	[134]
System Call Graph	[106]
System Components	[8]
System Entities	[107]
UML Artifacts	[66], [86], [3], [85],[154], [145], [10]
UML Diagrams, Architecture Elements, and Architecture Rationales	[152]

Table 5.8: Data source used to perform change impact analysis. (Part II)

Source Code Analysis When using the source code as the data source in the studies, we investigated the nature of the source code analysis. It may be dynamic, static, or hybrid. *Dynamic analysis* is used to capture data thru the source code run-time execution. The *static analysis* collects data when the source code is not executed, and the *hybrid approach* occurs when the model or tool uses both analysis types to perform source code analysis. Table 5.9 shows the categorization result. static analysis is the most used source code analysis - 91.6% of the works apply it. The hybrid analysis is the second type most used - nine papers (6.3%) applied it. The dynamic analysis was applied in only three works (2.2%).

Source Code Analysis	Papers
Dynamic	[69], [117], [53]
Hybrid	[84], [46], [33], [138], [114], [140], [20], [131], [24]
Static	[116], [92], [95], [82], [5], [76], [157], [121], [125], [164], [16], [112], [60], [158], [101], [38], [108], [73], [65], [96], [14], [159], [167], [168], [102], [136], [42], [27], [43], [141], [28], [91], [71], [34], [155], [21], [118], [172], [29], [22], [135], [150], [52], [166], [104], [151], [103], [149], [13], [62], [70], [143], [72], [74], [12], [115], [51], [87], [111], [35], [79], [40], [165], [41], [169], [56], [97], [127], [142], [100], [45], [107], [86]

Table 5.9: Type of source code analysis applied by the works.

5.4.2.4 Technique

We analyzed which techniques were applied by researchers when developing methods and tools for change impact analysis. Tables 5.10 and 5.11 exhibit the results.

We track 55 different techniques among the analyzed works. The most common technique researchers use is Graph Analysis - 61 works (42.6%). Data Mining is the second most used technique - 17 works (12.7%). The use of Metrics for change impact analysis is also relevant - 11 works (7.9%).

Observing the results, we saw that research usually only applies one technique when developing methods for change impact analysis. Apart from Graph Analysis, Data Mining, and Metrics, the researchers have used various techniques, which show how vast the change impact analysis field is.

Graph Analysis As shown in Section 5.4.2.4, Graph Analysis is extensively used in research related to change impact analysis. Aiming to understand such a technique better, we analyzed which type of graph was used in these studies. Table 5.12 shows the results. We identified 27 types of graphs. The dependency graph and call graph are the most used, corresponding to 27 (44.3%) and 17 (28%), respectively.

Technique	Papers
Architecture Graph	[173]
Architecture Reflection	[87]
Association Rule Mining	[139]
Association Rules	[124] [123]
AST and Control Dependence Graph	[159]
Belief Desire Intention (BDI) and Object Constraint Language (OCL)	[54]
Breadth First Search	[86]
Breadth First Search and Method Execution Path	[20]
Change Proximity	[4]
Components Linkage and Graph Analysis	[50] [45]
Contextual Information	[161] [160]
Data Mining	[9] [130]
Data Mining - Association Rules	[17] [116] [48] [122]
Data Mining - Clustering	[147] [23] [146]
Data Mining and Co-change	[170]
Data Mining and Information Retrieval	[69] [60]
Data Mining and Machine Learning - Random Forest	[119]
Data Mining and Metrics	[16] [6]
Dependency Analysis	[115] [145] [134]
Dependency Matrix	[27] [31] [132] [28]
Dynamic mapping feature	[3]
Execution Traces	[53]
Formal Concept Analysis (FCA)	[149] [103]
Genetic Algorithm and Mining Execution Trace	[110]
Graph Analysis	[168] [156] [95] [102] [136] [73] [65] [96] [97] [56] [141] [46] [34] [155] [21] [171] [33] [174] [152] [165] [121] [172] [68] [29] [8] [40] [138] [107] [44] [38] [77] [150] [47] [114] [52] [131] [92] [166] [104] [169] [93] [151] [106] [19] [13] [143] [14] [158] [79] [12] [71]
Graph Analysis and AST	[157] [22]
Graph Analysis and Lexical analysis	[167]
Graph Analysis and Metrics	[18]
Impact Rules	[100]
Information Retrieval	[135] [83]
Information Retrieval, Execution Trace and Graph Analysis	[35]

Table 5.10: Techniques applied by the works. (Part I)

Technique	Papers
Information Retrieval and Graph Analysis	[85]
Logical Prediction	[163]
Machine Learning	[164] [144]
Machine Learning - Association Rules	[39]
Machine Learning - Bayesian Networks	[2] [1]
Machine Learning - Random forest classifier	[112]
Metrics	[41] [75] [129] [51] [5] [101] [43]
Metrics and Matrix Dependency	[118]
Model Slicing	[113]
Predicate Logic (Horn clause)	[7]
Program Slicing	[70] [154]
Program Slicing and Graph Analysis	[140]
Reverse Engineering	[91] [84]
Semantic Analysis	[74]
Semantic Analysis and Information Retrieval	[125]
Semantic Analysis and Trace Execution	[62]
Semantic Trace	[72]
Semantic Analysis and Version Comparison	[82]
Heuristics set	[117] [76]
Simulation	[108]
Stochastic model	[111]
Trace Model	[66]
Trace Model	[80]
Traceability Analysis	[24]
Traceability Links and Graph Analysis	[127]
Traceability Matrix	[142]
Traceability-based algorithm and Rule-based inference engine	[61]
User Requirements Notation	[10]
Version Comparison	[42]

Table 5.11: Techniques applied by the works. (Part II)

5.4.2.5 Analyzed Elements

The analyzed elements cover which elements of the artifacts were used by the methods when performing changing impact analysis. We identified 47 elements considered by the works on change impact analysis. Tables 5.13 and 5.14 exhibits the results.

For most papers (43.67%), the analyzed element is related to the source code change. The methods and tools analyzed have at least one of these elements: Class, Field, Method, Function, Variable, Module, Code - any change in the source code, Statement - any command in a procedural language. Among these elements, the use of Classes as the only data source for the change analysis is the most common, 33.9%. Another

element that showed a relevant percentage (10.6%) of use were the changes made to the files. We observed that 8.5% of the papers analyzed elements related to the change in the requirements and features of the software. Among the mapped studies, we found the following keywords related to the system's requirements: *requirement changes*, *system requirement change*, *feature changes*.

Graph Type	Papers
Abstract Semantics Graph	[42]
Abstract System Dependency Graph	[46]
Annotation graph	[174]
Architectural Software Components Model Graph	[77]
AST and Dependency Graph	[108]
Bayesian Belief Networks	[121]
Call Graph	[172] [40] [141] [34] [104] [50] [45] [76] [116] [79] [106] [140] [131]
Call and Dependency Graph	[114]
Call, Dependency and Propagation Graphs	[165]
Change Guide graph	[93]
Conceptual Dependency Graph	[171]
Conditional Probability graph	[157]
Conditional system dependence Graph	[12] [13] [14]
Conceptual Graph	[33]
Control Flow Graph	[21] [95] [22] [29] [38] [127] [155] [173]
Dependency Graph	[97] [102] [86] [35] [150] [138] [8] [156] [136] [71] [96] [92] [167] [47] [169] [152] [166] [56] [65]
Dependency and Change Impact Graph	[68]
Dependency and Execution Path Graph	[20]
Domain-based Dependency Graph	[18]
Domain-based Coupling Graph	[19]
Reachability Graph	[85]
Impact Graph	[73] [151]
Intra-agent Dependency Graph	[52]
McCabe's complexity measure	[168] [28]
Random walk on a graph - a special case of a Markov chain	[158]
Relationship Graph	[44]
Simulation in Oriented Dependency Graph	[107]
Unified Dependency Graph	[143]

Table 5.12: Types of graphs used to perform change impact analysis when the applied technique is graph analysis.

Analyzed Elements	Papers
Artifact Change	[73]
Atomic Change	[38], [172]
Atomic Change in Aspect J	[38], [172]
Business Task Change	[40], [165], [41]
Change Request	[69], [60], [134], [3], [43], [142]
Change Set	[102], [103]
Class Change	[2], [138], [163], [86], [20], [4], [130], [48], [147], [16], [6], [118], [149], [34], [104], [71], [35], [75], [5], [101], [111], [42]
Class and Method Change	[62], [22]
Class, Method and Fields Change	[141]
Class, Method, Statement or Expression Change	[159]
Class, Method and Field Change	[150], [151] [159]
Class, Requirement and Code Change	[158]
Class, Requirement and Design Change	[24]
Classes Coupling	[135] [51]
Code Change	[169], [110], [74], [72]
Commit	[164]
Component Change	[173], [9], [77], [19], [21], [8], [154]
Data Modification	[97]
Data, Class or Method Modification	[95]
Diagram Change	[44]
Documentation Changes	[156]
Entity Change	[122], [123], [65], [136], [114], [143], [107], [50], [45], [100], [85], [117], [76], [127]
Feature Change	[132], [10]
File Change	[146], [23], [161], [160], [157], [83], [112], [129], [170], [144], [17], [39], [139]
File and Method Change	[93]
Function Change	[116], [68], [33]
Function, Arguments, Type and Global variables modification	[46]
Line Change	[174]

Table 5.13: Analyzed elements used to perform change impact analysis (Part I).

Analyzed Elements	Papers
Method Change	[155], [79], [106], [125], [124], [1], [140]
Method, Class, Data, Design Documentation, Network Chanel Change	[96]
Method, Class and Field Change	[131]
Method Change and Variable Change	[115]
Model Change	[54]
Module Change	[92], [14], [168], [167]
Package Change	[121]
Procedure Modification	[82]
Requirement Change	[145], [47], [108], [80], [66], [61], [171], [7]
Software Components	[56]
Software Life-cycle objects	[31]
Source Code and Test Code Change	[119]
System Change Request	[113]
System Specification Change	[18]
Textual Change Request, Method and Class Changes	[84]
UML elements	[152]
Variable Change	[70],[28],[27]
Variable and Function/Method Change	[29], [87]
Variable and Statement Change	[166], [12]
Variable, Method and Class changes	[91]

Table 5.14: Analyzed elements used to perform change impact analysis (Part II).

5.4.2.6 Supported Languages

We categorized the works according to the programming language or paradigm they considered in their respective method or tool proposal. Table 5.15 reports the results.

The Object-oriented paradigm and the programming languages that apply this paradigm - such as Java and C++ - are the most used among the works - 80 (56.7%). Java is considered in 25 works (17.7%). We identified that 23 works (16%) present models that do not depend on the programming language used in the studied software systems. Also, we found works presenting models and tools for change impact analysis for other paradigms and programming languages: three for the Agent-oriented paradigm; two for the AspectJ; one for Java Script; one for MatLab, and one for Multi-language product lines.

Supported Language	Papers
Agent-oriented	[53], [52], [54]
AspectJ	[38], [172]
C	[68], [33], [46], [29], [28]
C/C++	[83]
C++	[142], [20], [24], [171], [91]
Java	[40], [41], [69], [60], [103], [147], [16], [149], [104], [71], [35], [62], [22], [159], [150],[151], [72], [136], [100], [23],[157], [155], [79], [1], [108]
Java Script	[74]
Language Independent	[156], [65], [7], [27], [134], [4], [130], [110], [164], [19], [8], [122], [132], [129], [93], [139], [106], [125], [131], [47], [119], [70], [87]
Multi-language - Product Lines	[13]
Object Oriented Paradigm	[95], [73], [165], [3], [43], [141], [86], [48], [6], [118], [34], [75], [5], [101], [111], [158], [2], [138], [163], [51], [42], [135], [77], [21], [154], [97], [44], [114], [50], [45], [85], [117], [76], [127], [10], [112], [144], [17], [116], [124], [96], [92], [14], [145], [80], [66], [56], [169], [84], [102], [166]
Simulink/MatLab	[113]

Table 5.15: Programming languages supported by the CIA proposals.

5.4.3 Methods and Metrics Evaluation

In this section, we answer *RQ3 - Which methods and metrics did the studies use in evaluating these approaches and tools?* - analyzing which methods were used by the researchers when evaluating the methods and tools proposed by them. Tables 5.16 and 5.17 present the results.

Evaluation Methods. The most commonly applied evaluation method is Empirical Evaluation and Empirical Studies (31%). Among these studies, we identified that the sample of software systems they considered ranges from 1 to 13. Case Study is the second most applied method, present in 29% of the works. Comparative Analysis and Usage Examples were the third evaluation method, used in 9.9% of the analyzed studies. The remaining methods were applied in 6.4% of the studies.

Evaluation Metrics. When analyzing the data, we seek to identify the main metrics used by researchers when evaluating their works. We identified metrics used in the proposal evaluation in 68.3% of the analyzed papers. In 31.7% of them, we did not locate evaluation metrics. Table 5.18 presents the findings. Recall and Precision

are the primary set of metrics used by researchers when evaluating their results - 16.2% of the analyzed papers use these metrics. The second most used metrics set is Recall, Precision, and F-measure, used in 8.5% of the works.

Evaluation Method	Papers
Algorithm complexity	[167]
Case Study - 1 System	[163], [40], [145], [24], [19], [93], [87], [165], [41], [142], [20], [48], [2], [51], [132],[68], [131], [80], [108], [152], [71], [135], [73], [42], [82],[27], [17], [46], [56], [91], [116], [44]
Case Study - 2 Systems	[52], [35], [138], [100], [18], [106]
Case Study - 3 Systems	[75]
Case Study - 4 Systems	[119], [103]
Comparative Analysis with 2 approaches	[70] [155]
Comparative Analysis with 4 approaches	[70]
Comparative Study - 4 systems	[129], [33], [1]
Comparative Study with another approach - 1 system	[158], [14]
Comparative study with CodeSurfer - 2 systems	[169]
Comparative Study with HMS - 3 systems	[151]
Comparative Study with two other techniques - 5 systems	[114]
Comparative Study with two other techniques - 3 pilot projects	[84]
Comparison with a previous approach - 1 system	[96], [150]
Controlled Experiment - 1 embedded system	[66]
Demonstration Example with a self-made system.	[86]
Empirical Evaluation - 9 systems	[29]
Empirical Evaluation - 3 Systems	[170]
Empirical Evaluation	[140]
Empirical Evaluation of 4 systems. Comparison with previous heuristics.	[117]
Empirical Study	[10], [53]
Empirical Study - 1 System	[107], [22], [85], [121], [74]
Empirical Study - 2 systems	[62], [21], [161], [92], [23]
Empirical Study - 3 systems	[113], [147], [149], [110]

Table 5.16: Evaluation methods applied in the studies (Part I).

Evaluation Method	Papers
Empirical Study - 4 systems	[115], [130], [104], [159], [125],[101], [69]
Empirical Study - 4 systems and Comparative Study - 2 approaches	[164]
Empirical Study - 5 systems	[4], [16], [76], [83], [79]
Empirical Study - 6 systems	[139]
Empirical Study - 7 systems	[124]
Empirical Study - 8 systems	[172], [122]
Empirical Study - 9 systems	[12], [5]
Empirical Study - 10 systems	[111], [123], [160], [112]
Empirical Study - 11 systems	[28]
Empirical Study - 12 systems	[72]
Empirical Study - 13 systems	[6]
Example	[38], [3], [77], [45], [54], [61], [166], [168], [7], [102]
Proof of Concept for the tool	[97]
Qualitative Study	[146]
Quantitative Study - 2 Systems	[9], [143]
Tool Evaluation - in situ	[157]
Usage Example	[60], [34], [144],[136]

Table 5.17: Evaluation methods applied in the studies (Part II).

5.5 Discussion

The analysis carried out in this SMR shows that change impact analysis is an area with many demands, challenges, and opportunities, as discussed in the sequel.

Many models, fewer tools. Automation is indispensable for a proposal to be applied in practice. Thus, one of the main challenges identified is the creation of appropriate tools for change impact analysis. Among the evaluated tools, five were developed as plug-ins for the Eclipse IDE and one as a JBuilder5 extension. The development of tools linked to a specific IDE is a limitation since developers may prefer other types of IDE and choose not to use those only available in a particular environment. In other cases, the proposed tools are not available for use, only described in the paper. In addition, the evaluations of these tools involved few systems, which impacts the possibility of practical use.

The primary sources for change impact analysis are source code and change history. Static source code analysis is the most used approach in the literature. However, the main challenge in this approach is identifying dependencies not appar-

ent in the source code. In this respect, analyzing the history of changes can help to identify these dependencies. Although, a key limitation of change history analysis is that it does not apply to software systems with few change records, for example, the systems in their early life.

Graphs are a fundamental structure in change impact analysis. Graphs are the primary technique for change impact analysis, especially dependency and call graphs. However, software systems can have many elements and relationships, which makes manipulating the graphs representing the systems challenging in terms of processing time.

Software metrics have been timidly applied. Despite the large number of software metrics proposed in the literature, the technique of using metrics to aid change impact analysis has been little explored in the literature. One of the reasons for this may be the lack of adequate software metrics collection and analysis tools. In particular, it is worthwhile to note that, despite graphs having been used to aid change impact analysis, network metrics have not been explored. Exploring such metrics may improve the graph-based change impact analysis techniques.

Data mining and machine learning are emerging techniques. Data mining and machine learning have been applied in some studies on change impact analysis and have shown promise. However, it is not easy to apply these techniques because they require a significant volume for good results, which does not apply to software in the early stages of evolution. In addition, the processing time involved in approaches that apply data mining and machine learning is also a problem to be overcome to make the application of the technique viable in software development practice.

Other programming languages need to be explored. Object-oriented is the most explored paradigm, and Java is the most considered language in studies on change impact analysis. Creating approaches aimed at other programming languages, especially those widely applied in practice, such as Python and JavaScript, is essential. Multilingual approaches are also of interest, given the existence of software that applies more than one programming language.

Proper evaluation is challenging. Most of the works we analyzed use a few systems to evaluate the effectiveness of the proposed method. In addition, the evaluated systems are medium and small-sized, which means that the results obtained may be different in large-sized systems, which are the systems that exist in the practice of software development.

Evaluation Metrics	Papers
Accuracy	[4], [122], [143], [10]
Accuracy and Impact set size	[144]
Accuracy Predictors	[170]
Accuracy and Scalability	[169]
ANOVA	[41]
ANOVA and Ducal	[75]
Association Rules Confidence	[17]
Average affected code lines, and classes	[91]
Changeability Measures	[48]
Completeness , Correctness and Kappa value	[84]
Correlation Consistency	[16]
Correlation Metrics and Linear Regression	[130]
Magnitude of relative error (MRE)(MMRE) and Prediction Percentage (PRED)	[24]
Paired t-test	[110]
Pearson and JASP Correlation	[6]
Pearson and Spearman Correlation	[111]
Pearson Correlation	[124]
Person Coefficient	[29], [28]
Precision	[35], [62], [21], [93], [33]
Precision and Performance	[155]
Recall	[74]
Recall and Precision	[53], [52], [69], [103], [20], [149], [104], [113], [22], [159], [151], [123], [114], [100], [85], [1], [76], [160], [39], [139], [46], [79], [106],
Recall, Precision and Wilcoxon rank	[117]
Recall, Precision, and F-measure	[83], [5], [150], [158], [163], [135], [164], [23], [112], [116], [115], [92]
Recall, Precision, Accuracy, and Poit-Biserial Correlation	[121]
Recall, Precision, F-measure, Area Under the Curve and Mathews Correlation Coefficient	[161]
Recall, Precision, F-measure and Area Under the Curve	[119]
Recall, Precision, F-measure and Feedback Metric	[19]
Score, Focus, and Spread of Clusters	[147]
Significance (correlation coefficient R2)	[51]
Spearman Coefficient	[101]
Support and target rules, and true positives.	[125]
True, and False Positive	[18]

Table 5.18: Evaluation metrics applied in the studies.

5.6 Final Remarks

Since modifying the software systems and their artifacts is essential for its evolution, the creation of methods and tools to support the execution of this task is necessary. Over the past few years, some works related to change impact analysis have been developed. In this chapter, we presented a systematic mapping review of the literature on change Impact analysis. A total of 141 studies published between 1978 and 2021 were analyzed.

We extended the framework proposed by Li et al. [105] and extracted data from these publications as presented in the sequel. We categorized the studies according to:

(i) the type of the proposed method;
(ii) the applied scientific method;
(iii) the level of abstraction used;
(iv) the data source;
(v) the kind of source code analysis;
(vi) the technique developed in the modification propagation evaluation method;
(vii) the type of graph analysis, when the graph was the technique used;
(viii) the type of the elements analyzed in the proposed method;
(ix) the method and the evaluation metrics;
(x) the programming language supported by the method.

The results of this systematic mapping study reveal that:

- The studies proposed more methods than tools for change impact analysis, so the development of tools to support them is necessary.
- The scientific method most used by researchers is the empirical method.
- The methods and tools proposed for CIA are generally based on the analysis of artifacts that are at the same level of abstraction, i.e., the dependency abstraction level.
- Source code is the most common artifact used in methods and tools for CIA, and the static analysis of the code is the most applied.
- The most commonly applied technique for CIA is graph analysis.
- The most used types of graphs are the dependency graph and the call graph.
- Most of the methods and tools for CIA support object-oriented software systems.
- Empirical evaluation methods are the most used by researchers
- Precision and Recall are the metrics most used to evaluate the proposal.

The main conclusion we achieved from the results of this SMR and of the state of the practice, presented in Chapter 4, is that there is a relevant demand for more practical and effective approaches for change impact analysis. From the results described in the literature, we considered that change history analysis based on commits' data and static analysis are complementary and promising approaches. Therefore, in this Ph.D. dissertation, we propose a hybrid approach for change impact analysis based on change history and static analysis. However, the commits' data analysis approaches have essential fragilities mainly because they do not explore commit characteristics. For this reason, we conducted an empirical study on commits characterization, presented in Chapter 6. Based on the results of this characterization, we proposed a new heuristic to analyze co-changes in software systems, presented in 7. This heuristic will be applied to the definition of a new approach to change impact analysis, as described in Chapter 8.

Chapter 6

Commits Characterization

In this chapter, we present a study that aims to characterize commits according to the following aspects: categories of activities performed in the commits, co-occurrences of activities in commits, the size of commits in the total number of files, the size of commits in the number of source-code files; the size of commits by category; and the time interval of commits performed by contributors.

We organized this chapter as follows. Section 6.1 presents the study design and the creation of the dataset used in this work. Section 6.2 presents the study results, and Section 6.3 discusses them. Section 6.4 reports the threats to validity. Section 6.5 brings the final remarks.

6.1 Study Design

This section presents the method we applied to construct the dataset analyzed in this study, the data extraction, and the commits categorization process.

6.1.1 Dataset

As this work aims to characterize commits, we selected well-known open-source software systems with many commits. To identify the corpus of systems to consider in this work, first, we identified the 900 highest-rated Java repositories. We found many repositories that do not contain source code among these projects. Those repositories were mainly used as libraries - they have books, “how to”, and similar files. Thus, we removed those repositories and obtained 846 repositories of Java software. From these 846 repositories, we selected 24 open-source systems with the

highest number of commits to be the subject of this study. We restricted the number of systems to 24 due to the long time it takes to collect the data we analyzed in this study.

We developed a Python script using GraphQL API to mine GitHub to retrieve the projects. The data returned by this API contain the repositories' name, owner, age in years, URL, commits, forks, issues, and the number of stars. Java language was used as the primary selection criteria to define the projects considered in the analysis. We chose Java because studies on software engineering commonly consider this language.

Table 6.1 shows the name, age, number of commits, and number of stars of the systems analyzed in this study. The dataset comprises mature and well-known systems aged between 3 and 11 years and rated between 52K and 2,6K stars. All the systems have high commits, varying from 22.9K to 92K. Besides, the dataset is diverse in terms of application domains.

6.1.2 Data Extraction

The first step of the data extraction was to create a copy of all the 24 systems' repositories using the `git clone` command.[1] We developed a Python script using *GitPython* API to perform this cloning process. The script collects all the commits' information for each repository: author, date, description message, and the modified files, and exports all the data to a .csv file.[2]

In our analysis, we considered data from the first-parent line. We support our decision with the findings of Kovalenko et al.'s study [94]. The results of their study show that considering complete file histories, i.e., including branches, may modestly increase the performance of reviewer recommendation, change recommendation, and defect prediction techniques. On the other hand, collecting the entire file history demands extra effort, e.g., the time to collect the data may be unreasonable. Therefore, the increase in performance may not justify such an effort.

[1] We cloned all repositories in January 2021.

[2] The data was exported as a .csv file and is available at *https://figshare.com/s/fab86b2522ded083f81c*

System	Age	#Commits	#Stars
ballerina-lang/ballerina-platform	3	96,121	2,644
neo4j/neo4j	8	69,702	8,315
jdk/openjdk	2	62,947	6,553
elasticsearch/elastic	10	57,414	52,228
camel/apache	11	50,138	3,489
graal/oracle	4	53,665	13,950
languagetool/languagetool-org	7	46,224	4,114
vespa/vespa-engine	4	46,403	3,363
lucene-solr/apache	4	34,703	3,863
rstudio/rstudio	9	34,292	3,423
alluxio/Alluxio	7	31,587	4,805
hazelcast/hazelcast	8	30,936	4,033
jenkins/jenkinsci	9	31,136	16,463
sonarqube/SonarSource	9	30,480	5,272
beam/apache	4	30,519	4,362
spring-boot/spring-projects	8	30,671	51,678
bazel/bazelbuild	6	28,662	15,673
shardingsphere/apache	4	28,457	12,387
ignite/apache	5	27,401	3,518
selenium/SeleniumHQ	7	26,432	19,074
cassandra/apache	11	25,994	6,278
flink/apache	6	25,543	14,626
hadoop/apache	6	24,584	11,041
tomcat/apache	9	22,909	4,984

Table 6.1: Dataset systems sorted by number of commits.

6.1.3 Commits Categories

This work analyzes the main activities registered in the system's commits to answer the request question RQ3. For this purpose, we classified each commit into six categories:

- **Merge**: specific GitHub activities of merge and pull requests.
- **Corrective Engineering**: changes performed in the code to correct bugs, errors, or defects.
- **Forward Engineering**: inclusion of new features or requirements.
- **Reengineering**: changes performed in the code to enhance its quality.
- **Management**: activities not related to codification, such as documentation.
- **Other**: when the commit does not match any of the five categories.

We used the same categories proposed by Hattori and Lanza [78] and included a new one: Merge. In Hattori and Lanza's work, "merge" was a keyword of the Management category. We considered Merge a particular category because a merge is a specific activity that differs from the other management activities in GitHub. Unlike Hattori and Lanza's approach, we do not use a hierarchy to set only one category for a commit, i.e., in our approach, a commit may be classified in more than one category. We did that to cover the cases in which a developer proceeds a commit corresponding to more than one activity type, e.g., Corrective Engineering and Reengineering. This type of commit is called *tangled commit* [58].

Similar to other works [78, 37], we based our approach to categorizing a commit on the analysis of keywords extracted from the commit's messages. We chose to analyze the messages because it presents a complete description of the commits' activities. We developed a Python script to identify the commits' activity categories using the *flashtext* API. Given the vast number of commits ($\approx 1M$), we used this API because its performance is better than the search using regex. The *flashtext* API counts an instance of a word only if there is an exact match in the text with the word. Therefore, it was necessary to build a dictionary containing the keywords corresponding to the commits' categories we considered and their variations, e.g., add, addition, adding, added, and adds. We started the construction of the dictionary having as basis the keywords used by Hattori and Lanza [78]. Then, we ran the classification and manually inspected the results considering a set of randomly selected $\approx$ 500 commit messages. We included new keywords extracted from the commits' messages in the dictionary based on the manual inspection. We executed such a process iteratively until we found a correct classification of the set of commits selected for manual inspection. Table 6.2 exhibits the final primary keywords set. It is worth noting that the complete dictionary contains variations of these words.

To assess the approach used to classify the commits' activities, we calculated the precision and the recall considering a random sample containing 500 commits. In this evaluation, we manually analyzed each commit and tagged each categorization result as:

- True positive (TP): when the script indicates that a commit belongs to a category and this categorization is correct;
- False Positive (FP): when the script shows that a commit belongs to a category and this categorization is wrong; and
- True Negative (TN): when the script indicates that a commit does not belong to a category, this result is right.

Precision is given by TP/(TP+FP) and indicates how many positive classifications

1lightgray

Category	Keywords
Merge	merge, pull request
Corrective	bug, fix, correct, miss, proper, broken, corrupted, failure, fault, deprecate, throw/catch exception, crash, typo
Forward	implement, add, request, new, test, increase, expansion, include, initial, create, introduce, launch, define, determine, support, extend, set
Reengineering	parallelize, optimization, adjust, update, delete, remove, expunge, cut off, refactor, replace, modification, improve, is/are now, change, rename, eliminate, duplicate, obsolete, enhance, restructure, alter, rearrange, withdraw, conversion, revision, simplify, move, relocate, downgrade, exclude, reuse, revert, extract, reset, redefine, edit, readd, revamp, decouple
Management	clear, license, release, structure, integration, copyright, documentation, manual, Javadoc, migrate, review, polish, upgrade, style, standardization, TODO, migration, organization, normalize, configure, ensure, resolve conflict, bump, dump, comment, format code, do not use

Table 6.2: Primary keywords used to identify the activity category of commits.

are correct. As shown in Table 6.3, all categories showed precision above 52%. Merge is the category with the highest precision (96%). A Recall is given by TP/(TP+FN) and indicates how many situations the script should detect as true positives were correctly detected. The results show that the categorization's recall reaches 99%.

	Precision	Recall
Merge	0,96	0,99
Bug	0,78	0,98
Reengeneering	0,84	0,79
Foward	0,59	0,93
Management	0,52	0,77
Others	0,74	0,70

Table 6.3: Categorization Results: Precision and Recall

6.1.4 Research Questions

In this study, we seek to answer the following research questions:

RQ1. How often are the activity types performed in commits? With this research question, we aim to assess the number of commits involving each activity category identified in this study.

RQ2. How often do co-occurrences between the activity types appear in commits? We analyze the frequency of co-occurrence between the different types of activities a contributor may perform in a repository via a commit.

RQ3. What is the size of commits in software system repositories? This question investigates the developers' behavior regarding the number of files they use to commit together. Such a result may guide establishing the granularity of commits when carrying out research using commit data. For instance, in research investigating co-changes, the files changed together in a commit may be considered a co-change instance. However, the number of commits involving many files may bias the research results.

RQ4. What is the size of commits involving only *.java* files in software system repositories? In RQ3, we consider all types of files in the commits. In RQ4, we aim to analyze only source-code files. As we focus on Java-based software systems, we considered Java source-code files to answer this research question.

RQ5. What is the size of commits according to their aims? We analyze the number of files that usually are involved in different activities, such as reengineering, managing, Corrective maintenance, and Forward maintenance. Answering this RQ will bring insights into the proportion of each type of activity performed along the software systems' life cycle.

RQ6. What is the time interval a developer registers a commit in a repository? We analyze the interval of time the contributors usually perform commits in a repository. The results of this analysis may aid studies that define heuristics to co-change and change impact analysis.

6.2 Results

This section presents the results of this study by answering the research questions.

RQ1. How often are the activity types performed in commits?

To answer this research question, we categorized the commits as described in Section 6.1.3. Figure 6.1 shows the percentage of commits corresponding to each category: Merge, Corrective Engineering, Forward Engineering, Reengineering, Management, and others, in the 24 software systems analyzed in this study, giving the percentage values by *(number of commits of a category) / (total number of commits registered in the project)*.

Figure 6.1: Percentage of commits by category.

Reengineering is the most frequent activity, registered in 32.97% of the $\approx 1M$ commits analyzed in this work. This category has the highest percentage of commits in 13 out of the 24 systems. Besides, 83% (20/24) of the systems have at least 1/4 of the commits to Reengineering activities. *Cassandra* is the system with the lowest percentage of Reengineering activities (15%), and *elasticsearch* has the highest one (51%).

Forward Engineering is the second most frequent activity. The percentage of commits labeled as Forward Engineering ranges from 15.6% (*ignite* and *jdk*) to 47.7% (*bazel*). Forward Engineering comprises 28.2% of the commits. It is the most frequent category in three systems: *vespa*, *alluxio*, and *shardingsphere*.

Corrective Engineering is the third main activity. This category corresponds to 25% of the commits. *Shardingsphere* is the system with the lowest rate of commits of Corrective Engineering - only 9%. *Tomcat* presents the highest rate of commits tagged as Corrective Engineering and has this category as its main activity (39.6%).

Management corresponds to 18.7% of the commits. The results show that 75% (18/24) of the systems have less than 1/4 of their commits registering Management activities. The lowest number of Management activities is present in the *language tool* (6.6%), and *spring-boot* presented the highest one (43.6%).

Merge corresponds to 16,49% of the commits. This category presents a considerable disparity among the systems. The number of commits in the Merge category ranges from 0.6% (*selenium*) to 43.4% (*cassandra*). Only five systems present a percentage higher than 25%: *hazelcast* (27.3%), *spring-boot* (30%), *vespa* (33.47%), *ballerina-lang* (34.2%), and *cassandra*.

Others tag messages whose content could not be categorized with any other five categories. This category corresponds to 16% of the commits, and the percentages of commits in this category range from 4.7% (*ballerina-lang*) to 32.9%(*rstudio*). Besides *rstudio*, only two systems have more than 25% of commits tagged as Others - *graal* (25.7%) and *hadoop* (28%).

RQ2. How often do co-occurrences between the activity types appear in commits?

A commit may involve more than one activity type. As described in Section 6.1.3, our approach allows classifying a commit with more than one category. We found that 30% of all commits analyzed in this work involve more than one activity type. We calculated the percentages of all possible co-occurrence between the two categories. Table 6.4 shows the results.

	Merge	**Corrective**	**Forward**	**Reengineering**
Management	1.7%	4.6%	5.3%	**6%**
Reengineering	2.6%	**8%**	**8%**	
Forward	2.3%	5%		
Corrective	2.8%			

Table 6.4: Percentage of co-occurrences of commits categories.

The Merge category presented the lowest rate of co-occurrences, 1.6% to 2.8%. The highest rates of co-occurrence among the activity types are Reengineering with Corrective Engineering (8%), Forward Engineering (8%), and Management (6%).

Summary. A significant part of all commits involves more than one activity type, 30%. The highest percentage of co-occurrence of activity types are Reengineering with Corrective Engineering (8%) and Reengineering with Forward Engineering (8%).

RQ3. What is the size of commits in software system repositories?

The first step in answering this research question was to analyze the data distribution. Therefore, we calculated the number of files changed by each commit. Figure 6.2 shows the results, where a boxplot represents the distributions of the number of files per commit for each system. We marked the distributions' median as red dots in the boxplots.

The result presents some standard behaviors. All systems, observing the boxplots' shapes, show a long tail distribution because a high concentration of commits involves few files. The boxes are placed on the chart bottom. The extensive lines ranging from the third quartile to the outliers indicate that commits registering a higher number of changed files are atypical events, i.e., there are few commits with this behavior. The median ranges from 1 to 3, with two being the median in 58% of the systems. The boxes' height (i.e., the difference between the first and the third quartiles) is not high and is very similar. They range from 1 to 3 in 41.67% of the systems, from 4 to 6 in 41.67%, and from 7 to 9 in 16.67%. Table 6.5 presents the first, median, and third quartile values.

We may take *jdk* as an example of the enormous data disparity in the number of files per commit. In *jdk*, the 80th percentile is 14, i.e., 80% of the commits register the modification of a maximum of 14 files. In contrast, Table 6.5 shows a commit in this system that modified 56K files.

We detailed the analysis of the distributions' tails to verify whether there is a pattern of developers committing a high number of files in a single transaction. We consider outliers' values greater than the upper outer fence, i.e., values higher than $Q3 + 3 * IQR$, where Q3 is the third quartile, and IQR is given by *3rd quartile - 1st quartile*. Figure 6.3 shows the distribution's outliers. The outliers' distribution is also heavy-tailed. We can see that by observing the violin plots' shape: the most significant part of the plot is placed at the chart's bottom, and as the y-axes increase, the plots' shape becomes thinner. There is a big difference between the median values of the

Systems	Q1	Median	Q3	80th %	Max Files
alluxio	1	1	4	5	2448
ballerina-lang	1	3	10	13	21405
bazel	1	2	5	6	2733
beam	1	2	4	6	7206
camel	1	2	4	5	17925
cassandra	1	2	4	5	645
elasticsearch	1	2	6	8	14916
flink	1	3	8	10	11013
graal	1	2	5	6	11103
hadoop	2	3	6	8	5194
hazelcast	1	2	6	8	8674
ignite	1	3	10	15	9971
jdk	1	2	9	14	56923
jenkins	1	1	3	5	8949
languagetool	1	1	2	2	1266
lucene-solr	1	2	5	6	5570
neo4j	1	2	7	9	10716
rstudio	1	2	4	5	4624
selenium	1	2	4	5	3619
shardingsphere	1	2	6	8	5259
sonarqube	1	3	7	9	9263
spring-boot	1	1	3	4	4616
tomcat	1	1	3	3	1157
vespa	1	2	6	8	18589

Table 6.5: Percentiles of number of files per commit, where, Q1 = 1st quartile, Q3 = 3rd quartile.

outliers. Unlike the distribution shown in Figure 6.2, the medians vary between 56 and 198, and the values contained in the interquartile are more dispersed.

Summary. In general, the total files per commit range between 1 and 10. Nevertheless, some commits modify a very high number of files. Among the outliers, the medians vary between 56 and 198.

RQ4. What is the size of commits involving only *.java* files in software system repositories?

To answer this research question, we conducted the same analysis of RQ1; however, observing only Java source-code files, i.e., files with the extension .java. Figure 6.4 exhibits the results. We observe that the number of .java files committed in a single transaction also has a long tail distribution. The medians range between 0 and 2. In 70.8% of the systems, the median is 1, 16.7% is 2, and 12.5% is 0. In 66.5% of the analyzed systems, the first quartile is 0. The third quartile has the main values of modified java files per commit: 4 files, 29.2% of the systems, and three files, 20.8%

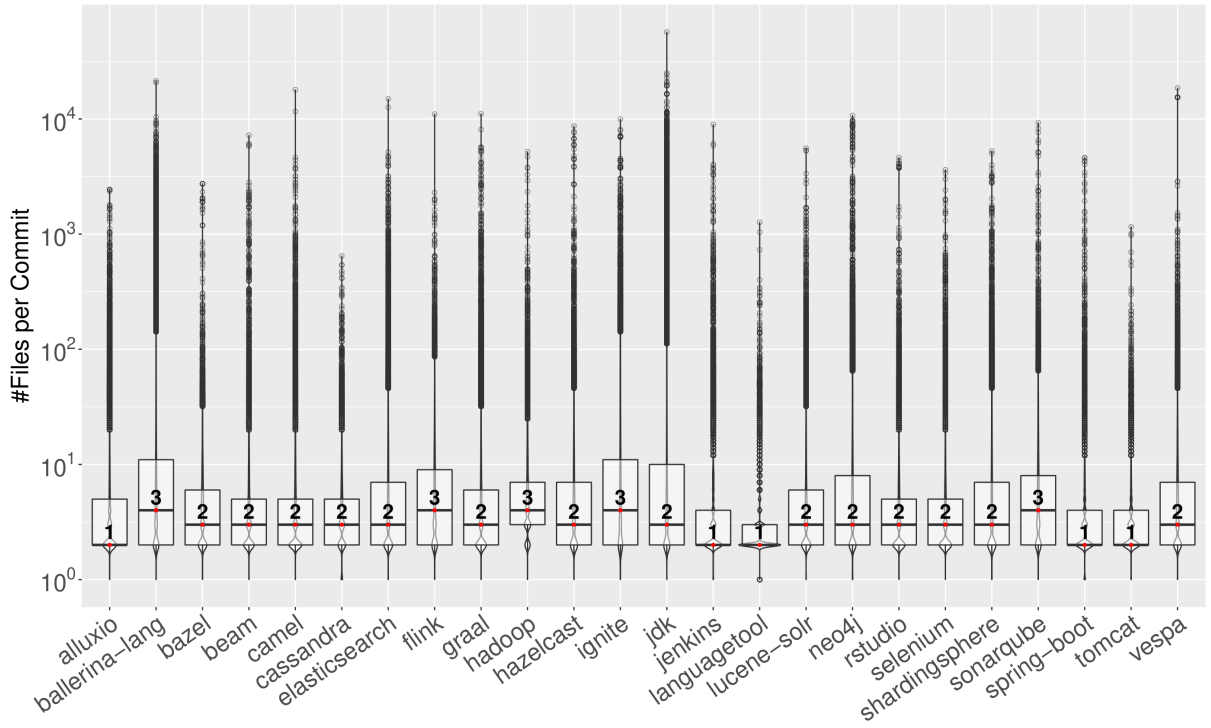

Figure 6.2: Distribution of files modified in commits.

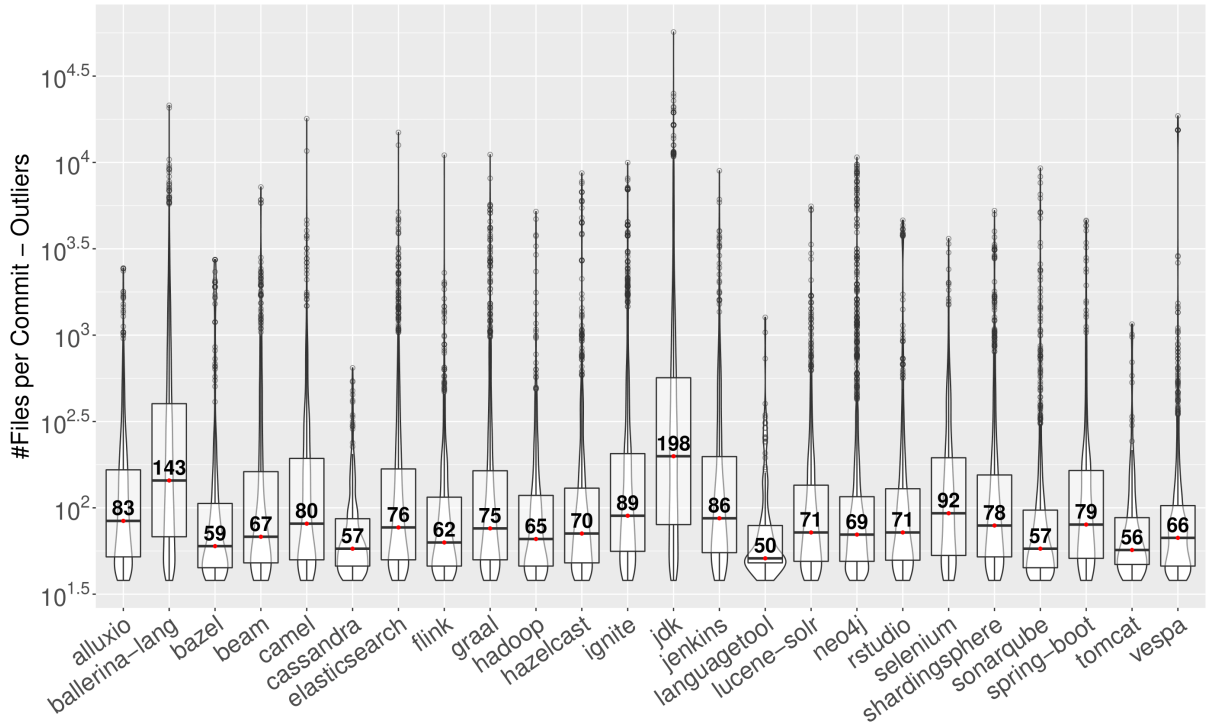

Figure 6.3: Distribution of files modified in commits - upper outer fence outliers.

of systems.

We observed the same behavior found in the analysis of RQ1. The results show that

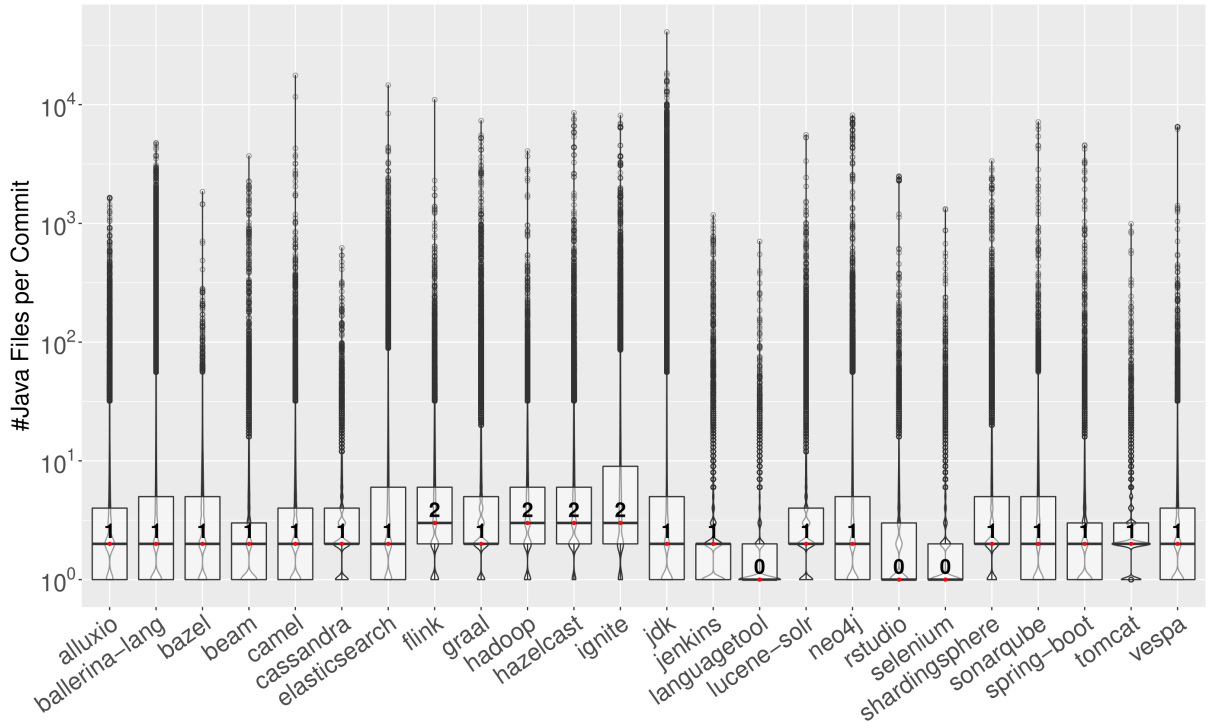

Figure 6.4: Distribution of Java files modified in commits.

jdk also presents the most considerable disparity between the number of .java files registered in a commit. The system's 80th percentile is 182, and the third quartile is 6.

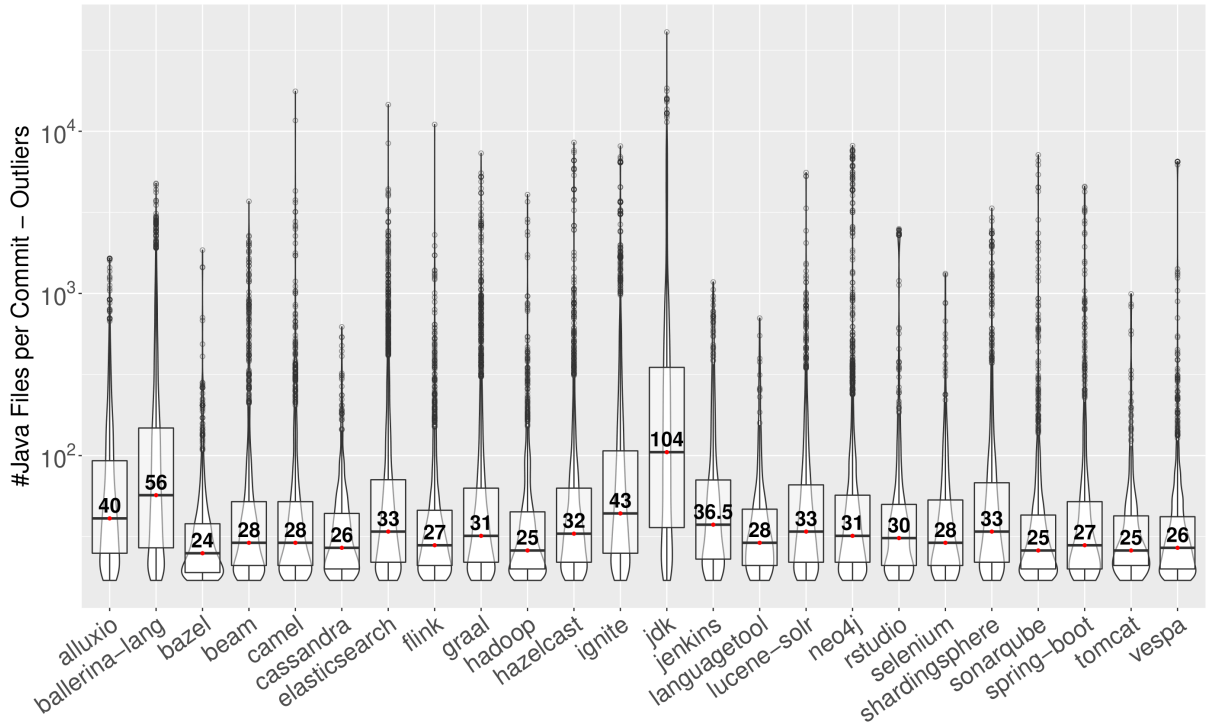

Figure 6.5: Distribution of Java files modified in commits - Upper outer fence outliers.

Figure 6.5 shows the analysis of the distributions' tail of the number of .java files modified in a commit. Among the outliers, the median ranges from 25 to 104. In the same way as the previous distributions' plots, *jdk* system shows a particular behavior, with the highest median value, 104, and higher dispersion. Such characteristic is essential to be considered when performing studies about this system.

Summary. The number of .java files modified per commit follows a heavy-tailed distribution. The systems generally have between 1 and 4 .java files modified per commit. Among the outliers, the median ranges from 25 to 104.

RQ5. How do practitioners perform change impact analysis?

To answer this research question, we calculated the number of files modified by each commit category: Merge, Corrective Engineering, Forward Engineering, and Management.

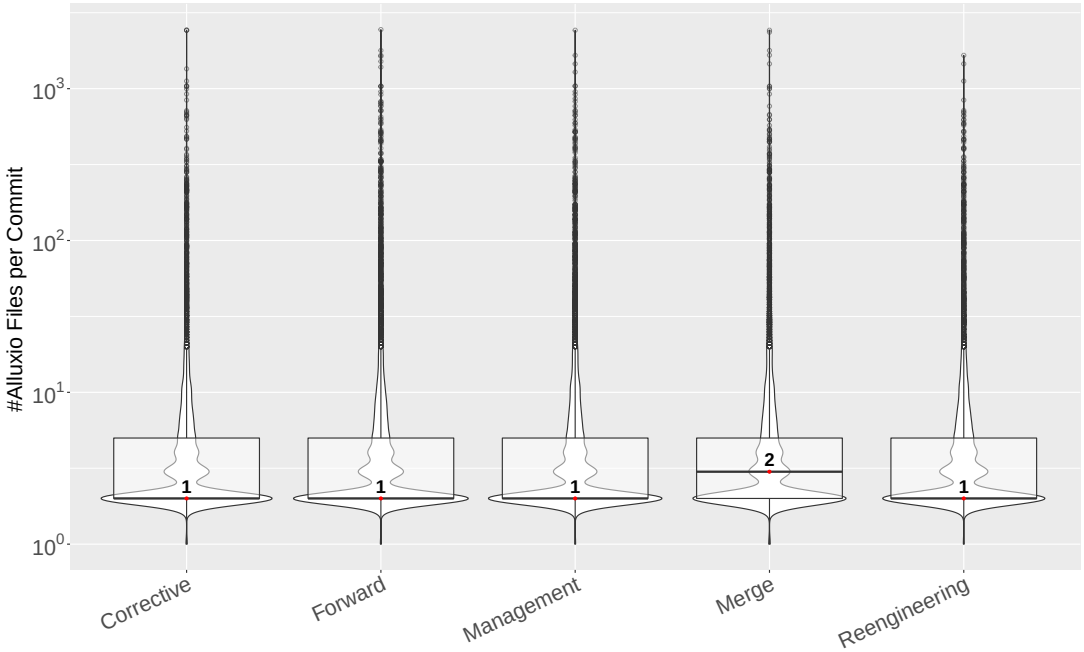

Figure 6.6: Alluxio distribution of files modified in commits grouped by category.

Figure 6.6 shows the results of *alluxio*. The median values are low in the data distribution, ranging from 1 to 3 files. The other systems presented a similar result, except *jdk*. Due to space limitations, we do not show all graphics with the results of this research question in this paper. However, we make them available online.[3]

Figure 6.7 shows the results of *jdk*. Reengineering, Forward Engineering, Corrective Engineering, and Management have the same distribution pattern, and the median

[3] `https://figshare.com/s/ffd7b22c520abdc7129c`

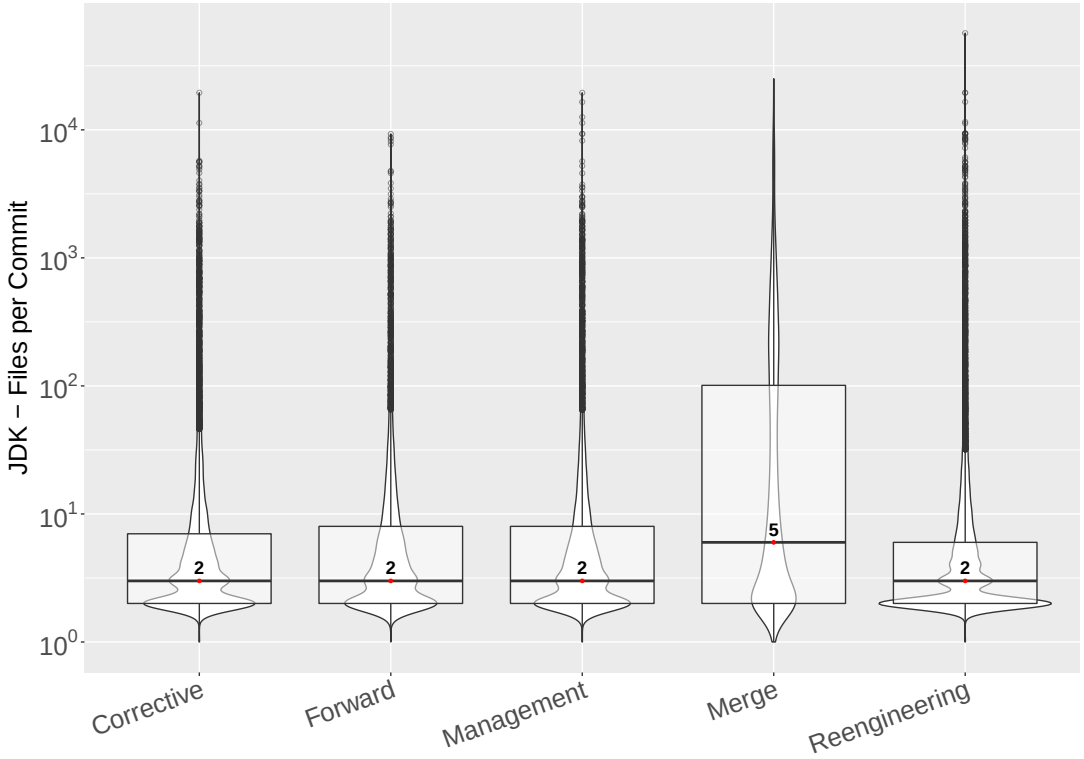

Figure 6.7: JDK distribution of files modified in commits grouped by category.

value is 2. The merge category presents a different result: it has the largest interquartile range:[4] 99 files. An important characteristic observed in *jdk* is that commits that did not change files were categorized exclusively as merge.

Summary. The number of files modified in a commit does not significantly differ regarding the activity type.

6.3 Discussion

Understanding the dataset's characteristics is critical for conducting a good experiment, and working with an inadequate dataset will lead any well-designed study to reach inaccurate results. This section discusses the main lessons learned from our research and their implications for studies that consider commits' data.

The commits *nature* should be considered by the studies. This study found

[4]Difference between the first and third quartiles. In *jdk*, there are, respectively, 1 and 100 files.

that most commits register Reengineering activities, followed by Forward Engineering and Corrective Engineering. A possible explanation for this characteristic is that as open-source software projects are developed collectively, it may demand refactoring the system more often. Besides, as the systems are publicly available, their users may continuously report defects and failures on them. This result indicates the need to properly select the commits in studies on refactoring and faults in software since Reengineering commits correspond to only 32.97% and Corrective Engineering to 25% of the commits in the systems. The percentage of Merge, Management, and Other activities should not be ignored: 18.7%, 16.49%, and 16%, respectively. If these activity types can impact the analysis in a study, they need to be identified when collecting the data. The systems analyzed in this study are popular and very active, which may be a reason for the high number of Forward Engineering. Therefore, future work on Forward Engineering may consider the sample analyzed in this work.

The Quantification of the Tangled Changes Problem. We found that 30% of the commits involve more than one activity type. This result indicates the extent of tangled changes in software repositories. Therefore, works threatened by tangled changes should perform characterization of commits in terms of activities because the amount of co-occurrence of activities is expressive. For example, this care is critical in studies on change impact analysis. Many studies consider a commit as a basic unit of correlated changes. In the face of the results found in this work, the analysis performed in those works may be biased.

Reengineering has the highest co-occurrence with other activity types, but this does not happen too often. The incremental software development methodologies, such as the Agile methodologies, favor Reengineering, Corrective Engineering, and Forward Engineering to occur in parallel. For example, it is possible that correcting a bug or introducing a new feature in the system may cause a reengineering. Then both types of activities may be committed together. However, the results of this study show that these co-occurrences do not happen very often. The highest frequency of co-occurrences is between Reengineering and Corrective Engineering and between Reengineering and Forward Engineering, 8% in both cases. Studies on refactoring based on commit analysis should verify if this amount of co-occurrence introduced bias in their results. Intuitively, we may consider that when a system is well constructed, making changes will be more comfortable; therefore, it will demand fewer refactoring activities. Consequently, we raise two hypotheses that may explain the low percentage of co-occurrence between reengineering with corrective and for-

ward engineering: (i) or fixing bugs and changing a piece of the system usually demands little refactoring in the system, (ii) or the practice of developers is to commit the refactor of the system before fixing a bug or changing the system.

The size of the commit matters. The results show that the size of commits follows a heavy-tailed distribution. Therefore, although most commits involve just a few files, a relevant number of them involve many files. A single commit may include hundreds of files. In contrast with what one may intuitively assume, large commits do not occur only in Merge or Management activities. This result is significant to studies that consider the set of files in a commit, which is the case of studies on change impact analysis and code authorship. In these works, disregarding that a relevant number of commits (more than 50%) involve a very high number of files may introduce bias in the analysis. In change impact analysis studies, the files in large commits may be more likely not to relate to a common cause of the change. In authorship analysis, a commit registered by a contributor involving many files may not express authorship.

JDK is an exception. Many empirical studies have considered JDK, and our results revealed that JDK is an exception regarding the number of files per commit. The commits of JDK involve a higher number of files per commit than the other systems. Therefore, the study design based on commit analysis should consider this characteristic if JDK is part of its analysis.

Commits' size is not Normal. All results in this study lead us to a simple but not-so-obvious conclusion: one of the essential characteristics regarding commits' size is that we cannot apply the Normal distribution statistical analysis methodologies to them. The number of files modified in a commit has a long tail distribution, and besides, there is no standard distribution for the number of commits considering the activity type.

The time intervals of commits by developers. Understanding the developers' behavior when registering commits in the repositories may aid practitioners, especially in management tasks. We investigated if there is a pattern of time intervals in which developers register commits in the repositories. The results indicate that, in a project, the distribution of the time intervals is approximately a Normal distribution, i.e., the distribution tends to be symmetric, and the mean is representative. The results also show that the time intervals vary among the projects. In this work, we do not investigate the causes of the behavior of developers when performing commits. Further works should investigate if the projects' nature, application domain, and the number of contributors may influence the frequency of commits by developers. Nevertheless, the results found indicated that, in general, a contributor registers a commit

every eight hours. We applied this result to define a new heuristic for co-change analysis.

6.4 Threats to Validity

To answer RQ3, we relied on the approach defined in our previous work to categorize commits [63]. That approach is based on the automatic search for keywords in the commits' messages. Therefore, as described in Section 6.1, it was necessary to build a dictionary containing the keywords and their variations. We constructed the dictionary manually, which may cause us to forget some keywords. To mitigate this threat, we built the dictionary based on the keywords described by a previous work [78] and added new words that we found in the manual inspection of ≈ 500 commits. We also evaluated the approach via manual inspection and found high precision and recall.

We considered the field "author" of the commit's data to identify the developers when calculating the time interval between commits. Depending on the type of study, such an approach may introduce substantial bias in the results, which is the case, for instance, of studies on code ownership because a developer may have more than one GitHub username. However, this is not the case in the present work because we are interested in analyzing sequential commits performed by a user whose name may be appropriately identified.

This research focused on Java-based software systems and considered data from 24 Java-based systems hosted on GitHub. GitHub has about 20 million public repositories. Therefore, it is not possible to ensure the generalization of the results found in this study. However, as this study concentrates on commit's data, we selected the most rated systems containing the highest number of commits, resulting in a dataset containing $\approx 1M$ commits of mature systems from well-known owners, such as Apache.

6.5 Final Remarks

The system's data hosted in GitHub have been profusely used in software engineering works. Commits data are one of the most used analysis sources in such works. However, not knowing or not considering the characteristics of commits may introduce biases in research. Besides, investigating the characteristics of commits may bring insight into the developers' practices and, hence, provide important information to practitioners to improve the management and the planning of the software activities.

We carried out an empirical study to characterize commit data in this research. We evaluated the 24 most popular and active Java-based projects hosted on GitHub. We analyzed $\approx 1M$ commits.

The main findings of the study described in this chapter revealed that:

(i) Reengineering is the most frequent activity, followed by Foward Engineering and Corrective Engineering.
(ii) Although low the frequency of Merge and Management activities are relevant.
(iii) 30% of commits involve more than one type of activity.
(iv) The most common co-occurrences are between Reengineering and Forward Engineering and between Reengineering and Corrective Engineering.
(v) The size of commits follows a heavy-tailed distribution; (vi) most commits involve one to 10 files and one to four source-code files.
(vi) Many commits involve hundreds of files and those commits not only refer to Merge or Management.
(vii) The distribution of the time intervals is approximately a Normal distribution, i.e., the distribution tends to be symmetric, and the mean is representative, and
(viii) On average, a developer proceeds a commit every eight hours.

The results of this study lead to some lessons that researchers should consider in empirical studies based on commit analysis. We applied the results of this study to define a new heuristic for co-change detection, presented in Chapter 7.

Chapter 7

A Heuristic for Co-change

This chapter presents a heuristic for detecting co-change in classes through mining data from software repositories - specifically commits. We incorporate into this heuristic the commits' characteristics presented in Chapter 6. We used data from 32 open-source Java systems to evaluate the heuristic and compare it to a heuristic that does not use commit information, only the modification data it presents.

In this chapter, Section 7.1 presents the proposed heuristic, Section 7.2 describes the method used to evaluate the heuristic, Section 7.3 shows the obtained results and Section 7.4 discusses them, Section 7.5 reports the threats to validity, and Section 7.6 presents the final remarks.

7.1 The Heuristic

We defined our heuristic for Java-based systems. Hence, given the clone of a Java system repository, the first step of the heuristic proposed by us is to extract the commit data from that repository. We collected authorship data, the file name (full path), the commit message, the hash, the linked issue, and the date (day and hour).

We based the proposed heuristic on three points:

1. Discard commits that involve a high number of files. The reason for this approach is that commits with a high number of files may be related to activities such as merge and, hence, introduce biases in the result of the co-change analysis. An essential point here is to define the threshold of the number of files to consider. In our study about the characteristics of commits, we found out that a commit changes between 1 and 10 files. Hence, commits that change more than ten files are outliers. Therefore, we apply this threshold in our heuristic and compute only commits whose number of files was less than or equal to 10.

2. Consider the issue number as a co-change indicator. In GitHub, an issue is a text-based description of tasks, bugs, changes, and updates that can be linked to a commit. So, commits linked to the same issue are highly likely to be related. Therefore, commits related to the same issue number represent co-changes. In our heuristic, we grouped commits with the same issue number.
3. Consider a block of commits of the same contributor. Previous studies consider atomic commits as units of co-changes. However, such an approach may bring biases in the analysis. Suppose a contributor should modify three classes in a given change task. Hence, he changed each class a time and performed three commits, one for each class. In this illustrative example, considering commits as units of co-change will bring wrong results because it will not reveal that the three classes were co-changed. Therefore, our heuristic identifies blocks of commits by the same contributor. An essential point here is to define the time interval a contributor will likely proceed with commits related to a change task. We found that, on average, a contributor performs a commit every eight hours. Thus, we grouped subsequent commits by the same author within eight hours. The rationale of this assumption is that sequential commits registered in intervals lower than the average may be considered a block of commits.

Figure 7.2 exhibits the steps of the data processing of the proposed heuristic. We developed a Python script that implements these steps and outputs a .csv file containing all the pairs of files that are part of a co-change.

7.2 Study Design

In this section, we present the study design. First, we define the evaluation approach and the research questions we aim to investigate in the study. We describe the dataset we used to evaluate the heuristic. Finally, we explain how we collected and processed the data.

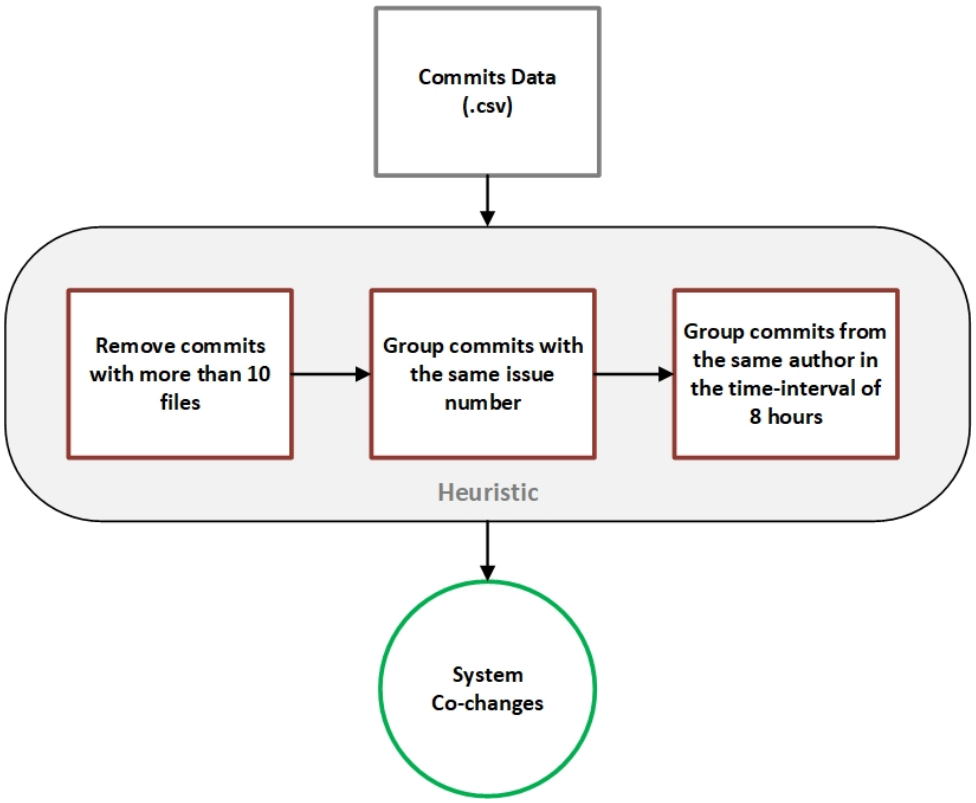

Figure 7.1: Heuristic steps performed to co-change detection.

7.2.1 Evaluation Approach

To assess the heuristic results, we compared the dependencies found by the co-change heuristic with the actual dependencies between the classes in the source code. For this purpose, we constructed a co-change graph and a dependency graph. In both graphs, the vertices correspond to the classes. The co-change graph will have an edge between A and B if the heuristic detects a co-change between classes A and B. In the dependency graph, there is an edge between A and B when there is a dependency between A and B.

We decided to apply this evaluation approach due to a theoretical rationale and to previous empirical results found in the literature. Theoretically, it is considered that the coupling among modules in a software system is a cause of change propagation [120]. Besides, the link between co-change and static dependencies has been empirically proved by previous studies [67].

We compared the Proposed Heuristic (PH) results with the Commit Heuristic (CH) results. In the commit heuristic, there is a co-change of two classes if at least one commit involves both classes.

We conducted the evaluation aiming to answer the following research questions:

RQ1. How precise is the proposed heuristic? We considered the systems' dependency graph an oracle, seeking to identify a connection between the co-change classes detected by the heuristics. The dependency graph represents the connections between the system's classes according to the static analysis of the source code. Therefore, in this RQ, we seek to identify the precision of the heuristics when detecting explicit dependencies between classes by comparing them.

RQ2. Does the amount of commits in a system influence the accuracy of the heuristics? To understand whether the systems' number of commits can influence the results, we categorized the systems into three types: *Small*, *Medium*, and *Large*. The number of commits for systems in the Small category varies from 255 to 302 commits; in the Medium category, this number varies from 704 to 636 commits; and in the Large category, the number of commits ranges from 1,027 to 2,907. In this RQ, we analyzed the accuracy of the heuristics according to their types.

RQ3. Does the distance between the classes influence their co-change? In this research question, we seek to identify whether classes that co-change more with each other tend to be closer to each other in the dependency graph. Then, we analyzed the correlation between the distance of two classes in the dependency graph and the number of co-changes these classes presented in the heuristics.

7.2.2 Dataset

We analyzed data from 32 open-source systems hosted on the GitHub platform. All systems have Java as the primary programming language. We collected their data in November 2022. We selected repositories well-rated on the platform according to their number of stars. Table 7.1 presents the systems repositories' full names, descriptions, number of commits, and stars. The systems' number of commits ranges from 256 (*KunMinX/Jetpack-MVVM-Best-Practice*) to 2,907 (*azkaban/azkaban*). They have several purposes: Frameworks, i.g., *uber/RIBs* and *alibaba/ARouter*; Android libraries, i.g., *Justson/AgentWeb* and *Tencent/tinker*; APIs ,i.g., *airbnb/DeepLinkDispatch*, among others domains.

Table 7.1: Description of the analyzed systems and their respective number of commits and star rate on Github.

System	Description	#Commits	#Starts (K)
azkaban/azkaban	Workflow manager.	2907	4.2
Justson/AgentWeb	Library based on Android WebView.	1027	8.8
zo0r/react-native-push-notification	Library for local and Remote Notifications	818	6.5
Tencent/tinker	Hot-fix solution library for Android.	815	16.7
eirslett/frontend-maven-plugin	A Maven plugin to manage Node and NPM locally.	773	3.9
alibaba/Sentinel	Flow control component for microservices.	771	20.4
google/open-location-code	Library to generate digital addresses.	708	3.8
NLPchina/ansj_seg	Library used for word segmentation.	705	6.2
square/dagger	Dependency injector for Android and Java.	704	7.3
citerus/dddsample-core	Domain-driven design application.	679	4.3
h6ah4i/android-advancedrecyclerview	Library for advanced features of RecyclerView.	673	5.2
j-easy/easy-rules	Java rules engine.	659	4.3
Genymobile/gnirehtet	System that provides reverse tethering	658	4.8
oldmanpushcart/greys-anatomy	Online troubleshooting tool for Java.	653	3.9
gabrielemariotti/cardslib	Android Library to build a UI Card.	652	4.7
socketio/socket.io-client-java	Socket.IO Client Library for Java.	328	5
alibaba/ARouter	A framework for Android componentization.	302	14.1
huanghaibin-dev/CalendarView	Calendar Widget on Android.	302	8.6
goldze/MVVMHabit	Set of libraries based on the MVVM design pattern.	298	7.2
ragunathjawahar/android-saripaar	UI form validation library for Android.	296	3.2
roncoo/roncoo-pay	Open-source online payment system.	292	4.4
airbnb/DeepLinkDispatch	API that provides access to Facebook.	291	4.3
facebookarchive/react-native-fbsdk	Wrapper for Facebook integration in React Native apps.	289	3
apache/dubbo-spring-boot-project	Java-based RPC framework.	287	5.4
orhanobut/hawk	Key-value storage for Android.	281	3.9
aurelhubert/ahbottomnavigation	Library to reproduce the behavior from Material Design.	280	3.9
rey5137/material	Library to convert components to pre-Lolipop Android.	280	6
nytimes/Store	Android Library for Async Data Loading and Caching	261	3.6
uber/RIBs	Uber's cross-platform mobile architecture framework.	260	7.3
vinc3m1/RoundedImageView	Android Library to support rounded shapes in design.	259	6.4
Meituan-Dianping/Robust	Plugin for Android hot-fix solution	258	4.4
KunMinX/Jetpack-MVVM-Best-Practice	Framework to build applications based on containers	256	4.3

7.2.3 Data Processing

The data extraction process consists of cloning the repositories, extracting data from the commits, applying the proposed heuristic, generating dependency data, generating the dependency and co-change graphs, and finally, performing the comparison between the co-change and the dependency graphs. Figure 7.2 shows the data extraction flow. We describe each step of the data processing in the following sections.

7.2.3.1 Cloning the Repositories

The first step of the data extraction was cloning the repositories and extracting the commits' data. To make the system data available in a local space, we developed a Python script using the *GitPython* library. Given a list containing the repositories'

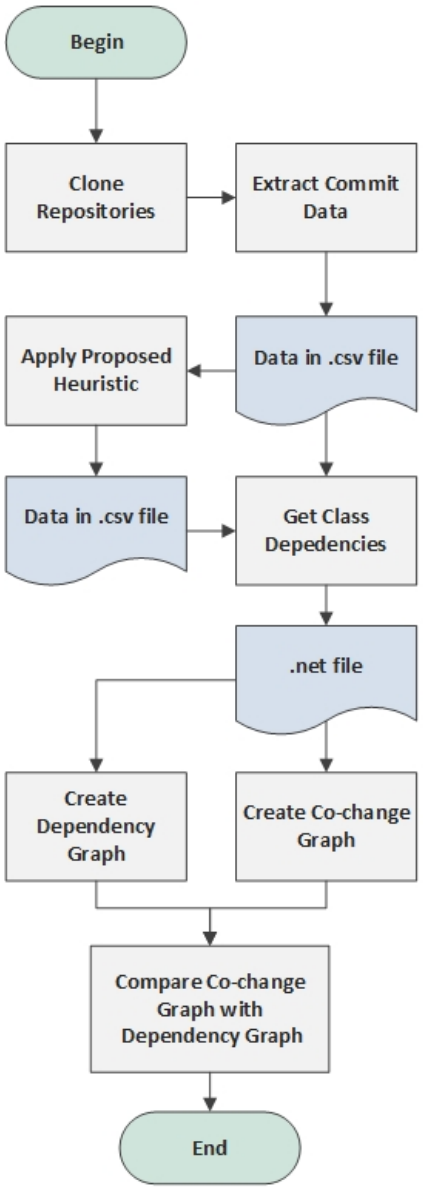

Figure 7.2: Data extraction flow

names and their GitHub URL, the script clones them into the chosen local space.

7.2.3.2 Extracting the Data from Repositories

After cloning the repositories, we generated a .csv file for each of them through a Python script that uses the *GitPython* library to extract the commits' data. For each commit in the system, the file contains the hash that identifies the commit, the author, the date (hour/min/sec), the number of modified files, the path of the files,

the message summary, and the complete message used to describe the modifications recorded in the commit.

7.2.3.3 Applying the Proposed Heuristic

The third step of the data processing was generating the .csv files for the two heuristics we compared in the study. The first heuristic, which we will call the Commit Heuristic (CH), considers that each commit corresponds to co-change in the system, i.e., there is no treatment of the commit content. The second one, the Proposed Heuristic (PH), identifies co-change according to the intrinsic characteristics of commits, as described in Section 7.1.

7.2.3.4 Getting Classes Dependencies

We modified the CK tool [15] to generate the dependency graph. CK is a system that calculates a series of Java code metrics based on the static analysis of the source code. The modification occurred in the FANIN and FANOUT metrics. The FANIN metric calculates class input coupling, i.e., the number of classes a class has as input dependency. The FANOUT metric calculates the output coupling of a class, i.e., the number of classes that a class has as output dependency. Our modification in the tool allowed CK returns the number of output and input classes and the classes' full path for these metrics. Then, we exported the data to a .txt file.

7.2.3.5 Building the Graphs

Based on the connections among the classes given by the CK tool, we generated a file (.net) containing the graph vertices and edges. The exported data followed the standard of the .net file from the system Pajek [25], an open-source system developed for the analysis and visualization of large networks. We also generate .net files for the co-change graphs. In this case, classes that belong to the same co-change are FANIN and FANOUT each other. We chose to generate the graphs

in the Pajek format to allow further data analysis using Pajek.

7.2.3.6 Comparing Co-change and Dependency Graph

The last step of the data processing was to compare the co-change data of each heuristic and the dependency graph. In this comparison, we seek to identify whether the set of classes belonging to a co-change depends on each other. For this, use the Python library *networkx* to search if there was a path between these classes in the system dependency graph. For example, a heuristic detects that classes A, B, and C are part of a co-change; then we verify whether there is a path between A-B, A-C, and B-C in the dependency graph; if the path exists, we calculate the distance between the classes. We exported the results of this comparison to files in .csv format.

This step is the most time-consuming in the data extraction flow. Due to the large number of vertices returned by the Commit Heuristic, the processing time of systems with a large number of commits exceeded 14 hours, which made collecting the data impracticable, restricting the number of systems in our dataset.

7.3 Results

RQ1. How precise is the proposed heuristic?

To answer this research question, we used the precision metric. *Precision* corresponds to the sensitivity of the proposed heuristic, i.e., it is the probability of the analyzed heuristic to identify that two classes are related and a path connects them in the dependency graph. We exhibit this metric in Equation 7.1.

$$Precision = \frac{TP}{TP \cup FP} \tag{7.1}$$

We computed a **True Positive (TP)** when the heuristic identifies the co-change between two classes, A and B, and there is a path from A to B in the dependency

graph, i.e., B is reachable from A. On the other hand, we computed a **False Positive (FP)** when the heuristic identifies the co-change between classes A and B. However, there is no path from A to B in the dependency graph, i.e., B is unreachable from A.

Figure 7.3: Precision of the Proposed Heuristic (PH) and the Commit Heuristic (CH).

Figure 7.3 shows the violin plots with the precision values for each heuristic. For the proposed heuristic (PH), the first quartile is 0.24, the median is 0.36, and the third quartile is 0.64. For the co-change heuristic without optimization (CH), the first quartile is 0.11, the median is 0.38, and the third quartile is 0.55. Therefore, PH has higher precision than CH, i.e., when analyzing co-changes based on commit data, the heuristic proposed is more likely to identify a true dependency between classes.

RQ2. Does the amount of commits in a system influence the accuracy of the heuristics?

To answer this research question, we calculated the precision values of the heuristics according to the number of commits in the repositories of the analyzed systems. As described in Section 7.2.1, we divided the systems into three groups: Large, Medium, and Small, as exhibited in Table 7.3. Figure 7.4 shows the violin plots with the results of precision.

	Repository	Precision	
		Proposed Heuristic	**Commit Heuristic**
Large	azkaban	0.69	0.52
	AgentWeb	0.24	0.20
Medium	react-native-push-notification	0.92	0.47
	tinker	0.63	0.64
	frontend-maven-plugin	0.70	0.55
	Sentinel	0.77	0.55
	open-location-code	0.34	0.89
	ansj_seg	0.01	0.00
	dagger	0.44	0.42
	dddsample-core	0.21	0.01
	android-advancedrecyclerview	0.63	0.65
	easy-rules	0.36	0.45
	gnirehtet	0.19	0.29
	greys-anatomy	0.28	0.12
	cardslib	0.62	0.56
Small	socket.io-client-java	0.03	0.00
	ARouter	0.62	0.41
	CalendarView	0.33	0.70
	MVVMHabit	0.19	0.18
	android-saripaar	0.02	0.01
	roncoo-pay	0.96	0.79
	DeepLinkDispatch	0.48	0.40
	react-native-fbsdk	0.68	0.37
	dubbo-spring-boot-project	0.04	0.00
	hawk	0.13	0.06
	ahbottomnavigation	0.79	0.53
	material	0.99	0.61
	Store	0.26	0.01
	RIBs	0.31	0.28
	RoundedImageView	0.35	0.23
	Robust	0.42	0.10
	Jetpack-MVVM-Best-Practice	0.25	0.15

Table 7.2: The precision of each analyzed system.

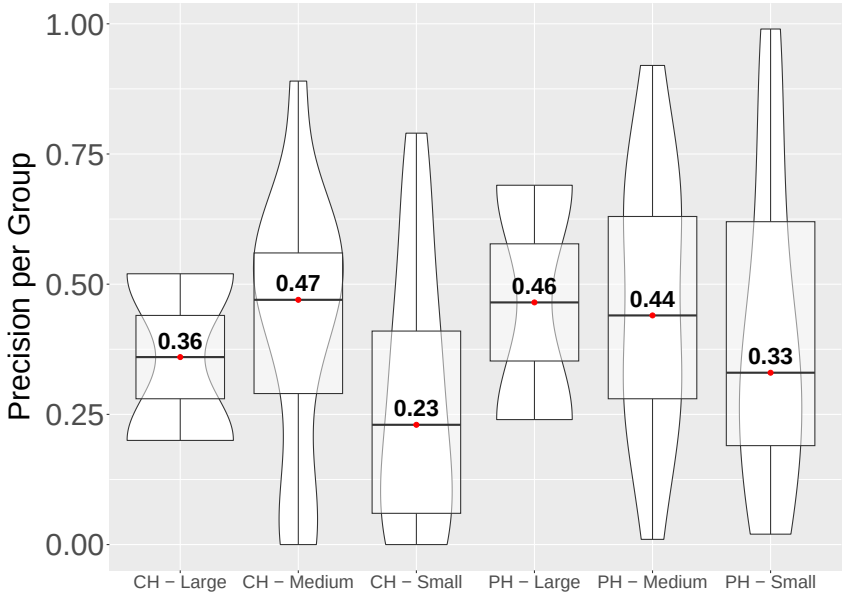

Figure 7.4: Precision of the proposed heuristic (PH) and the commit heuristic (CH) according to the commits value range.

For the system in the Large category, the Proposed Heuristic (PH) has the first quartile equal to 0.35, the median equal to 0.47, and the third quartile equal to 0.58. The Commit Heuristic (CH) has the first quartile equal to 0.28, the median equal to 0.36, and the third quartile equal to 0.44. In this category, PH presents more accurate results than CH.

In the Medium category, PH has the first quartile equal to 0.28, the median equal to 0.44, and the third quartile equal to 0.63. For CH, the first quartile is 0.29, the median is 0.47, and the third quartile is 0.56. The results demonstrate that for systems in this category, CH is more accurate than PH.

In the Small category, for PH, the first quartile is 0.19, the median is 0.33, and the third quartile is 0.62. For CH, the first quartile is 0.06, the median is 0.23, and the third quartile is 0.41. Therefore, PH has greater accuracy than CH in this category.

The results demonstrate that, in general, the accuracy of the heuristic is not associated with the number of commits analyzed by them - they perform similarly in all analyzed categories.

RQ3. Does the distance between the classes influence their co-change?

In this research question, we sought to identify whether there is a correlation between the number of times two classes underwent co-change and the distance (number of edges) between them. For that, we calculated the Pearson correlation coefficient for each system, analyzing only the true positives - co-changes that could be reflected in the dependency graph. This coefficient varies between -1 and 1, where -1 indicates a negative correlation - as one variable increases -, the other decreases, and 1 expresses that the two variables behave similarly. In some systems, the number of true positives was low, and it was impossible to calculate the coefficient. We identified them using the acronym ID (insufficient data) in Table 7.3, which presents the Pearson coefficients for each system. The results show that for PH, 62.5% of the systems have a negative Person coefficient, and for CH, this value equals 71.9%.

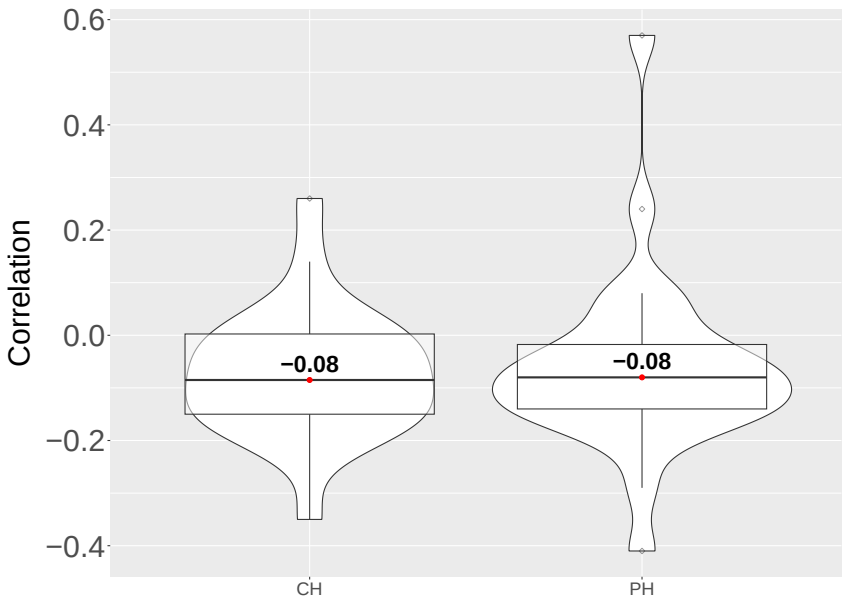

Figure 7.5: Correlation between classes co-change and classes distance.

Figure 7.5 exhibits the violin plots with the correlation results. The PH has the first quartile equal to -0.13, the median equal to -0.08 e third quartile equal to -0.01. For the CH, the first quartile is -0.16, the median is -0.08, and the third quartile is -0.02. Therefore, the negative correlation between the classes' co-change and distance is slightly higher in CH than in PH.

These results indicate that classes that change together more frequently tend to be closer to each other, i.e., the greater the number of co-changes between classes, the smaller the distance between them.

Repository	Correlation	
	PH	CH
azkaban	-0.05	-0.17
AgentWeb	-0.07	-0.20
react-native-push-notification	0.08	0.01
tinker	-0.07	-0.19
frontend-maven-plugin	0.05	-0.06
Sentinel	-0.05	-0.18
open-location-code	0.57	-0.09
ansj_seg	ID	ID
dagger	-0.01	-0.17
dddsample-core	-0.35	-0.02
android-advancedrecyclerview	0.00	-0.29
easy-rules	-0.15	-0.14
gnirehtet	-0.08	-0.07
greys-anatomy	-0.15	-0.16
cardslib	0.14	0.24
socket.io-client-java	ID	ID
ARouter	-0.09	-0.14
CalendarView	ID	-0.22
MVVMHabit	-0.11	-0.14
android-saripaar	-0.41	0.26
roncoo-pay	-0.09	-0.10
DeepLinkDispatch	-0.13	-0.04
react-native-fbsdk	ID	-0.02
dubbo-spring-boot-project	ID	ID
hawk	-0.12	0.08
ahbottomnavigation	-0.15	-0.14
material	-0.05	-0.17
Store	0.03	0.01
RIBs	ID	-0.31
RoundedImageView	0.03	-0.10
Robust	-0.21	-0.02
Jetpack-MVVM-Best-Practice	-0.06	0.01

Table 7.3: Correlation values between the number of co-changes and the distance between classes. The abbreviation ID means insufficient data.

7.4 Discussion

In our analysis, we considered the dependency graph as the basis of comparison to assess the precision of the heuristics. One may argue that if the dependency graph is the accurate set of change impact, then the co-change analysis performed employing commits' data is unnecessary. Indeed, as described mainly in the literature, the couplings between modules in a software system determine change impact proneness.

Hence, comparing the results of a co-change approach with the dependency graph is essential to assess to which extent the co-change identifies the actual dependencies among the modules.

However, it is worth noting that some types of coupling between classes may be hidden in the dependency graph. Hence, analyzing the change history is essential to capture such dependencies.

The results of the evaluation indicate that the precision of our heuristic (PH) outperforms the commit-only heuristic (CH) in terms of precision, regardless of the systems' category (small, medium, and large). Besides, the higher the number of commits in a repository, the higher the size of the historical data and, hence, the higher the expectation for the precision of the heuristic. In this sense, the precision of our approach (PH) was higher than the other one (CH) in 10 out of the 15 systems categorized as Medium and Large.

The evaluation also indicated that the results of both heuristics, in general, have a negative correlation with the distance between the classes in the dependency graph, which means that the higher the distance between the two classes, the lower the chance that a change made in a class will propagate to the other one.

7.5 Threats to Validity

Our heuristic considers the number of files changed in a commit and defining the threshold to this number is an essential point of our heuristic. We rely on the study's results described in Chapter 6 to consider commits with up to ten files.

Another strategy applied in our heuristic is to group the commits a contributor performed in eight hours. The rationale for doing so is that a contributor may conduct several sequential commits related to the same change task. Our approach's definition of the time interval was not arbitrary since we rely on the results of the study described in Chapter 6, which found that the average time interval a contributor does sequential commits is eight hours.

When evaluating the heuristics, we considered commits of 32 Java-based software systems hosted in GitHub. Therefore, the results may not be generalized to other programming languages. To mitigate this threat, we built a dataset with systems developed for different domains, such as frameworks, libraries, APIs, and others.

The number of commits in the systems varies from 255 to 2,907. The number of

commits is a threat to the evaluation since many systems have many more commits than those considered in this work. This limitation was due to the extensive processing time required to generate the Commit Heuristic data. Systems with more than 2,907 commits took more than 14 hours to process the data from these heuristics, which made it unfeasible to collect systems with more commits.

7.6 Final Remarks

In this study, we defined a new commit-based heuristic to co-change analysis. Unlike proposals found in the literature, our heuristic considers important characteristics of commits that recent results have revealed. Based on those characteristics, in our heuristic, we:

- discard commits that have more than ten files;
- group commits related to the same issue;
- group commits of the same contributor registered in the interval of eight hours.

We compared the proposed heuristic (PH) with the main approach applied by previous works, which considers all the files registered in a commit as a Co-change. We named this heuristic Commit Heuristic (CH). The basis for comparison was the dependency graph of the software systems. We constructed this graph by applying static analysis. In this graph, the nodes correspond to the classes, and the edges, are the relationships between the classes. We represented the Co-changes relations found by our approach (PH) as a graph and compared it with the dependency graph. We did the same with the Commit Heuristic (CH). Then, we compared the similarity between the dependency graphs with the resulting graphs of the two approaches. And the results indicate that our approach leads to results nearer to the actual dependencies among the classes. Besides, we also found an inverse correlation between the distance of the classes in the dependency graph and the Co-changes in both approaches. However, our approach presented a higher inverse correlation. This result indicates that the higher the distance between two classes, the lower the co-change between them.

We will apply the heuristic presented in this chapter to define a new change impact analysis approach, as described in Chapter 8.

Chapter 8

The Proposed Change Impact Model

This Ph.D. dissertation aims to define a new method for change impact analysis in object-oriented software, which is capable of, starting from the indication of the classes that will be initially changed, identifying the set of other classes that may suffer impacts from these initial changes. For this, we propose to define a hybrid probabilistic model. This chapter describes the structure of the proposed model.

8.1 Proposal Description

As a final result of the work developed in this Ph.D., we propose a probabilistic model for change impact analysis. It is a hybrid model based on the history of co-change between classes and on data from the system's dependency graph. Figure 8.1 shows the model's data source.

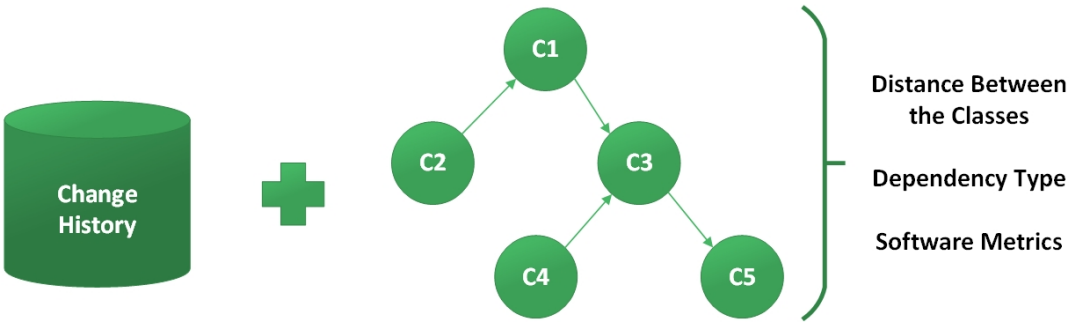

Figure 8.1: The data sources to be used in the proposed model.

To perform the change history analysis, we will apply the co-change heuristic we defined in Chapter 7 to detect co-change between the systems' classes. In the evaluation of the heuristic described in Chapter 7, we collected data from 32 Java-

based open-source software systems. We will extend this data set aiming to generate a more representative sample. For this purpose, we defined a set of 90 software systems to be analyzed, as shown in Tables 8.1, 8.2 and 8.3 at the end of this chapter. The criteria to select these software systems were the following: (i) software systems developed in Java and (ii) the best-evaluated Java systems in GitHub, according to the number of stars of the repository. We used three ranges to divide the dataset according to the number of commits: Small (241 to 292 commits), Medium (599 to 721 commits), and Large (1,891 to 2,543 commits).

Given the co-change data set, we will extract the probabilities of change impact between the classes considering the following main aspects: the type of structural dependency - inheritance, use of method, and use of fields-; the distance between them in the dependency graph; and software metrics that may be related to change propagation proneness, such as cohesion, coupling, number of methods, number of fields, and size of the parameter list. The probabilities will be defined for the data set as a whole; that is, the proposal is to use the change history data from the 90 software systems, mined by applying our co-change heuristic, to find such probabilities.

As described in Chapter 7, we extended the CK tool to collect the dependency graph. CK Tool also provides the extraction of the software metrics. However, CK Tool does not collect the type of structural dependency. Hence, we will need to extend the tool to provide this functionality.

Therefore, the proposed model will have as entry:

- the dependency graph of the software system, in which the vertices are the system's classes and the edges are their connections;
- the software metrics of the classes;
- the distance between the classes;
- the set of classes that will be initially changed.

The edges' weights in the dependency graph will be defined according to the probabilities found in the change history analysis. To calculate these weights, we will apply logistic regression. Logistic regression is a statistical model that allows the analysis of elements whose predictions are of the binary type, e.g., yes or no. We chose to use this type of regression because it allows using continuous (software metrics, distance) or categorical (co-change identification) predictors. Furthermore, logistic regression supports the use of multiple predictors.

8.2 Evaluation Method

We will base our evaluation method on an oracle of modifications built from data from open-source software systems. We will select mature and well-rated systems on the GitHub hosting platform. Besides, we will base the oracle on data extracted from issues existing on the platform since issues are used to tag and track changes. Issues can receive labels, for example, bugs. Therefore, we will build a database composed of issues related to bugs and the files that were changed related to each issue. We will adapt the scripts already developed in this work for collecting the data for oracle.

We will compare the results obtained by the proposed method with the oracle. For this, we will randomly select classes from the set of classes modified together in the oracle - this is the oracle impact set. For each class, we will compare the resulting impact set presented by our model with the oracle impact set. The evaluation will be done by calculating the accuracy and recall of the method. For false positives and false negatives, we will perform a manual analysis to analyze these results.

Category	Name	Owner	Age	#Commits	#Stars
	azkaban	azkaban	8	2543	3432
	glide	bumptech	7	2541	30209
	cryptomator	cryptomator	6	2516	4695
	quasar	puniverse	7	2494	4122
	graphql-java	graphql-java	5	2451	4549
	junit4	junit-team	11	2449	8004
	smile	haifengl	5	2446	5056
	apollo	ctripcorp	4	2425	22837
	immutables	immutables	7	2379	2780
	EhViewer	seven332	6	2376	5851
	karate	intuit	3	2339	4149
	angel	Angel-ML	3	2331	6031
	testcontainers-java	testcontainers	5	2314	4343
Large	conductor	Netflix	3	2283	3066
	zookeeper	apache	11	2237	8765
	lucida	claritylab	6	2202	4849
	mapdb	jankotek	8	2181	4128
	joda-time	JodaOrg	9	2163	4521
	Hystrix	Netflix	7	2109	20617
	shiro	apache	11	2103	3242
	shardingsphere-elasticjob	apache	5	2093	6606
	MPAndroidChart	PhilJay	6	2068	32106
	volley	google	3	2045	2876
	APIJSON	Tencent	3	2027	8431
	wiremock	tomakehurst	9	2012	4164
	rest-assured	rest-assured	10	2009	5011
	lettuce-core	lettuce-io	6	1964	3597
	android-volley	mcxiaoke	7	1932	4387
	smali	JesusFreke	8	1892	4661
	halo	halo-dev	2	1891	17208

Table 8.1: Dataset - the systems in the large category

Category	Name	Owner	Age	#Commits	#Stars
	spring-boot-starter	mybatis	5	721	3144
	spring-petclinic	spring-projects	7	720	4506
	Mapper	abel533	5	710	6085
	ansj_seg	NLPchina	8	702	5683
	dagger	square	8	701	7235
	react-native-push-notification	zo0r	4	700	5298
	xUtils3	wyouflf	5	697	5836
	AgentWeb	Justson	3	685	7607
	android-advancedrecyclerview	h6ah4i	5	672	5005
	frontend-maven-plugin	eirslett	7	665	3418
	dddsample-core	citerus	5	660	3076
	gnirehtet	Genymobile	3	657	2906
	greys-anatomy	oldmanpushcart	7	653	3557
Medium	cardslib	gabrielemariotti	7	652	4756
	SpringBoot-Labs	YunaiV	2	651	8453
	Sentinel	alibaba	2	648	14366
	open-location-code	google	6	648	3252
	zuihou-admin-cloud	zuihou	2	645	3271
	react-native-image-picker	react-native-image-picker	5	640	6687
	tinker	Tencent	4	639	15455
	easy-rules	j-easy	7	636	2705
	okdownload	lingochamp	2	624	3939
	flutter_boost	alibaba	1	618	4606
	Fragmentation	YoKeyword	4	612	9598
	android-classyshark	google	5	610	6588
	ListViewAnimations	nhaarman	7	609	5660
	SpringCloud	zhoutaoo	3	606	5332
	mosby	sockeqwe	5	603	5447
	Timber	naman14	5	602	6363
	nanohttpd	NanoHttpd	8	599	5668

Table 8.2: Dataset - the systems in the medium category.

Category	Name	Owner	Age	#Commits	#Stars
	ip2region	lionsoul2014	5	292	8684
	android-saripaar	ragunathjawahar	8	292	3170
	DeepLinkDispatch	airbnb	5	290	3864
	ARouter	alibaba	3	289	12565
	roncoo-pay	roncoo	4	288	3776
	piggymetrics	sqshq	5	286	9425
	ActiveAndroid	pardom-zz	7	284	4735
	android-tips-tricks	nisrulz	4	284	4342
	react-native-fbsdk	facebook	5	282	2911
	hawk	orhanobut	5	281	3728
	material	rey5137	5	280	6051
	ahbottomnavigation	aurelhubert	4	280	3842
	dubbo-spring-boot-project	apache	2	279	4831
Small	MVVMHabit	goldze	3	277	5951
	CalendarView	huanghaibin-dev	3	276	7125
	socket.io-client-java	socketio	7	268	4480
	RxGalleryFinal	FinalTeam	4	263	2702
	Store	nytimes	3	261	3603
	Robust	Meituan-Dianping	3	260	3801
	RoundedImageView	vinc3m1	7	258	6074
	RIBs	uber	3	257	5836
	Jetpack-MVVM-Best-Practice	KunMinX	1	255	5306
	jd-gui	java-decompiler	5	253	9263
	groupie	lisawray	4	253	3024
	elasticsearch-analysis-ik	medcl	8	252	10926
	ice	Teevity	7	251	2735
	okhttp-OkGo	jeasonlzy	4	250	10163
	UETool	eleme	2	250	2956
	Android-CleanArchitecture	android10	6	244	14569
	ViewPagerIndicator	JakeWharton	9	241	10215

Table 8.3: Dataset - the systems in the small category.

Chapter 9

Conclusion

The modification propagation analysis is an important activity for the maintenance and evolution of a software system. It is also a challenging activity: to carry it out, the developer must understand the impacts that a modification can have on a system, and this requires time and a high level of knowledge about the software structure - elements that are not always available in the daily life of the software development. Seeking to overcome the challenges faced by developers, many researchers have proposed methods for CIA over the years. However, using these methods in software development is not practical since many are complex, require extensive manual data collection, and lack tooling support. In this context, the objective of this Ph.D. is to propose a new change propagation analysis model.

As described in Chapter 2, up to the present moment, the results obtained in the development of the Ph.D. dissertation were:

- A study to understand how software engineering research has evolved and identify the general status of software maintenance research.
- A survey identifying how software maintenance has been done in practice.
- A systematic literature review on change impact analysis.
- An empirical study on commit characterization.
- A new co-change heuristic.

The results we obtain so far in this Ph.D. dissertation proposal were published in five papers. Besides, two papers were submitted to international venues.

9.1 Next Steps

To conclude this Ph.D. dissertation, we will implement and evaluate a novel change impact analysis method, as described in Chapter 8. The main difference between

our proposal and other change impact methods proposed in the literature is that it combines characteristics of change history analysis and three different types of structural analysis. In our method, the change history analysis is not done for a single software system but by a large set of software systems. The results of this analysis are, then, applied to the dependency graph of the software system, which may make the proposed method easy to be used in practice. We will perform the following steps to construct and evaluate our change impact analysis method:

1. Modify the CK tool to obtain data on types of dependencies between classes the systems' classes.
2. Run the co-change heuristic in the remaining data set.
3. Perform an empirical analysis to find the probabilities of change impact considering the structural dependency type, the distance between the classes, and the software metrics.
4. Define and implement the change impact analysis method.
5. Evaluate the proposed method, as described in Section 8.2.
6. Write the chapters of the Ph.D. dissertation describing the proposed change impact analysis method and its evaluation.
7. Write a paper about the proposed change impact analysis method.
8. Present the Ph.D. dissertation.

Table 9.1 presents the schedule for developing the activities described above.

Schedule									
	2023								
Activity	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**11**
1	x								
2		x							
3			x	x					
4					x	x			
5							x	x	
6			x	x	x	x	x	x	
7							x	x	
8									x

Table 9.1: Schedule of activities to be developed.

Bibliography

[1] Hani Abdeen, Khaled Bali, Houari Sahraoui, and Bruno Dufour. Learning dependency-based change impact predictors using independent change histories. Information and Software Technology (Inf. Softw. Technol.), 67:220–235, 2015.

[2] MK Abdi, Hakim Lounis, and H Sahraoui. Predicting change impact in object-oriented applications with bayesian networks. In 33rd International Computer Software and Applications Conference (COMPSAC), volume 1, pages 234–239. IEEE, 2009.

[3] C. Ackermann and M. Lindvall. Understanding Change Requests to Predict Software Impact. In 30th Annual IEEE/NASA Software Engineering Workshop (SWE), pages 66–75, 2006.

[4] A. Agrawal and R. K. Singh. Ruffle: Extracting co-change information from Software Project Repositories. In 1st International Conference on Smart Systems and Inventive Technology (ICSSIT), pages 88–91, 2018.

[5] Anushree Agrawal and R. K. Singh. Identification of Co-Changed Classes in Software Applications Using Software Quality Attributes. Journal of Information Technology Research (JITR), (2):110–128, 2020.

[6] Anushree Agrawal and Rakesh K. Singh. Predicting co-change probability in software applications using historical metadata. IET SOFTWARE, 14(7):739–747, 2020.

[7] Kiyoshi Agusa and Yutaka Ohno. SUPPORTING SYSTEM FOR SOFTWARE MAINTENANCE - RIPPLE EFFECT ANALYSIS OF REQUIREMENTS DESCRIPTION MODIFICATION. Journal of Information Processing, 8(3):179–189, 1985.

[8] A. Ahmad, H. Basson, and M. Bouneffa. Software evolution control towards a better identification of change impact propagation. In 4th International Conference on Emerging Technologies (ICETIS), pages 286–291, 2008.

[9] Aakash Ahmad, Claus Pahl, Ahmed B. Altamimi, and Abdulrahman Alreshidi. Mining Patterns from Change Logs to Support Reuse-Driven Evolu-

tion of Software Architectures. Journal of Computer Science and Technology (JCST), 33(6):1278–1306, 2018.

[10] Hasan Alkaf, Jameleddine Hassine, Taha Binalialhag, and Daniel Amyot. An automated change impact analysis approach for User Requirements Notation models. Journal of Systems and Software (JSS), 157:110397, 2019.

[11] Abeer AlSanad and Azeddine Chikh. The impact of software requirement change–a review. New Contributions in Information Systems and Technologies, pages 803–812, 2015.

[12] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. Change impact analysis for maintenance and evolution of variable software systems. Automated Software Engineering (Autom. Softw. Eng.), 26(2):417–461, 2019.

[13] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. Configuration-aware change impact analysis (t). In 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 385–395, 2015.

[14] Florian Angerer, Herbert Prähofer, and Paul Grünbacher. Modular change impact analysis for configurable software. In 32nd International Conference on Software Maintenance and Evolution (ICSME), pages 468–472, 2016.

[15] Maurício Aniche. Java code metrics calculator (CK), 2015. Available in https://github.com/mauricioaniche/ck/.

[16] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. A Method for Assessing Class Change Proneness. In 21st International Conference on Evaluation and Assessment in Software Engineering (EASE), pages 186–195, 2017.

[17] A. Aryani, I. D. Peake, and M. Hamilton. Domain-based change propagation analysis: An enterprise system case study. In 26th International Conference on Software Maintenance (ICSM), pages 1–9, 2010.

[18] A. Aryani, I. D. Peake, M. Hamilton, H. Schmidt, and M. Winikoff. Change Propagation Analysis Using Domain Information. In 20th Australian Software Engineering Conference (ASWEC), pages 34–43, 2009.

[19] A. Aryani, F. Perin, M. Lungu, A.N. Mahmood, and O. Nierstrasz. Predicting dependences using domain-based coupling. Journal of software: Evolution and Process (J. Softw.: Evol. Process), 26(1):50–76, 2014.

[20] M. H. Asl and N. Kama. A Change Impact Size Estimation Approach during the Software Development. In 22nd Australian Software Engineering Conference (ASWEC), pages 68–77, 2013.

[21] L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In 12th Asia-Pacific Software Engineering Conference (APSEC), pages 9 pp.–, 2005.

[22] Linda Badri, Mourad Badri, and Daniel St-Yves. Predicting change propagation in object-oriented systems: a control-call path based approach and associated tool. In 20th International Conference on Software Engineering Knowledge Engineering (SEKE), pages 103–110, 2008.

[23] Megan Bailey, King-Ip Lin, and Linda Sherrell. Clustering source code files to predict change propagation during software maintenance. In 50th Annual Southeast Regional Conference (ACMSE), pages 106–111, 2012.

[24] Sufyan Basri, Nazri Kama, Saiful Adli, and Faizura Haneem. Using static and dynamic impact analysis for effort estimation. IET Software, 10(4):89–95, 2016.

[25] Vladimir Batagelj and Andrej Mrvar. Pajek, 2022. Available in http://mrvar.fdv.uni-lj.si/pajek/.

[26] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[27] Sue Black. REST — A tool to Measure the Ripple Effect of C and C++ Programs. In Software Measurement: Current Trends in Research and Practice, pages 159–172. 1999.

[28] Sue Black. Computing ripple effect for software maintenance. Journal of Software Maintenance and Evolution (JSME), 13(4):263–279, 2001.

[29] Sue Black. Deriving an approximation algorithm for automatic computation of ripple effect measures. Information and Software Technology (Inf. Softw. Technol.), 50(7):723 – 736, 2008.

[30] B. W. Boehm. Software engineering. IEEE Trans. Comput., 25(12):1226–1241, 1976.

[31] S. A. Bohner. Extending software change impact analysis into COTS components. In 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW), pages 175–182, 2002.

[32] Andréa Sabedra Bordin and Fabiane Barreto Vavassori Benitti. Software maintenance: What do we teach and what does the industry practice? In 32th Brazilian Symposium on Software Engineering (SBES), pages 270–279, 2018.

[33] B. Breech, M. Tegtmeyer, and L. Pollock. Integrating Influence Mechanisms into Impact Analysis for Increased Precision. In 22nd IEEE International Conference on Software Maintenance (ICSME), pages 55–65, 2006.

[34] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: a tool for program comprehension during incremental change. In 13th International Workshop on Program Comprehension (IWPC), pages 149–152, 2005.

[35] Zhengong Cai, Xiaohu Yang, Xinyu Wang, and Aleksander J. Kavs. Feature location in source code by trace-based impact analysis and information retrieval. In IEICE Transactions on Information and Systems, pages 205 – 214, 2012.

[36] Jeffrey C. Carver, Oscar Dieste, Nicholas A. Kraft, David Lo, and Thomas Zimmermann. How practitioners perceive the relevance of esem research. In 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 56:1–56:10, 2016.

[37] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. Gitcproc: A tool for processing and classifying github commits. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, page 396–399, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Luca Cavallaro and Mattia Monga. Unweaving the Impact of Aspect Changes in AspectJ. In Workshop on Foundations of Aspect-oriented Languages (FOAL), pages 13–18, 2009.

[39] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. An eclectic approach for change impact analysis. In 32nd International Conference on Software Engineering (ICSE), pages 163–166, 2010.

[40] Brian Chan, King Chun Foo, Lionel Marks, and Ying Zou. An approach for estimating the time needed to perform code changes in business applications. International Journal on Software Tools for Technology Transfer (STTT), 11(6):503, 2009.

[41] Brian Chan, Lionel Marks, and Ying Zou. An approach for estimating code changes in e-commerce applications. In 10th International Symposium on Web Site Evolution (WSE), pages 111–120, 2008.

[42] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. In 3rd European Conference on Software Maintenance and Reengineering (CSMR), pages 130–138, 1999.

[43] M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. In 4th European Conference on Software Maintenance and Reengineering (CSMR), pages 45–54, 2000.

[44] Marsha Chechik, Winnie Lai, Shiva Nejati, Jordi Cabot, Zinovy Diskin, Steve Easterbrook, Mehrdad Sabetzadeh, and Rick Salay. Relationship-based change propagation: A case study. In 3rd ICSE Workshop on Modeling in Software Engineering (MiSE), pages 7–12, 2009.

[45] Chung-Yang Chen and Pei-Chi Chen. A holistic approach to managing software change impact. Journal of Systems and Software (JSS), 82(12):2051 – 2067, 2009.

[46] K. Chen and V. Rajlich. RIPPLES: Tool for change in legacy software. In 9th IEEE International Conference on Software Maintenance (ICSME), pages 230–239, 2001.

[47] Yan Chen, Ping Cheng, and Jing Yin. Change propagation analysis of trustworthy requirements based on dependency relations. In 2nd IEEE International Conference on Information Management and Engineering (ICIME), pages 246–251, 2010.

[48] J. K. Chhabra and A. Parashar. Prediction of changeability for object oriented classes and packages by mining change history. In 27th Canadian Conference on Electrical and Computer Engineering (CCECE), pages 1–6, 2014.

[49] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476–493, June 1994.

[50] Chung-Yang Chen, Cheung-Wo She, and Jia-Da Tang. An object-based, attribute-oriented approach for software change impact analysis. In 14th IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), pages 577–581, 2007.

[51] M. Dahane, M.K. Abdi, M. Bouneffa, A. Ahmad, and H. Basson. Using design of experiments to analyze open source software metrics for change impact estimation. International Journal of Open Source Software and Processes (IJOSSP), (1):16–33, 2019.

[52] H. K. Dam and A. Ghose. Automated change impact analysis for agent systems. In 27th IEEE International Conference on Software Maintenance (ICSM), pages 33–42, 2011.

[53] H.K. Dam and A. Ghose. Supporting change impact analysis for intelligent agent systems. Science of Computer Programming (SCP), 78(9):1728–1750, 2013.

[54] K. H. Dam, M. Winikoff, and L. Padgham. An agent-oriented approach to change propagation in software evolution. In 17th Australian Software Engineering Conference (ASWEC), pages 10 pp.–318, 2006.

[55] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. Traceability management for impact analysis. In 2008 Frontiers of Software Maintenance, pages 21–30. IEEE, 2008.

[56] L. Deruelle, M. Bouneffa, N. Melab, and H. Basson. A change propagation model and platform for multi-database applications. In Proceedings IEEE International Conference on Software Maintenance. ICSM, pages 42–51, 2001.

[57] Ankit Dhamija and Sunil Sikka. A systematic study of advancements in changeimpact analysis techniques. International Journal of Soft Computing and Engineering, 8:1 – 9, 2019.

[58] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 341–350, 2015.

[59] Philipp Diebold and Antonio Vetrò. Bridging the gap: Se technology transfer into practice: Study design and preliminary results. In 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 52:1–52:4, 2014.

[60] Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. ImpactMiner: A Tool for Change Impact Analysis. In 36th International Conference on Software Engineering (ICSE), pages 540–543, 2014.

[61] Jessica Díaz, Jennifer Pérez, Juan Garbajosa, and Alexander L. Wolf. Change Impact Analysis in Product-Line Architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, 5th European Conference on Software Architecture (ECSA), pages 114–129, Berlin, Heidelberg, 2011.

[62] Hamzeh Eyal Salman, Abdelhak-Djamel Seriai, and Christophe Dony. Feature-level change impact analysis using formal concept analysis. International Journal of Software Engineering and Knowledge Engineering (Int. J. Softw. Eng. Knowl. Eng), 25(01):69–92, 2015.

[63] Mívian Ferreira, Diego Golçalves, Kecia Ferreira, and Mariza Bigonha. Inside commits: An empirical study on commits in open-source software. In Brazilian Symposium on Software Engineering, SBES '21, page 11–15, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Mívian M. Ferreira, Bruno L. Sousa, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. The software engineering observatory portal. In Proceedings of the 18th International Conference on Scientometric & Informetrics, pages 407–412. International Society for Scientometrics and Informetrics (I.S.S.I.), 2021.

[65] M.J. Fyson and C. Boldyreff. Using application understanding to support impact analysis. Journal of Software Maintenance, 10(2):93 – 110, 1998.

[66] Spyridon K. Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. International Journal of Automation and Computing (IJAC), 6(2):109–123, 2009.

[67] Markus Michael Geipel and Frank Schweitzer. The link between dependency and cochange: Empirical evidence. IEEE Transactions on Software Engineering, 38(6):1432–1444, 2012.

[68] D. M. German, G. Robles, and A. E. Hassan. Change Impact Graphs: Determining the Impact of Prior Code Changes. In 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 184–193, 2008.

[69] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In 34th International Conference on Software Engineering (ICSE), pages 430–440, 2012.

[70] Chetna Gupta, Maneesha Srivastav, and Varun Gupta. Software change impact analysis: an approach to differentiate type of change to minimise regression test selection. International Journal of Computer Applications in Technology (IJCAT), 51(4):366–375, 2015.

[71] S. Gwizdala, Yong Jiang, and V. Rajlich. Tracker - a tool for change propagation in Java. In 7th European Conference on Software Maintenance and Reengineering (CSMR), pages 223–229, 2003.

[72] Alex Gyori, Shuvendu K Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 318–328, 2017.

[73] J. Han. Supporting impact analysis and change propagation in software engineering environments. In 8th International Workshop on Software Technology and Engineering Practice (STEP), pages 172–182, 1997.

[74] Q. Hanam, A. Mesbah, and R. Holmes. Aiding Code Change Understanding with Semantic Change Impact Analysis. In 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 202–212, 2019.

[75] S. Hassaine, F. Boughanmi, Y. Guéhéneuc, S. Hamel, and G. Antoniol. A seismology-inspired approach to study change propagation. In 27th IEEE International Conference on Software Maintenance (ICSM), pages 53–62, 2011.

[76] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In 20th IEEE International Conference on Software Maintenance (ICSME), pages 284–293, 2004.

[77] M. O. Hassan, L. Deruelle, and H. Basson. A knowledge-based system for change impact analysis on software architecture. In 4th International Conference on Research Challenges in Information Science (RCIS), pages 545–556, 2010.

[78] L. P. Hattori and M. Lanza. On the nature of commits. In 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 63–71, 2008.

[79] Chunling Hu, Bixin Li, and Xiaobing Sun. Mining variable-method correlation for change impact analysis. IEEE Access, 6:77581–77595, 2018.

[80] N. Ibrahim, W. M. N. W. Kadir, and S. Deris. Propagating Requirement Change into Software High Level Designs towards Resilient Software Evolution. In 16th Asia-Pacific Software Engineering Conference (APSEC), pages 347–354, 2009.

[81] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. Detecting asynchrony and dephase change patterns by mining software repositories. Journal of Software: Evolution and Process, 26(1):77–106, 2014.

[82] Jackson and Ladd. Semantic Diff: a tool for summarizing the effects of modifications. In 2nd International Conference on Software Maintenance (ICSM), pages 243–252, 1994.

[83] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. Empirical Software Engineering Journal (Empir. Softw. Eng. J.), 18(5):933–969, 2013.

[84] Nazri Kama and Faizul Azli. A change impact analysis approach for the software development phase. In 19th Asia-Pacific Software Engineering Conference (APSEC), volume 1, pages 583–592, 2012.

[85] D. Kchaou, N. Bouassida, and H. Ben-Abdallah. UML models change impact analysis using a text similarity technique. IET Software, 11(1):27–37, 2017.

[86] Prateek Khurana, Aprna Tripathi, and Dharmender Singh Kushwaha. Change impact analysis and its regression test effort estimation. In 3rd IEEE International Advance Computing Conference (IACC), pages 1420–1424, 2013.

[87] T. Kim, K. Kim, and W. Kim. An Interactive Change Impact Analysis Based on an Architectural Reflexion Model Approach. In 34th Annual Computer Software and Applications Conference (COMPSAC), pages 297–302, 2010.

[88] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical report, Keele University and Durham University Joint Report, 2007.

[89] Barbara A Kitchenham, David Budgen, and O Pearl Brereton. Using mapping studies as the basis for further research–a participant-observer case study. Information and Software Technology, 53(6):638–651, 2011.

[90] Barbara A. Kitchenham and Shari L. Pfleeger. Personal Opinion Surveys, chapter 3, pages 63–92. Springer London, London, 2008.

[91] Antje von Knethen. A trace model for system requirements changes on embedded systems. In 4th International Workshop on Principles of Software Evolution (IWPSE), page 17–26, 2001.

[92] Kenichi Kobayashi, Akihiko Matsuo, Katsuro Inoue, Yasuhiro Hayase, Manabu Kamimura, and Toshiaki Yoshino. Impactscale: Quantifying change impact to predict faults in large software systems. In 27th IEEE International Conference on Software Maintenance (ICSM), pages 43–52, 2011.

[93] T. Kobayashi, N. Kato, and K. Agusa. Interaction histories mining for software change guide. In 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE), pages 73–77, 2012.

[94] Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. Mining file histories: should we consider branches? In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 202–213, 2018.

[95] Kung, Gao, Hsia, Wen, Toyoshima, and Chen. Change impact identification in object oriented software maintenance. In 2nd International Conference on Software Maintenance (ICSM), pages 202–211, 1994.

[96] Kyung-Hee Kim, Jai-Nyun Park, and Yong-Ik Yoon. A graph of change impact analysis for distributed object-oriented software. In IEEE International Conference on Fuzzy Systems (FUZZ), pages 1137–1141 vol.2, 1999.

[97] M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS, pages 61–70, 2000.

[98] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, Software Process Technology, pages 108–124, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[99] S. Lehnert. A review of software change impact analysis. 2011.

[100] S. Lehnert, Q. Farooq, and M. Riebisch. Rule-Based Impact Analysis for Heterogeneous Software Artifacts. In 17th European Conference on Software Maintenance and Reengineering (CSMR), pages 209–218, 2013.

[101] Andrew Leigh, Michel Wermelinger, and Andrea Zisman. Evaluating the effectiveness of risk containers to isolate change propagation. Journal of Systems and Software (JSS), 176:110947, 2021.

[102] Li and Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In 4th Proceedings of International Conference on Software Maintenance (ICSM), pages 171–184, 1996.

[103] Bixin Li, Xiaobing Sun, and Jacky Keung. Fca–cia: An approach of using fca to support cross-level change impact analysis for object oriented java programs. Information and Software Technology (IST), 55(8):1437–1449, 2013.

[104] Bixin Li, Xiaobing Sun, and Hareton Leung. Combining concept lattice with call graph for impact analysis. Advances in Engineering Software, 53:1 – 13, 2012.

[105] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability, 23(8):613–646, 2013.

[106] Bixin Li, Qiandong Zhang, Xiaobing Sun, and Hareton Leung. Using water wave propagation phenomenon to study software change impact analysis. Advances in Engineering Software, 58:45 – 53, 2013.

[107] L. Li, G. Qian, and L. Zhang. Evaluation of Software Change Propagation Using Simulation. In 1st WRI World Congress on Software Engineering (WCSE), pages 28–33, 2009.

[108] L. Li, L. Zhang, L. Lu, and Z. Fan. Assessing Object-Oriented Software Systems Based on Change Impact Simulation. In 10th IEEE International Conference on Computer and Information Technology (ICCIT), pages 1364–1369, 2010.

[109] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In 10th Joint Meeting on Foundations of Software Engineering (FSE), pages 415–425, 2015.

[110] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining Performance Regression Inducing Code Changes in Evolving Software. In 13th Working Conference on Mining Software Repositories (MSR), pages 25–36, 2016.

[111] Kecia A. M. Ferreira, Mariza A. S. Bigonha, Roberto S. Bigonha, Bernardo N. de Lima, Bárbara M. Gomes, and Luiz Felipe O. Mendes. A model for estimating change propagation in software. Software Quality Journal (SQJ), 26(2):217–248, 2018.

[112] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting build co-changes with source code change and commit categories. In 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 541–551, 2016.

[113] B. Mackenzie, V. Pantelic, G. Marks, S. Wynn-Williams, G. Selim, M. Lawford, A. Wassyng, M. Diab, and F. Weslati. Change impact analysis in Simulink designs of embedded systems. In 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 1274–1284, 2020.

[114] Mirna Carelli Oliveira Maia, Roberto Almeida Bittencourt, Jorge Cesar Abrantes de Figueiredo, and Dalton Dario Serey Guerrero. The hybrid technique for object-oriented software change impact analysis. In 14th European Conference on Software Maintenance and Reengineering (CSMR), pages 252–255. IEEE, 2010.

[115] Mrinaal Malhotra and Jitender Kumar Chhabra. Improved Computation of Change Impact Analysis in Software Using All Applicable Dependencies. In 2nd Futuristic Trends in Network and Communication Technologies (FTNCT), pages 367–381, 2019.

[116] H. Malik and E. Shakshuki. Predicting Function Changes by Mining Revision History. In 7th International Conference on Information Technology: New Generations (ITNG), pages 950–955, 2010.

[117] Haroon Malik and Ahmed E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In 24th IEEE International Conference on Software Maintenance (ICSME), pages 177–186, 2008.

[118] N. Mansour and H. Salem. Ripple effect in object oriented programs. Journal of Computational Methods in Sciences and Engineering (JCMSE), 6(5-6):S23–S32, 2006.

[119] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining Co-change Information to Understand When Build Changes Are Necessary. In 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 241–250, 2014.

[120] Bertrand Meyer. Object-oriented software construction. Pearson, 2000.

[121] S. Mirarab, A. Hassouna, and L. Tahvildari. Using Bayesian Belief Networks to Predict Change Propagation in Software Systems. In 15th IEEE International Conference on Program Comprehension (ICPC), pages 177–188, 2007.

[122] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Associating Code Clones with Association Rules for Change Impact Analysis. In 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 93–103, 2020.

[123] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. HistoRank: History-Based Ranking of Co-change Candidates. In 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 240–250, 2020.

[124] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In 21st International Conference on Program Comprehension (ICPC), pages 103–112, 2013.

[125] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pages 358–362, 2014.

[126] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In 2013 21st International Conference on Program Comprehension (ICPC), pages 103–112. IEEE, 2013.

[127] M. M. El Nemr and D. S. Elzanfaly. A Framework for Advancing Change Impact Analysis in Software Development Using Graph Database. In 24th International Conference on Computer and Applications (ICCA), pages 434–438, 2018.

[128] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pages 250–260. IEEE, 2015.

[129] Anshu Parashar and Jitender Kumar Chhabra. Measurement of package-changeability by mining change-history. Procedia Computer Science (Procedia Comput. Sci.), 46:443–448, 2015.

[130] Anshu Parashar and Jitender Kumar Chhabra. Mining software change data stream to predict changeability of classes of object-oriented software system. Evolving Systems, 7(2):117–128, 2016.

[131] Prem Parashar, Rajesh Bhatia, and Arvind Kalia. Change impact analysis: A tool for effective regression testing. In 5th International Conference on Information Intelligence, Systems, Technology and Management (ICISTM), pages 160–169, 2011.

[132] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. Change impact analysis of feature models. In 18th International Conference on Information and Software Technologies (ICIST), pages 108–122, 2012.

[133] S.L Pfleeger. Understanding and improving technology transfer in software engineering. Journal of Systems and Software (JSS), 47(2):111 – 124, 1999.

[134] G. Pirklbauer, C. Fasching, and W. Kurschl. Improving Change Impact Analysis with a Tight Integrated Process and Tool. In 7th International Conference on Information Technology: New Generations (ITNG), pages 956–961, 2010.

[135] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. Empirical Software Engineering, 14(1):5–32, 2009.

[136] V. Rajlich. A model for change propagation based on graph rewriting. In 5th International Conference on Software Maintenance (ICSM), pages 84–91, 1997.

[137] Jr. Samuel T. Redwine and William E. Riddle. Software technology maturation. In 8th International Conference on Software Engineering (ICSE), pages 189–200, 1985.

[138] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. Approach for solving the feature location problem by measuring the component modification impact. IET Software, (4):292–311, 2009.

[139] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley. Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis. In 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 201–212, 2016.

[140] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In 28th IEEE International Conference on Software Maintenance (ICSM), pages 109–118, 2012.

[141] B.G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 46–53, 2001.

[142] M. Shahid and S. Ibrahim. Change impact analysis with a software traceability approach to support software maintenance. In 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST), pages 391–396, 2016.

[143] T. Sharma and G. Suryanarayana. Augur: Incorporating Hidden Dependencies and Variable Granularity in Change Impact Analysis. In 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 73–78, 2016.

[144] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. In 1st International Conference on Software Testing, Verification, and Validation (ICST), pages 268–277, 2008.

[145] Maryam Shiri, Jameleddine Hassine, and Juergen Rilling. A requirement level modification analysis support framework. In 3rd International IEEE Workshop on Software Evolvability (Software-Evolvability), pages 67–74, 2007.

[146] Luciana L. Silva, Marco Tulio Valente, and Marcelo A. Maia. Co-change patterns: A large scale empirical study. Journal of Systems and Software (JSS), 152:196–214, 2019.

[147] Luciana Lourdes Silva, Marco Tulio Valente, and Marcelo de A. Maia. Assessing Modularity Using Co-change Clusters. In 13th International Conference on Modularity (MODULARITY), pages 49–60, 2014.

[148] Bruno L. Sousa, Mívian M. Ferreira, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. Software engineering evolution: The history told by icse. In Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES '19, page 17–21, New York, NY, USA, 2019. Association for Computing Machinery.

[149] X. Sun, H. Leung, B. Li, and B. Li. Change impact analysis and changeability assessment for a change proposal: An empirical study. Journal of Systems and Software (JSS), 96:51–60, 2014.

[150] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang. Change Impact Analysis Based on a Taxonomy of Change Types. In 34th Annual Computer Software and Applications Conference (COMPSAC), pages 373–382, 2010.

[151] X. Sun, B. Li, W. Wen, and S. Zhang. Analyzing impact rules of different change types to support change impact analysis. International Journal of Software Engineering and Knowledge Engineering (Int. J. Softw. Eng. Knowl. Eng), 23(3):259–288, 2013.

[152] Antony Tang, Yan Jin, and Jun Han. A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software (J. Syst. Softw.), 80(6):918–934, 2007.

[153] CMMI Product Team. CMMI for Development, Version 1.3. Software Engineering Institute, 2010.

[154] Tie Feng and J. I. Maletic. Using Dynamic Slicing to Analyze Change Impact on Role Type based Component Composition Model. In 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR), pages 103–108, 2006.

[155] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. IEEE Transactions on Software Engineering (IEEE Trans. Softw. Eng.), 31(7):601–614, 2005.

[156] R.J. Turver and M. Munro. An early impact analysis technique for software maintenance. Journal of Software Maintenance: Research and Practice (J SOFTW MAINT EVOL-R), 6(1):35–52, 1994.

[157] Robert J. Walker, Reid Holmes, Ian Hedgeland, Puneet Kapur, and Andrew Smith. A Lightweight Approach to Technical Risk Estimation via Probabilistic Impact Analysis. In 3rd International Workshop on Mining Software Repositories (MSR), pages 98–104, 2006.

[158] Chengcheng Wan, Zece Zhu, Yuchen Zhang, and Yuting Chen. Multi-perspective change impact analysis using linked data of software engineering. In 8th Asia-Pacific Symposium on Internetware (Internetware), page 95–98, 2016.

[159] Wanzhi Wen, Jianping Chen, Jiaqi Yuan, and Xiaoyong Chen. Evolution slicing-based change impact analysis. In 3rd International Conference on Big Data Computing Service and Applications (BigDataService), pages 293–298, 2017.

[160] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Igor Steinmacher, Gustavo Ansaldi Oliva, Reginaldo Ré, Christoph Treude, and Marco Aurelio Gerosa. Pieces of contextual information suitable for predicting co-changes? An empirical study. Software Quality Journal (SQJ), 27(4):1481–1503, 2019.

[161] I.S. Wiese, R. Ré, I. Steinmacher, R.T. Kuroda, G.A. Oliva, C. Treude, and M.A. Gerosa. Using contextual information to predict co-changes. Journal of Systems and Software (JSS), 128:220–235, 2017.

[162] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in software engineering. Springer Science & Business Media, 2012.

[163] S. Wong and Y. Cai. Predicting change impact from logical models. In 25th IEEE International Conference on Software Maintenance (ICSM), pages 467–470, 2009.

[164] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. Cross-project build co-change prediction. In 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 311–320, 2015.

[165] Hua Xiao, Jin Guo, and Ying Zou. Supporting Change Impact Analysis for Service Oriented Business Applications. In International Workshop on Systems Development in SOA Environments (SDSOA), pages 6–, Washington, DC, USA, 2007.

[166] Z. Xiao-Bo, J. Ying, and W. Hai-Tao. Method on Change Impact Analysis for Object-Oriented Program. In 4th International Conference on Intelligent Networks and Intelligent Systems (ICISN), pages 161–164, 2011.

[167] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In 2nd International Computer Software and Applications Conference (COMPSAC), pages 60–65, 1978.

[168] S.S. Yau and J.S. Collofello. Some stability measures for software maintenance. IEEE Transactions on Software Engineering (IEEE Trans. Softw. Eng.), SE-6(6):545–552, 1980.

[169] A. R. Yazdanshenas and L. Moonen. Fine-grained change impact analysis for component-based product families. In 28th IEEE International Conference on Software Maintenance (ICSM), pages 119–128, 2012.

[170] L. Yu and S.R. Schach. Applying association mining to change propagation. International Journal of Software Engineering and Knowledge Engineering (Int. J. Softw. Eng. Knowl. Eng), 18(8), 2008.

[171] M. Zalewski and S. Schupp. Change Impact Analysis for Generic Libraries. In 22nd IEEE International Conference on Software Maintenance (ICSME), pages 35–44. ISSN: 1063-6773.

[172] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In 24th IEEE International Conference on Software Maintenance (ICSM), pages 87–96, 2008.

[173] J. Zhao, H. Yang, L. Xiang, and B. Xu. Change impact analysis to support architectural evolution. Journal of Software Maintenance and Evolution, 14(5):317–333, 2002.

[174] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead, Jr. Mining Version Archives for Co-changed Lines. In 3rd International Workshop on Mining Software Repositories (MSR), pages 72–75, 2006.