Università della Svizzera italiana

Software Institute

Mining Code Change Patterns to Aid Software Development

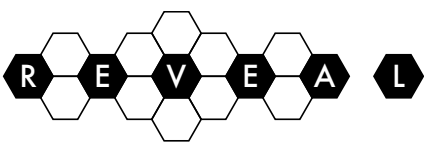
Fengcai Wen

Research Advisor

Prof. Dr. Gabriele Bavota

Research Co-Advisor

Prof. Dr. Michele Lanza



Dissertation Committee

Prof. Cesare Pautasso	Università della Svizzera italiana, Switzerland
Prof. Carlo Alberto Furia	Università della Svizzera italiana, Switzerland
Prof. Liliana Pasquale	University College Dublin, Ireland
Prof. Filippo Ricca	University of Genova, Italy

Dissertation accepted on 20 September 2021

Research Advisor	Co-Advisor
Prof. Dr. Gabriele Bavota	**Prof. Dr. Michele Lanza**
Ph.D. Program Co-Director	Ph.D. Program Co-Director
Prof. Walter Binder	**Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Fengcai Wen
Lugano, 20 September 2021

Abstract

Mining Software Repositories (MSR) has become a complete and mature research field, also due to the increasing number of open source projects publicly available. Repository hosting services such as GitHub provide unprecedented access to millions of events generated during development activities (*e.g.,* code commits, pull requests), that can be mined and analyzed to extract new pieces of knowledge. By analyzing the source code of a large corpus of software systems, recent work showed that most software is *natural*, meaning that it is likely to be repetitive and predictable. In other words, development and maintenance activities are likely the results of *unexposed code change patterns* that, if properly exploited, can be used to support code-related activities (*e.g.,* implementing a new feature).

Starting from these observations, we formulate our thesis statement: *Mining code change patterns from open source repositories enables researchers to gather large-scale, historical information about development and maintenance activities performed by developers. The collected empirical knowledge, once converted into actionable items, can support software developers on code-related tasks.*

We investigated the possibility of acquiring new empirical knowledge from mining three specific types of code change patterns in open source repositories: (i) the introduction and fix of code-comment inconsistencies, (ii) omitted code changes in developers' commits and (iii) implementation patterns followed by developers when implementing a new feature. We leveraged the knowledge acquired from the last type of patterns, to design and build FeaRS, an approach and a tool that, given the methods developers already wrote in the IDE, is able to suggest the complete code of the next method they are likely to implement. Our results show that mining unexposed code change patterns from open source repositories can help in better understanding development activities and potentially support developers during software development.

Acknowledgements

My Ph.D. has been an incredible journey – like any journey, it has had its ups and downs, but I am extremely grateful for this experience as it has enabled me to work with so many brilliant people at such an excellent research institute.

I would like to express my gratitude to my supervisors Prof. Gabriele Bavota and Prof. Michele Lanza, who have been great mentors guiding me with research advice and words of wisdom throughout my Ph.D. journey. Your passion and vision for doing research have been a deep inspiring driving force for me. Moreover, thank you for always being supportive and encouraging in every aspect of my life. I feel so fortunate and proud to be one of your Ph.D. students.

I also owe many thanks to Csaba Nagy, who has been closely working with me throughout my entire Ph.D. journey as a postdoc fellow. I would never become an independent researcher without your ever-present support. In addition, thank you for inviting me for those long-distance hikes we had accomplished together, which shows me the importance of believing in my own potential. I would also like to thank Emad Aghajani for collaborating with me, always being responsive and reliable. Thank you also to Bin Lin, Alejandro Mazuera Rozo, and Diego Marcilio - we always discuss everything together and it has been a privilege to "grow old" with you during my Ph.D. journey. A big shout-out to Roberto Minelli and Elisa Larghi, thank you for organizing and managing all the activities and events I have been involved with and I really appreciate that. Of course, I would like to thank the people who have made it a joy to work in the research group: Jevgenija Pantiuchina, Luca Pascarella, Matteo Ciniselli, Antonio Mastropaolo, Rosalia Tufano, Susanna Ardigò, Marco Raglianti and Aron Fiechter.

Thank you so much all of my dissertation committee members: Prof. Cesare Pautasso, Prof. Carlo Alberto Furia, Prof. Liliana Pasquale and Prof. Filippo Ricca, for accepting my invitation, taking time to review this thesis, attending my defense and sharing your wisdom.

Last but certainly not least, thank you to my family. Thank you my parent for much needed moral support and great advice throughout my Ph.D. journey. Thank you Yiwei, my wife, for always staying with me and loving me. And a special thank you to my little daughter, Leah, for bringing me the unique happiness.

Contents

Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Thesis Statement . . . 2
- 1.2 Research Contributions . . . 2
  - 1.2.1 Mining change patterns to build new empirical knowledge about development and maintenance activities performed in open source systems 3
  - 1.2.2 Leverage mined patterns to develop techniques and tools supporting developers in code-related tasks . . . 5
- 1.3 Outline . . . 6

2 State of the Art 7
- 2.1 Introduction . . . 7
- 2.2 Empirical Studies on Developers' Commits . . . 7
  - 2.2.1 Reasons for Changes . . . 8
  - 2.2.2 Effects of a Change on Code Quality . . . 8
  - 2.2.3 Changes and Time . . . 10
  - 2.2.4 Change Patterns . . . 10
  - 2.2.5 Bias and Noise in Mining Change Histories . . . 10
  - 2.2.6 Summing Up . . . 12
- 2.3 Introduction and Fix of Code-Comment Inconsistencies . . . 12
  - 2.3.1 Empirical Studies on Code Comments . . . 12
  - 2.3.2 Automatic Assessment of Comments Quality . . . 14
  - 2.3.3 Summing Up . . . 15
- 2.4 Source Code Recommender Systems . . . 15
  - 2.4.1 Code Completion Techniques . . . 16
  - 2.4.2 Code Search Engines . . . 17
  - 2.4.3 Summing Up . . . 18
- 2.5 Conclusion . . . 18

3 Studying Code-Comment Inconsistencies 21
- 3.1 Introduction . . . 21
- 3.2 Study Design . . . 22

3.2.1 Data Collection and Analysis 23
3.3 Results 27
3.3.1 To what extent do different code change types trigger comment updates? 27
3.3.2 What types of code-comment inconsistencies are fixed by developers? 31
3.3.3 Discussion and Implications 35
3.4 Threats to Validity 37
3.5 Conclusion 38

4 Quick Remedy Commits and Their Impact on Mining Software Repositories 39
4.1 Introduction 39
4.2 Study I: Quick Remedy Commits Performed by Developers 43
4.2.1 Study Design 43
4.2.2 Results 46
4.3 Study II: Impact of Reverted Commits on MSR Data Collection 53
4.3.1 Study Design 53
4.3.2 Results 56
4.3.3 Summing Up 59
4.4 Threats to Validity 59
4.5 Conclusion 60

5 Using Code Change Patterns for Code Recommendations 61
5.1 Introduction 61
5.2 FeaRS 64
5.2.1 Mining Android Apps 64
5.2.2 Identifying Methods Added in Commits 65
5.2.3 Clustering Similar Methods 65
5.2.4 Association Rule Mining 66
5.2.5 FeaRS Web Service and IDE Plugin 67
5.3 Study Design 68
5.3.1 Context Selection and Data Collection 68
5.3.2 Data Analysis 71
5.4 Results Discussion 73
5.4.1 FeaRS Parameters Tuning 73
5.4.2 Quantitative Results 74
5.4.3 Qualitative Examples 76
5.5 Threats to Validity 79
5.6 Conclusions 80

6 Conclusions and Future Work 83
6.1 Conclusions 83
6.2 Future Work 84
6.2.1 Detecting and Fixing Simple Omission Errors 84
6.2.2 Improving FeaRS 85
6.3 Closing Words 85

Bibliography **87**

Projects' References **103**

Figures

3.1 Code-comment evolution: MCC and CCC by change category (Table 3.2) . . 28
3.2 Statistical comparison of the chance of triggering method (top) and class (bottom) comment update by change category (Table 3.2) 29
3.3 Taxonomy of Code Comment-Related Changes 32

4.1 Example of quick remedy commit 41
4.2 Time differences (in minutes) between subsequent commits (without outliers) 44
4.3 Web application used to run the manual tagging 45
4.4 Taxonomy of Quick Remedy Commits 47
4.5 Impact of reverted commits on bug-fixing commits. 56
4.6 Impact of reverted commits on refactoring commits. 58

5.1 An implementation pattern in Android 62
5.2 The FeaRS pipeline 64
5.3 The FeaRS Android Studio plugin 67
5.4 Study Design 69
5.5 Data splitting and processing 70
5.6 An example of $dist_{tokens}$ calculation 72
5.7 Tuning of FeaRS's parameters 73
5.8 Correct recommendation to the usage of external storage in Android. 76
5.9 Correct recommendation to provide a custom back navigation for an Android DrawerLayout. 77
5.10 Correct recommendation for the creation of a GoogleMap instance from the Google Maps SDK for Android. 77
5.11 Unmatched recommendation for user credential validation in sign-up activity. 78
5.12 Unmatched recommendation for creating custom filter for filterable adapter in Android. 79

Tables

3.1 Dataset Statistics 23
3.2 Categories of AST-level Code Changes 25

5.1 FeaRS parameters tuning options 70
5.2 Performance when considering all methods 75
5.3 Performance when excluding short methods 76

1

Introduction

The rise of open source development and the adoption of version control systems as a *de facto* standard when developing software, has resulted in an unprecedented amount of data that can be mined and analyzed to collect new knowledge about how software systems are developed and maintained. In the last decade, this resulted in the growth of a new research field named Mining Software Repositories (MSR) [1], having the aim of analyzing and cross-linking the rich data available in software repositories to discover interesting and actionable information about software systems and projects. While most MSR studies focus on gathering empirical knowledge about specific research questions of interest [2, 3, 4, 5, 6, 7], researchers are also using the data mined from software repositories to create a new generation of recommender systems supporting software developers in everyday activities [8]. The goal of these tools is to increase the productivity of software developers, by lowering their learning curves when dealing with unfamiliar code, and by maximizing the quality of the code they write.

One research direction enabled by MSR is the identification of *unexposed code change patterns* (*i.e.,* recurrent coding patterns performed by software developers). Recent research showed that source code is natural [7, 9, 10, 11, 12, 13, 14], meaning that most of it is likely to be repetitive and predictable. This observation led to many studies investigating the phenomenon. For example, Nguyen *et al.* [15] found that, on average, 71% of an API's usage in a project is covered by API usage patterns (*i.e.,* repetitive code snippets using an API and sharing the same variables and control structures). They also indicated that 12% of the routines are repeated within a project [16], where a routine is defined as a portion of code that performs a specific task independently and can be called by the remaining code (*e.g.,* a procedure, function, or method). Based on these findings, we believe that useful and important code change patterns can be uncovered from the huge amount of data collected from open source repositories. For example, a change to the code which writes data to a file may require changes to the code which reads data from the file, despite the absence of dependencies between these two pieces of code.

Despite the many works in the MSR field, there is still little evidence about how unexposed code change patterns can be exploited to support and automate code-related tasks. This is the main goal of this thesis.

1.1 Thesis Statement

We formulate our thesis statement as follows:

> *Mining code change patterns from open source repositories enables researchers to gather large-scale, historical information about development and maintenance activities performed by developers. The collected empirical knowledge, once converted into actionable items, can support software developers on code-related tasks.*

To validate our thesis, we investigated the possibility of acquiring new empirical knowledge from mining three specific types of code change patterns in open source repositories: (i) the introductions and fixes of code-comment inconsistencies, (ii) omitted code changes in developers' commits and (iii) implementation patterns followed by developers when implementing a new feature. We leveraged the knowledge acquired from the last type of patterns, to design and build FeaRS, an approach and a tool that, given the methods developers already wrote in the IDE, is able to suggest the complete code of the next method they are likely to implement.

The three code change patterns we address, while not strictly related to each other, have been selected since they share the following characteristics: (i) all of them can be exploited, in principle, to support developers during coding tasks (*e.g.,* by automatically identifying code-comment inconsistencies or, as we did, by recommending the next method to implement); (ii) while there are a few studies on these code change patterns, most of them have been performed on a small scale, not taking full advantage of recent MSR techniques; (iii) they can all be investigated by looking at the same data source (*i.e.,* historical data in open source repositories), making it easier to integrate, in future, techniques built on top of them.

Also, these three code change patterns cover the four major activities in developers' commits [17]: forward engineering (*e.g.,* implementing a new feature) as a development activity; and reengineering (*e.g.,* refactoring the code), corrective engineering (*e.g.,* fixing an issue) and management (*e.g.,* updating documentation) as maintenance activities. For instance, the introduction of code-comment inconsistencies often result from a reengineering activity and the fixing of those inconsistencies always represent a management activity. Thus, in the most ideal situation, a highly integrated technique can be developed to support practitioners on different stages during the software development process. In addition, the empirical knowledge we gained from this thesis can provide a hint for researchers to explore other unexposed code change patterns (*e.g.,* see Section 4.2.2).

1.2 Research Contributions

The contributions of our research can be grouped in two high-level categories: i) Mining change patterns to build new empirical knowledge about development and maintenance activities in open source systems, and ii) Developing techniques and tools leveraging mined patterns to support developers in code-related tasks.

1.2.1 Mining change patterns to build new empirical knowledge about development and maintenance activities performed in open source systems

Introductions and Fixes of Code-Comment Inconsistencies

Any code-related activity lays its foundations in program comprehension: before fixing a bug, refactoring a class, or writing new tests, developers first need to acquire knowledge about the involved code components. While code comments usually plays an important role to comprehend the source code, developers do not always have the chance to carefully comment new code and/or to update comments as consequence of code changes. This latter scenario might result in the introduction of code-comment inconsistencies, manifesting when the source code does not co-evolve with the related comments. To raise the knowledge about code-comment inconsistencies, we performed a large-scale empirical study on the co-evolution of code and comments. We mined the complete change history of 1,500 Java projects and, for each commit, we captured fine-grained changes performed in code (*e.g.,* change of a *selection* statement) as well as update, delete, and insert operations performed in the related comments. Overall, this process resulted in a database of ∼476 GB containing ∼1.3 Billion AST-level operations impacting code or comments. Using this data, we studied the extent to which code changes impacting different code constructs (*e.g., literals*, *iteration statements*) trigger the update of the related code comments (*e.g.,* the developer adds a try statement and updates the method comment to "document" the changed code behavior). Then, we manually analyzed 500 commits related to the fixing of code-comment inconsistencies. The output of this analysis is a taxonomy of code comment-related changes implemented by developers, from which we present relevant cases related to code-comment inconsistencies, and discuss implications for researchers and practitioners. Our investigation resulted in the following publication:

> **A Large-Scale Empirical Study on Code-Comment Inconsistencies**
> Fengcai Wen, Csaba Nagy, Gabriele Bavota and Michele Lanza. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC 2019) – Technical Research*, pp. 53-64, 2019

Omitted Code Changes in Developers' Commits

During software development and maintenance, a single cohesive change (*e.g.,* a bug fix) could be split across several commits. This can be due to omitted code changes and/or the need for fixing a mistake in the first attempt to implement the change. Park *et al.* [18] showed that 22% to 33% of bugs require more than one fix attempt (*i.e.,* supplementary patches). Also, studying supplementary patches can be instrumental in designing recommender systems able to reduce omission errors by alerting developers. In a subsequent work by Park *et al.* [19], the authors tried to predict additional change locations for real-world omission errors. Due to the limited empirical evidence about the nature of omitted changes, this is still an open challenge. Indeed, while the work by Park *et al.* [18] investigates omitted changes, it explicitly focuses on supplementary patches for bug-fixing activities, ignoring other types of code changes (*e.g.,* implementation of new features, refactoring).

To understand the types of omitted code change in software repositories, we performed

a qualitative study focusing on "*quick remedy commits*" performed by developers. We define as *quick remedy commits* those commits that (i) *quickly* succeed a commit performed by the same developer in the same repository; and (ii) aim at *remedying* to issues introduced as the result of code changes omitted in the previous commit (*e.g.,* fix references to code components that have been broken as a consequence of a rename refactoring) and/or of introduced errors. We decided to focus on remedy commits that are temporally close to the original change they fix for two reasons. First, it is easier to establish a clear link between two commits by the same developer if they are performed within a few minutes one from the other. Second, it is challenging to prevent omission errors automatically; thus, we decided to focus on omission errors that, since fixed within few minutes, are likely not so complex. This allows gathering empirical knowledge to take a first step in automating the prevention of a basic set of omission errors that, as we show, can be responsible for bugs and major code inconsistencies if not promptly fixed. We manually analyzed 500 *quick remedy commits* to identify the rationale behind them and to define a taxonomy categorizing the types of issues introduced by developers during commit activities that trigger a remedy commit. Our investigation resulted in the following publication, awarded with an ACM Distinguished Paper Award:

> **An Empirical Study of Quick Remedy Commits**
>
> Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC 2020) – Technical Research*, pp. 60-71, 2020 (ACM Distinguished Paper Award)

In a followup study, we investigated the impact of quick remedy commits in MSR studies. We assume that, depending on the specific research questions an MSR study wants to answer, different choices must be taken about whether quick remedy commits should be considered separately from the original commit they are fixing. For example, in studies aimed at characterizing the expertise of software developers by analyzing their past code changes, the two commits should be likely considered as a single change. Instead, for studies looking for bug-introducing commits, the two commits could be considered separately. In this thesis, we particularly focus on investigating the impact of one specific type of quick remedy commits (*reverted commits*) on two frequently performed "data collection tasks" in MSR studies (identifying bug-fixing commits and mining refactoring operations). The results show the potential impact that these commits have on the data collection in MSR studies. The results of this work has been accepted by the Springer Journal of Empirical Software Engineering as an extension of the previous publication:

> **Quick Remedy Commits and Their Impact on Mining Software Repositories**
>
> Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. In *Springer Journal of Empirical Software Engineering 27, 14 (2022)*,

Implementation Patterns Followed by Developers when Implementing a New Feature

Code completion techniques assist developers in speeding up the implementation of new code. However, they are only able to recommend the next few code token(s) developers are

likely to write. One goal of our research is to collect knowledge about implementation patterns followed by developers at a more abstract granularity level (*e.g.,* which are the methods usually implemented together?). We started by building a knowledge base of method-level implementation patterns followed by Android developers. We mined from commits performed in the change history of open source Android apps, the set of new methods created in each commit. Using this information, we identified implementation patterns repeatedly followed by Android developers, *e.g.,* the implementation of m_1 could imply the implementation of $m_2, \ldots, m_n$, and exploit them to build a recommender system pushing automated coding beyond the capabilities of modern code completion.

1.2.2 Leverage mined patterns to develop techniques and tools supporting developers in code-related tasks

Recommending the Next Full Method to Implement in a Given Context

Using the knowledge base previously described, we can identify implementation patterns repeatedly adopted by Android developers. However, the identification of these implementation patterns is far from trivial. Indeed, two commits c_k and c_j performed in two different repositories may implement different sets of new methods (*e.g.,* $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that, however, represent the same implementation pattern (*i.e.,* $m_1 = m_3$ and $m_2 = m_4$). Recognizing this situation requires to identify groups of methods that are repeatedly implemented together in different commits/apps, and not just by chance in a single/few commit(s).

By combining MSR with clustering techniques and static code analysis, we built FeaRS, an approach and an IDE plugin that monitors the code written by Android developers in the IDE. It recommends the complete code of the next method (*i.e.,* signature and method body) they are likely to implement based on method(s) they have already implemented. The performed empirical evaluation showed that FeaRS can successfully generate useful recommendations with a precision of 72% in a simulated real usage scenario, but also highlights some future challenges. This work has been published on ICSE 2021:

> **Siri, Write the Next Method**
> Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza and Gabriele Bavota. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021) – Technical Track*, pp. 138-149, 2021

We also present FeaRS as a tool demonstration at ICSME 2021 focusing on the engineering part of building FeaRS and release FeaRS as an open source project:

> **FeaRS: Recommending Complete Android Method Implementations**
> Fengcai Wen, Valentina Ferrari, Emad Aghajani, Csaba Nagy, Michele Lanza and Gabriele Bavota. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME 2021) – Tool Demo Track*, To appear

1.3 Outline

This dissertation is structured in the following chapters:

Chapter 2 presents an overview of the state of the art, including the most relevant literature according to the three sub-topics covered in this thesis: (i) studying code-comment inconsistencies, (ii) investigating quick remedy commits and their impact on mining software repositories, and (iii) using code change patterns for code recommendations.

Chapter 3 describes our investigation in the co-evolution of code and comments in a large-scale setting. This chapter also presents our qualitative analysis on the fixing of code-comment inconsistencies in the studied systems.

Chapter 4 presents our investigation on omitted code changes in software repositories. To be more specific, we defined and studied "*quick remedy commits*" which are related to the introduction and fix of simple omission errors.

Chapter 5 presents our approach for a source code recommender system that can take existing code written by developers as input, and recommends the next full method they are likely to implement.

Chapter 6 concludes this dissertation by summarizing our work and indicating future research directions based on the results we achieved.

2

State of the Art

2.1 Introduction

With the rapid growing of MSR research field, researchers have conducted many studies mining and leveraging change patterns from open source repositories to build empirical knowledge and support developers in coding tasks. In this chapter, we introduce the most relevant literature according to the three sub-topics covered in this thesis: (i) studying code-comment inconsistencies, (ii) investigating quick remedy commits and their impact on mining software repositories, and (iii) using code change patterns for code recommendations. Our goal is to broaden the empirical knowledge in these three directions and move one step forward to support developers leveraging the acquired knowledge, by quantitatively and qualitatively mining open source repositories in a larger scale at the same time.

2.2 Empirical Studies on Developers' Commits

There is a vast literature of empirical studies investigating developers' commits for various purposes: *e.g.,* summarizing the reasons for code changes, investigating the effects of a change on code quality, analyzing the relation between code changes and time, and identifying code change patterns. Meanwhile, considering the validity of those MSR studies, researchers also have realized that some commits can be meaningless or deceptive which might introduce bias and noise in mining change histories. In this section, we present an overview of the related work that close to the topic of our study described in Chapter 4. Compared to the literature we are going to discuss, our study defines a new type of commits (*i.e., quick remedy commits*) where developers claimed to fix or improve a change recently committed. We build a taxonomy trying to understand the rational behind this specific type of commits and their possible impact on code quality. We also look into the possible bias and noise introduced by this type of commits while performing some data collection tasks in MSR studies.

2.2.1 Reasons for Changes

Mockus *et al.* [20] studied a large legacy telecommunication system to identify reasons for software changes. Using an automatic classification algorithm, they discovered three primary reasons for changes according to maintenance activities: adding new functionality (*adaptive*), repairing faults (*corrective*), and restructuring the code to accommodate future changes (*perfective*). Besides, they noticed that several changes fall under the fourth category of inspection rework changes, *i.e.,* changes to implement the recommendations of code inspections. They also found a strong relationship between the type and size of a change and the difficulty of a change.

Hattori and Lanza [17] conducted an empirical study on nine large open source systems. They defined the size of a commit based on the number of involved files. They classified commits according to the comments information into development (forward engineering) or maintenance (reengineering, corrective engineering, and management) categories.

Hindle *et al.* [4] conducted a study on large commits and created a taxonomy of the purpose of large commits. They also found that large commits are more focused on perfective maintenance, while small commits are more related to corrective maintenance.

2.2.2 Effects of a Change on Code Quality

Small Changes. Purushothaman and Perry [21] investigated small source code changes (*i.e.,* one-line changes) during the development process. An interesting finding of their work is that there is less than a four percent probability that a one-line change introduces a fault in the code.

Large Changes. Sliwerski *et al.* [22] studied fix-inducing changes (*i.e.,* changes that lead to problems indicated by fixes) in Eclipse and Mozilla. They located bug-fix commits by extracting information from bug report, and determined the earlier changes at the same location as fix-inducing commits. Then they investigated the size of those commits (*i.e.,* number of files touched) and found out that large commits have higher chances of introducing bugs.

Social Characteristics. Eyolfson *et al.* [23] investigated the bug-fix time as the time from the earliest commit that introduced the bug to the bug-fixing commit. Their findings suggest that the time and date of a code update may affect the quality of the code. Sliwerski *et al.* [22] also investigated the relation between the quality of code changes and the day of week they were applied. The results show that commits performed on Friday are more likely to lead to problems.

In an earlier study, Claes *et al.* [24] also studied developers' working hours by investigating the timestamps of commit activities. They found that developers mainly work in regular office hours, and they did not find support that project maturation would decrease irregular working hours.

Bird *et al.* [25] mined commits in Windows Vista and Windows 7 to investigate the relationship between code ownership and software quality. They found that high levels of ownership, specifically high values for the proportion of ownership for the top owners, or high values for major, and low values of minor contributors, are associated with fewer defects.

Rahman *et al.* [26] found that implicated code is more closely related to the contribution of a single developer. Their findings also indicate that an author's specialized experience in the target file is more important than general experience.

Gonzales-Barahona *et al.* [27] investigated in FLOSS projects from the Mozilla community whether contributors fixing a bug are the same introducing and seeding them in the first place. Their results show that in 80% of the cases, the bug-fixing activity involves source code modified by at most two developers. Hence, in most of the cases, the bug fixing process is not carried out by the same developers.

Supplementary Patches. Park *et al.* [18] studied bugs whose initial patches were later considered incomplete and to which programmers applied supplementary patches. They examined three open source projects: Eclipse JDT core, Eclipse SWT, and Mozilla. They found that a significant portion of bugs fall in this category while their causes are often diverse, *e.g.,* missed port changes, incorrect handling of conditional statements, or incomplete refactoring. In their follow-up work [19, 28] they further investigated supplementary patches, and the results showed that only 7 % to 17 % of supplementary patches had content similar to their initial patches, which implies that a separate code clone analysis could not predict the supplementary patch location.

An *et al.* [29] found that supplementary bug fixes accounted for 10.3% to 26.9% of total bug reports. Also, in the subject systems, a high percentage of the supplementary fixes (*i.e.,* from 21.6% to 33.8%) had been re-opened.

Consecutive Changes. Dai *et al.* [30] investigated the relationship between consecutive changes and software quality. They introduced two novel concepts of consecutive changes: chain of consecutive bug-fixing file versions, and chain of consecutive file versions where each pair of adjacent versions has different authors. They found that those consecutive changes have a negative impact on the later file versions in the short term, especially when the length of the change chain is four or five.

Inconsistent Changes. Bettenburg *et al.* [31] conducted an empirical study on inconsistent changes to code clones in two large open source software systems. They observed that the number of defects caused by inconsistent changes to code clones was substantially lower at the release level, compared to the revision level. Their findings suggest that developers can effectively manage and control the evolution of cloned code at the release level.

Incorrect Changes. Yin *et al.* [32] presented a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases, including Linux, OpenSolaris, and FreeBSD. They found that at least 14.8%-24.4% of sampled fixes for post-release bugs in these large operating systems were incorrect.

Changes and Refactoring. Palomba *et al.* [33] conducted a quantitative investigation of the relationship between different types of code changes and different refactoring types. They found that developers tend to apply a higher number of refactoring operations when they are fixing bugs.

Bavota *et al.* [34] presented a study aimed at investigating to what extent refactoring activities induce faults. They showed that refactorings involving hierarchies (*e.g.,pull down method*) induce faults very frequently. Conversely, other kinds of refactorings are likely to be harmless in practice.

Differently from the above-discussed studies, we focus on investigating what types of omission errors performed by developers can be quickly spotted and fixed, and their possible impact on code quality.

2.2.3 Changes and Time

Rodriguez-Perez *et al.* [35] conducted two case studies and studied the *Time To Notify* (TNN) metric which describes how much time it takes for a bug to be notified/reported since the bug was introduced into the source code. They examined how this metric is related to software maintenance and evolution. Interestingly, they found relatively high mean values of TTN in the projects: 312 and 431 days.

Kim *et al.* [36] studied the bug-fix time of files in ArgoUML and PostgreSQL. Their statistics showed that fixing 50% of the bugs requires 100 to 300 days, while the median bug-fix time is about 200 days.

2.2.4 Change Patterns

Pan *et al.* [37] presented an automatic approach in which software history data is mined to find patterns in bug fix changes and automatically categorize bugs. They defined bug fix patterns (*e.g.,* method call with different actual parameter values) which covered 45-63 % of bug fixes in seven open source projects.

Zhao *et al.* [38] conducted an empirical study to investigate the characteristics of change types in bug fixing code. They proposed a change classification schema and developed an automatic classification tool to categorize changes into five change types. They found that interface related code changes are the most frequent bug-fixing changes.

In a related research thread, Martinez and Monperrus [39] presented Coming, a tool to mine change pattern (*i.e.,* a set of changes between two revisions and the elements affected by those changes) instances from git commits. The main purpose of this tool is to help a researcher to (i) filter the revisions of a studied system that are relevant for his research, and (ii) captures the commits that introduce a set of particular code changes (*e.g.,* adding if-return statements).

Change patterns have also been exploited recently to train neural networks in order to automatically reproduce code changes implemented by developers in pull requests of open source projects [40] or to learn how to automatically fix bugs [41].

2.2.5 Bias and Noise in Mining Change Histories

Many approaches and studies depend on the quality of the dataset produced by mining change histories. Discussions about the bias in data collected by mining repositories have gained more attention recently. As Bird *et al.* [3] say in a study on bias in bug-fix datasets: "*bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalizability of hypotheses tested on biased data*". Here, we overview the potential causes and impact of bias and noise in mining studies.

Impact of Non-Essential Changes. Kawrykow and Robillard [42] observed that software changes are often accompanied by non-essential modifications, such as local variable refactorings, or textual differences induced as part of a rename refactoring. They studied code changes in over 24,000 changesets of seven open-source systems and observed non-essential changes in their history. They found that up to 15.5% of a system's method updates were due to non-essential differences among interesting observations.

The authors also investigated the impact of non-essential changes on change-based analyses in their same research work [42]. They implement a method-pair association rule mining analysis similar to the approach of Zimmermann *et al.* [43]. This approach, given a set of changes, suggests and predicts likely further changes. They found that removing non-essential method updates improved the precision of the recommendations by 10.5% and decreased their recall by 4.2%.

Impact of Tangled Changes. Herzig *et al.* [44] defined a tangle change as a single commit which consists of separate changes (*e.g.,* fixing a bug and adding a new feature). They found that up to 15% of all bug fixes include tangled changes.

Later, they also showed that tangled changes could significantly impact the accuracy of defect prediction models assessed in empirical studies [45].

Impact of Untracked Changes. Hora *et al.* [46] claimed that changes affecting code entities' names (untracked changes) present a potential threat to MSR studies. For example, a method rename could be misinterpreted as the deletion and the addition of a method, thus, splitting its history. Based on an empirical analysis of 15 Java systems, they found that between 10 and 21% of the method level changes are untracked, hence, should be systematically considered by MSR studies.

Bias in Bug Localization and Prediction. Kochhar *et al.* studied biases in bug localization [47]. They identified potential causes that can impact the validity of the results reported in studies. One of the main reasons is that files modified in commits that fix the bugs might not contain the bug. Instead, files are often changed because of refactorings or modifications to program comments.

Kim *et al.* [48] measured the impact of noise on defect prediction models built using historical defect data obtained by mining software repositories. They consider false positives and false negatives as noise in such dataset. They found that, for large defect datasets, noises alone do not lead to substantial performance differences. However, their prediction performance decreased significantly when the dataset contained 20%-35% of both FP and FN noises.

Rahman *et al.* [49] assessed whether the size of the dataset or bias affects the performance of defect prediction approaches. Similar to the findings of Kim *et al.* [48], they conclude that size matters at least as much as bias.

Noise in History Slicing. Li *et al.* [50] presented a semantic history slicing approach to extract changes related to a particular functionality. As they say, state-of-the-art techniques tend to over-approximate the inferred changes, and their slice histories may contain irrelevant changes. Their approach implements a method to untangle unrelated changes introduced in a single commit.

Threats in Aggregating Software Repository Data. Robillard *et al.* [51] investigated po-

tential threats to validity associated with metrics that summarize software repository data. They conducted a case study in which they retrieved and analyzed every file considered abandoned to investigate the files' properties, including size, file type, and amount of comments. As a result, they identified eight major threats that can generalize to software process metrics derived from repository data. These threats are fragility, file content, file role, comment, contributor involvement, quantization, architectural sensitivity, and exceptional action.

Apart from these studies, our research also investigates the possible bias and noise introduced by one type of *quick remedy commits* while mining the change history of open source repositories.

2.2.6 Summing Up

As discussed above, previous work investigated code changes from several different points of view. However, to the best of our knowledge, the study presented in Chapter 4 is the first shedding some light on the phenomenon of *quick remedy commits*. Indeed, while previous studies looked at supplementary bug-fixes [18, 19, 28], their focus was limited to bug-fixing activities, while we looked at remedy commits from a broader perspective. For this reason, our work complements previous findings reported in the literature. Also, we complement studies related to bias and noise in MSR studies by investigating the role played by *reverted commits* (*i.e.,* a specific type of *quick remedy commits* in which changes from a previous commit are undone) on data collected for MSR studies.

2.3 Introduction and Fix of Code-Comment Inconsistencies

In Chapter 3 we study code-comment inconsistencies to understand how code and comments co-evolve, to identify coding activities triggering/not-triggering the introduction of code-comment inconsistencies, and to investigate the types of inconsistencies fixed by developers. In this section, we discuss related work concerning (i) empirical studies on code comments, and (ii) approaches for the detection of code-comment inconsistencies. Compared to these works, we present the largest empirical study to date on code-comment evolution and inconsistencies. As a result, our research provided an extensive database and a taxonomy which already enabled other works on detecting code-comment inconsistencies [52, 53].

2.3.1 Empirical Studies on Code Comments

Woodfield *et al.* [54] conducted a user study with 48 programmers and showed that commented code is better understood by developers as compared to non-commented code.

Ying *et al.* [55] analyzed the usage of code comments in the IBM internal codebase. They show that comments are not only a means to document the source code, but also a communication channel towards colleagues, *e.g.,* to assign tasks and keep tracks of ongoing coding activities. Their work also showed that comments are very challenging to analyze automatically because they have ambiguous context and scope.

Arnaoudova *et al.* [56] analyzed how developers perceive linguistic antipatterns (LAs), *e.g.,* poor software documentation practices that result in code-comment inconsistencies. Aghajani *et al.* [57] showed that LAs lead to a 29% higher chance of introducing bugs, highlighting that outdated code comments have a negative impact on code quality.

McBurney and McMillan [58] compared code summaries written by code authors and readers (*i.e.,* non-authors performing code understanding). They used the Short Text Semantic Similarity (STSS) metric to assess the similarity between source code and summaries written by the authors and compared it to the similarity between the code and the summaries written by the readers. They found that readers rely more on source code than authors when summarizing the code.

Padioleau *et al.* [59] proposed a taxonomy based on meanings of comments and manually classified 1,050 comments. They found 52.6% of these comments can be leveraged to improve software reliability and increase programmer productivity. Pascarella and Bacchelli [6] presented a hierarchical taxonomy of types of code comments for Java projects and experimented with automatically classifying code comments. Such a taxonomy, composed of six top categories and 16 inner categories, was built by manually analyzing 2,000 code comments. Later, Pascarella [60] conducted a similar study in the context of Java mobile applications, and the result shows only a marginal difference exists between desktop and mobile apps. The taxonomy presented later in our study, differently from the one in [6, 59], aims at classifying the types of code-comment inconsistencies fixed by software developers.

Other authors studied the evolution of code comments. Jiang and Hassan [61] conducted a study on the evolution of comments in PostgreSQL. They investigated the trend over time of the percentage of commented functions in PostgreSQL. Their results reveal that the proportion of commented functions remains constant over time.

Arafat *et al.* [62] studied the density of comments (*i.e.,* the number of comment lines divided by the number of code lines) in the history of 5,229 open source projects written in different programming languages. They show that the average comment density depends on the programming language (with the highest one of 25% measured for Java systems), while it is not impacted by the project and team size.

Ibrahim *et al.* [63] investigated the relationship between comment update practices and software bugs introduction in three open-source systems. Their findings show that abnormal comment update behavior (*e.g.,* missing to update a comment in a subsystem whose comments are always updated) leads to a higher probability of introducing bugs.

Linares-Vasquez *et al.* [64] looked at how developers described database usages in method comments and found that database-related method comments are far less frequently updated than the source code.

Fluri *et al.* [65] investigated how comments and source code co-evolved over time in three open source systems. They observed that 23%, 52%, and 43% of all comment changes in ArgoUML, Azureus, and JDT Core respectively, were due to source code changes, and in 97% of these cases the comment changes occurred in the same revision as the associated code change. However, newly added code barely got commented.

In a follow-up work, Fluri *et al.* [66] investigated the co-evolution between code and comment in eight systems. They found that the ratio between the growth of code and com-

ments is constant but confirmed the previous observation about the frequent lack of comment updates for newly added code. They also found that: (i) the type of code entity impacts its likelihood of being commented (*e.g.,* `if` statements are commented more often than other types of statements), (ii) 90% of comment changes represent a simultaneous co-evolution with code (*i.e.,* they change in the same revision), and (iii) surprisingly API comments are often adapted retroactively.

2.3.2 Automatic Assessment of Comments Quality

Researchers have developed tools and metrics to capture the quality of code comments. Khamis *et al.* [67] developed JavadocMiner, an approach to assess the quality of Javadoc comments. JavadocMiner exploits Natural Language Processing (NLP) to evaluate the "quality" of the language used in the comment as well as its consistency with the source code. The quality of the language is assessed using several heuristics (*e.g.,* checking whether the comment uses well-formed sentences including nouns and verbs) combined with readability metrics such as the Gunning Fog Index. The consistency between code and comments is also checked with a heuristic-based approach, *e.g.,* a method having a return type and parameters is expected to have these elements documented in the Javadoc with the `@return` and `@param` tags.

Steidl *et al.* [68] also proposed an approach for the automatic assessment of comments' quality. First, their approach uses machine learning to classify the "type" of code comment (*e.g.,* copyright comment, header comment). Second, a quality model is defined to assess the comments' quality. Also in this case, the model is based on a number of heuristics (*e.g.,* the coherence of the vocabulary used in code and comments). On a similar line of research, Scalabrino *et al.* [69] used the semantic (textual) consistency between source code and comments to assess code readability, conjecturing that the higher this consistency, the higher the readability of the commented code.

Other authors explicitly focused on the automatic detection of code-comment inconsistencies. Seminal in this area are the works by Tan *et al.* [70, 71]. First, they presented iComment [70], a technique using NLP, machine learning, and program analysis to detect code-comment inconsistencies. iComment is able to detect inconsistencies related to the usage of locking mechanisms in code and their description in comments. This technique was evaluated on four systems (Linux, Mozilla, Wine, and Apache) showing its ability to identify inconsistencies confirmed by the original developers.

In a follow-up work, Tan *et al.* [71] also presented @TCOMMENT, an approach able to test the consistency between Javadoc comments related to `null` values and exceptions with the behavior of the related method's body. @TCOMMENT has been experimented on seven open source projects, identifying inconsistencies confirmed by developers.

Similarly, Zhou *et al.* [72] devised a first-order logic-based approach detecting inconsistencies related to parameter constraints and exceptions in API documentation. The approach was able to detect 1,146 defective document directives with a ∼80% precision.

A rule-based approach named Fraco was proposed by Ratol *et al.* [73] to detect code-comment inconsistencies resulting from rename refactoring operations performed on iden-

tifiers. Their evaluation shows the superior performance ensured by FRACO as compared to the rename refactoring support implemented in Eclipse.

Recent works also started investigating on detecting just-in-time inconsistencies instead of pre-existing inconsistencies. Liu *et al.* [74] analyzed historical versions of existing projects to train a machine learner able to identify comments that need to be updated during code changes. The approach uses a random forest classifier with 64 hand-engineered features capturing, for example, the *diff* of the implemented changes, to automatically detect outdated comments. The authors report a ∼75% detection precision for their approach.

Stulova *et al.* [52] present a preliminary investigation of a technique that uses BOW-based similarity metrics to map a comment to the AST nodes of the method signature (before the code changes). This mapping is used to check whether the code changes have triggered an inconsistency in the corresponding comments.

Liu *et al.* [53] propose a novel approach, namely CUP, to not only detect just-in-time inconsistent comments, but also automatically update them. CUP is based on a neural sequence-to-sequence (seq2seq) model learning comment update patterns from historical data. Their experiments show that CUP achieves significant improvements over three baselines on just-in-time comment updating.

Finally, a related research thread is the one presenting techniques to detect self-admitted technical debt (SATD) in code comments [75, 76, 77, 78, 79]. These techniques, while not directly related to the quality of code comments, use these latter to make the development team aware of SATD.

2.3.3 Summing Up

While seminal work investigating the co-evolution of code and comments [61, 63, 65, 66] limited their analyses to the change history of a few software systems (less than 10), our study presented in Chapter 3 is performed on a much larger scale, involving the change history of 1,500 projects. Also, we complement this quantitative analysis with a manually defined taxonomy of code-comment inconsistencies fixed by developers. While our work is not related to the automatic assessment of comments' quality, it still provides empirical knowledge useful to devise novel approaches for the detection of "problematic" code comments (*e.g.,* the dataset built as output of our study has been already exploited in works on detecting inconsistent comments [52, 53]).

2.4 Source Code Recommender Systems

One of the contributions of this thesis is FeaRS, an approach that recommends to developer the next method to write (see Chapter 5). FeaRS is one of the many recommender systems proposed in the software engineering literature. The latter have been proposed to support many different tasks such as, the recommendation of formal and informal documentation (see *e.g.,* [80, 81, 82]), the automatic generation of code for different purposes (*e.g.,* [40, 41, 83, 84, 85, 86]), or the recommendation of relevant code examples/discussions for a task at

hand (*e.g.,* [87, 88, 89, 90, 91]). We focus our discussion on the most related works, and in particular on those dealing with code completion techniques and code search engines.

2.4.1 Code Completion Techniques

The first attempts to support code completion in IDEs mostly relied on the static type system of a programming language and did not consider other actual code context: Suggestions were usually sorted in alphabetical order. As a result, relevant recommendations were not always easy to identify.

An alternative approach was presented by Bruch *et al.* [9]. Their *intelligent code completion system* provides context-sensitive recommendations for method calls against object instances of a given framework. Their approach is based on a variant of the machine learning K-nearest neighbors algorithm, called Best Matching Neighbors (BMN). The context of variables is extracted and variables used in similar circumstances are searched in an example code base, followed by method suggestions synthesized from these closest clippings.

Proksch *et al.* [92] proposed a Bayesian network for code completion based on similar ideas. Method call completion was also explored by Asaduzzaman *et al.* [93]. Their approach, called CSCC (Context-sensitive code completion), relies on a database of method call usage contexts collected from open source projects and applies a hash function to find relevant recommendations.

Another context-aware approach called GraccPacc was developed by Nguyen *et al.* [11], which is a graph-based, pattern-oriented code completion method based on a database of API usage patterns. *GraPacc* uses graphs to model API usage patterns, where nodes represent actions (*e.g.,* method calls) and control points (*e.g.,* while), and edges represent control and data flow dependencies between nodes. It also collects context-sensitive characteristics (*e.g.,* the relation between API elements and other code elements) from the code being modified and utilizes them to find and rank the patterns that best fit it. When a pattern is chosen, the graph-based code technique is used to finish the present code.

Robbes and Lanza used information extracted from the change history to improve code completion of method calls and class names: Whenever a method is modified, this method and all the references to other methods and classes in its body are more likely to be reused by developers. Their evaluation on different code completion strategies shows that giving the priority to the matches with most recent change date can outperform the alphabetical and structure-aware ordering. Their tool is able to propose a correct match in the top-3 results in 75% of cases [13].

Machine learning-based language models have also been used for code completion. The main idea is to avoid extracting hand-coded features and rely on structural representations of code (*e.g.,* lexemes, ASTs, dataflow) and learn how to perform code prediction using these representations. In their seminal work on the naturalness of software, Hindle *et al.* [10] developed a code completion engine for Java based on an n-gram language model. Their work has been extended by Nguyen *et al.* [7] who performed a large-scale study on the repetitiveness of source code at the routine level (*i.e.,* a portion of code that performs a specific task independently). They found that 12.1% of the routines are repeated within a project, and

14.3% of the routines have all of their subroutines repeated, which provide implications to code completion tools for better recommendations. In addition to the naturalness of source code, Tu *et al.* [12] also noticed that source code tends to have a localness property, *i.e.,* tokens tend to be repeated within specific areas of source code.

Another language model approach, named SLANG, has been implemented by Raychev *et al.* [14]. SLANG is a Java code completion tool that generates full method (API) call sequences, including the parameters of each call. They extract sequences of method calls from a large codebase to train a model able to take as input code snippets lacking API calls and can suggest which APIs to invoke. Their approach achieves an accuracy of 90% when considering the top three results.

In the most recently proposed code recommender systems, deep learning models have been more and more exploited, thanks to their ability of learning coding patterns from a large amount of data. Karampatsis *et al.* [94] suggested that neural networks are the best language-agnostic algorithm for code completion, showing its superiority compared to the state-of-the-art language model [95]. Starting from this work, several researchers have leveraged the latest Transformer-based models [96] to create code recommender systems (see *e.g.,* [97, 98, 99]) which outperform previous approaches. The recently proposed GitHub Copilot tool [100] builds on top of this literature.

Popular IDEs have also recognized the importance of supporting context-sensitive recommendations. For example, IntelliJ IDEA has a feature called *Smart completion* to filter and show suggestions applicable to the current context. NetBeans has a *Smart Code Completion* feature to display at the top of the suggestions the most relevant ones for the context. Eclipse has plugins to extend its core code completion, among these, *aiX Code Completer*[101] and *Codota*[102] use AI techniques and can even recommend a full line of code.

2.4.2 Code Search Engines

FeaRS is also related to approaches implementing code search engines that allow retrieving code samples and reusable open source code from the Web. Indeed, FeaRS explores the history of thousands of open source repositories and generate a pre-processed database containing reusable, representative code samples which will be recommended to developers when specific rules are triggered.

Umarji *et al.* [103] performed a web-based survey to better understand why programmers search for code. They classified code search objectives along two orthogonal dimensions: motive (reuse *vs.* reference example) and size of search target. They observed that developers search for code components at different granularity levels (*e.g.,* from a single line of code to a subsystem, to libraries). The majority of queries are performed to look for code examples, finding a library implementing specific features, or learning how to use an API.

Early online code search engines (*e.g.,* codesearch.google.com, koders.com, and krugle.org) offered keyword-based search and file-level retrieval. These approaches could be improved by considering structural and semantic information of code.

Bajracharya *et al.* [104] developed Sourcerer, a code search engine that extracts structural information from the code and stores it in a relational model so it can be queried for

code search. It supports queries for control structures, Java types, and micro patterns (*e.g.,* implementation of Semaphore). Reiss developed an approach to combine code search with transformations to map the retrieved code, to meet user specifications [105]. For the searching, it allows the user to specify multiple semantic rules, which also form the basis for the transformations.

RACS [50] generates an action connection graph from query data and depicts API use patterns as method dependency graphs from the gathered code snippets. In this manner, the issue of identifying comparable method dependency graphs for a given action connection graph is simplified to the problem of code search. The findings have been used in query formulation by CodeExchange and CodeLikeThis. The former uses a set of criteria to find comparable results for new queries, whereas the latter looks directly for results that are similar to existing results [106].

Thummalapenta *et al.* developed an approach to support code search engines with static analysis to return fewer, but more relevant code samples for search queries [107, 108]. Their primary goal was to support a user in reusing a given API. Later they extend their approach with SpotWeb [109] to assist users by detecting hotspots that can serve as starting points for reusing APIs.

API usage was also proposed by McMillan *et al.* [110, 111] to return highly relevant matches for a source code search engine. Their approach combines three sources of information to locate relevant software: the textual descriptions of applications, the API calls used inside each application, and the dataflow among those API calls.

2.4.3 Summing Up

As discussed above, previous code completion techniques are undoubtedly valuable to speed up code writing. However, they are limited to recommendations related to the next few tokens the developer is likely to type given the current context. In the best case, they can recommend a few APIs that the developer is likely to use next. With FeaRS we forge another step ahead, to predict the next full method a developer is likely to implement. FeaRS also relies on an extensive database of methods' source code in open source applications, compared to code search engines. These methods are organized in clusters based on a similarity algorithm implemented in the ASIA clone detector [112]. In addition, FeaRS does not require the user to write a "query" to identify relevant pieces of code, but extrapolates this need by monitoring the IDE.

2.5 Conclusion

In this chapter we reviewed the state of the art focusing on three major lines of research related to our studies in this thesis, namely (i) Empirical studies on developers' commits, (ii) Introductions and fixes of code-comment inconsistencies, and (iii) source code recommender systems.

In Section 2.2, we discussed the related work in investigating developers' commits from different aspects. Many studies tackle when/where/why/how developers change their source

code. However, there has been little research on quick fixes, or consecutive changes performed by software developers, as well as on the impact of some specific types of commits in the data collection of MSR studies. In Chapter 4, we define a new type of commits (*i.e., quick remedy commits*) which usually represent a quick fix or a consecutive change associated to a previous code change. We also analyze when/where/why/how developers perform this type of commit and their possible impact on MSR studies.

In Section 2.3, we reviewed the state of the art related to code-comment inconsistencies. Some researchers studied the co-evolution of code and comments, while others proposed techniques and tools able to detect code-comment inconsistencies automatically. These techniques are able to identify specific types of code-comment inconsistencies (*e.g.,* inconsistencies introduced as result of rename refactoring operations). Still, more research is needed in this area to increase the types of code-comment inconsistencies that can be automatically identified. Also, the empirical evidence provided by studies that pioneered the investigation of code-comment evolution is limited to the analysis of the change history of a few software systems. In Chapter 3, we present the largest empirical study at date on code-comment evolution and inconsistencies, which has also contributed to further investigation on code-comment inconsistencies detection.

In Section 2.4, we presented an overview of related work on source code recommender systems, and we focused the discussion on code completion techniques and code search engines in particular. For several years, most of the effort in code completion development targeted the improvement of the recommendations in terms of accuracy (i.e., the ability to correctly predict the code tokens the developer is going to type). However, little progress has been made regarding the type of support these tools can provide to developers. Indeed, techniques and tools able to recommend more complex code elements such as entire statements or even functions have been proposed only very recently. In Chapter 5, we present FeaRS, an approach and a tool that can recommend the next full method to be implemented in a given context. Also, compared to typical code search engines, FeaRS relies on an extensive database of source code in open source repositories, and does not need a written “query” to trigger the recommendations.

3

Studying Code-Comment Inconsistencies

Code comments are a primary means to document source code. Keeping comments up-to-date during code change activities requires substantial time and attention. For this reason, researchers have proposed methods to detect code-comment inconsistencies (*i.e.,* comments that are not kept in sync with the code they document) and studies have been conducted to investigate this phenomenon. However, these studies were performed at a small scale, relying on quantitative analysis, thus limiting the empirical knowledge about code-comment inconsistencies.

In this chapter, we present the largest study at date investigating how code and comments co-evolve. The study has been performed by mining 1.3 Billion AST-level changes from the complete history of 1,500 systems. Moreover, we manually analyzed 500 commits to define a taxonomy of code-comment inconsistencies fixed by developers. Our analysis discloses the extent to which different types of code changes (*e.g.,* change of *selection* statements) trigger updates to the related comments, identifying cases in which code-comment inconsistencies are more likely to be introduced. The defined taxonomy categorizes the types of inconsistencies fixed by developers. Our results can guide the development of tools aimed at detecting and fixing code-comment inconsistencies.

3.1 Introduction

Any code-related activity lays its foundations in program comprehension: before fixing a bug, refactoring a class, or writing new tests, developers first need to acquire knowledge about the involved code components. As recently shown by Xia *et al.* [113], this results in 58% of developers' time spent comprehending code. Besides the code itself, code comments are considered as the most important form of documentation for program comprehension [114]. Indeed, not surprisingly, studies showed that commented code is easier to comprehend than uncommented code [54, 115]. This empirical evidence also pushed researchers to consider code comments as a pivotal factor to study technical debt [74, 75, 76], or to assess code quality [69, 116]. While the importance of code comments is undisputed, developers do not always have the chance to carefully comment new code and/or to update comments

as consequence of code changes [68]. This latter scenario might result in the introduction of code-comment inconsistencies, manifesting when the source code does not co-evolve with the related comments. For example, if a method comment is not updated after major changes to the method's application logic, the comment might provide misleading information to developers comprehending the method, hindering program comprehension rather than fostering it. Furthermore, recent studies have shown that code-comment inconsistencies lead a higher chance of introducing bugs, highlighting that outdated code comments have a negative impact on code quality [57].

To raise the knowledge about the co-evolution of code and comments and the introduction/fixing of code-comment inconsistencies, we present a large-scale empirical study quantitatively and qualitatively analyzing these phenomena. We mine the complete change history of 1,500 Java projects hosted on GitHub for a total of 3,323,198 analyzed commits. For each commit, we use GUMTREEDIFF [117] to extract AST operations performed on the files modified in it. In this way, we captured fine-grained changes performed in code (*e.g.,* change of a *selection* statement) as well as update, delete, and insert operations performed in the related comments. Overall, this process resulted in a database of ∼476 GB containing ∼1.3 Billion AST-level operations impacting code or comments. Using this data, we study the extent to which code changes impacting different code constructs (*e.g., literals*, *iteration statements*) trigger the update of the related code comments (*e.g.,* the developer adds a try statement and updates the method comment to "document" the changed code behavior).

Then, we manually analyze 500 commits identified, via a keywords-matching mechanism, as likely related to the fixing of code-comment inconsistencies. The output of this analysis is a taxonomy of code comment-related changes implemented by developers, from which we present relevant cases related to code-comment inconsistencies, and discuss implications for researchers and practitioners.

As a contribution to the research community, we make the database of fine-grained code changes publicly available [118]. This enables the replication of this work, making also other types of investigations possible.

Structure of the Chapter

Section 3.2 presents the design of the study we performed to how we study code-comment inconsistencies quantitatively and qualitatively. Section 3.3 reports the results of the study including a statistic analysis on fine-grained code-comment co-evolution and a taxonomy related to the introduction and fix of code-comment inconsistencies. Finally, after a discussion of threats that could affect the validity of our results(Section 3.4), Section 3.5 concludes this chapter.

3.2 Study Design

The *goal* of the study is to investigate code-comments inconsistencies from a quantitative and a qualitative perspective. The *purpose* is to (i) understand how code and comments

co-evolve, to identify coding activities triggering/not-triggering the introduction of code-comment inconsistencies; (ii) define a taxonomy of inconsistencies that developers tend to fix. The study addresses the following research questions (RQ):

RQ_1: *To what extent do different code change types trigger comment updates?* This RQ studies the code-comments co-evolution in open source projects. We investigate the extent to which different types of fine-grained code changes (*e.g., changes to selection statements*) trigger the update of the related code comments. This analysis provides empirical evidence useful to quantify the cases in which code-comment inconsistencies could possibly be introduced and to identify the types of code changes having a higher chance of introducing these inconsistencies. This evidence can be used, for example, to develop context-aware tools warning developers when code changes are likely to require code comments' updates.

RQ_2: *What types of code-comment inconsistencies are fixed by developers?* This research question aims at identifying the types of code-comment inconsistencies that are fixed by software developers *e.g.,* updating a comment as a consequence of a previously performed refactoring that renamed an identifier. Knowing the types of code-comment inconsistencies fixed by developers can guide the development of tools aimed at automatically detecting them.

3.2.1 Data Collection and Analysis

Table 3.1. Dataset Statistics

	Overall	**Per Project**		
		Mean	Median	Std. Dev.
Java files	1,599,323	1,068	360	2,838
Effective LOC	162,243,714	108,379	31,392	305,704
Stars	2,895,219	1,930	762	3,455
Commits analyzed	3,323,198	2,215	832	5,089

To answer RQ_1 we mine the fine-grained changes at AST (Abstract Syntax Tree) level performed in commits from the change history of 1,500 open source Java projects hosted on GitHub. Then, we analyze the extent to which different types of code changes trigger updates in the related code comments. The 1,500 projects representing the context of our study have been selected from GitHub in November 2018 using the following constraints:

Programming language. We only consider projects written in Java since, as it will be clear later, Java is the reference language for the infrastructure used in this study.

Change history. Since in RQ_1 we study the co-evolution of code and comments, we only focus on projects having a long change history, composed of at least 500 commits.

Popularity. The number of stars [119] of a repository is a proxy for its popularity on GitHub. Starring a repository allows GitHub users to express their appreciation for the project. Projects with less than ten stars are excluded from the dataset, to avoid the inclusion of likely irrelevant/toy projects.

6,563 projects satisfy these constraints. Then, we manually filtered out repositories that do not represent real software systems (*e.g.,* JAVA-DESIGN-PATTERNS [169] and SPRING-PETCLINIC [170]), and checked for projects with shared history (*i.e.,* forked projects). When we identified a set of forked projects, we only selected among them the one with the longest commit history (*e.g.,* both FINDBUGS [171] and its successor SPOTBUGS [172] fall under our search criteria, but we only kept the latter one). Finally, considering the high computational cost of the data extraction process needed for our study (details follow), we decided to only analyze a subset of the remaining projects: We sorted the projects in descending order based on their number of stars (*i.e.,* the most popular on top), and we selected from the list the top 1,500 projects for our study. Table 3.1 reports descriptive statistics for size, change history, and popularity of the selected projects. The complete list of considered projects is available in our replication package [118].

We cloned the 1,500 GitHub repositories and extracted the list of commits performed over the change history of each project. To do so, we iterated through the commit history related to all branches of each project with the `git log --topo-order` command. This allowed us to analyze all branches of a project, without intermixing their history and avoiding unwanted effects of merge commits. We then excluded commits unrelated to Java files (*i.e.,* commits that do not impact at least one Java file). For each remaining commit c_i, we use GumTreeDiff [117] with its JavaParser generator to extract AST operations performed on the files modified in c_i.

GumTreeDiff considers the following edit actions performed both on code and comment nodes: (i) *updatedValue* replaces the value of a node in the AST; (ii) *add/insert* inserts a new node in the AST; (iii) *delete*, which deletes a node in the AST; (iv) *move*, which moves an existing node in a different location in the AST. Also, to store more details of the changed AST nodes, such as their parent method and class (needed to know the code component to which a comment AST node belongs to), we extended GumTree with our own reporter. Overall, we extracted 1.3 Billion AST-level changes, resulting in a 476 GB database (excluding indexes) we make publicly available [118].

From our analysis we disregard any file added/deleted in c_i, since our primary goal is to study how changes to different types of code constructs trigger (or not) updates in code comments. In an added/deleted file, all code and comment AST nodes would trivially be added or deleted. Also, we work at method-level granularity, meaning that we only focus on code changes affecting the body or the signature of methods, discarding code changes impacting *e.g.,* a class attribute. This is done since it is easy, from the AST, to identify the comment related to a method (and, thus, to study how changes in the method impact the related comments) while it is not trivial to precisely link a class attribute to its related comments. Finally, we ignore the *move* actions detected by GumTreeDiff because we noticed that they generate a lot of noise in the data, since also deleting a blank line can result in an AST node move.

Once collected the list of AST operations performed in each commit on the code and comments of modified files, we classified the code changes into categories as shown in Table 3.2. The idea is to group together AST-level operations performed on related code constructs. For example, all operations performed on `if` and `switch` statements are grouped into the

Table 3.2. Categories of AST-level Code Changes

Category	GumTreeDiff Changes
Annotation	MarkerAnnotationExpr, MemberValuePair, NormalAnnotationExpr, SingleMemberAnnotationExpr
Array	ArrayAccessExpr, ArrayCreationExpr, ArrayCreationLevel, ArrayInitializerExpr
Casting	CastExpr, InstanceOfExpr
Constructor	ConstructorDeclaration
Empty Statement	EmptyStmt
Exception Handling	CatchClause, ThrowStmt, TryStmt
Expression	AssignExpr, BinaryExpr, ClassExpr, ConditionalExpr, EnclosedExpr, FieldAccessExpr, SuperExpr, ThisExpr, UnaryExpr
Iteration	BreakStmt, ContinueStmt, DoStmt, ForeachStmt, ForStmt, WhileStmt
Lambda Expression	LambdaExpr, MethodReferenceExpr
Literal	BooleanLiteralExpr, CharLiteralExpr, DoubleLiteralExpr, IntegerLiteralExpr, LongLiteralExpr, NullLiteralExpr, StringLiteralExpr
Method Invocation	ExplicitConstructorInvocationStmt, MethodCallExpr
Method Signature	MethodDeclaration, Parameter
Name	Name, SimpleName
Others	AssertStmt, BlockStmt, InitializerDeclaration, LabeledStmt, ObjectCreationExpr, SynchronizedStmt
Return	ReturnStmt
Selection	IfStmt, SwitchEntryStmt, SwitchStmt
Type	ArrayType, ClassOrInterfaceDeclaration, ClassOrInterfaceType, IntersectionType, LocalClassDeclarationStmt, PrimitiveType, TypeExpr, TypeParameter, UnionType, VoidType, WildcardType
Variable Declaration	VariableDeclarationExpr, VariableDeclarator

Selection category. Such a grouping is done for the sake of easing the RQ_1 data analysis. In particular, for each code change category CH_i in Table 3.2, we compute $MCC(CH_i)$ as the percentage of AST changes falling in the CH_i category that triggered a Method Comment Change in comments related to the impacted method. Using the AST, we classify as comments related to the method those present in the method body plus its Javadoc comment (if any). As "Comment Changes" we consider (i) the addition of a comment inside the method or of the Javadoc; (ii) modifications applied to any of the already existing method-related comments; and (iii) deletions of any of the existing method-related comments. Important to highlight is that, to better isolate the *triggering effect* of the CH_i type of change on the method's comments, we only consider CH_i's changes performed in **isolation** on a given method when computing $MCC(CH_i)$. Let us explain this design choice with an example: In a given commit two methods are modified, M_1 and M_2. Both methods are subject to AST changes belonging to the category CH_i, but M_2 is also affected by changes of type CH_j, with $i \neq j$. When computing $MCC(CH_i)$, we consider the changes in M_1, since possible M_1's comments updates are likely to be triggered by the change type CH_i, while this is not true for possible comment updates observed in M_2, since this latter has been subject to different categories of changes.

Since a comment in a method could also have a major impact on the responsibilities implemented by a class, for each CH_i we also compute $CCC(CH_i)$ as the percentage of changes falling in the CH_i category that triggered a Class Comment Change in comments related to the class the impacted method belongs to. In this case, we only focus on the Javadoc comment of the class, since the comments related to the methods it implements are already considered by the MCC metric. Also in this case, we only consider changes performed in isolation for a given change category, as explained for the MCC.

We answer $\mathbf{RQ_1}$ by comparing the distributions of MCC and CCC for the change categories reported in Table 3.2 via bar charts, showing the percentage of times that each change category CH_i triggered comment updates. We also use the Fisher's exact test [120] to test whether the chance of triggering method's and class's comments update significantly differ across change categories. To control the impact of multiple pairwise comparisons (*e.g.,* the chance of triggering method's comment changes of the *Array* category is compared against that of 17 other categories), we adjust p-values using the Holm's correction [121]. We use the Odds Ratio (OR) [120] as effect size measure. An OR of 1 indicates that the event under study (*i.e.,* the chance of triggering comment updates) is equally likely in two compared groups (*e.g.,* Array *vs* Casting).

An OR greater than 1 indicates that the event is more likely in the first group, while an OR lower than 1 indicates that the event is more likely in the second group.

Concerning $\mathbf{RQ_2}$, we manually analyzed a set of commits in which the developers fixed code-comment inconsistencies. We extracted, from the same set of 1,500 systems used in RQ_1, all commits having a commit note matching lexical patterns likely indicating the fixing of code-comment inconsistencies. To define these lexical patterns, we experimented with different combinations of words and inspected the resulting commits (details in [118]). we found the following pattern to be the best suited for the identification of the commits of interest: (*update** or *outdate**) and *comment(s)*. In other words, all commit notes containing

the word *update* or *outdate* in different derivations (*e.g.,* updates, updated) **and** the word *comment* or *comments* have been selected, for a total of 3,641 commits matched. From this set, we randomly selected for the manual analysis a sample of 500 commits, representing a 99% statistically significant sample with a 5% confidence interval.

The 500 commits were randomly distributed among three researchers including the author, making sure that each commit was classified by two researchers. The goal of the process was to identify the exact reason behind the changes performed in the commit. If the commit was unrelated to code comments, the evaluator classified it as *false positive*. Otherwise, a tag explaining the reason for the change (*e.g., update comment to correct a wrong method's parameter description*) was assigned, even in the case the commit was not related to a code-comment inconsistency, but just to changes in a comment (*e.g., fixed a typo in a comment*). We did not limit our analysis to the reading of the commit message, but we analyzed the source code diff of the changes implemented in the GitHub commit. The tagging process was supported by a Web application that we developed to classify the commit and to solve conflicts between the researchers. Each researcher independently tagged the commits assigned to him by defining a tag describing the reason behind the commit. Every time the researchers had to tag a commit, the Web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags. Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the commit) is extremely high, such a choice helps using consistent naming and does not introduce a substantial bias. In cases for which there was no agreement between the two evaluators (51% of the classified commits), the document was assigned to an additional evaluator to solve the conflict.

After having manually tagged all commits, we defined a taxonomy of code comment-related changes through an open discussion involving all the researchers (see Fig. 3.3). We qualitatively answer RQ_2 by discussing specific categories of commits likely related to the fixing of code-comment inconsistencies. For each category, we present interesting examples and common solutions, and discuss implications for researchers and practitioners.

3.3 Results

3.3.1 To what extent do different code change types trigger comment updates?

Fig. 3.1 compares the MCC (top) and the CCC (bottom) for the categories of AST-level changes described in Table 3.2. It is worth remembering that the MCC and the CCC values for a change category CH_i represent the percentage of times that a change of type CH_i triggered a change in a related method (MCC) or class (CCC) comment. Fig. 3.2 summarizes the results of the statistical comparison between the chance of triggering method (left) and class (right) comment changes for different categories of change categories in the form of a heatmap: A white block indicates that the difference between two categories is not statistically significant (adjusted p-value ≥ 0.05) or that the odds ratio between the two categories indicate a similar chance of triggering changes in code comments ($0.8 \leq d \leq 1.25$).

Blocks with four different grayscale values from light to dark represent a significant dif-

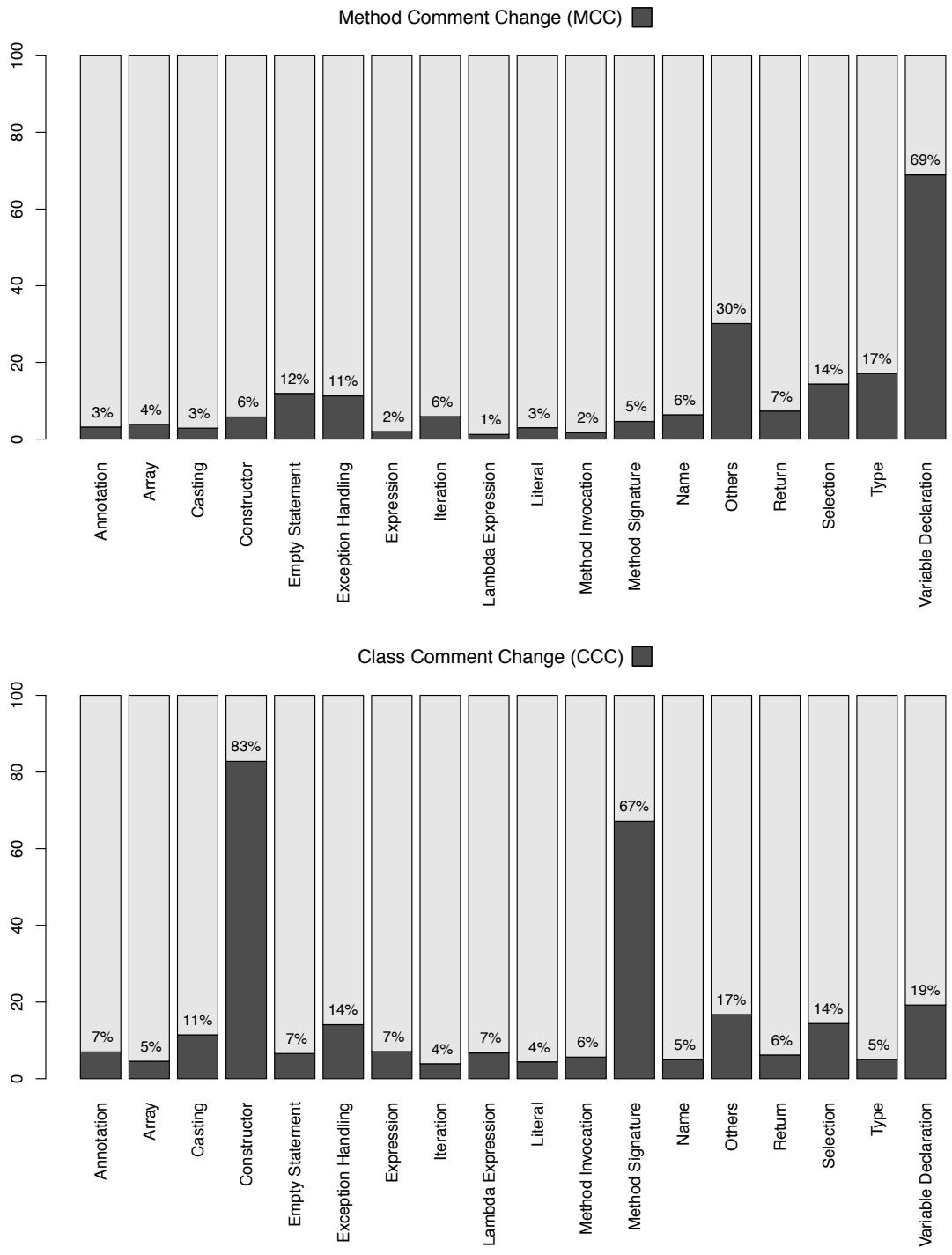


Figure 3.1. Code-comment evolution: MCC and CCC by change category (Table 3.2)

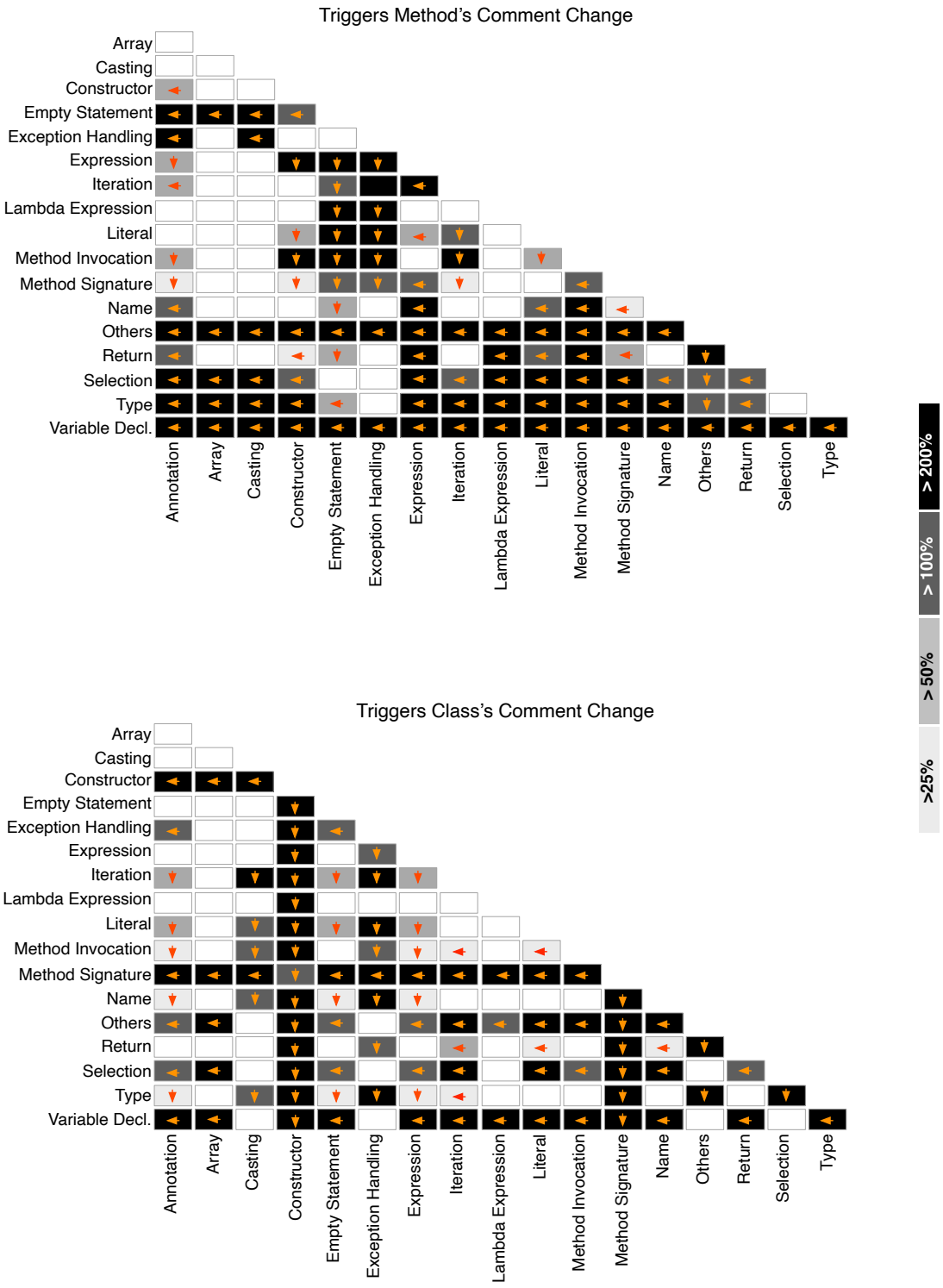


Figure 3.2. Statistical comparison of the chance of triggering method (top) and class (bottom) comment update by change category (Table 3.2)

ference between two change categories CH_i and CH_j accompanied by an odds ratio indicating that CH_i's changes have at least 25%, 50%, 100%, or 200% higher chance of triggering method or class comment changes than CH_j (or *vice versa*). The arrows in the heatmap point to the change category having the highest chance of triggering comment changes among the compared two. For example, when comparing the categories *Type* and *Constructor*, Fig. 3.2-left shows that *Type*'s changes have a higher chance of triggering updates in the related comments (at least 200% higher — black square). The detailed results with adjusted p-values and odds ratio are available in our online appendix for all comparisons [118].

From the analysis of Fig. 3.1 and 3.2 it is clear that different categories of code changes have a different likelihood of triggering updates in the related method and class comments. Also, the MCC and the CCC values show that changes to method and class comments are triggered by different categories of code changes. For example, changes impacting the *Constructor* are much more likely to trigger updates in the class comment ($CCC = 0.83$) as compared to the method comment ($MCC = 0.06$). This is expected since changes to the constructor can impact the way the whole class is instantiated and, as a consequence, are likely to require updates to the class's comment description. A similar trend can be seen for changes impacting the *Method Signature* ($CCC = 0.68$ *vs* $MCC = 0.04$), while the opposite is true for *Variable declaration*-related changes, *i.e.,* these changes trigger more frequently updates in the related method comments ($MCC = 0.69$) than in the class comment ($CCC = 0.19$). This result is reasonable, considering that we only take into account code changes affecting the methods' body, as we previously explained. Thus, changes to a variable declared inside a method are likely to only impact the logic of that method, without necessarily involving the overall class functioning.

One general trend that can be observed from Fig. 3.1 is that most of the code change categories rarely trigger changes in the related method and class comments. Our results point in the same direction of the findings by Fluri *et al.* [65]: They found that 23%, 52%, and 43% of all comment changes in ArgoUML, Azureus, and JDT Core respectively, were triggered by source code changes. Working on a much larger corpus of 1,500 systems, when considering all change types together we observe a co-evolution of code and comments happening in 7% of cases for method's comments and 13% of cases for class's comments. This means that, according to our data, 13% to 20% of code changes trigger a comment change in the class and/or in the methods' comments: 13% in case there is complete overlap between the two sets of changes (*i.e.,* those triggering methods' and those triggering class's comments changes), 20% in case they are completely disjointed.

Categories exhibiting low values of both MCC and CCC and showing statistically significant lower chance to trigger comment updates when compared to most of the other categories are *Array*, *Lambda Expression*, *Iteration*, *Literal*, *Method Invocation*, and *Name*. Due to the lack of space, we only discuss two exemplary cases from these categories, while more qualitative analysis will be reported in RQ_2.

The *Name* category includes changes performed on identifiers (*e.g.,* rename refactoring). We found cases of code-comment inconsistencies introduced as result of renamed identifiers. For example, in a commit performed in the `alluxio` project [173], the developer implements a rename refactoring on the `mIn` identifier, changing it to `mStream`.

This change affects several methods implemented in the class, but only one of them, namely `openStream()`, mentions the renamed identifier in its comment. In this commit, the developer forgets to update the comment, thus referring in it to an identifier that does not exist anymore in the code. The issue is fixed 20 days later [174].

The second example of inconsistency refers to the *Literal* category, grouping changes related to fixed values in code. A commit from the `bitcoinj` project [175] changed the value of a String literal from `"connectionTimeoutMillis"` to `"connectTimeoutMillis"`. This literal was used as a parameter value in a call to the `setOption` method. As explained in the commit note, this change was needed to fix a bug: *"Fix typo that prevented connection timeouts from being set properly"*. Indeed, the parameter value `"connectionTimeoutMillis"` was not a valid one. While the commit fixed the problem in the code, it did not fix an example reported in one of the comments of the class including an invocation to the `setOption` method, still using the old, wrong parameter value. The problem has been fixed in a later commit [176].

We answer RQ_1 with the following observations:

We confirm previous findings in the literature [65], showing that between 13% and 20% of code changes trigger comment updates. This does not imply that in the remaining ∼80% of cases code-comment inconsistencies are introduced, but they represent a possibility, as we observed through manual inspection, and as further demonstrated by the qualitative analysis we present in RQ_2.

Code changes to the Array, Lambda Expression, Iteration, Literal, Method Invocation, and Name categories are the ones less frequently triggering comment updates. This is also confirmed by the statistical analysis (Fig. 3.2), in which these categories exhibit, as compared to other categories, statistically significant lower chance of triggering comment updates, accompanied by at least a "small" and in most cases by a "large" effect size.

Change categories Variable Declaration and Selection are among those more likely to trigger comment updates, both at the method and at the class level. This is possibly due to the fact that these changes could severely impact the application logic (*Selection*) or the data manipulated in the code *Variable Declaration*. Also, changes in the *Method Signature* and *Constructor* categories are often accompanied by changes to the class's comment.

*The other change categories (*e.g., *Return, Annotation, etc.) exhibit MCC and CCC values mostly in the range 0.1-0.2*, showing that they still represent possible scenarios for the introduction of code-comment inconsistencies.

3.3.2 What types of code-comment inconsistencies are fixed by developers?

We addressed RQ_2 by labeling 500 commits identified as candidates to fix code-comment inconsistencies (see Section 3.2). We identified 138 false positives and 362 commits actually related to comment changes. Note that, while not all these commits are strictly related to code-comment inconsistencies, they are all related to improvement actions performed on comments. Overall, we identified 69 types of comment changes tackled by developers, 25 of which relevant for code-comment inconsistencies.

Fig. 3.3 presents the results in the form of a hierarchical taxonomy composed by six root

categories: Application Logic, Code Design/Quality, Maintenance, Formatting/Readability, Copyright/License and Others. The more specific types of comment-related changes are represented either as intermediate nodes or leaves, and changes relevant for the fixing of code-comment inconsistencies are marked with a ❗ sign.

For each root category, we next describe representative examples and, at the end of this section, we discuss implications for researchers and/or practitioners derived from our findings.

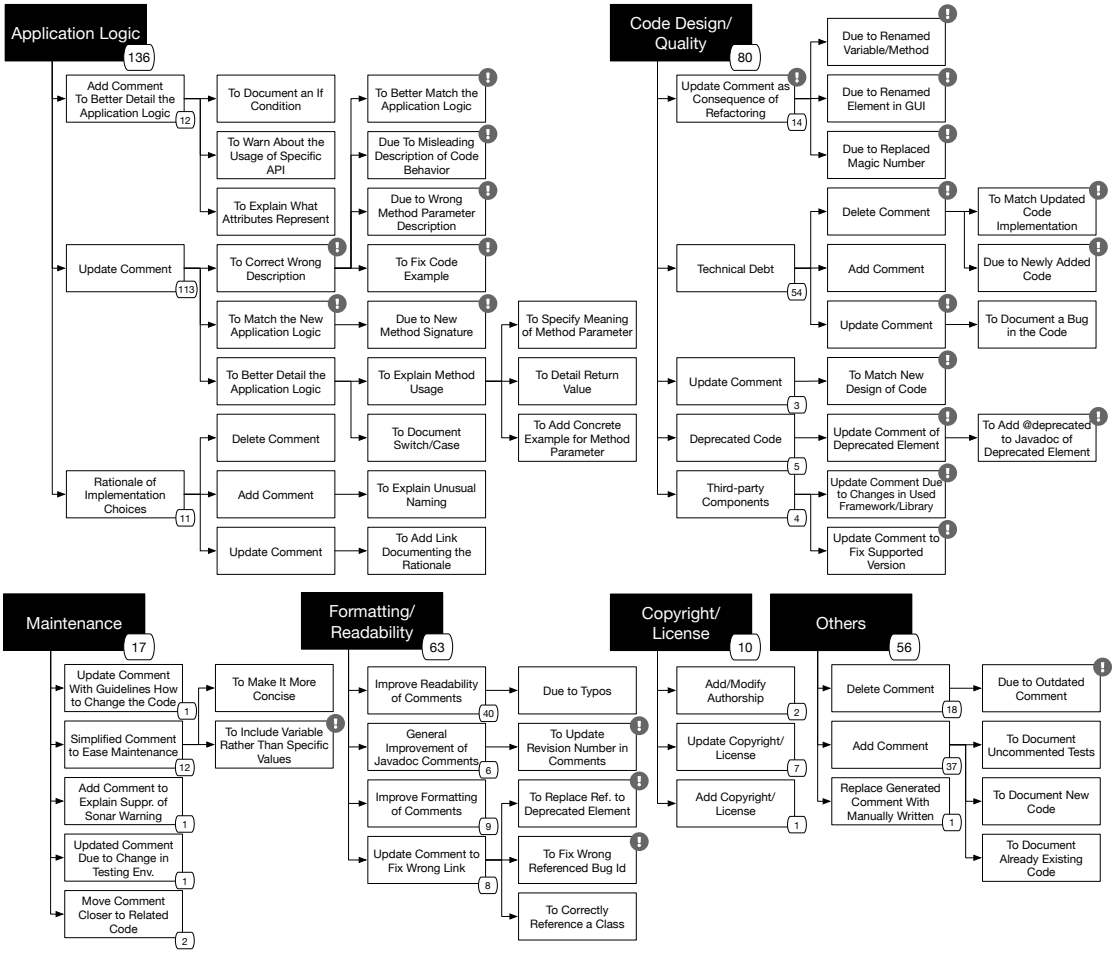


Figure 3.3. Taxonomy of Code Comment-Related Changes

Application Logic (136)

This category groups comment changes in which the impacted comments are related to the implemented application logic, such as a Javadoc describing the steps of an algorithm implemented in a method, its parameters or return type. In most cases, the change occurred in the form of a comment update (113), while in a few cases (12) a new comment was added. We

observed three main reasons why developers update comments: (i) the comment wrongly describes the application logic (35), due to an error done when the comment was written in the first place or to an inconsistency introduced later (in these cases we were not able to trace back to the specific cause of the problem); (ii) the comment needs to be updated as a consequence of a new implementation logic (25); (iii) the comment is improved to explain the implementation in more details (53).

For instance, in a commit of `QRCodegenerator` [177] an inline comment describing how an array element is calculated was updated to fix a copy/paste mistake done when the code was firstly written. The comment was copied from another line of code also calculating an array, but in a different way. This commit fixed the inconsistency between the code implementation and the comment description.

In `WordPressforAndroid` [178], the previously misleading comment of the `getPath()` method was replaced from "*descendants must implement this to send their specific request to the stats api*" to "*descendants must implement this to return their specific path to the stats rest api*". Similarly to the example discussed in RQ_1, also in RQ_2 we found cases in which the comment was fixed to update a code usage example reported in the comment and not aligned with the actual code implementation (see [179]).

Comments can also be used to explain the rationale for implementation choices (*e.g.,* to justify the usage of a specific collection type to represent data). We found cases in which after a code change, these comments became outdated, pushing developers to fix the discrepancy by simply deleting the comments (see [180]). In other commits, comments documenting the rationale were added, as in the case of the `ApacheCassandra` project [181], in which a comment was added in 2017 to explain why a variable introduced in 2015 was named `nulls`.

Code Design/Quality (80)

This category groups comment improvements that originate as consequence of actions related to code quality and design (*e.g.,* refactoring activities).

We observed three cases in which changes to the class hierarchy resulted in inconsistent comments.

One of these is from the `ApacheCordovaAndroid` project [182], in which we found a commit accompanied by the note: "*Update documentation comments to match implementation*". In 2012, the developers refactored the class hierarchy and converted the abstract class `CordovaInterface` to an interface [183]. However, its Javadoc comment has only been updated one year later, in 2013 [182].

Most of the cases in this group are related to *Technical Debt* comments (54), *i.e.,* comments describing known issues or 'TODOs' in the code. Such comments are often deleted (26) as a consequence of the developers *paying back* the debt. While the comment is usually removed in the same commit in which the technical debt is paid back, we found cases in which developers fixed the technical debt issue but left the comment by mistake. This required a subsequent commit aimed at removing the comment, *e.g.,* "*Issue #326: Remove forgotten outdated comment*" [184], from the `JavaParser` project.

In 19 commits the 'TODO comments' were updated due to a change in the code, for example to keep track of progress done in the fixing of the documented technical debt.

Related to technical debt, there were also cases in which comments were added to document the fact that a class/method was deprecated (see [185], [186]).

Updated comments following refactorings were also frequent (14), particularly after renamed methods/variables (11), but we found interesting examples also following renamed GUI elements (2), *e.g.,* [187], or a replace magic number refactoring (1), *e.g.,* [188]. In the latter case the developer replaced the inline comment "*Keep only 1000 batches worth data in queue*" with "*Keep only numBatchesInQueuePerPartition batches worth data in queue*", to match a replace magic number refactoring performed in a previous commit, thus fixing the code-comment inconsistency.

Finally, related to *Code Design/Quality* are comment changes aimed at fixing inconsistencies originated outside of the project scope in third-party libraries. An example we found is from the `PSIProbe` project [189]: "*Update comment about support as TomEE now supports tomcat 8.5*". Here the code implementation already provided support for a new Tomcat version that, however, was not officially released yet.

This was documented in the code through a code comment, that became obsolete once Tomcat 8.5 was released, pushing developers to delete it.

Maintenance (17)

In this category fall comment changes aimed to ease the future maintenance of comments, for example by making them more concise. Interesting are the changes implemented in `ApacheGroovy` to fix an outdated comment in such a way to also avoid uptodateness issues in the future: They modified the comment in order to use a newly introduced variable ("*The parameter can take one of the values in* `@link ALLOWED_JDKS`") rather than listing the complete list of supported JDK versions ("*[...] can take one of the values 1.7, 1.6, 1.5, 1.4*") [190] [191]. This makes unnecessary in the future to update these comments when new versions are supported or old versions are not supported anymore.

Another example of comment change aimed at avoiding future uptodateness issues is the commit "*remove comment that can be very easily outdated*" from `JetBrainsAndroid` [192]. Here the developer extracts from the Javadoc comment of the `IntellijCodeNavigator` class three paragraphs detailing the logic of its main method (`getNavigatable`), in particular related to branches of an `if` statement it implements. Each of the extracted paragraphs has then been moved closer to the specific lines of code it documents, to allow for an easier maintainability and to avoid that future changes to the code would not be reflected in the comment.

Formatting/Readability (63)

Changes intended to improve the formatting or readability of comments are grouped in this category. Not surprisingly, we found many commits in which developers just improved the wording of the comments (31) or fixed typos (9). We also grouped in this category comments aimed at implementing general improvements in the Javadoc (6), with a mix of

changes aimed at fixing typos, improving readability, formatting, etc. (*e.g.,* "*Fix comments to update javadoc for a bunch of methods*" from `Aluxio` [193]).

Although this type of changes is usually not related to code-comment inconsistencies, we found cases in which references (*e.g.,* related to other code elements, bug reports) became obsolete, resulting in invalid/outdated references in comments. For example, in `GoogleGuava` a commit says: "*Updated a comment in ListenerCallQueue to point at SequentialExecutor instead of the deprecated SerializingExecutor wrapper interface*" [194].

Copyright/License (10)

We grouped fixes related to copyright/license comments separately under this category as we found a considerable amount of commits working on updating licensing information.

These changes were mostly related to simple maintenance of copyright headers in source files, *i.e.,* updating authorship [195] or copyright year [196].

We also spotted cases, however, where outdated copyright comments remained in source files for several years. In 2011 a developer of the `ApacheGroovy` project updated the copyright header of a Java file from a BSD variant to Apache License v2 [197], although the project had already changed its license back in 2007 [198].

Others (56)

This category groups comment changes that, while not being false positives (*i.e.,* they are related to code comments), do not fit any of the previous categories. Comments were added in 37 cases to document new or already existing code. In one case, automatically generated comment skeletons were replaced with manually written comments [199], while the comment deletions were generally related to outdated comments left in the code by mistake. An example can be seen in a commit of the `CrateDB` project [200] where the developer deletes the description of the error handling of SQL operations that was rewritten in earlier commits.

3.3.3 Discussion and Implications

Our large-scale study in RQ_1 confirmed previous findings reported in the literature and showing that, in most of the cases, code and comments do not co-evolve. It is important to highlight that a code change does not always result in the need for updating the corresponding comment. Thus, we are not claiming that do not updating comments as a consequence of code changes is a bad practice. However, our qualitative analysis disclosed several cases in which developers introduced (RQ_1) or fixed previously introduced (RQ_2) code-comment inconsistencies, providing us with a number of lessons learned. In the following we discuss implications for researchers (indicated with the ⚗ icon) and/or practitioners (🐙 icon) derived from our findings.

The maintainability of comments is as important as that of source code 🐙. As it happens for code, comments should also deal with "functional" and "non-functional" requirements. The functional aspect here is the proper documentation of the source code, and it has always been recognized as a fundamental support for code comprehension. Not less

important are, however, the non-functional aspects of code comments, such as their readability and *maintainability*. As shown in our study, a simple idea such as using a variable referenced in the comment to document the supported JDK versions as opposed to explicitly listing them [190] can dramatically simplify the maintenance of the comment, that will not require any future update in case of changes to the supported JDKs. Basically, as source code is often designed to isolate and minimize future changes, the same should happen for comments.

Refactoring code comments ⚗. From the researchers' perspective, our study stresses the importance of investing effort in the development of tools to support code comments refactoring. Indeed, most of the effort in this field has been devoted to the automatic assessment of comment quality (see *e.g.,* [67, 68, 71, 74]) without, however, recommending how to automatically refactor it. Our findings provide insights for the future development of approaches able to both detect and fix issues in code comments. For example, we have seen as simple copy/paste can introduce code-comment inconsistencies, due to a wrong reuse of comments across semantically different instructions (*i.e.,* the same comment is reused for two different instructions, wrongly documenting one of the two) [177]. Identifying different code components documented with the same comment can help in identifying these problems.

Concerning the automatic comment refactoring, a first step in this direction could be the definition of a catalogue of operations for comments, similarly to what has been done for source code [122]. For example, we observed an instance of what can be defined as an "*extract comment refactoring*" [192], aimed at splitting a large comment into several comments to place each one closer to the exact instruction it documents.

Code comments are first-class citizens in code refactoring ⚗ Y. We observed several code-comment inconsistencies introduced as consequence of refactoring activities. For example, the application of a replace magic number refactoring [188] caused the magic number to be removed from the code but not from the related comments. Similarly, major refactorings to the class hierarchy [182] caused outdated references in comments. This highlights: (i) the need for developers to consider the effects on comments when applying refactoring operations; (ii) the opportunity for researchers to investigate how to integrate better comment support into refactoring tools.

Code-comment traceability is still an open problem ⚗. Related to the previously discussed points, one major research challenge is the code-comment traceability (*i.e.,* automatically identifying the code instructions documented by a given comment). In 1988, Kaelbling argued that programming languages should not have comment statements [123], but *scoped comments*, explicitly indicating the code elements they refer to. As of today, documentation tools such as Javadoc help developers to explicitly comment certain elements by referring to them. IDEs also provide support, *e.g.,* by showing related comments of selected items. However, popular programming languages are still bound to line and block comments. There are many research opportunities here both for language designers and researchers to facilitate the code-comment traceability. Solving this problem will in turn help to make substantial steps ahead in the identification/fixing of code-comment inconsistencies.

3.4 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

RQ_1: Computation of the MCC and CCC metrics. As explained in Section 3.2 these metrics, for a specific type of code change category CH_i, measure the percentage of times that method (MCC) or class (CCC) comments are inserted/deleted/modified in response to CH_i's changes. Clearly, if a modified method/class has no comments, these metrics cannot capture the deletion or modification of the method's comments, but only the insertion of new comments. Considering the scale of our study and the focus on long-lived and popular systems unlikely to have many undocumented methods, these imprecisions should not have a major impact on the outcome of our study.

RQ_1: Imprecision introduced by GumTreeDiff. As any differencing tool, GumTreeDiff could generate wrong information. For example, we noticed that in some cases the update of a variable type (*e.g.,* from `double` to `int`) was reported as the deletion of a variable (the `double` one) followed by the addition of a new variable (the `int` one). However, GumTree is a state-of-the-art differencing tool and at least we tried to reduce possible noise caused by "move" operations.

RQ_2: Subjectivity in the manual classification. We identified through manual analysis the reasons behind commits performed by developers to (likely) fix code-comment inconsistencies. To mitigate subjectivity bias in such a process, every commit was assigned to two researchers who manually analyzed it independently. Then, in the case of a disagreement, a third researcher was assigned to the commit to solve the conflict.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. One aspect could be related to the selection of projects being considered. As explained by Kalliamvakou *et al.* [124] mining GitHub can be risky because projects may contain very few commits. To mitigate this threat, we applied strict criteria (*i.e.,* more than 500 commits, more than 10 stars) when selecting the context of our study. To reinforce the internal validity, when possible, we integrated the quantitative analysis with a qualitative one.

Threats to *external validity* concern the generalizability of our findings. RQ_1 tries to achieve a high generalizability in terms of mined projects that, however, are all written in Java. Future work should focus on systems written in different languages to confirm or contradict our findings. RQ_2 analysis is limited to a specific set of 500 commits we randomly selected as output of a keyword-based mechanism used for the pre-selection of commits likely related to code-comment inconsistencies. Because of this procedure, our taxonomy surely omits types of code-comment inconsistencies fixed in commits we did not analyze and/or documented in diverse data sources (*e.g.,* issues on GitHub).

3.5 Conclusion

We presented the largest study at date about the co-evolution of code and comments. The study involved the analysis of the complete change history of 1,500 Java systems. Then, we manually analyzed 500 commits likely related to the improvement of code comments, classifying 362 of them (the non-false positives) into a taxonomy of comment-related changes (Fig. 3.3). The results achieved with our quantitative and qualitative analyses have been used to distill lessons learned resulting in actionable items for both researchers and practitioners (Section 3.3.3).

4

Quick Remedy Commits and Their Impact on Mining Software Repositories

Most changes during software maintenance and evolution are not atomic changes, but rather the result of several related changes affecting different parts of the code. It may happen that developers omit needed changes, thus leaving a task partially unfinished, introducing technical debt or injecting bugs.

In this chapter, we present a study investigating "*quick remedy commits*" performed by developers to implement changes omitted in previous commits. With *quick remedy commits* we refer to commits that (i) *quickly* follow a commit performed by the same developer, and (ii) aim at *remedying* issues introduced as the result of code changes omitted in the previous commit (*e.g.,* fix references to code components that have been broken as a consequence of a rename refactoring) or simply improve the previously committed change (*e.g.,* improve the name of a newly introduced variable). Through a manual analysis of 500 quick remedy commits, we define a taxonomy categorizing the types of changes that developers tend to omit. The taxonomy can (i) guide the development of tools aimed at detecting omitted changes and (ii) help researchers in identifying corner cases that must be properly handled. For example, one of the categories in our taxonomy groups the *reverted commits*, meaning changes that are undone in a subsequent commit. We show that not accounting for such commits when mining software repositories can undermine one's findings. In particular, our results show that considering completely reverted commits when mining software repositories accounts, on average, for 0.07 and 0.27 noisy data points when dealing with two typical MSR data collection tasks (*i.e.,* bug-fixing commits identification and refactoring operations mining, respectively).

4.1 Introduction

In the software life-cycle, change is the rule rather than the exception. Changes are generally performed through commit activities to add new functionality, repair faults, and refactor code [20]. Some of these commits can involve a substantial part of the source code, with

dozens of artifacts impacted [17]. This is often the result of what Herzig and Zeller [44] defined as *tangled commits*: Commits grouping together several unrelated activities, such as fixing a bug and adding a new feature.

In other cases, a single cohesive change (*e.g.,* a bug fix) is instead split across several commits. This can be due to omitted code changes and/or the need for fixing a mistake done in the first attempt to implement the change. Park *et al.* [18] showed that 22% to 33% of bugs require more than one fix attempt (*i.e.,* supplementary patches). Studying supplementary patches can be instrumental in designing recommender systems able to reduce omission errors by alerting software developers, as attempted in a subsequent work by Park *et al.* [19], where the authors tried to predict additional change locations for real-world omission errors. Due to the limited empirical evidence about the nature of omitted changes, this is still an open challenge. Indeed, while the work by Park *et al.* [18] investigates omitted changes, it explicitly focuses on supplementary patches for bug-fixing activities, ignoring other types of code changes (*e.g.,* implementation of new features, refactoring). Thus, there is no study broadly investigating the types of changes that developers tend to omit during implementation activities.

To fill this gap, we presented a qualitative study focusing on "*quick remedy commits*" performed by developers. We defined as *quick remedy commits* those commits that (i) *quickly* succeed a commit performed by the same developer in the same repository; and (ii) aim at *remedying* the issues introduced as the result of code changes omitted in the previous commit (*e.g.,* fix references to code components that have been broken as a consequence of a rename refactoring) and/or of introduced errors. In other words, we identified pairs of commits (c_i, c_{i+1}) that are temporally close (*i.e.,* c_{i+1} succeeds c_i by a few minutes), are performed by the same developer, and include in the commit note of c_{i+1} a reference to fixing issues introduced in c_i.

Fig. 4.1 shows an example of a quick remedy commit from our dataset, and in particular from the GitHub project `bardsoftware/ganttproject`. In the commit depicted in the top part of Fig. 4.1 (*i.e.,* commit `a43b8f2`), the developer implemented, among other changes, a refactoring aimed at simplifying the code of the `GPAction` class. In particular, instead of invoking three times the method `GanttLanguage.getInstance()` in different parts of the class, the `language` variable is instantiated, and reused where needed.

Two minutes later, the same author performs a *quick remedy commit* (bottom part of Fig. 4.1 — commit `2c40a07`) by reporting in the commit note: *Forgot 1 refactoring of 'language' in previous commit*. The remedy commit propagates the changes introduced by the refactoring to another location of the `GPAction` class, that was missed by mistake in the original commit.

We decided to focus on remedy commits (c_{i+1}) that are temporally close to the original change they fix (c_i) for two reasons. First, it is easier to establish a clear link between two commits by the same developer if they are performed within a few minutes. Second, as shown by Park *et al.* [19], it is challenging to prevent omission errors automatically; thus, we decided to focus on omission errors that, since fixed within few minutes, are likely not to be so complex.

This allows gathering empirical knowledge to take a first step in automating the preven-

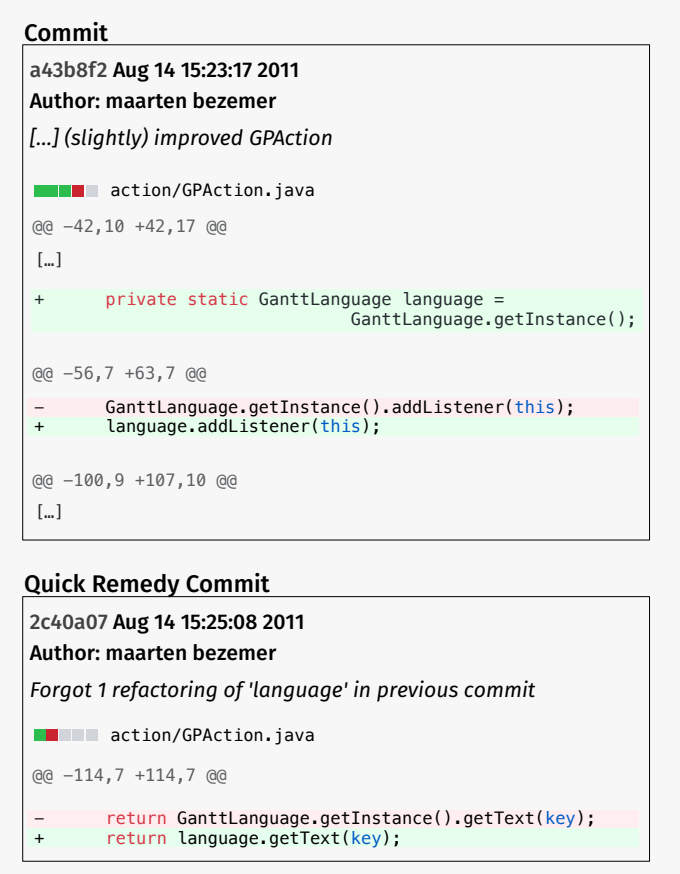


Figure 4.1. Example of quick remedy commit

tion of a basic set of omission errors that, as we show, can be responsible for bugs and major code inconsistencies if not promptly fixed.

We defined heuristics to identify *quick remedy commits* automatically, and mined the commits of interest from the complete change history of 1,497 Java projects hosted on GitHub. This allowed the identification of ∼1,500 candidates quick remedy commits. We manually analyzed 500 of them looking at the changes introduced in the remedy commit (c_{i+1}) and the previous commit (c_i) as well as the summary of changes provided in the commit notes.

The goal of the manual analysis was to identify the rationale of the remedy commits to define a taxonomy categorizing the types of issues introduced by developers during commit activities that trigger a remedy commit, discussing the implications of our taxonomy for researchers and practitioners.

As a following study, we further looked into the implications of a specific part of our taxonomy for researchers working in the Mining Software Repositories (MSR) field. In particular, we focused on a category in our taxonomy grouping together *reverted commits*, *i.e.,* remedy commits c_{i+1} in which the developers revert, completely or partially, the code changes they committed in the previous commit c_i. We defined a methodology to automatically iden-

tify these commits in a given repository and studied the impact they could have on MSR studies. The decision to focus on such a specific category in our taxonomy is two-fold: (i) as we will explain later in the thesis, it is the type of quick remedy commits that is more likely to affect the data collection process in MSR, possibly leading to the inclusion of noisy data points in the study; (ii) it is the only category for which a reliable and automated detection mechanism can be easily devised (*i.e.,* it is relatively easy to detect reverted commits as compared to other categories of commits in our taxonomy).

We took two "data collection tasks" frequently performed in MSR studies, namely the identification of bug-fixing commits (see *e.g.,* [41, 125, 126, 127, 128]) and the mining of refactoring operations performed in the history of a system (see *e.g.,* [128, 129, 130, 131, 132, 133]). Then, we applied these two tasks on 100 long-lived Github repositories; collecting refactoring operations performed in each commit and a set of bug-fixing commits. Finally, we cleaned the collected data by removing completely and partially reverted commits. For example, a researcher may identify a bug-fixing commit in the history of a software system. However, if such a bug-fix is later on reverted by the developer, we argue that it should not be considered a valid data point, since it basically represents noise. We show that, for each completely reverted commit kept in the collected data, there is a .07 increase in the number of detected bug-fixing commits and a 0.27 increase in the number of detected refactoring commits. The methodology we adopt to identify the *reverted commits* can be applied in MSR studies to help researchers in minimizing the impact of these commits on their findings. Clearly, the removal of *reverted commits* is subject to the goal of the study and the data analyses researchers are interested in performing. For example, if the goal of the study is to count the number of bugs introduced by a developer in a system, researchers may be willing to also count bug-introducing commits that have been later on reverted. Instead, if the goal is to assess the logical coupling between code components (*i.e.,* how frequently they co-change), researchers may want to ignore completely reverted changes in the coupling computation. All in all, our study confirms the importance of careful data cleaning when mining software repositories, as highlighted in previous works [134].

The data used in both our studies are publicly available [135].

Structure of the Chapter

In Section 4.2 we present the design and the results of our first empirical study, in which we investigate the types of quick remedy commits performed by developers. Section 4.3 presents the design and results of our second study, assessing the potential impact of reverted commits in MSR studies. In Section 4.4 we discuss the threats that could affect the validity of our two studies, while in Section 4.5 we conclude the chapter.

4.2 Study I: Quick Remedy Commits Performed by Developers

4.2.1 Study Design

The *goal* of the study is to qualitatively investigate quick remedy commits. The *purpose* is to define a taxonomy of quick remedy commits that developers perform to fix issues introduced in a previous commit and/or finalize an uncompleted implementation task. The study addresses the following research question (RQ):

RQ$_1$: *What types of quick remedy commits are made by developers in Java projects?*

This RQ aims at identifying the types of quick remedy commits that are performed by developers (*e.g.,* documenting through a code comment a piece of code introduced in the previous commit). Knowing the types of quick remedy commits made by developers can guide the development of tools to automatically alert developers when code changes they are committing may require a subsequent remedy commit. In some cases this could even avoid the introduction of bugs (*e.g.,* due to changes not propagated in all code areas where they are required).

Data Collection and Analysis

We started the study with the same list of 1,500 GitHub repositories we collected from Section 3.2.1 in Chapter 3. During the cloning of the 1,500 GitHub repositories, we got a cloning error for three of them. Thus, we extracted the list of commits performed over the change history of the remaining 1,497 projects. The complete list of considered projects is publicly available in our replication package [135].

To extract the history of the subject systems, we iterated through the commit history related to all branches of each project with the `git log --topo-order` command. This allowed us to analyze all branches of a project, without intermixing their history and avoiding unwanted effects of merge commits.

Then, given the commit history, our goal was to identify all pairs of subsequent commits (c_i, c_{i+1}) in which c_{i+1} had been performed by a developer D_j as a quick remedy fix for a commit c_i also authored by D_j. In other words, c_{i+1} must (i) have been authored by the same developer of c_i and performed within a relatively short time interval from c_i; (ii) clearly be a "compensatory" fix for c_i.

To identify the (c_i, c_{i+1}) pairs of interest, we adopt the following heuristic-based procedure. First, we computed the time interval between all adjacent (subsequent) commits in each system authored by the same developer. In *git* it is possible to retrieve the *author date* (*i.e.,* the date in which the change has been implemented by the author) or the *committer date* (*i.e.,* the date in which the change has been committed). Given the goal of our work, we considered the *author date*. We analyzed the distribution of these time intervals (see Fig. 4.2).

We considered the first quartile, exactly *five minutes*, as a candidate threshold to identify remedy commits: c_{i+1} commits performed as quick fixes for their predecessor c_i commit.

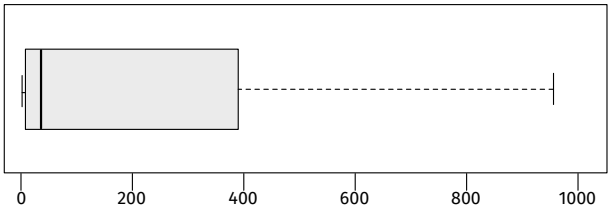


Figure 4.2. Time differences (in minutes) between subsequent commits (without outliers)

This allowed us to select pairs of commits meeting our first requirement: They were authored by the same developer and performed in rapid succession (*i.e.,* within five minutes). This filtering left us with 1,041,397 candidate commits.

Second, we set up a process to define lexical patterns allowing the identification of c_{i+1} commits in which the developer explicitly indicates in the commit note the fact that c_{i+1} is a remedy commit for changes introduced in the previous commit (c_i). We extracted from all 1,041,397 commits output of the previous filtering step the words and 2-grams used in their commit notes. This means that, from a commit note reporting "*Fixes a bug introduced in previous commit*", we would extract *fixes*, *a*, *bug*, etc. as the single words, and *fixes a*, *a bug*, *bug introduced*, etc. as 2-grams. To remove noise, stop words (*e.g.,* articles) and all single words shorter than four characters had been excluded from the set of single words (not from the 2-grams list). The remaining words and all 2-grams had then been sorted by frequency in descending order, excluding the long tail of those appearing in less than ten commits. Indeed, even if useful to identify remedy commits, lexical patterns defined from these words/2-grams are unlikely to retrieve a substantial amount of useful commits and, thus, are excluded *a priori* from reducing the inspection effort. For each remaining word/2-gram, we randomly extracted ten commit notes in which it appears.

This dataset, composed of words/2-grams and related commit notes, had been manually and independently inspected by three researchers including the author with the goal of defining the needed lexical patterns. After an open discussion in which each researcher presented his list of patterns, the three evaluators agreed on the following lexical pattern to identify remedy commits:

(*former* or *last* or *prev* or *previous*) and *commit*

This means that commit notes including *former commit*, *last commit*, *prev commit*, or *previous commit* would be matched and considered as relevant for our study. While this heuristic is quite strict, our goal was to maximize precision at the expense of recall, considering the fact that our study is qualitative in nature and does not target a large number of manually analyzed commits. At the end of this last filtering step, we obtained 1,577 c_{i+1} commits which (i) have been authored within five minutes from the commit c_i previously performed by the same author; and (ii) explicitly mention in the commit note a lexical reference to the previous commit that can be captured by the defined pattern. Given the high cost of the manual analysis process detailed in the following, we decided to focus our analysis on a randomly selected sample of 500 commits, representing a 99% statistically significant sample

with a 4.8% confidence interval.

The 500 commits were randomly distributed among three researchers including the author, making sure that each commit was classified by two researchers. The goal of the process was to identify the exact reason behind the changes performed in the commit. If the commit was unrelated to the previous one, the evaluator classified it as *false positive*.

Otherwise, a tag explaining the reason for the change (*e.g., remove debugging code from the previous commit*) was assigned.

We did not limit our analysis to the reading of the commit message, but we analyzed the source code diff of the changes implemented in the GitHub commits, both in the c_{i+1} commit as well as in its predecessor (c_i). The tagging process was supported by a Web application that we developed to classify the commit and to solve conflicts between the researchers. The Web application is shown in Fig. 4.3. Each researcher independently tagged the commits assigned to him by defining a tag describing the reason behind the commit. Every time the researchers had to tag a commit, the Web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags (visible in the bottom part of Fig. 4.3). Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the commit) is extremely high, such a choice helps using consistent naming and does not introduce substantial bias. In cases for which there was no agreement between the two evaluators (44% of the classified commits), the commit was assigned to an additional evaluator to solve the conflict. While such a percentage may look high, it is worth considering that our task was not to assign commits to a list of predefined categories, but to define the names for such categories during the tagging process. This naturally leads to a higher number of conflicts. Also, we considered as a conflict cases in which a different but "semantically equivalent" tag was used by the two evaluators (*e.g., remove unnecessary code vs remove unneeded code*). In this case, the third evaluator just made sure that consistent wording was used, and selected the proper tag. In a minority of cases, the two evaluators applied completely different tags and the third evaluator could choose whether to reuse one of the two labels or, instead, define a new tag by discussing and agreeing with the two original evaluators.

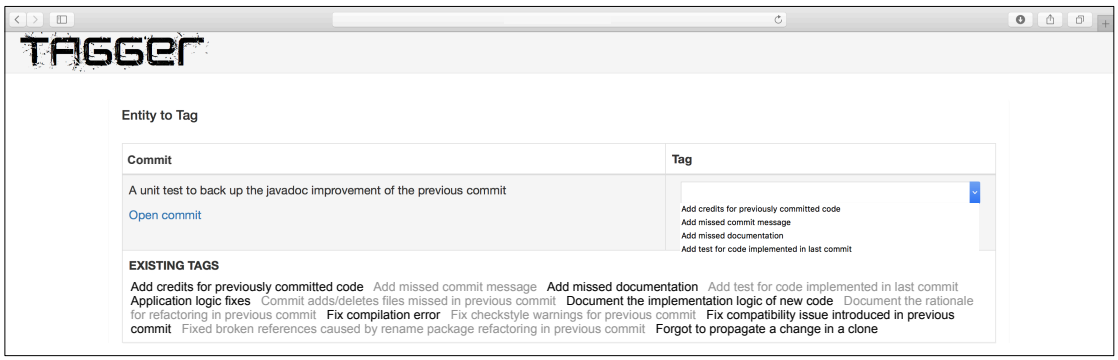


Figure 4.3. Web application used to run the manual tagging

After having manually tagged all commits, we defined a taxonomy of quick remedy commits through an open discussion involving all the researchers (see Fig. 4.4). We qualitatively answer our research question by discussing specific categories of commits likely related to the code changes developers often forget to implement and try to immediately remedy. For each category, we present interesting examples and discuss implications for researchers and practitioners.

4.2.2 Results

We addressed our research question by labeling 500 commits identified as candidates to being quick remedy commits (see Section 4.2.1). We identified 42 false positives (*i.e.,* commits c_{i+1} that were not related to the preceding c_i commit) and 458 commits actually classifiable as quick remedies [1]. Note that not all these quick remedy commits are compensatory fixes for issues caused by omitted changes. They also include fixes for previously introduced errors (*e.g.,* the developer realizes that her previous commit introduced a bug) as well as commits aimed at simply improving the previously committed change (*e.g.,* improve the name of a newly introduced variable). Finally, our taxonomy also features remedy commits aimed at fixing simple mistakes performed during the c_i commit process itself (*e.g.,* the developer forgot to include a modified file in commit c_i and thus commits it in c_{i+1}).

Overall, we identified 69 types of quick remedy commits made by developers, 20 of which relevant for changes omitted in the previous commit.

Fig. 4.4 presents the results in the form of a hierarchical taxonomy composed by six root categories: *Bug Fix*, *Code Refactoring/Clean Up*, *Build Issue*, *Missing Code Change*, *Documentation*, and *Reverted Commit*. The more specific types of quick remedy commits are represented either as intermediate nodes or leaves, and commits relevant for the fixing of issues caused by omitted changes are marked with a ❗ sign. For each category, we next describe representative examples and discuss implications for researchers (indicated with the ⚗ icon) and/or practitioners (🌿 icon) derived from our findings.

Bug Fix (79)

This category groups pairs of commits (c_i, c_{i+1}) in which the remedy commit (*i.e.,* c_{i+1}) fixes a bug introduced in c_i. We identified two main subcategories: *Fix Broken Test*, in which c_{i+1} has been triggered by test cases failing after the change implemented in c_i, and *Fix Implementation Logic*, in which the developer realized that she introduced a bug in c_i and quickly submits a patch.

The commits in the *Fix Broken Test* category targets the fixing of the production code or the test code modified in c_i and causing the test suite to break. For example, in the `Denominator` project of `Netflix`, a developer reported in the commit message: "*Fix tests broken by former commits*" [201].

[1] Our online appendix features an analysis of common keywords present in the commit message of these commits in comparison to non-quick-remedy commits [135].

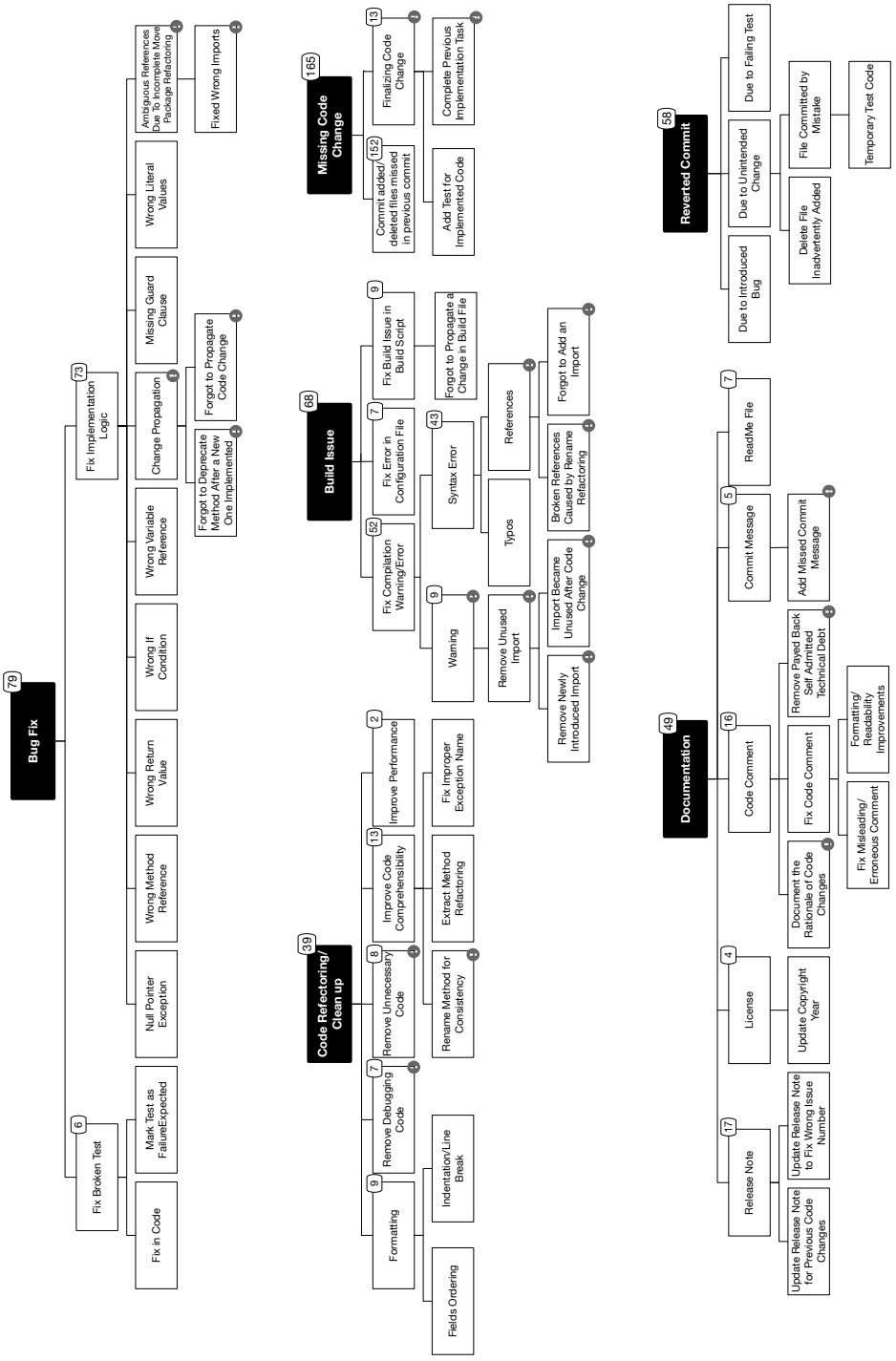


Figure 4.4. Taxonomy of Quick Remedy Commits

While in the cases we analyzed the issue was spotted and fixed quickly by the developer, there might be non-trivial cases in which only a subset of the test suite is executed for regression testing (*e.g.,* due to a limited testing budget) and a non-executed broken test is not identified by the developer.

For researchers, this is an opportunity to study test breaking-changes and to develop techniques able to alert the developer when a change she implemented might require a double check of (part of) the test suite. For practitioners, continuous integration practices can help in timely spotting these issues in most of the cases.

The fixes to the implementation logic are mostly classic bugs introduced but quickly recognized and fixed by developers (*e.g.,* errors in `if` conditions, wrong literal values, null pointer exceptions, etc.). While these are not related to omitted changes, they are interesting since they represent bugs fixed by developers within five minutes (due to our selection criteria for the commits).

This indicates that these bugs, while prevalent in our taxonomy (73 instances), are likely quite simple to fix. Thus, researchers could investigate the possibility of creating approaches able to learn from this data on how to avoid and/or automatically fix these bugs. For example, recent work applied Neural Machine Translation (NMT) models to automatically fix bugs [41]. However, given the complexity of this task and the non-trivial bugs that these models have to fix, they are usually only able to automatically fix a minority of the bugs provided as input [41]. Focusing on these simpler but quite frequent bugs could represent a good application scenario for the NMT-based bug fixing approach.

Some of the fixes in the *Fix Implementation Logic* category are related to omitted changes (see Fig. 4.4). This includes the *Forgot to Propagate Code Change* category in which developers do not consistently propagate a change across all relevant code components. This is typical of cases in which code clones are spread in the system and inconsistent changes are implemented in c_i [136]. An example of this can be seen in the $mathttTomP2P$ project. In a commit [202], the developers adapts a builder class (`PutBuilder`) to earlier changes of the original class and they implement new methods such as `isPutConfirm` and `isPutReject`. In a follow-up change [203], they fix a conditional statement to check the status of a `Put` object in a new branch. Then, only a few seconds later [204], they update a conditional check with a similar structure but in another class. For this last commit, the commit message says "*belongs to previous commit*". Another example can be seen in the $mathttspacewalk$ project. In a commit [205], they update a SQL script by adding a query for the removal of unnecessary data. Then, in the quick subsequent commit [206], they propagate the same schema changes into a database upgrade file.

These examples highlight the relevance for practitioners of approaches to guide code changes (see *e.g.,* the seminal work in the area by Zimmermann *et al.* [137]) as well as the need for the research community to continue improving these techniques and, possibly, making them easily pluggable into a continuous integration pipeline to foster developers' adoption.

Interesting in this category is also the introduction of *ambiguous references due to incomplete move package refactoring*. We found this case in the `apache/accumulo` project, where they migrate some classes to another package [207], but still keep the old ones.

In a follow-up commit [208], they realize that they use, however, the wrong references to the migrated classes. ⚗ Code clone detection techniques [138] could help in these cases by promptly pointing the developer to the presence of multiple copies of the same classes in the repository. The integration of these approaches in a just-in-time fashion could help in identifying clones introduced in the last commit, thus avoiding mistakes as the one in the discussed commit [207].

Code Refactoring/Clean up (39)

This category groups the pairs of commits (c_i, c_{i+1}) in which the remedy commit (*i.e.,* c_{i+1}) implements a refactoring/cleanup of the code changed in c_i (see Fig. 4.4). In these commits developers are either not satisfied of the code they implemented or are trying to address warnings received by static analyzers.

Some other subcategories include the simple removal of code that was only temporary implemented in c_i (*i.e., Remove Debugging Code*) or that becomes unnecessary after c_i's changes (*i.e., Remove Unnecessary Code*). Also, code formatting issues (*e.g.,* mainly the inconsistencies of indentations and line breaks introduced with code changes) were fixed by developers in the remedy commit (ie *Code Formatting*). Additionally, in 2 cases, developers changed the code implemented in c_i to improve its performance. An example can be seen in project `rzwitserloot/lombok` [209] where a developer fine tunes a cache clearing mechanism implemented in a previous commit by turning a variable volatile and moving the invocation for the cache clearing after a conditional check.

However, the main purpose of those code refactoring/clean up tasks is to improve the code understandability. Variable and method renaming refactoring (*i.e.,* renaming a variable or method to better reflect its functionality) is the most common way to make the code easier to comprehend. Also popular are code transformations aimed at replacing literal values with variables or splitting long functions through extract method refactoring. The latter allows not only to foster comprehensibility, but also the reusability of small code snippets.

Other interesting cases are the ones in which developers modify the previously committed code to promote consistency with the coding style of the project (see *e.g., Rename Method for Consistency*). For example, in a commit of the `liferay−portal` project [210], developers opened an issue to "*introduce tests to document current behavior*" [211]. Interestingly, in this process they very carefully review the used method names for better readability, and in a commit they say:

> "*[...] where specific method names are NOT accurate, go for a generic name to force the developer to read the code to find what the method actually does*".

The developers decided to change a method's name from `assertThatSearchResultHasVersion` to `assertSearchResult`. In the next commit [210], to remain consistent, they replace the method invocation of `assertThatEverythingButSummaryIsEmpty` (in another class) to `assertSearchResult`. For this last commit, the commit message says "*Match previous commit even though this method name was accurate*".

🧪 The inconsistencies fixed with simple refactorings point to the possibility for the software engineering research community to investigate techniques able to learn coding conventions used in a given system and recommend fixes for possible violations. To the best of our knowledge, the only attempt at date has been made by Allamanis *et al.* [139] with their NATURALIZE tool able to recommend meaningful identifier names and formatting guidelines. Other approaches focus only on rename refactoring suggestions [140]. While these techniques cover most of the inconsistencies fixed in the *Code Refactoring/Clean up* category (*e.g., Rename Method for Consistency*, *Fix Improper Exception Name*), others are left uncovered (*e.g., Fields Ordering*), indicating more potential for additional research in the area of recommending coding convention fixes.

Build Issue (68)

This category is related to commits fixing build issues introduced as a result of the c_i changes. The main subcategory here is the fix of the compilation errors/warnings issued by the compiler due to the changes in c_i (*i.e., Fix Compilation Warning/Error*). Unused import statements are the main cause for the warnings we identified (see Fig. 4.4), and the trigger for the remedy commits in this category. The unnecessary import statements are caused either by `import` statements introduced in c_i by the developer and then unused, or by previously existing `imports` becoming unused due to the changes implemented in c_i. These warnings are usually raised by static analysis checks performed at commit time and, thus, are easy to catch for developers.

In the *Syntax Error* category we found many cases of broken references due to rename refactoring operations performed in c_i. These rename refactorings are related to variables, methods, classes, as well as packages. An example can be seen in the commit [212] of the `DroidPlanner/Tower` project which followed a renaming of multiple classes. Some other cases were violating the syntax of the programming language due to introduced typos (*e.g.,* missing statement separators).

Considering the good refactoring support provided by modern IDEs, the identification of these broken references as a consequence of refactorings was quite surprising for us. 🧪🔧 This may indicate either that these refactorings were performed manually, leading to the introduction of broken references, or that bugs might affect refactoring engines, as already found by previous work in the literature [141]. Additional investigation focused on these specific types of errors is needed to understand the reasons behind them.

Other subcategories that also caused a build issue include the fix of introduced errors in configuration files (*i.e., Fix Error in Configuration File*) or in a build script (*i.e., Fix Build Issue in Build Script*). For example, in some remedy commits developers fixed broken tags in configuration files or incorrect filepath references in build scripts.

Missing Code Change (165)

This category groups the pairs of commits (c_i, c_{i+1}) in which the remedy commit (*i.e.,* c_{i+1}) adds some missing code changes that should be introduced within previous commit c_i. We

divided those commits into two subcategories: *Commit Added/Deleted Files Missed in Previous Commit* and *Finalizing Code Change*.

The first subcategory is related to fixing a previous commit error. In this case, we are not referring to the code changes implemented in c_i, but to the commit process itself. This issue is mainly caused by an incorrect selection of committed files by the developer. Also, sometimes IDE cache issues can lead to a similar situation (*e.g.,* the IDE cached the wrong version of a committed file or lost track of some code changes during the git commit process). While this subcategory is kind of unrelated to artifacts' changes, it still provides hints for interesting research directions. ⚗ For example, approaches to automatically identify the set of files to commit can be designed to reduce the possibility of missing files or to include unrelated changes. This could also go further and recommend to the developer *when* to commit in such a way to avoid tangled commits [44] and committing cohesive sets of code changes. To the best of our knowledge, the only step in this direction has been done by Bradley *et al.* [142] with a context-aware developer assistant able to identify the files to push towards the repository when the developer asks. However, more automation can be envisioned, with approaches also able to (i) recommend when to commit (as previously said, to *e.g.,* avoid tangled commits), and (ii) summarize the changes in a meaningful commit message (as attempted by Jiang *et al.* [143]).

The second subcategory (*i.e., Finalizing Code Change*) refers to code changes forgotten or left incomplete for other reasons in commit c_i that are then finalized in c_{i+1}. This includes cases in which developers add new test cases needed to test the production code introduced in the previous commit, or to complete an implementation task. For example, in a commit of the `openpnp` project [213], the developer claimed in the commit message that three new sub-features were introduced. However, the developer forgot to actually implement one of those sub-features and added the missing implementation in the following commit. In another interesting case from the `geoserver` project [214], the developer introduced a guard clause in commit c_i to check if a processed reference is `null`. Meanwhile, a debugging message was also added saying that "*the reference is null, reset it to default value*". However, the actual implementation for resetting this reference value was missing in commit c_i, and implemented in the remedy commit c_{i+1}. ⚗ While these issues are of different natures, some of them can be spotted automatically through techniques able to compare what described in the commit message and what has been actually implemented in the code change. For example, in the previously discussed example [213], a misalignment between the number of sub-features actually implemented and claimed in the commit message could be spotted and reported to the developer.

Reverted Commit (58)

This category groups remedy commits c_{i+1} in which the developers revert the code changes they committed in the previous commit c_i. The reasons pushing a developer to revert previous changes through a remedy commit include: (i) introduced bugs spotted after pushing the changes in c_i; (ii) unintended changes, pushed in c_i by mistake; (iii) failing test cases, possibly indicating a bug worth of investigation before applying the c_i's changes. In all these

cases, developers prefer to quickly bring the code back to its previous state to double check the implemented changes and understand the causes for the (possible) introduced issues.

It is also worth mentioning that in many cases, we were not able to understand the reasons behind the reverted changes by manually inspecting the subject commits. These cases are just grouped in the root category *Reverted Commit*. Also, we observed that sometimes the code changes were reverted backward and forward within a few subsequent commits.

Our study is not the first one investigating reverted commits in software repositories. Shimagaki *et al.* [144] conducted a study to gain a better understanding of why commits are reverted in large software systems. They found that 1%-5% of the commits from the systems they studies are reverted and this number could be reduced by improving team communication and developers' awareness. However, in some cases, commits are reverted due to external factors (*e.g.,* requirement change by end-users, customers, or remote teams) and, in this case, they are difficult to avoid. Yan *et al.* [145] proposed a model to automatically identify commits that will be reverted in the future. They also found that the developer who performs the change is the most important predictive feature among the three they studied (*i.e.,* code change, developer, commit message). Besides the recommendations to developers already provided by Shimagaki *et al.* [144], the presence of reverted commits in the history of software systems is also relevant for the mining software repositories (MSR) research community. For example, it could be debated whether studies analyzing the change-proneness of code components (*i.e.,* how frequently code components are subject to changes in software repositories) — *e.g.,* [146, 147, 148] — should take into account commits that are quickly reverted or, as currently done, should consider them. The same applies for works using the history of changes implemented by developers as a proxy for the developers' experience — *e.g.,* [149, 150]. In Section 4.3 we present an empirical study aimed at assessing the impact of considering (or not) reverted commits for typical MSR data collection tasks.

Documentation (49)

Our last category groups remedy commits related to software documentation. These commits impact a number of documentation artifacts that represent the main subcategories (see Fig. 4.4), namely: release notes, licensing statements, code comments, commit messages, and readme files.

The errors fixed in release notes, licenses and readme files are mostly minor. For example, some commits just update the copyright year in a previously committed file. Also, the fixes of commit messages rarely happen, and are mostly due to adding a missing commit message for the code changes implemented in the previous commit. Also these cases are interesting for the MSR community. For example, approaches using pairs $\langle$*code changes implemented in a commit* c_x*, commit message of* $c_x\rangle$ to train models able to learn how to generate commit notes [143], could be negatively biased by commit messages in a commit c_{i+1} referring to changes implemented in c_i.

Other remedy commits are related to code comments. In some cases, developers documented the rationale for a code change implemented in the previous commit. This is the

case of commit [215] performed in the `jitsi` project. In a commit [216] they fix a bug due to the wrong generation of a message where they mistakenly set a value of a parameter to an empty string instead of a `null` value.

In the next commit [215] they add a comment to explain the otherwise non-trivial difference in the generated message.

Interesting is also the missed removal of Self Admitted Technical Debt (SATD) instances [151], meaning technical debt documented by developers in the code with comments such as `TODO` : ..., `TOFIX` : ..., etc. We found cases in which developers payed-back the technical debt instance, but forgot to remove the comment documenting the SATD. This resulted in a code-comment inconsistency [152], that could possibly confuse developers comprehending the associated code components. One representative example of this scenario is the commit [217] performed in the `apache/tinkerpop` project where the developers "*Forgot to remove todo from previous commit*", as their commit message says. Indeed, in the remedy commit they remove a single-line comment which says "*todo: need a test to enforce this condition*", and just right in the previous commit [218] they had implemented the missing test case, thus paying back the technical debt.

The cases discussed above for the *Documentation* category provide us with some interesting lessons learned. First, identifying code components in which specific types of comments (*e.g.,* to document the rationale for a given implementation and/or to detail the application logic) are needed, can be a promising research direction. Second, automatically classify SATD as payed-back (or not) can help in identifying obsolete and misleading comments in the code. We believe this is another interesting research direction for the software engineering community.

4.3 Study II: Impact of Reverted Commits on MSR Data Collection

4.3.1 Study Design

The *goal* of the study is to investigate the impact of reverted commits (one subcategory of quick remedy commits) on data collection activities performed in the context of MSR studies. The *purpose* is to show the level of noise introduced by reverted commits in MSR studies collecting specific types of data. Our study addresses the following research question:

> **RQ_2:** *What is the impact of reverted commits on data collection tasks when mining Java projects?*

We instantiate RQ_2 on two popular "data collection tasks", namely the identification of bug-fixing commits [41, 125, 126, 127, 128] and of refactoring operations [128, 129, 130, 131, 132, 133] performed in the change history of software systems. We show the impact of filtering-out (or not) reverted commits while mining this data (*e.g.,* a refactoring operation mined in the system's history in commit c_i may have been reverted in commit c_{i+1}, thus questioning its validity as a study data point). The results of our study help to increase the awareness about noisy data points introduced by reverted commits, eventually leading to a better handling of data processing in MSR studies.

Data Collection and Analysis

To answer RQ_2, we sorted the 1,497 projects used in the context of RQ_1 based on the number of commits in their change history impacting at least one source code (*i.e.,* Java) file. We discarded seven projects having more than 100k of such commits since the data extraction process for refactoring operations on these systems is too costly in terms of time. In particular, we run the data collection process described in the following for two weeks, processing in parallel up to ten systems at a time. At the end of these two weeks, the seven systems we excluded were still far from being processed. We replaced these seven systems with those ranked in positions 101-107, still selecting a total of 100 repositories as context for RQ_2. The list of considered projects is available in our replication package [135].

From each of the 100 selected projects we extracted the following information:

- **Bug-fixing commits.** To identify bug-fixing commits in open-source repositories, we mined lexical patterns in commits, as done in previous work [153]. In particular, we used the pattern defined by Tufano *et al.* [154], who reported a precision of 97.6% (*i.e.,* 97.6% of commits identified by this heuristic as bug-fixes are true positives): The commit message must match the patterns *("fix" or "solve") and ("bug" or "issue" or "problem" or "error")* to classify the related commit as a bug-fix.

- **Refactoring operations.** To mine the refactoring operations in the history of a system at commit level we used the state-of-the-art tool RefactoringMiner [155, 156]. If at least one refactoring operation is identified in a given commit, we mark this commit as a "refactoring commit" and store the refactoring-related information (*i.e.,* performed refactoring operations, code lines impacted by the refactoring).

- **Reverted commits.** Before detailing the procedure we adopted to identify reverted commits, it is important to clarify that, in our study, we only focus on identifying commits reverting Java code changes from the previous commit. This means that, as for our previous study, we are still in a scenario in which we are looking at pairs of commits c_i and c_{i+1}, with c_i being the reverted commit and c_{i+1} the reverting one. We implemented an approach similar to the one by Yan *et al.* [145]. First, we identify reverting commits by scanning commit messages, looking for the pattern *reverts commit* c_i. Second, to identify reverting commits c_{i+1} not explicitly labeled as such in their commit note, we compare the code they change with the one changed in the previous commit c_i. To do this, we stored the changes performed in each commit in a vector having the format: *AddedFile, DeletedFile, ModifiedFile, AddedCode, DeletedCode*. We refer to this 5-element vector as a commit change vector V, in which *AddedFile* indicates the added file paths, *DeletedFile* the deleted file paths, *ModifiedFile* the modified file paths, and *AddedCode* and *DeletedCode* refer to the text in the inserted lines and removed lines, respectively, with each line added together with a prefix of the changed file path. Given the commits c_i and c_{i+1}, we mark c_i as a reverted commit and c_{i+1} as a reverting commit if they satisfy all of the following constraints:

  - $addedFile_{i+1} = deletedFile_i$,

  - $deletedFile_{i+1} = addedFile_i$,
  - $modifiedFile_{i+1} = modifiedFile_i$,
  - $addedCode_{i+1} = deletedCode_i$,
  - $deletedCode_{i+1} = addedCode_i$.

- **Partially reverted commits.** Similarly to the identification of completely reverted commits, given two commits c_i and c_{i+1}, we mark c_i as a partially reverted commit and c_{i+1} as a partially reverting commit if they satisfy all of the following constraints:

  - $addedFile_{i+1} \subset deletedFile_i$,
  - $deletedFile_{i+1} \subset addedFile_i$,
  - $modifiedFile_{i+1} \subset modifiedFile_i$
  - $addedCode_{i+1} \subset deletedCode_i$,
  - $deletedCode_{i+1} \subset addedCode_i$.

Once extracted the above described data from the change history of the 100 selected projects, we compute the impact of considering/not-considering completely and partially reverted commits when collecting bug-fixes and refactoring operations from the change history of software projects. In particular, given a task $T \in \{refactorings,\ bugfixes\}$, we compute for each project the average number of noisy data points introduced by a single reverted commit in the following way:

$$\frac{|DataPoints_{T_{all}} - DataPoints_{T_{cleaned}}|}{|reverted|}$$

where $DataPoints_{T_{all}}$ represents the total number of data points collected for the task T (*i.e.,* in our case, number of bug-fixes or number of refactorings); $DataPoints_{T_{cleaned}}$ is the number of data points collected for the same task T when removing reverted commits; and $|reverted|$ is the total number of reverted commits identified in the repository. To make an example, in the case of T = mining of bug-fixing commits, a value for this metric of 0.5 indicates that every reverted commit introduces in the collected data, on average, 0.5 noisy bug-fixing commits. We compute the same metric when considering both reverted and partially reverted commits:

$$\frac{|DataPoints_{T_{all}} - DataPoints_{T_{cleaned'}}|}{|reverted| + |partiallyreverted|}$$

In this case, the only difference is that $DataPoints_{T_{cleaned'}}$ represents the number of data points collected for the task T when removing both reverted and partially reverted commits.

4.3.2 Results

We start by commenting on the number of fully and partially reverted commits we identified in the 100 systems. Overall, we found 5,083 reverted (avg=51, median=30, Q1=15, Q3=60) and 958 partially reverted (avg=10, median=7, Q1=3, Q3=13) commits. While the number of reverted commits is non-negligible, we only found a limited number of partially reverted commits, with a maximum of 44 observed for `apache/hbase`. Also, fully reverted commits are found in all repositories, while for the partially reverted ones we did not find any instance in six of the analyzed projects. Note that the number of reverted commits found in our study is substantially lower as compared to the data reported in the work by Shimagaki *et al.* [144] and Yan *et al.* [145], in which up to 5% of commits in a repo were found to be reverted. However, it is worth noting that in our study, differently from previous work, we only considered reverting commits c_{i+1} that revert changes in c_i (*e.g.,* we do not consider c_{i+1} as reverting commit if it reverts changes performed in c_{i-1}).

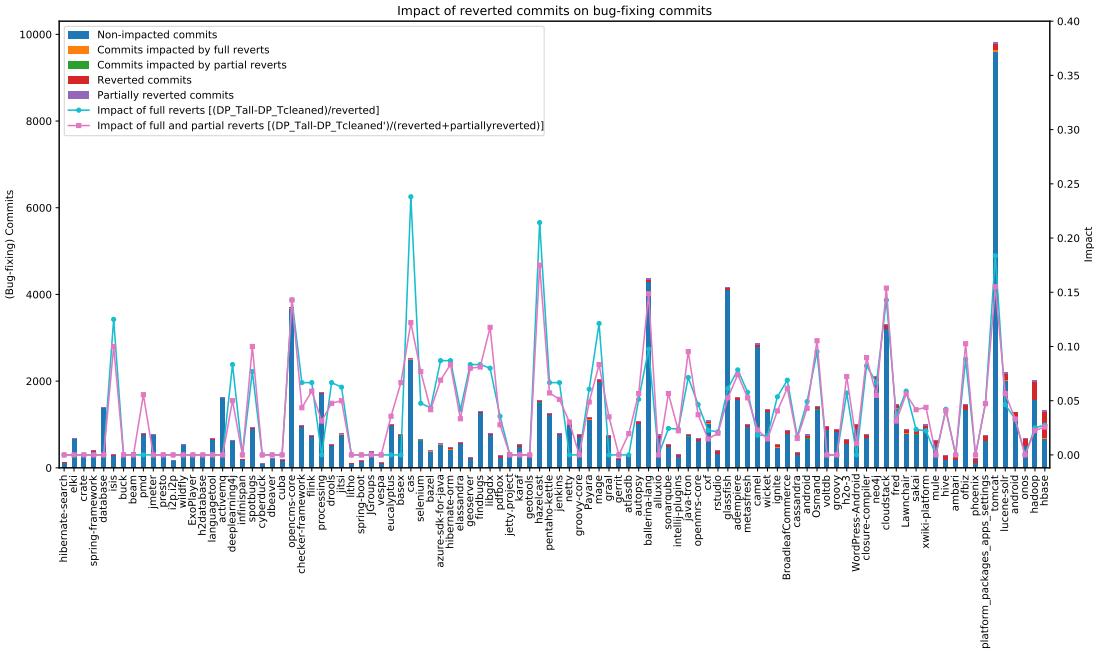


Figure 4.5. Impact of reverted commits on bug-fixing commits.

Fig. 4.5 shows the results achieved for the data collection task related to bug-fixing commits. The 100 projects are sorted from the left to the right in ascending order by the absolute number of completely reverted commits. For example, the first project on the left is `hibernate/hibernate-search` with only one reverted commit in its change history, while the last is `apache/hbase` with 617. The stacked bar chart shows the number of non-impacted bug-fixing commits (*i.e.,* commits that are not fully nor partially reverted)—blue bar, of fully reverted bug-fixing commits (orange bar) and of partially reverted bug-fixing commits (green bar), using the scale on the left y-axis. The partially reverted commits are hardly visible in

the chart due to their low number.

The line chart in Fig. 4.5 shows instead the average impact of fully reverted commits (cyan line) and of both fully and partially reverted commits (pink line) using the formulas presented at the end of Section 4.3.1. In this case, the reference y-axis is the one on the right. Since the number of partially reverted commits is very low, we limit our discussion to the impact of fully reverted commits on the collected bug-fixes. However, as it can be seen in the line chart in Fig. 4.5, the trend of the two lines is very similar.

Ignoring reverted commits from the data collection has an impact, in terms of collected data points, on 57 out of the 100 analyzed systems. The average impact goes from a minimum of 0.02 (*i.e.,* a reverted commit results, on average, in 0.017 "wrong" bug fixes collected) to a maximum of 0.24, with an average of 0.07 and a median of 0.06. The system resulting in the highest number of noisy data points for this task is `apache/tomcat`, in which the 147 reverted commits cause the collection of 27 reverted bug-fixes (on average, each reverted commit contributes 0.18 noisy data points).

We discuss a few examples of commits that were identified as a bug-fixing commit but had been reverted in the subsequent commit.

One commit of the `apache/hadoop` project was marked bug-fixing [219] as the log message said: "*Fix synchronization issues . . .* " The changes, however, were reverted by the next commit with the message " *Revert "Fix synchronization issues . . . " because forgot to add JIRA Number*." In this case, the reverted commit is indeed a bug-fixing commit, but the reverting commit should not be considered a valid bug-fix even though it contains the expression "Fix issues." In the worst case, a mining study might believe that there are already two bug-fixes in the change history after the revert. While in reality, the code does not implement the bug-fix after the revert.

In another commit of the `apache/tomcat` project [220], the author claimed in the commit message that a reported issue had been fixed. However, the fix was reverted in the subsequent commit as they noticed that "*it fixes the reported issue but introduces other issues*." Again, the fix was reverted, and the reverting commit should not be counted as a bug-fix.

Another interesting example can be seen in [221]. The bug-fix was reverted because the issue had been fixed before by someone else: "*Revert [. . .] Bug: 27700406" Framework bug was fixed by ag/900274, so this is no longer needed.*"

It is important to highlight that, while there is an impact of the reverted commits on the collected bug-fixes (and, as such, excluding them from the data analysis might be preferred), such an impact is overall limited. However, it is also worth reminding that in our study design we favored the precision in the identification of reverted commits rather than recall. Thus, the number of reverted commits we identify is certainly an underestimation of the real ones. Also, in case these reverted bug-fixes are used to compute additional data (*e.g.,* are provided as input to an SZZ algorithm as done in previous works [128]), such an error can further propagate and results in additional noisy data points. Basically, a cleaning of reverted commits when collecting bug-fixes is always desirable even though for specific study designs (*e.g.,* collection of bug-fixing commits for qualitative manual analysis) it might not be needed.

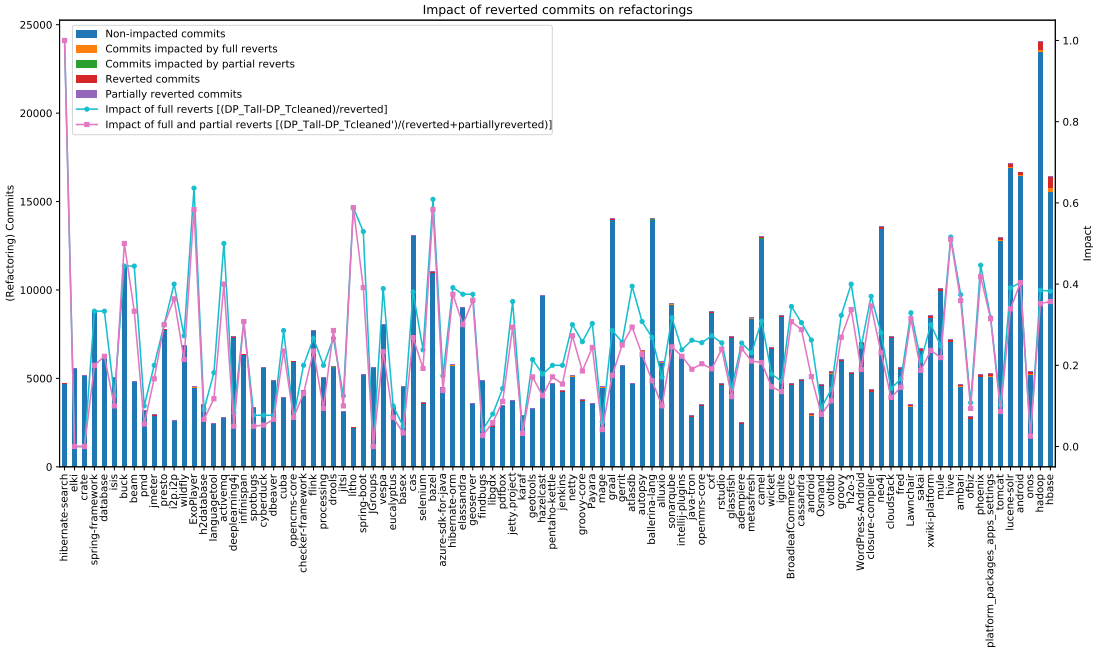


Figure 4.6. Impact of reverted commits on refactoring commits.

Fig. 4.6 shows the same data discussed before for the refactoring-related task, with the only difference that, in this case, the reverted and partially reverted commits are "refactoring commits", meaning commits featuring at least one refactoring operation. Also in this case we focus our discussion on the completely reverted commits.

Considering reverted commits during the data collection has an impact on 97 out of the 100 systems, with an average impact for a single reverted commit of 0.27 noisy data points (*i.e.,* reverted refactoring commits), median=0.26. The average impact goes from a minimum of 0.08 to a maximum of 1.00. The latter is a sort of outlier, since it refers to the `hibernate/hibernate-search` that, as said before, does only have one reverted commit that is indeed a refactoring commit.

In this case, the system that would be mostly affected by the presence of noisy refactoring commits collected when not handling reverted commits is `apache/hbase` with a total of 236 reverted refactoring commits that would be wrongly considered (result of the overall 617 reverted commits in this system).

An example of refactoring-related commit that has been reverted is the one commit performed in the `metasfresh` project [222]. The developer performed some refactoring operations (*e.g.,* rename parameter, change return type, rename method), but the commit message claimed that the refactoring was only partially. The subsequent commit reverted this partial refactoring. Thus, specific types of empirical studies mining refactoring operations may consider ignoring the refactorings detected in the first commit, since the refactorings were implemented and quickly reverted by the developer. In another commit performed in the `WordPress – Android` project [223], one of the private inner classes has been moved to

a public outer class through a move class refactoring. However, the author said that this refactoring was only for testing purpose and reverted the change in the subsequent commit.

As compared to the collection of bug-fix commits, reverted commits seem to have a higher impact when mining refactoring operations, with an overall of 1,447 reverted (noisy) refactoring commits that are identified across the 100 analyzed systems. Considering our conservative approach to identify reverted commits, we believe its cleaning is highly recommended when studying refactoring operations over the history of software systems.

4.3.3 Summing Up

Both the quantitative and qualitative results of this study point to an opportunity to obtain cleaner data by considering reverted commits: Reverted commits are noise in the recorded history of a system, and while it looks like a negligible phenomenon, we argue that the cleaner the data the better the analyses. In the spirit of the work by Kawrikow and Robillard [42] on cleaning out non-essential changes from any mining software repositories research, detecting and removing reverted commits could thus also become a part of the cleaning preprocessing before starting an actual analysis.

4.4 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to (Study I) the manual analysis we performed to identify the reasons behind the quick remedy changes performed by developers, and (Study II) the heuristics used to identify bug-fixing commits and reverted commits as well as to imprecisions introduced by the tool used to mine refactoring operations.

To mitigate subjectivity bias in the manual analysis (Study I), every commit was assigned to two researchers who manually analyzed it independently. Then, in the case of disagreement, a third researcher was assigned to the commit to solve the conflict. In addition to that, we used lexical patterns to identify candidate remedy commits. While these lexical patterns can return false positives, these have been excluded in our study through manual validation, and thus do not influence our findings.

Concerning Study II, the identification of bug-fixing commits was based on a heuristic defined and validated in previous work [154]. As for the reverted commits, we combined two types of heuristics based on the analysis of the commit message and of the code changes. Also, we limited the identification of reverted commits only to pairs of subsequent commits to increase the precision in our analysis. While this likely reduces the number of reverted commits we can identify (*i.e.,* recall), considering the analysis we performed (*i.e.,* assessing the average "cost" in terms of noisy data of a single reverted commit) our findings should not be substantially affected. Finally, refactoring operations have been mined by relying on the state-of-the-art tool RefactoringMiner [156].

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. One aspect could be related to the selection of projects being considered. As explained by Kalliamvakou *et al.* [124] mining GitHub

can be risky because projects may contain very few commits. To mitigate this threat, we applied strict criteria (*i.e.,* more than 500 commits, more than ten stars) when selecting the context of our study. Also, we manually looked into the set of retrieved projects to exclude repositories that do not represent real software systems (*e.g.,* tutorials, collections of code examples) and forked projects. Also, in Study I all considered data points (*i.e.,* commits) have been manually checked, strengthening its internal validity.

Threats to *external validity* concern the generalizability of our findings. Our analysis in Study I is limited to a specific set of 500 commits we randomly selected as the output of a keyword-based mechanism used for the pre-selection of commits likely to be "remedy" commits. Because of this procedure, our taxonomy inevitably omits types of remedy commits we did not analyze and/or documented in diverse data sources. Also, we set a 5-minute threshold to identify the *quick remedy commits* subject of our study. While our choice is justified by the temporal distribution plotted in Fig. 4.2, changing this threshold value may result in different findings. This investigation is part of our future research agenda.

As for Study II, the reported findings are related to a set of 100 Java open source projects, which do not allow us to generalize our results to projects written in other languages which require additional investigations.

4.5 Conclusion

We presented two empirical studies related to *quick remedy commits*. In the first, we qualitatively investigate *quick remedy commits* performed by developers in GitHub projects. We defined *quick remedy commits* as commits performed by developers to remedy changes omitted or errors introduced in a previous commit, performed just a few minutes before. This study (Study I) is based on the manual analysis of 500 commits, that we classified by looking at the objective of the remedy commit. The output of this study is represented by the taxonomy depicted in Fig. 4.4. We used several qualitative findings to distill lessons learned resulting in actionable items for both researchers and practitioners.

Then, we investigated the impact of a specific type of quick remedy commits, namely reverted commits, on the data extracted for MSR studies. In particular, we focused on two data collection tasks performed in many previous works: (i) the identification of bug-fixing commits and (ii) the mining of refactoring operations over the change history of a system. Our analysis disclosed the amount of potential noise brought by reverted commits for these two data collection tasks.

5

Using Code Change Patterns for Code Recommendations

Code completion is one of the killer features of Integrated Development Environments (IDEs), and researchers have proposed different methods to improve its accuracy. While these techniques are valuable to speed up code writing, they are limited to recommendations related to the next few tokens a developer is likely to type given the current context. In the best case, they can recommend a few APIs that a developer is likely to use next.

In this chapter, we present FeaRS, a novel retrieval-based approach that, given the current code a developer is writing in the IDE, can recommend the next complete method (*i.e.,* signature and method body) that the developer is likely to implement. To do this, FeaRS exploits "implementation patterns" (*i.e.,* groups of methods usually implemented within the same task) learned by mining thousands of open source projects. We instantiated our approach to the specific context of Android apps. A large-scale empirical evaluation we performed across more than 20k apps shows encouraging preliminary results, but also highlights future challenges to overcome.

5.1 Introduction

Developing high-quality software while reducing time-to-market are two classical contrasting objectives in the software industry. This translates into the need for increasing the productivity of software developers, by lowering their learning curves when dealing with unfamiliar code, and by maximizing the quality of the code they write. In response to these needs, researchers have proposed recommender systems for software engineering, defined by Robillard *et al.* as "applications that provide information items valuable for a software engineering task in a given context" [8].

Some recommender systems pursue a long-lasting dream of software engineering research: The (semi-)automatic generation of source code. The goal of these tools is speeding up the implementation of new code. Code completion techniques are nowadays one of the killer features of IDEs [157]. Researchers have proposed different methods to improve code

completion accuracy and, more in general, its capabilities [7, 9, 10, 11, 12, 13, 14]. While these approaches are certainly valuable to speed up code writing, they are limited to recommendations related to the next few tokens a developer is likely to type given the current context. In the best case, they can recommend a sequence of APIs that a developer is likely to use next [11, 14].

We aim at reaching the next level in supporting developers during the writing of new code. We present FeaRS, an approach and an IDE plugin which monitors the code written by Android developers in the IDE and is able to recommend the complete code of the next method (*i.e.,* signature and method body) they are likely to implement based on method(s) they already have implemented.

FeaRS relies on a set of implementation patterns that we built by mining 20,713 open-source Android apps available on GitHub. To give a concrete example, the code snippet in Fig. 5.1 implements an options menu in an Android app. To perform such a task, tutorials recommend as first step to inflate the menu in the `onCreateOptionsMenu(...)` method and, then, to handle the item selection in the `onOptionsItemSelected(...)` method. Assuming the existence of this implementation pattern in several apps, FeaRS can learn it and recommend the implementation of `onOptionsItemSelected(...)` once `onCreateOptionsMenu(...)` has been implemented by the developer.

```
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.my_options_menu, menu);
    return true;
}

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.about:
            startActivity(new Intent(this, About.class));
            return true;
        case R.id.help:
            startActivity(new Intent(this, Help.class));
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Figure 5.1. An implementation pattern in Android

We analyzed 2,721,800 commits performed during the history of the subject apps to identify new methods that are implemented within the same commit. This results, for each analyzed commit c_k, in a set $M_k = \{m_1, m_2, \ldots, m_n\}$ of n new methods created in c_k. By extracting this information for thousands of commits, we can identify implementation patterns repeatedly followed by Android developers, *e.g.,* the implementation of m_1 could imply the implementation of $m_2, \ldots, m_n$. We refer to m_1 as the Left-Hand Side (LHS) of the pattern

and to $m_2, \ldots m_n$ as the Right-Hand Side (RHS).

The identification of these implementation patterns is far from trivial. Indeed, two commits c_k and c_j performed in two different repositories may implement different sets of new methods (*e.g.,* $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that, however, represent the same implementation pattern (*i.e.,* $m_1 = m_3$ and $m_2 = m_4$). Recognizing this situation is necessary to identify groups of methods that are repeatedly implemented together in different commits/apps, and not just by chance in a single/few commit(s).

FeaRS identifies clusters of methods likely to implement the same feature in the overall set of mined added methods. Going back to the previous example, this means that m_1 and m_3 are assigned to the same cluster C_1, and m_2 and m_4 to C_2. This results in the flattening of c_k and c_j to the same implementation pattern (*i.e.,* $M_k = M_j = \{C_1, C_2\}$). Once this processing is done for all mined commits, FeaRS applies association rule discovery [158] on all commits, thus creating the set of implementation patterns it relies on.

When monitoring the code written by a developer in the IDE, FeaRS identifies newly written methods and assigns, if possible, each of them to one of the clusters created in the previous step. Then, it checks if an implementation pattern having one or more of the newly implemented methods as LHS is available and, in case a pattern is found, the corresponding RHS is triggered as a recommendation to the developer.

We evaluated FeaRS in a study in which we simulated its usage in the change history of the same 20,713 apps we used to extract the implementation patterns. We used the first 80% of the apps' histories to extract the implementation patterns, the subsequent 10% to tune the FeaRS's parameters, and the last 10% to assess its performance (*i.e.,* test set). For each commit c in the test set, we simulated the scenario in which a developer implemented a subset S of the new methods added in c and used FeaRS to generate recommendations using S as LHS. Then, in case a recommendation is generated, we check if the RHS corresponds to one of the methods actually implemented in c and not part of S.

The achieved results show the feasibility of our approach, but also its strong limitations. Indeed, while FeaRS is able to generate meaningful recommendations for thousands of methods, several of them concern small methods that are not expected to substantially boost the developer's productivity.

Structure of the Chapter

Section 5.2 thoroughly describes the technical details of FeaRS, the core technique behind the system. Section 5.3 reports the design of the study we performed to build FeaRS by mining open source repositories and assess its performance, with Section 5.4 presenting the corresponding results. After the discussion of threats to validity (Section 5.5), Section 5.6 concludes this chapter.

5.2 FeaRS

Fig. 5.2 depicts the inner working of FeaRS.

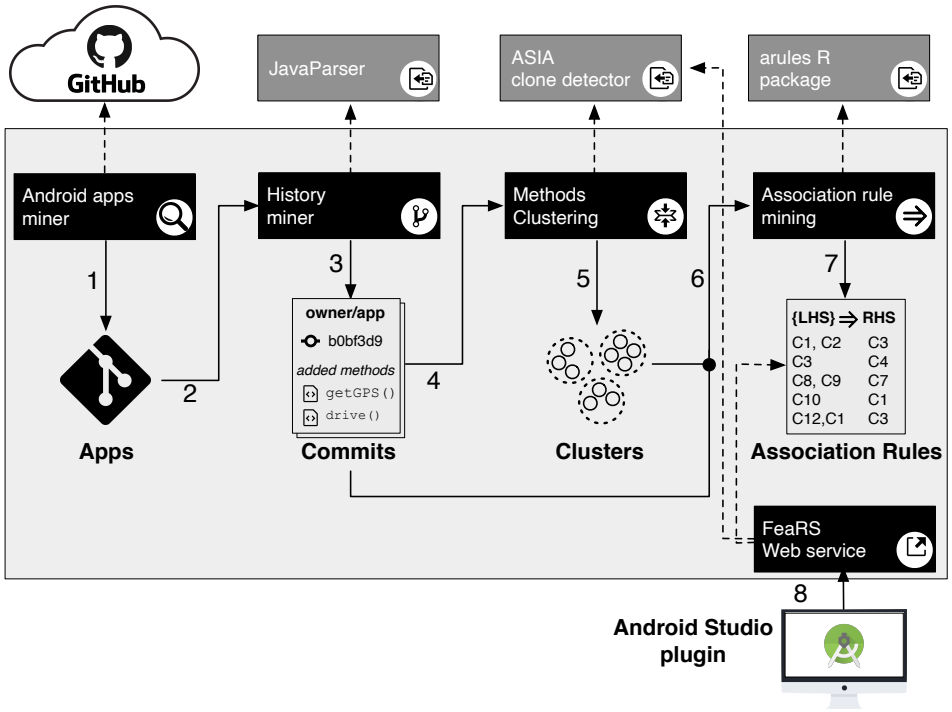


Figure 5.2. The FeaRS pipeline

The black boxes represent components that we developed; the grey boxes depict external tools we reused and/or adapted.

All components except the Android Studio IDE plugin reside on a central server providing an access point via the *FeaRS Web service*. Steps 1-7 are executed offline, and continuously mining existing or newly created Android apps to always learn new implementation patterns. Step 8 is executed every time the developer completes the implementation of a new method. We released FeaRS as an open source project [159].

5.2.1 Mining Android Apps

The *Android apps miner* identifies GitHub repositories related to Android apps. Their history is then analyzed to identify methods implemented within the same commit. We use the GitHub APIs to search for repositories satisfying the following criteria:

They are written in Java. While Android is transitioning to Kotlin as the official language, the majority of Android apps is still written in Java [160]. Note that while we instantiated FeaRS to the specific problem of recommending complete methods for Java Android apps, all the steps in Fig. 5.2 can be customized to any programming language.

They are Android apps. We ensure that the repository contains a `build.gradle` file with an explicit dependency towards the Android SDKs, indicating the usage of the Gradle build system, the default choice in Android Studio.

They have a non-trivial change history. We excluded apps with *less than 100 commits* since we are interested in identifying the new methods added by developers within the same commit.

5.2.2 Identifying Methods Added in Commits

The set of cloned repositories is provided as input to the *History miner* (step 2 in Fig. 5.2). This component extracts the list of commits performed in all branches of each repository by using the `git log --topo-order` command. This command allows analyzing all branches of a project without intermixing their history, avoiding unwanted effects of merge commits.

History miner uses JavaParser [161] to extract, from the Java files added or modified in each commit, the AST nodes which represent the callable declarations (*i.e.,* methods and constructors). In particular, we are interested in the callable declarations added in each commit. Commits not implementing at least two new methods and/or constructors are excluded at this stage, since we want FeaRS to learn implementation patterns in the form of $\{M\} \implies m_i$, where M represents a set of one or more methods and m_i a method that FeaRS can recommend based on the fact that the developer implemented M. Thus, assuming M to be a singleton, at least two new methods must be implemented in a commit (*i.e.,* the one in M and m_i) to make it useful for learning. We excluded commits adding more than 10 new methods, since these are likely to be tangled commits not representative of any specific implementation pattern [162].

These commits processed in this stage are provided as input to the module in charge of the methods clustering (step 4 in Fig. 5.2).

5.2.3 Clustering Similar Methods

To identify recurring implementation patterns in the considered commits, FeaRS applies clustering to group methods added in different commits, possibly from different systems, that implement equivalent or very similar functionalities. Two commits c_k and c_j performed in two different repositories may implement different sets of new methods (*e.g.,* $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that represent the same implementation pattern (*i.e.,* $m_1 = m_3$ and $m_2 = m_4$). FeaRS can identify, through association rule discovery, that these sets of methods represent a repetitive implementation pattern.

FeaRS builds a weighted undirected graph. Each method added in any of the commits is considered as a node. The weight on the edges connecting each pair of nodes represents the similarity between the two corresponding methods. To assess similarity we use the publicly available ASIA clone detector [112], since it (i) is designed to capture the similarity between two Android methods; and (ii) returns as output an easily interpretable value from 0 (min similarity) to 1 (max). We customized the ASIA similarity algorithm in two ways.

First, in the original implementation all terms in the two methods to compare are lowercased before computing their textual similarity. This is suboptimal in FeaRS, since high

precision in the identification of related methods is fundamental.

Experiments revealed that the similarity of methods is artificially boosted by lowercase transformation: Given two methods m_1 and m_2, it happens that a term appearing in the name of m_1 (*e.g.,* `date`) is matched with the type of an object appearing in m_2 (*e.g.,* `Date`). By not transforming `Date` to lowercase, the presence of these two terms does not influence positively the similarity between m_1 and m_2.

Second, while ASIA uses tf-idf (term frequency-inverse document frequency) as a weighting schema for the terms during the textual similarity computation, we only employ term frequency, because we noticed that a single term appearing in both methods and having a very high idf (*i.e.,* being very rare in the corpus) can result in a high similarity between the two methods, even if they implement completely different features. This is especially true in small methods, due to the low number of terms present in them and the strong impact a single shared term can have on their similarity.

We prune all edges with a weight below a threshold λ (λ will be tuned in our evaluation). This creates a set of disconnected subgraphs, each one representing a cluster of methods implementing strongly related functionalities. Within each subgraph (*i.e.,* cluster) we identify the cluster centroid: the method with the highest number of edges, which serves as representative for that cluster. The centroid is used later on by the FeaRS Web service when interacting with the IDE plugin.

5.2.4 Association Rule Mining

This module takes as input the list of commits generated by the *History miner* and the clusters output of the previous step (step 6 in Fig. 5.2) and creates a text file reporting in each line a set of methods added in the same commit and in the same file, using the cluster they belong to. For example, assuming a commit adding three methods m_1, m_2, and m_3 to a file F_i, and those methods being assigned to clusters C_{12}, C_8, and C_{71}, respectively, a line C_{12}, C_8, C_{71} will be added to the file. We decided to split methods added in the same commit but in different files to extract more "cohesive" association rules, and to avoid learning recommendations that span different files (*i.e.,* the developer is working on F_i and FeaRS recommends a method to add in F_j).

FeaRS analyzes the created file using Association Rule Mining [163] to identify implementation patterns, relying on the *R* `arules` package. The output is a set of association rules in the form {LHS} $\implies$ RHS, where the LHS can be composed by one or more methods, while the RHS always has a single method. This means that FeaRS can only recommend the next method to implement given the one(s) already implemented by the developer.

There are three parameters that we tune in our evaluation: minimum *support* (*sup*), *confidence* for the mined rules (con), and maximum size of the LHS (max_{LHS}).

The support (*sup*) indicates how frequently a rule is observed in the dataset and, in our case, represents the percentage of analyzed commits that contains the specific rule.

The confidence (con) assesses how often a given rule is actually true in the dataset. Given a rule {LHS} $\implies$ RHS, it is computed as the number of commits implementing in the same file all methods in the LHS and RHS divided by the number of commits implementing the

LHS in the same file (with or without the RHS). Finally, we also tune the maximum size of the LHS (max_{LHS}).

5.2.5 FeaRS Web Service and IDE Plugin

Fig. 5.3 shows the FeaRS Android Studio IDE plugin.

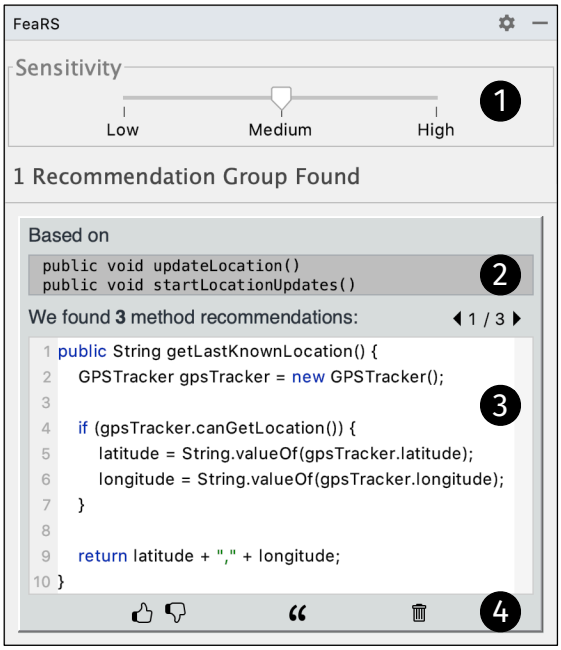


Figure 5.3. The FeaRS Android Studio plugin

The plugin interacts with the server through the Web service (step 8 in Fig. 5.2). The developer can start and stop FeaRS through simple ▶ and ■ icons in the IDE toolbar. By clicking ▶, FeaRS starts monitoring the code written by the developer and identifies when a new method is added. When this happens, the text of the new methods added by the developer since she pressed the start button is sent to the Web service.

The Web service identifies, for each received method, the cluster it belongs to. Our customized version of the ASIA clone detector computes the similarity between each received method and each centroid representative of the computed clusters. The similarity s for the most similar centroid is compared against a γ threshold (the fifth and last FeaRS parameter to tune): If $s > \gamma$, the method is assigned to the cluster represented by the most similar centroid, otherwise no match is found and the method is discarded.

All combinations of received methods that are matched with a centroid are used to generate different LHSs. For example, if three methods added by the developer are matched to clusters C_1, C_2, and C_3, we generate 7 possible LHSs: $\{C_1\}$, $\{C_2\}$, $\{C_3\}$, $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$, and $\{C_1, C_2, C_3\}$.

FeaRS checks if any of these LHSs is equal to the LHS of one of the association rules

previously extracted. In case of a match, a recommendation is generated. In the reported example, if $\{C_1, C_2\}$ is matched in a rule $\{C_1, C_2\} \implies C_9$, then the centroid of cluster C_9 is returned by the Web service to the plugin as a recommendation. For the same LHS several different RHSs may be recommended. The matching of the LHS of two rules can lead to redundant recommendations. In the example, let us assume that two rules are matched, one with $\{C_1\}$ and one with $\{C_1, C_2\}$ as LHS, and that both of them have C_9 as RHS. In this case, the Web service returns the centroid of C_9 reporting that it is recommended based on the LHS belonging to the rule having the highest confidence.

The generated recommendations are shown in the IDE as depicted in the bottom part of Fig. 5.3. ② shows the signatures of the methods implemented by the developer that are part of the LHS of the association rule used to recommend the method shown in ③ (*i.e.,* RHS of the rule). In case several recommendations share the same LHS, the plugin displays them as one recommendation allowing developers to switch between different RHSs using the arrow buttons above ③. The buttons at the bottom of the code snippet ④ allow to: (i) provide a feedback reporting if the recommendation was useful; (ii) copy the snippet; and (iii) delete the recommendation. The feedback, in our current implementation, is stored but not used. We plan to use it in future to adjust the confidence of the recommendations. If the developer decides to copy the snippet, a comment documenting the GitHub repository from when the snippet has been taken is added to the code, so that the developer can check its reusability from a legal perspective.

The slider at the top of the plugin GUI ① allows the developer to customize the "chattiness" of the plugin on three different levels. *Low*, *Medium*, and *High sensitivity* are three different FeaRS configurations that resulted from the calibration of its parameters presented in Section 5.4.1. By moving the slider towards *Low*, FeaRS becomes more strict and generates fewer, but higher quality, recommendations, while the opposite holds for *High*.

5.3 Study Design

The goal of this study is to assess the performance of FeaRS when used to recommend the next method to implement given one or more (already implemented) methods as input. It thus addresses the following research question:

RQ_1: *What is the accuracy of FeaRS in recommending complete methods in the context of Android apps?*

5.3.1 Context Selection and Data Collection

To conduct the empirical study for addressing the research question, we target to build a "static" version of FeaRS and assess its performance from mining a limited number of Android apps. Besides satisfying the criteria mentioned in Section 5.2.1, we also excluded apps having *more than 1,000 commits* while running the mining step. In this study, the *Android apps miner* was executed once, and identified 20,713 GitHub repositories. The set of apps that we use in this study is available in our replication package [164].

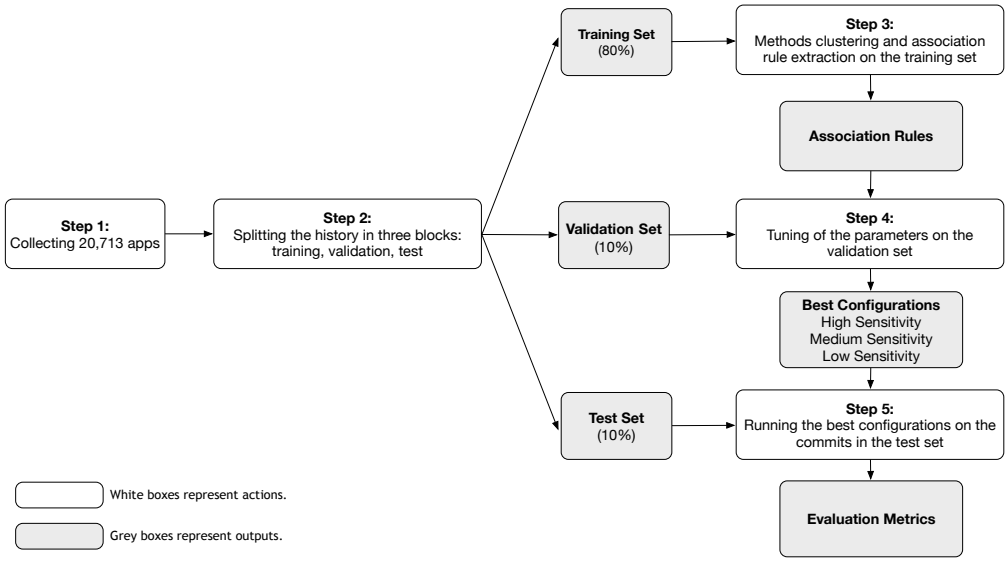


Figure 5.4. Study Design

Fig. 5.4 overviews the steps in our experimental design. We exploit the dataset of 20,713 Android apps as the context of our study. Then, we split such a dataset into three blocks namely training, validation, and test. Fig. 5.5 depicts how we create and use these three sets in our study.

The black arrows represent the change history of the apps considered in our study. Note that the history of the apps is not aligned, meaning that not all the apps exist in the same time period. The vertical dashed lines show how we divide the change history of the apps.

We use the first 80% to extract the association rules used by FeaRS to generate recommendations. We refer to this subset of the history as the "training set." The subsequent 10% is used to tune the parameters of FeaRS to identify the best configurations (*i.e.,* "validation set"), which are used to generate recommendations on the "test set" (*i.e.,* the last 10%), with the goal of assessing the performance of FeaRS.

One important clarification: We do not use the first 80% of each repository as the training set, due to the misalignment of the mined change histories. Instead, given d_s the date of the oldest commit present in all analyzed apps and d_e the date of the most recent commit, we take the first 80% of the time interval going from d_s to d_e as training set. As shown in Fig. 5.5, this may result in some apps exclusively contributing to the training set (or to the validation/test sets).

However, such a design is needed to avoid using "data from the future" when generating recommendations for the validation and test set and, thus, to simulate a real usage scenario for FeaRS. Indeed, by selecting the first 80% of the history of each app to learn the association rules, it could happen that a given App_x has the last commit of training set made on date d_x, while for App_y the latest commit of its entire history comes on date d_y, with $d_y < d_x$ (*i.e.,* d_y is older than d_x). This would mean that association rules learned on d_x will be applied to generate recommendations for commits performed on date d_y (that will be part of the test

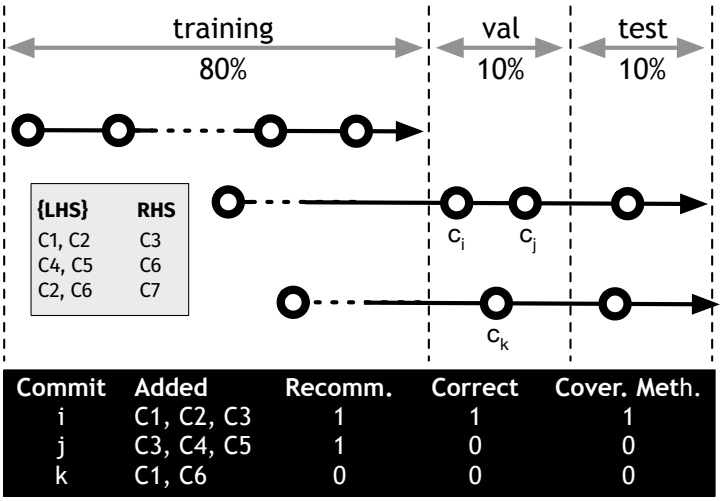


Figure 5.5. Data splitting and processing

set), thus using data from the future to learn how to trigger recommendations, something that cannot happen in a real usage scenario.

Table 5.1. FeaRS parameters tuning options

Parameter	Experimented values
con	0.05, 0.20, 0.35, 0.50, 0.65, 0.80
sup	8.00E-06, 4.80E-05, 8.80E-05, 1.28E-04, 1.68E-04
λ	0.80, 0.85, 0.90, 0.95
max_{LHS}	1, 2, 3, 4, 5, 6, 7, 8, 9

Once the association rules are learned, we assess the performance of FeaRS on the validation set with different parameter configurations (Table 5.1), for a total of 1,080 configurations. Given the number of mined commits, the minimum value of sup we experiment (*i.e.,* 8.00E-06) ensures that an association rule is learned from at least 5 commits to be considered valid.

In all combinations of parameters, we used $\gamma = \lambda$, meaning that the minimum similarity needed to cluster two methods together (*i.e.,* λ) is also the minimum similarity used when generating recommendations to assign a newly implemented method m to a cluster C (*i.e.,* γ, see Section 5.2.5).

As shown in Fig. 5.5, to identify the best configuration(s) we use 10% of the apps change history (validation set).

For each commit in the validation set (c_i, c_j, and c_k in Fig. 5.5) we match all newly added methods to the clusters that have been defined during the association rules extraction from the training set (using the same similarity threshold as for the clusters definition). This means that we simulated the scenario in which each of the added methods is written by the developer in the IDE, and the FeaRS plugin checks if the added method can be matched with

any of the existing clusters (*i.e.,* if its similarity with one of the centroids is higher than γ). If a method is not matched, no further action is taken, while all matched methods are assigned to the corresponding cluster.

Fig. 5.5 represents our running example, in which the grey box on the left shows the association rules learned on the training set, and the black box at the bottom shows how performance is computed for each commit in the evaluation set. In the case of commit i, three added methods have been matched to clusters C_1, C_2, and C_3. Then, we compute all possible combinations of the matched clusters involving all but one of them. In the case of commit i, this means all possible combinations having length lower than three: $\{C_1\}$, $\{C_2\}$, $\{C_3\}$, $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$. Then, we check if any of those combinations match the LHS of one of the rules learned from the training set. In Fig. 5.5 the pair $\{C_1, C_2\}$ matches the rule $\{C_1, C_2\} \implies C_3$. This means that, assuming C_1 and C_2 to be written before C_3 (more discussion on this assumption in our threats to validity), FeaRS would be able in a real usage scenario to successfully recommend the next method to implement (*i.e.,* the C_3 centroid). Thus, in Fig. 5.5, we count the number of recommendations generated by FeaRS (1), column "Recomm.", the number of correct recommendations (1), and the number of methods added in commit i that FeaRS would have potentially been able to recommend (1 out of 3), column "Cover. Meth." Concerning commit j, it would match the rule $\{C_4, C_5\} \implies C_6$ generating one wrong recommendation (see Fig. 5.5). No recommendation would be triggered for commit k, since no matched rules are found.

There are two special cases that must be handled:

First, when multiple association rules have the same RHS (*e.g.,* assume $\{C_1\} \implies C_3$ and $\{C_2\} \implies C_3$ are both available in the set of learned association rules). In this case, both rules could be applied, for example, in the context of commit i in Fig. 5.5. However, considering both rules as successful would inflate the performance of FeaRS since, in a real usage scenario, if $\{C_1\} \implies C_3$ is applied, $\{C_2\} \implies C_3$ cannot be applied, since C_3 already exists.

Second, in case of a "circular dependency" between the LHS and the RHS of two rules, *e.g.,* $r_1 = \{C_1\} \implies C_3$ and $r_2 = \{C_2, C_3\} \implies C_1$. The LHS of r_1 matches the RHS of r_2, and the RHS of r_1 is contained in the LHS of r_2.

In theory both rules could be applied to commit i in Fig. 5.5, but the application of one rule would exclude the other in a real usage scenario. If we apply r_1, it means that C_1 has been implemented by the developer and it does not make sense to recommend it with r_2. Similarly, if r_2 is applied, this means that C_3 already exists, making r_1 useless.

In both cases we select the rule with the highest confidence.

5.3.2 Data Analysis

We assess the performance of each experimented configuration by computing the following metrics:

Recall: $recall = \frac{Comm_{cor}}{Comm_v}$, where $Comm_{cor}$ is the number of commits for which FeaRS generated at least one *correct* recommendation and $Comm_v$ is the set of commits mined in the validation set. A correct recommendation is not necessarily an exact match to the

actual implemented code, but the similarity has to be above a certain threshold which is consistent with the predefined clusters. Recall indicates in how many commits FeaRS could be potentially useful for developers.

Precision: $precision = \frac{Comm_{cor}}{Comm_{rec}}$, where $Comm_{rec}$ is the number of commits for which FeaRS generated at least one recommendation (correct or wrong).

Cov$_{\text{commits}}$: $cov_{commits} = \frac{Comm_{rec}}{Comm_{v}}$. This metric indicates the percentage of commits from the validation set that could have triggered FeaRS to generate at least one recommendation (correct or wrong) for developers.

Cov$_{\text{meth}}$: $cov_{meth} = \frac{Meth_{cor}}{Meth_{Comm_{v}}}$, where $Meth_{cor}$ is the number of methods successfully recommended by FeaRS and $Meth_{Comm_{v}}$ is the total number of methods added in $Comm_{v}$. This coverage metric indicates the percentage of methods added in all commits from the validation set that could have been automatically generated by FeaRS.

#Recom: $\#recom$ is the number of recommendations generated by FeaRS in a commit for which it was triggered. We report both the mean and the median values.

Dist$_{\text{tokens}}$: $dist_{tokens}$ is the distance in number of tokens that must be modified, added or deleted by a developer when they receive a correct recommendation from FeaRS, which does not imply an exact match with the code actually implemented by the developer. Thus, we assess the effort needed by developers to adapt the received recommendation to their codebase (an example computation of such a metric is shown in Fig. 5.6).

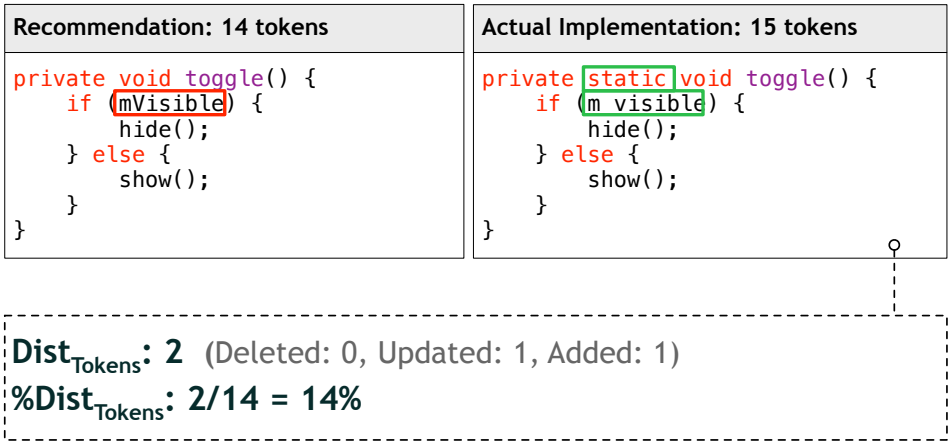
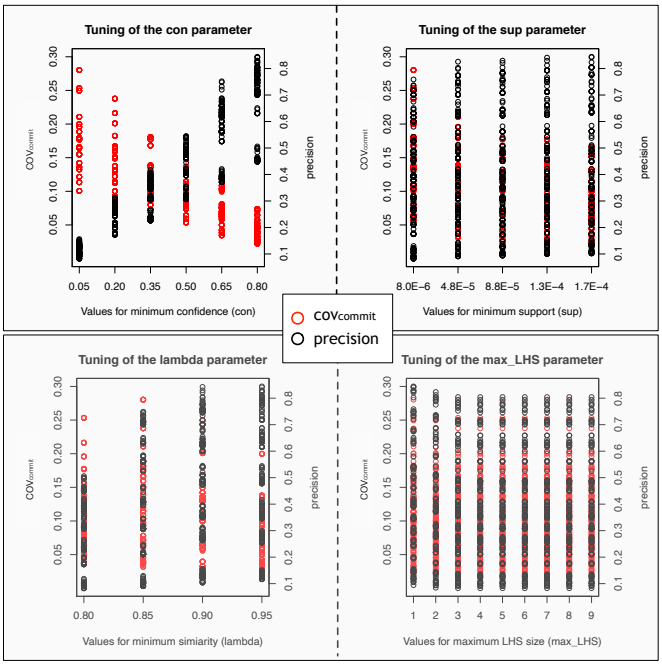


Figure 5.6. An example of $dist_{tokens}$ calculation



Selected Configurations

High Sensitivity	
Con = 0.50	sup = 8.0E-6
λ = 0.90	*maxLHS* = 1
Precision = 0.52 COVcommit = 0.13 Correct recommendations: 8,355	

Medium Sensitivity	
Con = 0.65	sup = 8.0E-6
λ = 0.90	*maxLHS* = 1
Precision = 0.64 COVcommit = 0.10 Correct recommendations: 7,092	

Low Sensitivity	
Con = 0.80	sup = 8.0E-6
λ = 0.90	*maxLHS* = 2
Precision = 0.72 COVcommit = 0.06 Correct recommendations: 5,801	

Figure 5.7. Tuning of FeaRS's parameters

5.4 Results Discussion

5.4.1 FeaRS Parameters Tuning

Fig. 5.7 shows the results of the parameters tuning performed on the validation set. Each of the four graphs reports on the x-axis the values experimented for a specific parameter; from left to right: minimum confidence (con), minimum support (sup), minimum similarity to cluster two methods (λ), and maximum size of the LHS (max_{LHS}). The y-axis reports the $cov_{commits}$ (left) and the precision (right) achieved, with red dots indicating $cov_{commits}$ values, and black dots precision values. We decided to use these two metrics, over the others, for the parameters tuning since we wanted to contrast the talkativeness of our tool (*i.e.,* in how many commits it generates a recommendation) against the precision of the generated recommendations. To better understand what the black and red dots represent, consider the con graph when its value is set to 0.05. The dots plotted in correspondence of this value represent the performance achieved when fixing $con = 0.05$ and varying all other parameters.

One first observation is related to the range of performance achieved by different configurations: The $cov_{commits}$ varies from 0.02 to 0.28, while the precision from 0.08 to 0.84. While the values of $cov_{commits}$ may look low, it is important to note that the validation set includes 70,562 commits.

The trends observed for the four parameters indicate that con has the strongest influence on performance. When the minimum confidence needed to trigger a recommendation grows,

as expected the precision linearly increases with a corresponding linear decrease of recall (left part of Fig. 5.7). Setting con lower than 0.50 does not ensure acceptable precision.

Concerning sup, increasing its minimum value does not substantially increase precision while having a strong negative effect on $cov_{commits}$. Low values of this parameter are preferable. Instead, increasing the λ parameter results in a notable increase in precision, especially when moving from 0.80 to 0.90/0.95. In this case, 0.90 seems to be a good compromise, also considering the minor loss of $cov_{commits}$ as compared to lower values. Finally, the max_{LHS} does not play a big role in the performance of FeaRS. As the output of this tuning process, we identified three configurations that we linked to the sensitivity bar in our IDE plugin and that are shown in the gray boxes at the right of Fig. 5.7.

These configurations have been picked using the following process. We started from the assumption that a precision level below 0.50 (*i.e.,* one out of two generated recommendations is correct) is not acceptable. Then, we picked as a *high sensitivity* configuration the one ensuring a precision of at least 0.50 and having the highest $cov_{commits}$. This configuration is able to generate 8,355 correct recommendations in the validation set, with a precision of 52%. Then, we increase the minimum acceptable precision by 10%, identifying the configuration ensuring at least a 60% precision with the maximum $cov_{commits}$. This resulted in the *medium sensitivity* configuration, that can successfully recommend useful methods in 7,092 cases, with a precision of 64%. Finally, a further increase of the precision level to at least 70%, led to the identification of the *low sensitivity* configuration, that can recommend 5,801 correct methods, with a precision of 72%. These three configurations are the ones we experiment with.

5.4.2 Quantitative Results

Table 5.2 reports the results achieved by the three FeaRS's configurations on the test set. The top part of the table reports the raw data used to compute the performance metrics in the bottom part of the table. In the top part, while "#commits w. corr. recomm." indicates the number of commits with at least one correct recommendation, "#corr. recomm." represents the number of correctly recommended methods, possibly more than one per commit.

The results achieved by the three configurations are in line with what we observed on the validation set: precision goes from 0.50 (*high sensitivity*) to 0.72 (*low sensitivity*), with recall moves in an inverse direction, decreasing from 0.07 (*high sensitivity*) to 0.04 (*low sensitivity*).

The recall values, while low, still correspond to thousands of methods correctly recommended. As we learned while performing the qualitative analysis in Section 5.4.3, a correct recommendation does not imply a "useful" recommendation. We noticed that many of the correct recommendations are due to small methods (*e.g.,* a getter method triggers the implementation of the corresponding setter), and decided to re-compute the performance of FeaRS only considering recommended methods with at least four lines of code (including signature but excluding annotations and the closing brace). To correctly compute recall, this also required us to exclude from our analysis the commits in which a successful recommendation would not be possible at all, due to the absence of newly implemented methods

Table 5.2. Performance when considering all methods

	high sensitivity	**medium sensitivity**	**low sensitivity**
#commits	69,480	69,480	69,480
#added methods	219,331	219,331	219,331
#commits w. recomm.	8,757	6,447	4,116
#commits w. corr. recomm.	4,878	4,167	3,110
#recommendations	14,642	9,996	7,170
#corr. recomm.	7,383	6,183	5,149
recall	0.07	0.05	0.04
precision	0.50	0.62	0.72
coverage$_{commits}$	0.13	0.09	0.06
coverage$_{meth}$	0.03	0.03	0.02
#recom(median)	1	1	1
#recom(mean)	1.67	1.55	1.74
distance$_{tokens}$(Q1,Q2,Q3)	0,1,2	0,1,2	0,1,2
distance$_{tokens}$(mean)	1.94	2.03	1.81
%distance$_{tokens}$(Q1,Q2,Q3)	0,13,22	0,13,22	0,13,20
%distance$_{tokens}$(mean)	14%	14%	13%

having at least four lines.

Table 5.3 reports the results achieved in this scenario. The precision values are in line with before (min: 0.59, max: 0.71), showing that the "quality" of the recommendations is not influenced by the length of the recommended methods. Instead, we observed a drop of recall, that does not go over 2%, with a number of correct recommendations ranging between 522 (low sensitivity) and 778 (high sensitivity).

The number of recommendations generated by FeaRS (#recom) is usually very low (median=1 and mean<2 in both scenarios). This shows that FeaRS does not generate many cases to inspect when triggered. Also, the results of distance$_{\text{tokens}}$ indicate that developers need to modify only a few tokens to adapt the received recommendations to their code.

While these results show the potential of FeaRS, they highlight (as in cases discussed for Table 5.2), that the recommended methods are short, with a potential small benefit for developers. Our qualitative analysis will help in better assessing the value of these recommendations.

Table 5.3. Performance when excluding short methods

	high sensitivity	medium sensitivity	low sensitivity
#commits	31,088	31,088	31,088
#added methods	83,562	83,562	83,562
#commits w. recomm.	900	763	564
#commits w. corr. recomm.	568	536	413
#recommendations	1,329	1,099	738
#corr. recomm.	778	742	522
recall	0.02	0.02	0.01
precision	0.59	0.68	0.71
coverage$_{commits}$	0.03	0.03	0.02
coverage$_{meth}$	0.01	0.01	0.01
#recom(median)	1	1	1
#recom(mean)	1.48	1.44	1.30
distance$_{tokens}$(Q1,Q2,Q3)	0,3,10	0,3,10	0,3,4
distance$_{tokens}$(mean)	5.08	5.07	3.98
%distance$_{tokens}$(Q1,Q2,Q3)	0,14,28	0,14,28	0,10,18
%distance$_{tokens}$(mean)	17%	16%	13%

5.4.3 Qualitative Examples

Correct Recommendations

Fig. 5.8 shows an example of a recommendation generated for the Memento app for Android Wear [165].

Repository: inertia-besi-c/Memento-AndroidWear **Commit:** 590449d

LHS

```
public static boolean
isExternalStorageReadable() {
    String state = Environment.
       getExternalStorageState();
    if
(Environment.MEDIA_MOUNTED.
      equals(state) ||
Environment.
      MEDIA_MOUNTED_READ_ONLY.
      equals(state)) {
        return true;
    }
    return false;
}
```

RHS

```
public static boolean
isExternalStorageWritable() {
    String state = Environment.

getExternalStorageState();
    if
(Environment.MEDIA_MOUNTED.
      equals(state)) {
        return true;
    }
    return false;
}
```

Figure 5.8. Correct recommendation to the usage of external storage in Android.

Suppose that the developer implements the `isExternalStorageReadable()` method to check whether the external storage of the device is mounted in read-only mode. FeaRS can

pop up and recommend the `isExternalStorageWritable()` method to check also if it is writable or not. This rule had four matching instances in our test set from four different repositories.

Fig. 5.9 shows an example of providing a custom back navigation for an Android `DrawerLayout`.

Repository: KaryaKita/karyakita-android **Commit:** e811795

LHS

```
@Override
public boolean
onNavigationItemSelected(MenuItem
item) {
  int id = item.getItemId();
  if (id == R.id.nav_camera) {
  } else if (id == R.id.nav_gallery) {
  } else if (id == R.id.nav_slideshow)
{
  } else if (id == R.id.nav_manage) {
  }
  DrawerLayout drawer = (DrawerLayout)
     findViewById(R.id.drawer_layout);
  drawer.closeDrawer(GravityCompat.
                                START);
  return true;
}
```

RHS

```
public void onBackPressed() {
  DrawerLayout drawer = (DrawerLayout)

findViewById(R.id.drawer_layout);
  if (drawer.isDrawerOpen(
    GravityCompat.START)) {
      drawer.closeDrawer
         (GravityCompat.START);
    } else {
      super.onBackPressed();
    }
}
```

Figure 5.9. Correct recommendation to provide a custom back navigation for an Android DrawerLayout.

Following the implementation of an `onNavigationItemSelected(...)` method that uses a `DrawerLayout`, FeaRS recommends a proper implementation for the `onBackPressed()` method. Interestingly, in case of a missing implementation, the `DrawerLayout` might not close properly, as it is discussed in a Stack Overflow question [166]. We found 19 matching instances for this rule in 17 different repositories.

Fig. 5.10 shows an example recommendation for the creation of a Google Map object from the Google Maps SDK.

Repository: p-hilosophers/TravelGuide **Commit:** f690635

LHS

```
@Override
protected void onCreate(Bundle
savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(
     R.layout.activity_maps);
  SupportMapFragment mapFragment =
   (SupportMapFragment)
      getSupportFragmentManager().
      findFragmentById(R.id.map);
  mapFragment.getMapAsync(this);
}
```

RHS

```
@Override
public void onMapReady(GoogleMap
                          googleMap) {
  mMap = googleMap;
  LatLng sydney = new
                 LatLng(-34, 151);
  mMap.addMarker(new
        MarkerOptions().
        position(sydney).
        title("Marker in Sydney"));
  mMap.moveCamera(
     CameraUpdateFactory.
     newLatLng(sydney));
}
```

Figure 5.10. Correct recommendation for the creation of a GoogleMap instance from the Google Maps SDK for Android.

We found 68 matches for this rule in 62 repositories. FeaRS matches an `onCreate(...)`

method in which an `Activity` creates a `SupportMapFragment` from the SDK. Next, it recommends an initial implementation for the `onMapReady(...)` method, that shows how to add a marker to the map. We found various implementations having a different initial marker position (*e.g.,* London, Sydney).

Unmatched Implementation Patterns

We present FeaRS's recommendations that have been triggered during the evaluation process (*i.e.,* their LHS has been matched in the test commits) but that have never been successful (*i.e.,* the RHS has not been matched).

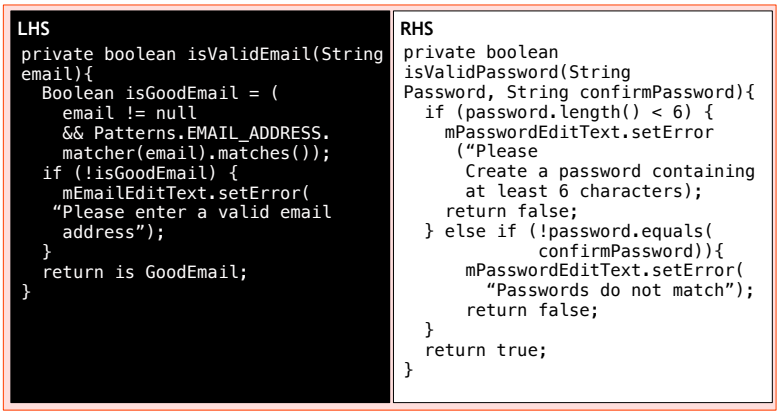


Figure 5.11. Unmatched recommendation for user credential validation in sign-up activity.

Fig. 5.11 shows an example of recommendation generated for the Artissans Android app [167].

Suppose that the developer implements the `isValidEmail()` method to check whether the email address provided when creating an new account is valid. FeaRS recommends the `isValidPassword()` method to check, in the same scenario, if the provided password/confirm password fields are valid (*i.e.,* they are composed by at least six characters, and they match each other). This rule had been triggered twice without finding a match for the RHS, thus being classified as an incorrect recommendation. However, when we looked into the two commits in which this recommendation was triggered, we found that both of them actually implemented an `isValidPassword()` method that, however, only validated the password based on its length, do not making the recommended method and the implemented one similar enough to be counted as a correct recommendation. This example is representative of others we found.

For example, Fig. 5.12 relates to the creation of a custom filter applied to a `RecyclerView.Adapter` in Android. The class `Filter` is used in Android to constrain data according to a specified pattern.

Following the implementation of a `UserFilter` constructor, FeaRS recommends a proper implementation of the overridden `publishResults` method from the `Filter` class that, as explained in the Android documentation, is *invoked in the UI thread to publish the filtering*

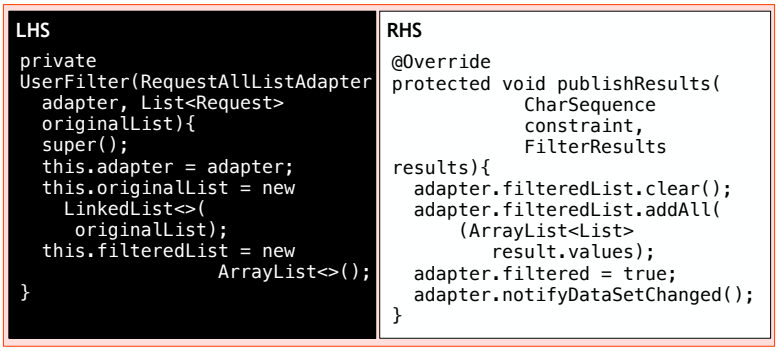


Figure 5.12. Unmatched recommendation for creating custom filter for filterable adapter in Android.

results in the user interface. Again, this recommendation was not matched (and considered wrong) during our study, but also in this case looking into the test commit [168] subject of the recommendation, we found that a similar overridden `publishResults` method was implemented as well following a custom filter constructor. Unfortunately, also in this case the similarity between the RHS of the rule and the implemented `publishResults` was not high enough to identify the recommendation as useful.

These cases show that our experimental design, while useful to provide a first indication about the quality of the recommendations triggered by FeaRS, has imprecisions in assessing FeaRS's performance. As previously said, only complementing this mining-based study with experiments with developers can help in better assessing FeaRS's usefulness.

5.5 Threats to Validity

Construct validity. In our experimental design we assumed that if a commit added three methods belonging to clusters C_1, C_2, and C_3 and FeaRS has an association rule $\{C_1\} \implies C_3$, FeaRS would have been useful in that commit to recommend C_3 to the developer. However, we cannot know whether C_3 was written before C_1, thus making FeaRS's recommendation useless in practice. Such a threat can only be addressed by (i) performing a user study in which developers code live using FeaRS, or (ii) recording IDE interaction data of programming sessions. While this is part of our future work, we preferred as first evaluation for FeaRS something that can be large-scale and fully automated, before moving to more costly studies requiring human involvement. In the design of our study, we only consider coding activities from one single commit might perform an implementation task, while ignoring those cases in which a given task can be separated into several commits. Actually we considered the idea of using close commits as a single data point, but we found out that it is hard to define a proper criterion for the selection of multiple commits and it might be risky for the cohesiveness of the task.

Another threat is related to the criterion we used to identify a generated recommendation as "correct." Given a commit c in which m_i and m_j are added, we assume that a recommendation $C_k \implies C_s$ is correct if m_i is matched to an existing cluster C_k and m_j is matched to

an existing cluster C_s (or *vice versa*, *i.e.,* m_i to C_s and m_j to C_k). This implies an assumption, meaning that the assignment of methods to cluster is correct or that, in other words, when a method is assigned to a cluster, the method actually implements functionalities related to those of the cluster. To partially address this threat, two researchers including the author manually analyzed a set of 100 methods assigned by FeaRS to a specific cluster, with the goal of verifying whether the assigned cluster actually implements the same feature of the method.

After solving conflicts arisen in 7% of cases, they reported an accuracy of 91%. Thus, we acknowledge possible imprecisions.

Internal validity. We tuned the FeaRS's parameters on a set of commits not used for the learning of the association rules nor for assessment of FeaRS's performance. We experimented with 1,080 combinations of parameters. However, it is possible that better performance can be achieved by considering other possible values. Thus, from this point of view, the reported performance is an underestimation. We adopted a careful experimental design to avoid using "data from the future" when tuning and testing our approach.

External validity. Overall, our study involves 20,713 open-source Android apps. The main issue is related to the fact that all used apps are open source, and might not be representative of commercial apps. Also, while FeaRS is general enough to be adapted to other contexts (*e.g.,* Java programming in general), we decided to focus on a more narrow scenario at least for this first work.

5.6 Conclusions

Code completion, while provenly useful and extensively used by developers [157] is just a step in the direction of an automated pair programmer, adding complete methods that a developer would have to add anyway and thus removing from the developer the burden of rote work. This was the ambitious goal that we set out to achieve with this work, embodied in the creation of FeaRS, an approach and a tool [164] to automatically recommend to developers the complete next method to write during implementation activities.

FeaRS relies on a simple but intuitive idea: programming is an eclectic activity, which some even go as far as calling it "natural" [10]. What a developer is doing has a high chance of having been done by someone else, somewhere else before. Leveraging this idea, FeaRS mines vast amounts of data to recommend complete methods given a set of methods being implemented by a developer. We evaluated FeaRS on the change history of 20,713 Android apps. The results show the potential of FeaRS, with hundreds of correct methods recommended even in its most conservative configuration.

However, our findings are not conclusive for what concerns the actual usefulness of the generated recommendations in a real usage scenario, in which developers use FeaRS during coding activities. This is due to two observations we made. First, some of the methods recommended by FeaRS are quite short and, while they can still be useful, they could also represent "trivial" recommendation for developers. We believe this can in part be made up by introducing a user feedback loop, which is part of our future work. The quantitative results show that around 15% of the tokens from the recommendations need to be modified,

added or deleted to fit the user's code base. One of our future plans is to integrate code adaption techniques into FeaRS to avoid potential conflicts or compilation errors with the user's code environment, and convert the coding convention into the user's style. Second, due to our experimental design, the "unmatched recommendations" are always considered false positives, while we observed that some are actually valuable recommendations. Thus, a deeper evaluation of FeaRS including a well-designed user study represents another main target of our future research.

6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we presented our research in the field of mining code change patterns with the aim of acquiring new empirical knowledge on software development activities and leveraging such a knowledge to facilitate code-related tasks. The research was organized into three directions, namely (i) studying code-comment inconsistencies, (ii) investigating quick remedy commits and their impact on mining software repositories, and (iii) using code change patterns for code recommendations. In each direction, we investigated one specific type of code change pattern and its possible applications in supporting developers. In this chapter, we sum up all the contributions and findings discussed in this dissertation, and outline possible directions for future work.

In Chapter 3, to raise the knowledge about code-comment inconsistencies, we presented the largest study at date about the co-evolution of code and comments, which involved the analysis of the complete change history of 1,500 Java systems. We investigated the extent to which different types of fine-grained code changes (*e.g.,* changes to selection statements) trigger the update of the corresponding code comments. This analysis provided empirical evidence useful to quantify the cases in which code-comment inconsistencies could possibly be introduced and to identify the types of code changes having a higher chance of introducing these inconsistencies. In this study, a database of ∼476 GB containing ∼1.3 Billion AST-level operations impacting code and comments has been created and made accessible for other types of inquiries. We also manually examined 500 commits likely related to the improvement of code comments, categorizing them into a taxonomy of comment-related changes, which aimed at identifying the types of code-comment inconsistencies that are fixed by software developers.

By analyzing the same 1,500 GitHub repositories we collected from Chapter 3, we qualitatively investigated another type of code change pattern performed by developers, namely *quick remedy commits* in Chapter 4. We defined *quick remedy commits* as commits performed by developers to remedy changes omitted or errors introduced in a previous commit, performed just a few minutes before. We defined a heuristic-based approach to identify quick

remedy commits by mining the history of open source repositories. By manually analyzing 500 quick remedy commits, we defined a taxonomy documenting the main motivations for these commits. Then, we investigated the impact of a specific type of quick remedy commits, namely reverted commits, on the data extracted for MSR studies. We focused on two data collection tasks performed in many previous works: (i) the identification of bug-fixing commits and (ii) the mining of refactoring operations over the change history of a system. Our analysis disclosed the amount of potential noise brought by reverted commits for these two data collection tasks.

In Chapter 5, we attempted to mine implementation patterns followed by developers at method level (*i.e.,* methods usually implemented together by developers) when implementing Android apps. We developed an approach that can identify these patterns. Then, we built a tool, called FeaRS, which is able to recommend the complete code of the next method developers are likely to implement while monitoring the code written in the IDE. We performed a large-scale empirical evaluation of FeaRS by simulating its usage on the change history of 20,713 Android apps. We reported promising results and discussed limitations of our approach.

6.2 Future Work

We discuss in this section our plans for future work, which mostly revolve around (i) the improvement of FeaRS and (ii) the detection and fixing of simple omission errors performed by developers.

6.2.1 Detecting and Fixing Simple Omission Errors

In our study of "*quick remedy commits*" (Chapter 4), we use strict selection criteria to identify commits suitable for manual inspection as we aimed at performing a qualitative study identifying the types of quick remedy commits performed by developers, organizing them into a taxonomy. While working on such a taxonomy, we found that some quick remedy commits are related to the introduction and the fix of omission errors. For example, a developer may forget to propagate code changes into other locations that are affected by the implemented change. Being errors that have been fixed within a few minutes from their introduction, we believe they represent a suitable scenario for applying automated techniques aimed at detecting and fixing them. However, to investigate this research direction, we first need to define new and less strict heuristics to collect a large dataset of quick remedy commits fixing omission errors. For example, we could inspect the diffs of adjacent commits to see whether (i) they impact the same files, and (ii) the more recent commit only implements relatively minor changes. Using the collected data, we can then investigate possible approaches to automatically spot and fix omission errors. While recent state-of-the-art techniques are exploiting more and more deep learning models, their applicability will depend on the amount of training data we can collect. Thus, we also plan to investigate other options based on static code analysis.

6.2.2 Improving FeaRS

In Chapter 5 we presented FeaRS and its empirical evaluation, which highlighted some limitations of our tool that could be addressed in future.

We plan to integrate into FeaRS code transformation techniques able to automatically improve the reusability of the generated code recommendations by adapting them to the developer's coding style. To be more specific, we can target: (i) compilation errors/bugs possibly introduced through the recommended code (*e.g.,* due to misused or undefined code entities in the recommended code); (ii) the adaptation of the recommended code to existing coding conventions (*e.g.,* from simple formatting to naming conventions). For the first point, we can use syntactic analysis to identify potential conflicts and errors caused by the usage of the recommended code in the developers' codebase. Classic examples here are accesses to undefined fields. We will focus on statements causing compilation errors and try to fix them by applying a set of predefined fixing rules (*e.g.,* replacing undefined fields with those available in the scope of the recommended code). As for the second point, we need to learn the developers' coding style from their codebase automatically. Then, the recommended code must be adapted accordingly to the extracted style. Both approaches aim to minimize the manual effort required by developers to reuse the recommended code. We expect the steps above to improve FeaRS' performance. To assess this, we also plan to conduct a deeper evaluation of FeaRS featuring a user study (*e.g.,* controlled experiments, surveys).

FeaRS could also benefit from additional features implemented in it. At the moment, the implementation patterns we learned are represented as a group of methods with their complete code. This implies that developers need to write at least one full method to trigger potential recommendations from FeaRS. To overcome this problem we could summarize each method based on specific textual and structural features it contains. In this way, when the developer starts implementing a method, we could match features in the incomplete implementation being able to recommend the developer how to autocomplete the current method(*e.g.,* from signature to method's body). Furthermore, we can also explore the possibility of learning implementation patterns at different granularity levels (*e.g.,* lines, blocks).

6.3 Closing Words

In this dissertation, we showed that mining unexposed code change patterns from open source repositories can help in better understanding development activities and potentially support developers during software development. We presented several empirical studies to build new knowledge on three specific types of code change patterns. Our research on these MSR related topics contributes to a better understanding of some partly hidden development activities and demonstrates other possibilities of converting those empirical knowledge into applications for code-related tasks.

At the end, we hope more researchers will continue working on exploring new code change patterns from open source repositories and utilizing existing empirical knowledge to build tools, in order to further enhance practitioners' awareness of the relevance of MSR for their daily work.

Bibliography

[1] "Mining Software Repositories research field. http://www.msrconf.org/."

[2] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 31–41.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 1–10.

[4] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 99–108.

[5] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 1–1.

[6] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *Proc. of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 227–237.

[7] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016)*, 2016, pp. 362–373.

[8] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[9] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2009, 2009, pp. 213–222.

[10] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.

[11] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 69–79.

[12] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 269–280.

[13] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[14] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2014, 2014, pp. 419–428.

[15] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 69–79.

[16] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 362–373.

[17] L. P. Hattori and M. Lanza, "On the nature of commits," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'08. Piscataway, NJ, USA: IEEE Press, 2008, pp. III–63–III–71.

[18] J. Park, M. Kim, B. Ray, and D. Bae, "An empirical study of supplementary bug fixes," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, June 2012, pp. 40–49.

[19] J. Park, M. Kim, and D.-H. Bae, "An empirical study of supplementary patches in open source projects," *Empirical Softw. Engg.*, vol. 22, no. 1, pp. 436–473, Feb. 2017.

[20] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ser. ICSM '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 120–.

[21] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 511–526, Jun. 2005.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[23] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 153–162.

[24] M. Claes, M. V. Mäntylä, M. Kuutila, and B. Adams, "Do programmers work at night or during the weekend?" in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 705–715.

[25] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 4–14.

[26] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 491–500.

[27] J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, and A. Capiluppi, "Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers," *Int. J. Open Source Softw. Process.*, vol. 3, no. 2, pp. 23–42, Apr. 2011.

[28] J. Park, M. Kim, and D.-H. Bae, "An empirical study on reducing omission errors in practice," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 121–126.

[29] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. re-opened bugs," in *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 205–214.

[30] M. Dai, B. Shen, T. Zhang, and M. Zhao, "Impact of consecutive changes on later file versions," in *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*, ser. EAST 2014. New York, NY, USA: ACM, 2014, pp. 17–24.

[31] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 760–776, Jun. 2012.

[32] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36.

[33] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. IEEE Press, 2017, pp. 176–185.

[34] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, 2012, pp. 104–113.

[35] G. Rodriguez-Perez, G. Robles, and J. M. Gonzalez-Barahona, "How much time did it take to notify a bug?: Two case studies: Elasticsearch and nova," in *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 29–35.

[36] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 173–174.

[37] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009.

[38] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, and B. Xu, "Towards an understanding of change types in bug fixing code," *Inf. Softw. Technol.*, vol. 86, no. C, pp. 37–53, Jun. 2017.

[39] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 79–82.

[40] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 25–36.

[41] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 832–837.

[42] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, p. 351–360.

[43] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.72

[44] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR 2013. IEEE Press, 2013, pp. 121–130.

[45] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Softw. Engg.*, vol. 21, no. 2, pp. 303–336, Apr. 2016.

[46] A. Hora, D. Silva, M. T. Valente, and R. Robbes, "Assessing the threat of untracked changes in software evolution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, p. 1102–1113.

[47] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. ACM, 2014, p. 803–814.

[48] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, p. 481–490.

[49] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, p. 147–157.

[50] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 690–701. [Online]. Available: https://doi.org/10.1145/2950290.2950341

[51] M. P. Robillard, M. Nassif, and S. McIntosh, "Threats of aggregating software repository data," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 508–518.

[52] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 65–69.

[53] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 585–597. [Online]. Available: https://doi.org/10.1145/3324884.3416581

[54] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, pp. 215–223.

[55] A. T. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: An exploration of eclipse task comments and their implication to repository mining," in *Proc. of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. ACM, 2005, pp. 1–5.

[56] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Softw. Engg.*, vol. 21, no. 1, p. 104–158, Feb. 2016. [Online]. Available: https://doi.org/10.1007/s10664-014-9350-8

[57] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 25–35.

[58] P. W. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, vol. 21, no. 1, pp. 17–42, Feb 2016.

[59] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers — taxonomies and characteristics of comments in operating system code," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 331–341.

[60] L. Pascarella, "Classifying code comments in Java mobile applications," in *Proc. of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: ACM, 2018, pp. 39–40. [Online]. Available: http://doi.acm.org/10.1145/3197231.3198444

[61] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. ACM, 2006, pp. 179–180.

[62] O. Arafat and D. Riehle, "The commenting practice of open source," in *Proc. of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 857–864.

[63] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293 – 2304, 2012.

[64] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code? (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 36–41.

[65] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proc. of the 14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 70–79.

[66] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, Dec 2009.

[67] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The JavadocMiner," in *Natural Language Processing and Information Systems*, C. J. Hopfe, Y. Rezgui, E. Métais, A. Preece, and H. Li, Eds. Springer, 2010, pp. 68–79.

[68] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. of the 21st IEEE International Conference on Program Comprehension (ICPC)*, May 2013, pp. 83–92.

[69] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *Proc. of the 24th IEEE International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[70] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or bad comments?*/," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 145–158, Oct. 2007.

[71] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 260–269.

[72] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proc. of the 39th IEEE International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 27–37.

[73] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 112–122.

[74] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, July 2018, pp. 154–163.

[75] G. Sridhara, "Automatically detecting the up-to-date status of ToDo comments in Java programs," in *Proc. of the 9th India Software Engineering Conference*, ser. ISEC '16. ACM, 2016, pp. 16–25.

[76] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, Feb 2018.

[77] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "SATD Detector: A text-mining-based self-admitted technical debt detection tool," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 9–12.

[78] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 179–188.

[79] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. of the 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 9–15.

[80] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from stack overflow," in *Proceedings of ICSE 2016 (38th International Conference on Software Engineering)*, 2016, pp. 392–403.

[81] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proceedings of ASE 2013 28th IEEE/ACM International Conference on Automated Software Engineering)*, 2013, pp. 562–567.

[82] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *Proceedings of ICSE 2017 (39th International Conference on Software Engineering)*, 2017, pp. 94–105.

[83] C. Lezos, G. Dimitroulakos, I. Latifis, and K. Masselos, "Automatic generation of code analysis tools: The castql approach," in *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, ser. RWDSL '16. ACM, 2016.

[84] R. L. Glass, "Some thoughts on automatic code generation," *SIGMIS Database*, vol. 27, no. 2, p. 16–18, Apr. 1996.

[85] H. Liao, J. Jiang, and Y. Zhang, "A study of automatic code generation," in *2010 International Conference on Computational and Information Sciences*, 2010, pp. 689–691.

[86] N. K. Singh, *EB2ALL: An Automatic Code Generation Tool*. London: Springer London, 2013, pp. 105–141.

[87] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proceedings of RSSE 2012*. IEEE Press, 2012, pp. 85–89.

[88] P. Rigby and M. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of ICSE 2013*, 2013, pp. 832–841.

[89] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of SUITE 2011*. ACM, 2011, pp. 17–20.

[90] R. Holmes, R. Walker, and G. Murphy, "Strathcona example recommendation tool," *SIGSOFT Software Engineering Notes*, vol. 30, pp. 237–240, 2005.

[91] ——, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE TSE*, vol. 32, no. 12, pp. 952–970, 2006.

[92] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2744200

[93] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 621–624.

[94] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: http://arxiv.org/abs/1903.05734

[95] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, p. 763?773.

[96] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[97] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*, 2021, pp. 150–162.

[98] G. A. Aye, S. Kim, and H. Li, "Learning autocompletion from real-world datasets," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*, 2021, pp. 131–139.

[99] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020. Association for Computing Machinery, 2020.

[100] "GitHub Copilot https://copilot.github.com."

[101] "aiX Code Completer. https://tinyurl.com/ydb2ux8x."

[102] "Codota. https://www.codota.com."

[103] M. Umarji and S. E. Sim, *Archetypal Internet-Scale Source Code Searching*. New York, NY: Springer New York, 2013, pp. 35–52. [Online]. Available: https://doi.org/10.1007/978-1-4614-6596-6_3

[104] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. ACM, 2006, p. 681–682.

[105] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, p. 243–253.

[106] L. Martie, A. v. d. Hoek, and T. Kwak, "Understanding the impact of support for iteration on code search," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 774–785. [Online]. Available: https://doi.org/10.1145/3106237.3106293

[107] S. Thummalapenta, "Exploiting code search engines to improve programmer productivity," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. ACM, 2007, p. 921–922.

[108] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. Association for Computing Machinery, 2007, p. 204–213.

[109] ——, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 327–336.

[110] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, p. 475–484.

[111] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

[112] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza, "Automated documentation of android apps," *IEEE Transactions on Software Engineering*, 2019.

[113] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.

[114] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information*, ser. SIGDOC '05, 2005, pp. 68–75.

[115] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.

[116] J. H. Hayes and L. Zhao, "Maintainability prediction: A regression analysis of measures of evolving systems," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05, 2005, pp. 601–604.

[117] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[118] "Replication package. https://github.com/USI-INF-Software/ICPC2019-code-comment-inconsistencies."

[119] "About stars (GitHub). https://help.github.com/articles/about-stars/."

[120] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[121] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[122] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[123] M. J. Kaelbling, "Programming languages should not have comment statements," *SIGPLAN Not.*, vol. 23, no. 10, pp. 59–60, Oct. 1988.

[124] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proc. of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 92–101.

[125] G. Rodriguez-Perez, G. Robles, and J. M. Gonzalez-Barahona, "How much time did it take to notify a bug? two case studies: Elasticsearch and nova," in *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2017, pp. 29–35.

[126] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. González-Barahona, "What if a bug has a different origin? making sense of bugs without an explicit bug introducing change," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3239235.3267436

[127] C. Wang, Y. Li, L. Chen, W. Huang, Y. Zhou, and B. Xu, "Examining the effects of developer familiarity on bug fixing," *Journal of Systems and Software*, vol. 169, p. 110667, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220301266

[128] M. D. Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 556–567.

[129] A. Peruma, "A preliminary study of android refactorings," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 148–149.

[130] M. Mahmoudi, S. Nadi, and N. Tsantalis, "Are refactorings to blame? an empirical study of refactorings in merge conflicts," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 151–162.

[131] B. Lin, C. Nagy, G. Bavota, and M. Lanza, "On the impact of refactoring operations on code naturalness," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 594–598.

[132] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 2–12.

[133] E. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, 2019, pp. 51–58.

[134] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *35th International Conference on Software Engineering, ICSE '13*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 832–841.

[135] "Replication package. https://github.com/USI-INF-Software/EMSE-ICPC2020-quick-remedy-commit."

[136] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 170–178.

[137] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[138] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[139] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 281–293.

[140] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and NLP to promote a consistent usage of identifiers," in

17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017, 2017, pp. 81–90.

[141] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, 2007, pp. 185–194.

[142] N. C. Bradley, T. Fritz, and R. Holmes, "Context-aware conversational developer assistants," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 993–1003.

[143] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 135–146.

[144] J. Shimagaki, Y. Kamei, S. McIntosh, D. Pursehouse, and N. Ubayashi, "Why are commits being reverted?: A comparative study of industrial and open source projects," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 301–311.

[145] M. Yan, X. Xia, D. Lo, A. E. Hassan, and S. Li, "Characterizing and identifying reverted commits," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2171–2208, Aug 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09688-8

[146] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in oo software through visualization," in *11th IEEE International Workshop on Program Comprehension, 2003.*, 2003, pp. 44–53.

[147] G. Catolino and F. Ferrucci, "An extensive evaluation of ensemble techniques for software change prediction," *Journal of Software: Evolution and Process*, vol. 31, no. 9, 2019.

[148] M. F. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, 2018.

[149] M. M. Rahman, C. K. Roy, and R. G. Kula, "Predicting usefulness of code review comments using textual features and developer experience," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 215–226.

[150] M. Tufano, G. Bavota, D. Poshyvanyk, M. D. Penta, R. Oliveto, and A. D. Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *Journal of Software: Evolution and Process*, vol. 29, no. 1, 2017.

[151] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[152] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 53–64.

[153] M. Fischer, M. Pinzger, and H. C. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, 2003, p. 23.

[154] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[155] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[156] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[157] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[158] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 1995, pp. 3–14.

[159] "FeaRS github project. https://github.com/Em11FW/FeaRS."

[160] R. Coppola, L. Ardito, and M. Torchiano, "Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github," in *Proceedings of the International Workshop on App Market Analytics*, 2019, pp. 8–14.

[161] "JavaParser. https://javaparser.org/."

[162] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 121–130.

[163] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.

[164] "Replication package. https://github.com/anonymousfears/fears."

[165] “Memento for Android Wear. https://github.com/inertia-besi-c/Memento-AndroidWear.”

[166] “StackOverflow question. https://tinyurl.com/y7pge4l9.”

[167] “Artissans Android app. https://github.com/Wess58/Artissans.”

[168] “Artie Android app. https://github.com/manbradcalf/Artie-Android/commit/34ccfa3.”

Projects' References

[169] "Project java-design-patterns on GitHub." [Online]. Available: https://github.com/iluwatar/java-design-patterns

[170] "Project spring-petclinic on GitHub." [Online]. Available: https://github.com/spring-projects/spring-petclinic

[171] "Project findbugs on GitHub." [Online]. Available: https://github.com/findbugsproject/findbugs

[172] "Project spotbugs on GitHub." [Online]. Available: https://github.com/spotbugs/spotbugs

[173] "Commit to alluxio project on GitHub." [Online]. Available: https://github.com/Alluxio/alluxio/commit/727a301dbf

[174] "Commit to alluxio project on GitHub." [Online]. Available: https://github.com/Alluxio/alluxio/commit/0ebbfb887b

[175] "Commit to bitcoinj project on GitHub." [Online]. Available: https://github.com/bitcoinj/bitcoinj/commit/8923af57

[176] "Commit to bitcoinj project on GitHub." [Online]. Available: https://github.com/bitcoinj/bitcoinj/commit/3211fe59

[177] "Commit to qr-code-generator project on GitHub." [Online]. Available: https://github.com/nayuki/QR-Code-generator/commit/12360be

[178] "Commit to wordpress-android project on GitHub." [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android/commit/04ee958211

[179] "Commit to gwtp project on GitHub." [Online]. Available: https://github.com/ArcBees/GWTP/commit/656406523

[180] "Commit to cassandra project on GitHub." [Online]. Available: https://github.com/apache/cassandra/commit/370b0ec2f2

[181] "Commit to lucee project on GitHub." [Online]. Available: https://github.com/lucee/Lucee/commit/b84c88f8

[182] "Commit to cordova-android project on GitHub." [Online]. Available: https://github.com/apache/cordova-android/commit/dfb89df4

[183] "Commit to cordova-android project on GitHub." [Online]. Available: https://github.com/apache/cordova-android/commit/79935d31

[184] "Commit to javaparser project on GitHub." [Online]. Available: https://github.com/javaparser/javaparser/commit/c1cde434c

[185] "Commit to opennms project on GitHub." [Online]. Available: https://github.com/OpenNMS/opennms/commit/7c70d666ad6

[186] "Commit to findbugs project on GitHub." [Online]. Available: https://github.com/findbugsproject/findbugs/commit/906b4d22c

[187] "Commit to wordpress-android project on GitHub." [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android/commit/533cc9bfad

[188] "Commit to voltdb project on GitHub." [Online]. Available: https://github.com/VoltDB/voltdb/commit/963f93002b

[189] "Commit to psi-probe project on GitHub." [Online]. Available: https://github.com/psi-probe/psi-probe/commit/9755fd67

[190] "Commit to groovy project on GitHub." [Online]. Available: https://github.com/apache/groovy/commit/2db6e8d41a

[191] "Commit to groovy project on GitHub." [Online]. Available: https://github.com/apache/groovy/commit/8341c825fa

[192] "Commit to android project on GitHub." [Online]. Available: https://github.com/JetBrains/android/commit/3b81254b74

[193] "Commit to alluxio project on GitHub." [Online]. Available: https://github.com/Alluxio/alluxio/commit/f615234fdf

[194] "Commit to guava project on GitHub." [Online]. Available: https://github.com/google/guava/commit/6d8d9d97f

[195] "Commit to elasticsearch-hadoop project on GitHub." [Online]. Available: https://github.com/elastic/elasticsearch-hadoop/commit/899e2bb5

[196] "Commit to classgraph project on GitHub." [Online]. Available: https://github.com/classgraph/classgraph/commit/526b3f12

[197] "Commit to groovy project on GitHub." [Online]. Available: https://github.com/apache/groovy/commit/86963ae8b0

[198] "Commit to groovy project on GitHub." [Online]. Available: https://github.com/apache/groovy/commit/85898e99

[199] "Commit to openmrs-core project on GitHub." [Online]. Available: https://github.com/openmrs/openmrs-core/commit/0e72cf5c8

[200] "Commit to crate project on GitHub." [Online]. Available: https://github.com/crate/crate/commit/7b56616dc

[201] "Commit to denominator project on GitHub." [Online]. Available: https://github.com/Netflix/denominator/commit/e727b9d

[202] "Commit to TomP2P project on GitHub." [Online]. Available: https://github.com/tomp2p/TomP2P/commit/3db803c

[203] "Commit to TomP2P project on GitHub." [Online]. Available: https://github.com/tomp2p/TomP2P/commit/8802c5e

[204] "Commit to tomp2p project on GitHub." [Online]. Available: https://github.com/tomp2p/TomP2P/commit/4bc6e824

[205] "Commit to spacewalk project on GitHub." [Online]. Available: https://github.com/spacewalkproject/spacewalk/commit/6df7327

[206] "Commit to spacewalk project on GitHub." [Online]. Available: https://github.com/spacewalkproject/spacewalk/commit/fec7040

[207] "Commit to Accumulo project on GitHub." [Online]. Available: https://github.com/apache/accumulo/commit/2ad672a

[208] "Commit to accumulo project on GitHub." [Online]. Available: https://github.com/apache/accumulo/commit/b8859513a

[209] "Commit to lombok project on GitHub." [Online]. Available: https://github.com/rzwitserloot/lombok/commit/57f59074

[210] "Commit to liferay-portal project on GitHub." [Online]. Available: https://github.com/liferay/liferay-portal/commit/1b5c378d4785

[211] "Liferay Portal Issue LPS-44476." [Online]. Available: https://issues.liferay.com/browse/LPS-44476

[212] "Commit to tower project on GitHub." [Online]. Available: https://github.com/DroidPlanner/Tower/commit/72132d049

[213] "Commit to openpnp project on GitHub." [Online]. Available: https://github.com/openpnp/openpnp/commit/aeef4cb0e4

[214] "Commit to geoserver project on GitHub." [Online]. Available: https://github.com/geoserver/geoserver/commit/22c89ad106

[215] "Commit to jitsi project on GitHub." [Online]. Available: https://github.com/jitsi/jitsi/commit/6a361bbf6

[216] "Commit to jitsi project on GitHub." [Online]. Available: https://github.com/jitsi/jitsi/commit/a74af45

[217] "Commit to tinkerpop project on GitHub." [Online]. Available: https://github.com/apache/tinkerpop/commit/a4c62be7a5

[218] "Commit to tinkerpop project on GitHub." [Online]. Available: https://github.com/apache/tinkerpop/commit/aa3d538

[219] "Commit to Hadoop project on GitHub." [Online]. Available: https://github.com/apache/hadoop/commit/cb64e8eb192

[220] "Commit to Tomcat project on GitHub." [Online]. Available: https://github.com/apache/tomcat/commit/f711963768

[221] "Commit to Aosp Mirror project on GitHub." [Online]. Available: https://github.com/aosp-mirror/platform_packages_apps_settings/commit/d3dcce029d

[222] "Commit to Metasfresh project on GitHub." [Online]. Available: https://github.com/metasfresh/metasfresh/commit/7875c81632

[223] "Commit to WordPress-Android project on GitHub." [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android/commit/c363f2ff2d