Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

4-2018

How We Refactor and How We Mine it ? A Large Scale Study on Refactoring Activities in Open Source Systems

Eman Abdullah AlOmar
eaa6167@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

AlOmar, Eman Abdullah, "How We Refactor and How We Mine it ? A Large Scale Study on Refactoring Activities in Open Source Systems" (2018). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

How We Refactor and How We Mine it ? A Large Scale Study on Refactoring Activities in Open Source Systems

by

Eman Abdullah AlOmar

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

April 2018

The thesis "How We Refactor and How We Mine it ? A Large Scale Study on Refactoring Activities in Open Source Systems" by Eman Abdullah AlOmar has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor
Thesis Committee Chair

Dr. Christian D. Newman
Assistant Professor

Dr. Scott Hawker
Associate Professor
Graduate Program Director

To my family and friends, for all of their endless love, support, and encouragement

Acknowledgments

First and foremost, I am extremely grateful to the Almighty God for enabling me to successfully complete this master research.

I would like to express my sincerest gratitude and profound appreciation to my adviser, Dr. Mohamed Wiem Mkaouer, for his assistance, constant guidance, and immeasurable support throughout the process of conducting this research. His enthusiasm and encouragement helped me carried out this assignment.

Besides my advisor, I would like to thank Dr. Christian Newman for his constructive feedback and suggestions for my thesis project, and to the faculty members of the Department of Software Engineering for exposing me to many different areas of software engineering throughout my duration of my Master's program.

My appreciations also go to my Software Engineering colleagues who willingly helped me with their abilities.

For financial support, I extend my thanks to the Saudi Arabian Cultural Mission (SACM) for providing me with this wonderful opportunity to pursue master degree and their academic support through two years of graduate school.

Last but not the least, I must acknowledge my thanks to my parents for supporting me spiritually and being my primary source of inspiration and encouragement during two years of study.

Abstract

Refactoring, as coined by WIlliam Obdyke in 1992, is the art of optimizing the syntactic design of a software system without altering its external behavior. Refactoring was also cataloged by Martin Fowler as a response to the existence of design defects that negatively impact the software's design. Since then, the research in refactoring has been driven by improving systems structures. However, recent studies have been showing that developers may incorporate refactoring strategies in other development related activities that go beyond improving the design. In this context, we aim in better understanding the developer's perception of refactoring by mining and automatically classifying refactoring activities in 1,706 open source Java projects. We perform a *differentiated replication* of the pioneering work by Tsantalis et al. We revisit five research questions presented in this previous empirical study and compare our results to their original work. The original study investigates various types of refactorings applied to different source types (i.e., production vs. test), the degree to which experienced developers contribute to refactoring efforts, the chronological collocation of refactoring with release and testing periods, and the developer's intention behind specific types of refactorings. We reexamine the same questions but on a larger number of systems. To do this, our approach relies on mining refactoring instances executed throughout several releases of each project we studied. We also mined several properties related to these projects; namely their commits, contributors, issues, test files, etc. Our findings confirm some of the results of the previous study and we highlight some differences for discussion. We found that 1) feature addition and bug fixes are strong motivators for developers to refactor their code base, rather than the traditional design improvement motivation; 2) a variety of refactoring types are applied when refactoring both production and test code. 3) refactorings tend to be applied by experienced developers who have contributed a wide range of commits to the code. 4) there is a correlation between the type of refactoring activities taking place and whether the source code is undergoing a release or a test period.

Contents

Acknowledgments . . . . **i**

Abstract . . . . **ii**

1 Introduction **1**

2 Related Work **5**

2.0.1 Refactoring Detection 5

2.0.2 Refactoring Motivation 6

2.0.3 Commits Classification 8

2.0.4 Study Replication 9

3 Methodology **13**

3.0.1 Phase 1: Selection of GitHub Repositories 14

3.0.2 Phase 2: Refactoring Detection 15

3.0.3 Phase 3: Commits Classification 17

4 Analysis & Discussion **20**

4.1 **RQ1**: Do software developers perform different types of refactoring operations on test code and production code? 21

4.2 **RQ2**: Which developers are responsible for refactorings? 24

4.3 **RQ3**: Is there more refactoring activity before project releases than after? . 27

4.4 **RQ4**: Is refactoring activity on production code preceded by the addition or modification of test code? 32

4.5 **RQ5**:What is the purpose of the applied refactorings? 35

5 Threats to Validity **40**

6 Conclusion & Future Work **41**

Bibliography **43**

List of Tables

2.1 Refactoring Detection Tools in Related Work. 6
2.2 Commits Classification in Related Work. 10

3.1 Refactoring Types (Extracted from Fowler [14]). 14
3.2 Projects Overview. 15
3.3 Classification Categories. 16
3.4 Comparison of Different Classifiers. 18

4.1 Characteristics of Three Software Projects under Study. 20
4.2 Refactoring Frequency in Production and Test Files in all Projects Combined. 21
4.3 Frequency of Refactorings per Type on Production and Test Files in Hadoop, OrientDB and Camel. 23
4.4 Percentage of Refactorings on Production and Test Files in all Projects Combined. 29
4.5 Refactoring Level Percentages per Category. 36

List of Figures

3.1 Approach Overview. 13

4.1 Top 1 and all the Rest of Refactoring Contributors for all Projects Combined 24
4.2 Total Refactoring and Non-Refactoring Commits for Refactoring Contributors for all Projects Combined 25
4.3 Refactoring Contributors in Production and Test Files in Hadoop, OrientDB and Camel. 26
4.4 Refactoring Density - Release Point Comparison. 27
4.5 Refactoring Density - Release Point Comparison (Production Files). 28
4.6 Refactoring Density - Release Point Comparison (Test Files). 28
4.7 Frequency Occurrence of each Refactoring Operation before and after Release 30
4.8 Refactoring Activity Comparison (Release)- Hadoop. 31
4.9 Refactoring Activity Comparison (Release)- OrientDB. 31
4.10 Refactoring Activity Comparison (Release)- Camel. 31
4.11 Refactoring Activity Comparison (Test)- Hadoop. 34
4.12 Refactoring Activity Comparison (Test)- OrientDB. 34
4.13 Refactoring Activity Comparison (Test)- Camel. 34
4.14 Percentage of Classified Commits per Category in all Projects Combined. . 35
4.15 Distribution of Refactoring Types per Category 37
4.16 Percentage of Classified Commits per Category in Hadoop, OrientDB and Camel. 38

Chapter 1

Introduction

The success of a software system depends on its ability to retain high quality of design in the face of continuous change. However, managing the growth of the software while continuously developing its functionalities is challenging, and can account for up to 75% of the total development cost. One key practice to cope with this challenge is refactoring. Refactoring is the art of remodeling the software design without altering its functionalities [14][10]. It was popularized by Fowler, who identified 72 refactoring types and provided examples of how to apply them in his catalog [14].

Refactoring is a critical activity in software maintenance and is regularly performed by developers for multiple reasons [42][40][34]. Because refactoring is both a common activity and can be automatically detected [11][36][42], researchers can use it to analyze how developers maintain software during different phases of development and over large periods of time. This research is vital for understanding more about the maintenance phase; the most costly phase of development [7][12].

By strict definition, refactoring is applied to enforce best design practices, or to cope with design defects. However, recent studies have shown that, practically, developers interleave refactoring practices with other maintenance and development related tasks [33]. For instance, this previous work [32] distinguished two refactoring tactics when developers perform refactoring: *root-canal refactoring*, in which programmers explicitly used refactoring for correcting deteriorated code, and *floss refactoring*, in which programmers use refactoring as means to reach other goals, such as adding a feature or fixing a bug. Yet, there is little evidence on which refactoring tactic is more common; there is no consensus

in research on what events trigger refactoring activities and how these correlate with the choice of refactoring (i.e., extract method, rename package, etc.); and there is no consensus on how refactoring activities are scheduled and carried out, on average, alongside other software engineering tasks (e.g., testing, bug fixing).

The purpose of this study is to further our understanding of how refactoring is performed and the motivation behind different forms of refactorings. Therefore, our work is a *differentiated replication* [46][1] of the study of Tsantalis et al. [42]; extending it primarily by:

- Analyzing the commit and refactoring history of 1,706 curated open source Java projects. The intention behind mining such a large set of projects is to challenge the scalability of the previous related work's conclusions, which relied primarily on developer's feedback through interviews and a limited set of projects.
- Performing an automatic classification of commits which contain refactoring operations with the goal of identifying the developer's motivation behind every application of a refactoring; what caused the developer to refactor? To the best of our knowledge, no prior studies have automatically classified refactoring activities, previously applied by a diverse set of developers, belonging to a large set of varied projects.

In this study we revisit the following research questions:

1. **RQ1** Do software developers perform different types of refactoring operations on test code versus production code?

   This research questions seeks any patterns that developers specifically apply to either production or test codes.

2. **RQ2** Is there a subset of developers, within the development team, who are responsible for refactoring the code?

[1] Similarity and differences between the two studies are described in Section 2.

We answer whether the task of refactoring is equally distributed among all developers or it is the responsibility of a subset. We also verify whether it is proportional to the developers overall contribution to the project, in terms of commits.

3. **RQ3** Is there more refactoring activity before project releases than after?

   To answer this research question, we monitor the refactoring activities for a time window centered by the release date, then we compare the frequency of refactoring before and after the release of the new version.

4. **RQ4** Is refactoring activity on production code preceded by the addition or modification of test code?

   Similarly to the previous research question, we compare the frequency of refactoring before and after the version's testing period.

5. **RQ5** What is the developer's purpose of the applied refactorings?

   We determine the motivation of the developer through the classification of the commit containing these refactoring operations. It identifies the type of development tasks that refactorings were interleaved with, e.g., updating a feature, or debugging.

The results of our study will help strengthen our understanding of what events cause the need for refactorings (e.g., bug fixing, testing). Based on some of the results in this paper, tools that help developers refactor can better support our empirically-derived reality of refactoring in industry. One recent area of research that is growing in popularity is automating the construction of transformations from examples [3] [31] [38], which may be used for refactoring. The result of this may help improve the application and output of these technologies by helping researchers 1) choose appropriate examples based on what motivates the refactoring and 2) train these tools to apply refactorings using best practices based on why (e.g., bug fix, design improvement) the refactoring is needed.

Additionally, recommending or recognizing best practices for refactorings requires us to understand why the refactoring is being carried out in the first place. It may not be

true that best practices for refactorings performed for bug fixes are the same as those performed to improve design or support a new framework. This work adds to the empirical reality originally constructed by Tsantalis et al [42] and represents a step forward in a stronger, empirically-derived understanding of refactorings and promote potential research that bridges the gap between developers and refactoring in general.

The remainder of this paper is organized as follows. Section 2 enumerates the previous related studies. In Section 3, we give the design of our empirical study. Section 4 presents the study results while discussing our findings compared to the previous replicated paper's results. The next section reports any threats to the validity of our experiments, before finally concluding in Section 7.

Chapter 2

Related Work

This thesis's focuses on mining commits to detect refactorings, then classifying the commits to identify the motivation behind the detected refactorings. Thus, in this section, we are interested in refactoring detection tools, along with the recent research on commit classification. Finally, we report studies that analyze the human aspect in the refactoring-based decision making.

2.0.1 Refactoring Detection

Several studies have discussed various methodologies to identify refactoring operations between two versions of a software system. Our technique takes advantage of these tools to discover refactorings in large bodies of software. Dig et al. [11] developed a tool called Refactoring Crawler, which uses syntax and graph analysis to detect refactorings. Prete et al. [36] proposed Ref-Finder, which identifies complex refactorings using a template-based approach. Hayashi et al. [18] considered the detection of refactorings as a search problem, the authors proposed a best-first graph search algorithm to model changes between software versions. Xing and Stroulia [47] proposed JDevAn, which is a UMLDiff based, design-level analyzer for detecting refactorings in the history Object-Oriented systems. Tsantalis et al [42] presented Refactoring Miner, which is a lightweight, UMLDiff based algorithm that mines refactorings within git commits. The authors extended their tool [40] to enhance the accuracy of the 14 refactoring types that can be detected through structural constraints. Silva et al. [39] extended Refactoring Miner by combining the heuristics

based static analysis with code similarity (TF-IDF weighting scheme) to identify 13 refactoring types. Table 2.1 summarizes the detection tools cited in this study.

Table 2.1: Refactoring Detection Tools in Related Work.

Study	Year	Refactoring Tool	Detection Technique	No. of Refactoring
Dig et al. [11]	2006	Refactoring Crawler	Syntactic & Semantic analysis	7
Weissgerber and Diehl [45]	2006	Signature-Based Refactoring Detector	Signature-based analysis	10
Xing and Stroulia[48]	2008	JDevAn	Design Evolution analysis	Not Mentioned
Hayashi et al.[18]	2010	Search-Based Refactoring Detector	Graph-based heuristic search	9
Prete et al.[36], Kim et al.[23]	2010	Ref-Finder	Template-based rules reconstruction technique	63
Tsantalis et al.[42][43], Silva et al.[40]	2013 & 2016 & 2018	RefactoringMiner	Design Evolution analysis	14
Silva and Valente [39]	2017	RefDiff	Static analysis & code similarity	13

2.0.2 Refactoring Motivation

Silva et al. [40] investigate what motivates developers when applying specific refactoring operations by surveying Github contributors of 124 software projects. They observe that refactoring activities are mainly caused by changes in the project requirements and much less by code smells. Palomba et al.[34] verify the relationship between the application of refactoring operations and different types of code changes (i.e., *Fault Repairing Modification, Feature Introduction Modification, and General Maintenance Modification*) over the change history of three open source systems. Their main findings are that: 1) developers apply refactoring to improve comprehensibility and maintainability when fixing bugs, 2) improve code cohesion when adding new features, and 3) improve the comprehensibility when performing general maintenance activities. On the other hand, Kim et al. [22] do not differentiate the motivations between different refactoring types. They surveyed 328 professional software engineers of Microsoft to investigate when and how they do refactoring. When surveyed, the developers cited the main benefits of refactoring to be: improved readability (43%), improved maintainability (30%), improved extensibility (27%) and fewer bugs (27%). When asked what provokes them to refactor, the main reason provided was poor readability (22%). Only one code smell (i.e, code duplication) was mentioned (13%).

Murphy-Hill et al. [33] examine how programmers perform refactoring in practice by monitoring their activity and recording all their refactorings. He distinguished between

high, medium and low-level refactorings. High-level refactorings tend to change code elements signatures without changing their implementation e.g., *Move Class/Method, Rename Package/Class*. Medium-level refactorings change both signatures and code blocks e.g., *Extract Method, Inline Method*. Low level refactorings only change code blocks e.g., *Extract Local Variable, Rename Local Variable*. Some of the key findings of this study are that 1) most of the refactoring is floss, i.e., applied to reach other goals such as adding new features or fixing bugs, 2) almost all the refactoring operations are done manually by developers without the help of any tool, and 3) commit messages in version histories are unreliable indicators of refactoring activity because developers tend to not explicitly state refactoring activities when writing commit messages. It is due to this third point that, in this study, we do not rely on commits messages to identify refactorings. Instead, we use them to identify the motivation behind the refactoring. Tsantalis et al. [42] manually inspected the source code for each detected refactoring with a text diff tool to reveal the main drivers that motivated the developers for the applied refactoring. Beside code smell resolution, they found that introduction of extension points and the resolution of backward compatibility issues are also reasons behind the application of a given refactoring type. In another study, Wang generally focused on the human and social factors affecting the refactoring practice rather than on the technical motivations [44]. He interviews 10 industrial developers and found a list of *intrinsic* (e.g., responsibility of code authorship) and *external* (e.g., recognitions from others) factors motivating refactoring activity.

All the above-mentioned studies have agreed on the existence of motivations that go beyond the basic need of improving the system's design. Refactoring activities have been solicited in scenarios that have been coined by the previous studies as follows:

- Bug fix: refactoring the design is a part of the debugging process.
- Design improvement: refactoring is still the de facto for increasing the system's modeling quality and design defect's correction.
- Feature add/update: refactoring the existing system to account for the upcoming

functionalities.

- Non-functional attributes enhancement: refactoring helps in increasing the system's visibility to the developers. It helps in better comprehending and maintaining it.

Since these categories are the main drivers for refactoring activities, we decided to cluster our mined refactoring operations according to these groups. In order to perform the classification process, we review the studies related to commit and change history classification in the next subsection.

2.0.3 Commits Classification

A wide variety of approaches to categorize commits have been presented in the literature. The approaches vary between performing manual classification [40][28][21][9], to developing an automatic classifier [29] [16], to using machine learning techniques [20] [19] [2] [30] [27] [26] and developing discriminative topic modeling [49] to classify software changes. These approaches are summarized in Table 2.2.

Hattori and Lanza [17] developed a lightweight method to manually classify history logs based on the first keyword retrieved to match four major development and maintenance activities: Forward Engineering , Re-engineering, Corrective engineering, and Management activities. Also, Mauczka et al.[28] have addressed the multi-category changes in a manual way using three classification schemes from existing literature. Silva et al.[40] applied thematic analysis process to reveal the actual motivation behind refactoring instances after collecting all developers' responses. Further, in Chavez et al. [9], manual inspection and classification of refactoring instance in a commit as root-canal refactoring or floss refactoring have been proposed. An automatic classifier is proposed by Hassan [16] to classify commits messages as a bug fix, introduction of a feature, or a general maintenance change. Mauczka et al. [29] developed an Eclipse plug-in named Subcat to classify the change messages into Swanson's original category set (i.e., Corrective, Adaptive and Perfective [41]), with additional category "Blacklist". He automatically assesses if a change to the software was due to a bug fix or refactoring based on a set of keywords in

the change messages. Along with the progress of methodology, Hindle et al. [19] proposed an automated technique to classify commits into maintenance categories using seven machine learning techniques. To define their classification schema, they extended Swanson's categorization [41] with two additional changes: Feature Addition, and Non-Functional. They observed that no single classifier is best. Another experiment that classifies history logs was conducted by Hindle et al. in [20]. Their classification of commits involves the non-functional requirements (NFRs) a commit addresses. Since the commit may possibly be assigned to multiple NFRs, they used three different learners for this purpose beside using several single-class machine learners. Amor et al. [2] had a similar idea to [19] and extended the Swanson categorization hierarchically. They, however, selected one classifier (i.e. Naive Bayes) for their classification of code transaction. Moreover, Maintenance requests have been classified into type using two different machine learning techniques (i.e. Naive Bayesian and Decision Tree) in [27]. Another work [30] explores three popular learners to categorize software application for maintenance. Their results show that SVM is the best performing machine learner for categorization over the others. Recent work [26] automatically classified commits into three main maintenance activities using three classification models , namely, J48 , Gradient Boosting Machine (GBM) , and Random Forest (RF). They found that RF model outperforms the two other models.

2.0.4 Study Replication

Biegel et al. [6] replicated Weissgerber and Diehl's work [45], which considers a signature-based refactoring detection technique using CCFinder to determine the similarity between the body of old and new code elements. Biegel et al. then extend the replicated experiment by plugging in the two similarity metrics (JCCD [5] and Shingles [8]) to investigate their influence on the signature-based detection technique. Bavota and Russo [4] present a large-scale empirical study as a differentiated replication of the study by Potdar and Shihab [35] in the detection of Self-Admitted Technical Debt (SATD). In particular, they ran a study across 159 Java open source systems as compared to 4 projects analyzed in [35]

Table 2.2: Commits Classification in Related Work.

Study	Year	Single/Multi-labeled	Manual/Automatic	Classification Method	Category
Amor et al. [2]	2006	Yes/No	No/Yes	Machine Learning Classifier	Swanson’s category Administrative
Hattori & Lanza. [17]	2008	Yes/No	No/Yes	Keywords-based Search	Forward Engineering Reengineering Corrective Engineering Management
Hassan [16]	2008	Yes/No	No/Yes	Automated Classifier	Bug Fixing General Maintenance Feature Introduction
Hindle et al. [21]	2008	Yes/Yes	Yes/No	Systematic Labeling	Swanson’s category Feature Addition Non-Functional
Hindle et al. [19]	2009	Yes/No	No/Yes	Machine Learning Classifier	Hindle et al.[21] category
Mauczka et al. [29]	2012	Yes/No	No/Yes	Subcat tool	Swanson’s category Blacklist
Tsantalis et al. [42]	2013	Yes/No	Yes/No	Systematic Labeling	Code Smell Resolution Extension Backward Compatibility Abstraction Level Refinement
Mauczka et al. [28]	2015	Yes/Yes	Yes/No	Systematic Labeling	Swanson’s category Hattori & Lanza’s [17] category Non-Functional
Yan et al. [49]	2016	Yes/Yes	No/Yes	Topic Modeling	Swanson’s category
Chávez et al. [9]	2017	Yes/No	Yes/No	Systematic Labeling	Floss Refactoring Root-canal Refactoring
Levin & Yehudai [26]	2017	Yes/No	No/Yes	Machine Learning Classifier	Swanson’s category

to investigate their (1) diffusion (2) evolution over time, and (3) relationship with software quality. Although they aim to address same research questions reported in [35], they use different experimental procedures to answer these questions. For instance, to explore the evolution of SATD (i.e. the percentage of SATD removed after it's introduced), Potdar and Shihab perform an analysis at the release-level, whereas Bavota and Russo use commit history to detect SATD. Our work represents a replication and an extension of the work by Tsantalis et al. [42]. In summary, for each research question, 1) we replicate the small-scale/qualitative analysis by analyzing three projects: Hadoop, OrientDB, and Camel; 2) we extend the study by performing large-scale/quantitative analysis on a larger set of projects. For the qualitative analysis, we randomly selected Hadoop, OrientDB and Camel among 11 candiate projects, which were hand-selected in a previous study based on their popularity, coverage of a wide variety of domains such as IDE, DBMS, and integration [26]. Revisiting the qualitative analysis allows a direct comparison with the results of the previous studies [33, 42]. Additionally, the extended analysis over a larger set of projects not only challenges the scalability of the qualitative analysis, but also allows the discovery and exploration of refactoring trends that may not have appeared due to the previous study's limited set of projects and interviewed developers.

We summarize the similarities and differences between this study and the original study as follows:

- *Project size:* To increase the generalizability of the results, our study is conducted in 1,706 open source Java projects compared to the 3 projects analyzed in [42].
- *Refactoring detection tool:* Since the approach relies on mining refactorings, we use the Refactoring Miner tool, which detects 14 refactoring types. Tsantalis et al. [42] developed and used the previous version of this tool that supports only 11 refactoring operations.
- *Refactoring operations on production and test code (RQ1):* For each of the 3 projects,

both studies identify test-specific and production-specific code entities through package organization analysis followed by keyword analysis. Just like Tsantalis et al.[42], we conduct a manual inspection of the results related to the qualitative analysis. For the quantitative analysis, we follow a pattern matching approach to distinguish between production and test files.

- *Refactoring contributors (RQ2):* Both studies explore which developers are responsible for refactorings by comparing the percentage of refactorings performed by the top refactoring contributors with all the rest.

- *Refactoring activities around the release points (RQ3).* Tsantalis et al. [42] select windows of 80 days around each project major release date, dividing each release into two groups of 40 days to count refactorings before and after the release points. In our work, we consider all of the project releases, making sure that there is no overlap between refactorings for each release.

- *The relationship between refactoring and testing (RQ4):* Both this and the replicated study investigates whether the refactoring activity on production files is accompanied by the addition or modification of test files by detecting them in the commit history; focusing on testing periods with intense activity and setting the end of testing period as a pivot point in the window analysis.

- *Refactoring motivation (RQ5):* Both studies postulate the main motivations behind the applied refactoring. We add an automatic classification of commit messages using Random Forest classifier. The previous work [42] performs a systematic labeling of the refactoring instances by manually inspecting the text diff reports along with their commit logs.

Chapter 3

Methodology

To answer our research questions, we conducted a three phased approach that consisted of: (1) selection of GitHub repositories, (2) refactoring detection and (3) commits classification.

Figure 3.1: Approach Overview.

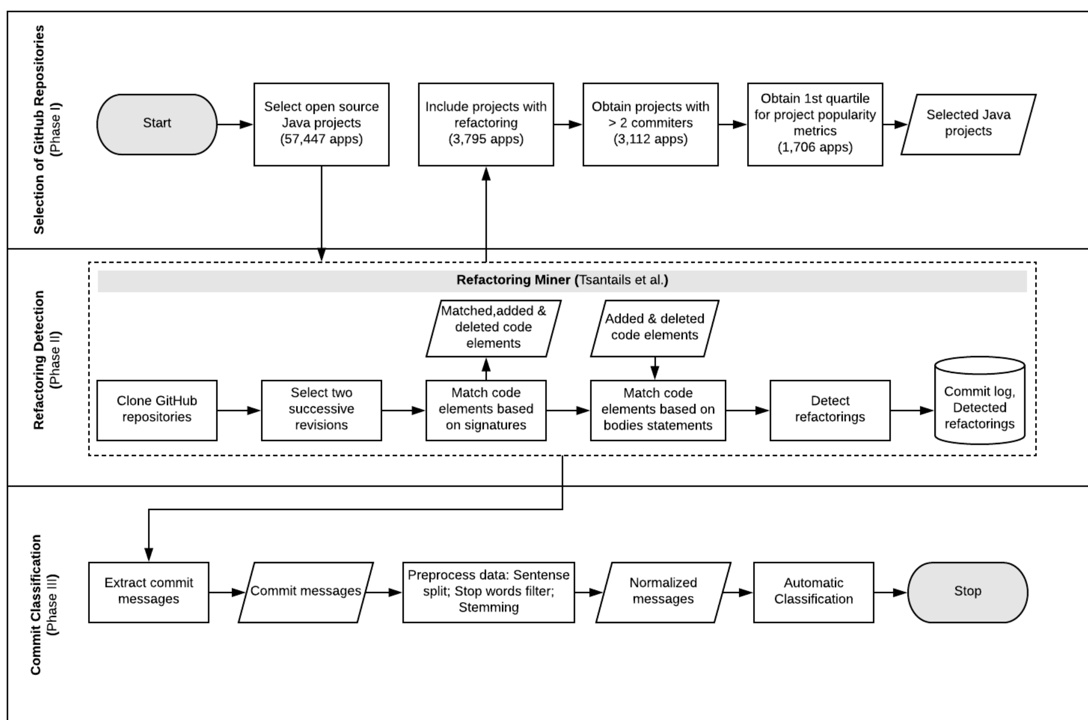


Table 3.1: Refactoring Types (Extracted from Fowler [14]).

Refactoring Type	**Description**
Extract Method	A code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.
Move Class	A class isn't doing very much. Move all its features into another class and delete it
Move Attribute	A field is, or will be, used by another class more than the class on which it is defined
Rename Package	The name of a package does not reveal its purpose. Change the name of the package
Rename Method	The name of a method does not reveal its purpose. Change the name of the method
Rename Class	The name of a class does not reveal its purpose. Change the name of the class
Move Method	A method is, or will be, using or used by more features of another class than the class on which it is defined
Inline Method	When method's body is just as clear as its name. Put the method's body into the body of its callers and remove the method
Pull Up Method	You have methods with identical results on subclasses. Move them to the superclass
Pull Up Attribute	Two subclasses have the same field.Move the field to the superclass
Extract Superclass	There are two classes with similar features. Create a superclass and move the common features to the superclass
Push Down Method	The Behavior on a superclass is relevant only for some of its subclasses. Move it to those subclasses
Push Down Attribute	A field is used only by some subclasses. Move the field to those subclasses
Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. Extract the subset into an interface

3.0.1 Phase 1: Selection of GitHub Repositories

To perform this study, we randomly selected 57,447 projects from a list of GitHub repositories [1], while verifying that they were Java based since that is the only language supported by Refactoring Miner. After scanning all projects, those with no detected refactoring were discarded, and 3,795 projects were considered. To ensure that the selected projects properly fit in answering the above-mentioned research questions. We apply further selection criteria using *Reaper* [**?**], an open source tool for selecting curated, and well engineered Java project, based on user-set rules. Candidate repositories were selected by ranking them based on the number developers who made commits to the project, the number of commits, the number of versions, the Stargazers count, number of forks, and number of subscribers. Similar to [37][40], we filter out the projects that have less than the 1st quartile for all of these engagement metrics to ensure that these projects have a significant maintenance activity, and maintained by a large set of developers for a significant period of time. The final selection results in 1,706 projects analyzed that have a total of 1,208,970 refactoring types. An overview of the projects is provided in Table 3.2.

Table 3.2: Projects Overview.

Item	Count
Total projects	57,447
Projects with releases	521
Projects with identified refactorings	3,795
Refactoring commits	322,479
Refactoring operations	1,208,970
Production-based refactoring operations	950,739
Test-based refactoring operations	258,231
Analyzed Projects - Refactored Code Elements	
Code Element	**# of Refactorings**
Class	329,378
Method	718,335
Attribute	97,516
Package	18,334
Interface	8,096

3.0.2 Phase 2: Refactoring Detection

For the purpose of extracting the entire refactoring history of each project, we used the Refactoring Miner tool, proposed by Tsantalis et al. [40]. Refactoring Miner is designed to analyze code changes (i.e., commits) in git repositories to detect any applied refactoring. The list of detected refactoring types by this tool is described in Table 3.1. Two phases are involved when detecting refactorings using the Refactoring Miner tool. In the first phase, the tool matches two code elements in top-down order (starting from classes and continuing to methods and fields) based on their signatures, regardless of the changes that occurred in their bodies. If two code elements do not have an identical signature, the tool considers them as added or deleted elements. In the second phase, however, these unmatched code elements (i.e. potentially added or removed elements) are matched in a bottom-up fashion based on the common statements within their bodies. The purpose of this phase is to find code elements that experienced signature changes. It is important to note that Refactoring

Miner detects refactoring in a specific order, applying the refactoring detection rules as discussed in Tsantalis et al. [43].

We decided to use Refactoring Miner for consistency, since it was both proposed and used by the authors of the study we replicated. Additionally, Refactoring Miner has shown promising results in detecting refactorings compared to the state-of-art available tools [40] and is suitable for a study that requires a high degree of automation since it can be used through its external API. The Eclipse plug-in refactoring detection tools we considered, in contrast, require user interaction to select projects as inputs and trigger the refactoring detection, which is impractical since multiple releases of the same project have to be imported to Eclipse to identify the refactoring history. It is important to note that, while we were conducting this study, a more recent tool named *refDiff* was developed with the same goal in mind; to mine refactorings from open source repositories. In future studies, it might be possible for us to use this tool for comparison purposes.

Table 3.3: Classification Categories.

Category	**Description**
Feature	Implementation of a new/updated feature(s).
BugFix	Application of bug fixes.
Design	Restructuring and repackaging the system's code elements to improve its internal design.
Non-Functional	Enhancement of non-functional attributes such as testability, understandability, and readability.

3.0.3 Phase 3: Commits Classification

Our classification process has three main steps: (1) choice of the classifier, (2) data preprocessing, and (3) preparation of the training set.

Choice of the classifier

Selecting the proper classifier for optimal classification of the commits is rather a challenging task[13]. Developers provide a commit message along with every commit they make to the repository. These commit messages are usually written using natural language, and generally convey some information about the commit they represent. In this study we were dealing with a multi-class classification problem since the commit messages are categorized into four different types as explained in Table 3.3. Since it is very important to come up with an optimal classifier that can provide satisfactory results, several studies have compared K-Nearest Neighbor (KNN), Naive Bayes Multinomial, Linear Support Vector Classifier (SVC), and Random Forest in the context of commit classification into similar categories [26, 25]. These studies found that Random Forest gives satisfactory results. We investigated each classifier ourselves and came to the same conclusion using common statistical measures (precision, recall, and F-score) of classification performance to compare each. Therefore, we used Random Forest as our classifier. One important observation we had for the selection of the classifier was that the precision, recall, and F-score results from the Linear SVC and Random Forest are pretty close. However, while applying the trained model on a new dataset, Linear SVC didn't provide satisfactory results. After further investigation, we figured out that since linear SVC is an inherently binary classifier and in this case it uses One-vs-Rest strategy for multi-class classification, it provides less satisfactory results when applied to a new dataset. Table 3.4 shows the performance comparison of different classifiers we evaluated in this study.

Table 3.4: Comparison of Different Classifiers.

Classifier	Precision	Recall	F1-Score
Random Forest	0.99	0.99	0.99
Linear SVC	0.99	0.98	0.99
Naive Bayes Multinomial	0.95	0.94	0.94
K-Nearest Neighbor	0.88	0.85	0.85

Data Preprocessing

We applied a similar methodology explained in [25] for text pre-processing. In order for the commit messages to be classified into correct categories, they need to be preprocessed and cleaned; put into a format that the classification algorithms will accept. The first step is to tokenize each commit by splitting the text into its constituent set of words. After that, punctuation, stop words, and numbers are removed since they do not play any role as features for the classifier. Next, all the words are converted to lower case and then stemmed using the Porter Stemmer in the NLTK package. The goal of stemming is to reduce the number of inflectional forms of words appearing in the commit; it will cause words such as "performance" and "performing" to syntactically match one another by reducing them to their base word– "perform". This helps decrease the size of the vocabulary space and improve the volume of the feature space in the corpus (i.e., a commit message). Finally, each corpus is transformed into vector space model (VSM) using the TF-IDF vectorizer in Python's SKlearn package to extract the features.

Building the Training Set

We started to build the training set by extending an existing labeled dataset [28], and manually classifying 3500 commit messages from Java open source projects. We used the same model for the manual classification of the commits as explained in [19]. Next, we asked a set of four software engineering graduate students to check and evaluate those commit messages. Each student was assigned to evaluate one category of the commit messages

and we asked them to mark the commit messages that do not belong to any category as unknown. The unknown category gathers all commits that were hard to manually classify. A commit was considered hard to classify if it was too short, ambiguous, or the software engineering students could not reach an agreement about how it should be classified. We used the unknown category's commits to train the classifier to recognize confusing commits so that similarly hard-to-classify commits ended up in their own category, causing our classifier to be more accurate. Also, in order to mitigate the subjectivity involved in the manual classification, we followed the method explained in [26]. We randomly selected commit messages from three different categories and had those commits evaluated again by one of the members of the graduate student team who did not evaluate those commits from those three categories during the first pass. It is important to note that by manual classification we provide the correct class label for each commit type. Finally, we divided the dataset randomly into 75% training set and 25% testing set to evaluate the performance of different classifiers explained at the beginning of this section.

Chapter 4

Analysis & Discussion

This section reports our experimental results and aims to answer the research questions in Section 1. The dataset of refactorings along with their classification is available online[1] for replication and extension purposes.

Because our study examines many systems and many refactorings, we present the results for each research question as both an agglomeration of all data in the study (i.e., in the subsections labeled *automated*) and as a sample of three systems (i.e., in the subsections labeled *manual*). These three systems were chosen based on the criteria used in [26]; system characteristics are illustrated in Table 4.1. Additionally, these systems contained a significant (95% confidence level) number of refactorings and test files. Therefore, we report both a global view and a fine-grained view of our results to provide a stronger understanding of the data.

Table 4.1: Characteristics of Three Software Projects under Study.

Software Project	All Commits	Ref Commits	Refactorings	Contributors	Refactoring Contributors
apache/hadoop[2]	17,510	1,462	4,207	114	73
orientechnologies/orientdb[3]	16,246	1,737	5,063	113	35
apache/camel[4]	30,781	2,863	8,422	368	73

[1] Link omitted for double-blind review

4.1 RQ1: Do software developers perform different types of refactoring operations on test code and production code?

To answer this research question, we start by scanning all the changed files in the set of analyzed commits. Then, we identify test files by tagging any file in the project's nested repositories whose code element (class, method) identifier contains *test*. Therefore, any refactoring operations performed on test files, are considered test refactorings.

Automated

Table 4.2: Refactoring Frequency in Production and Test Files in all Projects Combined.

Refactoring Type	In Production Files	In Test Files
Extract Method	**17.79%**	12.36%
Move Class	**15.86%**	**17.08%**
Move Attribute	5.81%	4.00%
Move Method	3.00%	2.16%
Inline Method	3.07%	1.71%
Pull Up Method	6.51%	3.90%
Pull Up Attribute	2.99%	2.71%
Extract Superclass	0.94%	0.74%
Push Down Method	0.75%	0.27%
Push Down Attribute	0.31%	0.32%
Extract Interface	2.82%	0.24%
Rename Package	1.86%	0.61%
Rename Method	**28.51%**	**39.02%**
Rename Class	10.54%	**14.88%**

An overview of the detected refactorings is provided in Table 4.2. We found that around 77% of all mined refactorings were applied, on average, to production files, and 23% applied to test files. For comparison, the previous study, Tsantalis et al. [42], found that, on average, 73% of refactorings were applied to production files while 27% were

applied to test files. Furthermore, we found that the topmost applied refactorings (shown in **bold** in Table 4.2) are respectively *Rename Method, Extract Method, and Move Class*. Our topmost used refactoring types overlaps with the findings of the previous study [40] since they ranked *Extract Method, and Move Class* as their most popular types. The absence of *Rename Method* from their findings can be explained by the fact that their tool did not support its detection when they conducted their study.

As stated prior, the difference in the distribution of refactorings in production/test files between our study and the previous one is also due to the size (number of projects) effect of the two groups under comparison. To mitigate this, we selected three samples in which both the number refactorings and the number of test files are statistically significant with respect to whole set of projects (95% confidence level). The project details are enumerated in Table 4.1.

Manual

We now examine on a per-system basis. In Hadoop, RefactoringMiner detected a total of 4124 refactorings, 3131 (76%) of them were applied on production code, while 993 (24%) were test code refactorings. Out of the total 5044 detected refactoring in OrientDB, 4369 (87%) were production code refactorings and 675 (13%) were test code refactorings. In Camel, there were 6283 (76%) production code refactorings and 2039 (24%) were test code refactorings out of 8322 total detected refactorings. For each project, we use the Mann–Whitney U test to compare between the group of refactorings applied to production and test files. We found that the number of applied refactorings in production files is significantly higher than those applied to test classes with a p-value=0.0030 (resp. 0.0013, 0.0019) in Hadoop (resp. OrientDB and Camel). Furthermore, we use the Kruskal Wallis H test to analyze whether developers use the same set of refactoring types with the same ratios when refactoring production and test files. In OrientDB (resp. Camel) the hypothesis of both groups having the same distribution was rejected with a p-value=0.0024 (resp. 0.0366). As for Hadoop, it was not statistically significant (p-value=0.0565).

Table 4.3: Frequency of Refactorings per Type on Production and Test Files in Hadoop, OrientDB and Camel.

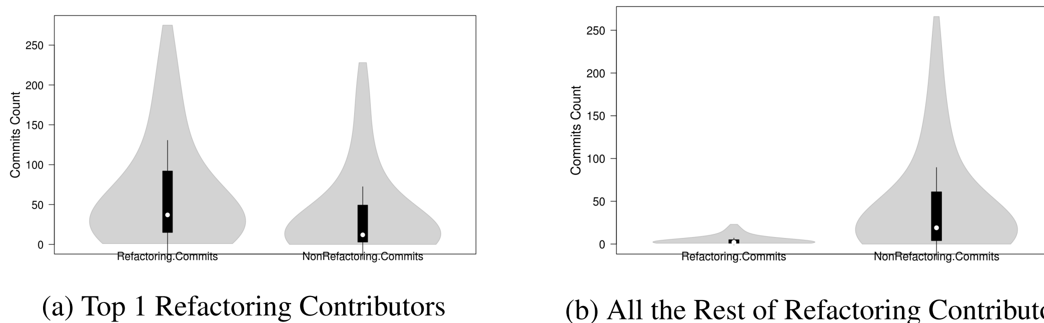
	Production Code			Test Code		
Refactorings	**Hadoop**	**OrientDB**	**Camel**	**Hadoop**	**OrientDB**	**Camel**
Extract Method	951	1,615	1,248	392	163	229
Move Class	220	200	416	66	90	229
Move Attribute	174	141	403	17	1	90
Move Method	220	101	326	29	21	21
Inline Method	142	280	116	23	13	43
Pull Up Method	176	267	489	41	38	171
Pull Up Attribute	187	127	171	62	8	76
Extract Superclass	27	31	57	17	11	10
Push Down Method	3	46	36	0	14	9
Push Down Attribute	2	19	16	0	2	5
Extract Interface	28	33	58	3	0	2
Rename Package	27	47	54	0	1	6
Rename Method	832	1,120	2,007	300	208	790
Rename Class	142	342	886	43	105	358
Total	3131	4369	6283	993	675	2039

Our findings are aligned with the previous study; we found that developers refactor production files significantly more than test files. We also could not confirm that developers uniformly apply the same set of refactoring types when refactoring production and test files.

4.2 RQ2: Which developers are responsible for refactorings?

Automated

Figure 4.1a and 4.1b show violin plots with the commit distribution of both the most frequent refactoring contributor (i.e., Figure 4.1a) and the rest of the refactoring contributors (i.e., Figure 4.1b) on refactoring and non-refactoring commits for all 1,706 systems. We also provide plots 4.2a and 4.2b to compare their contributions. From the figure, we see that all refactoring contributors perform refactoring and non-refactoring commits. However, we notice that a single developer has taken over the responsibility for applying refactorings due to the significant number of refactoring commits performed in comparison to non-refactoring commits. All the rest of the refactoring contributors are performing more non-refactoring commits and less refactorings than the main refactoring contributors. Across the large-scale projects, we cannot conclude the key role of the top 1 refactoring contributors to examine in contrast with the results found in the previous study.



(a) Top 1 Refactoring Contributors

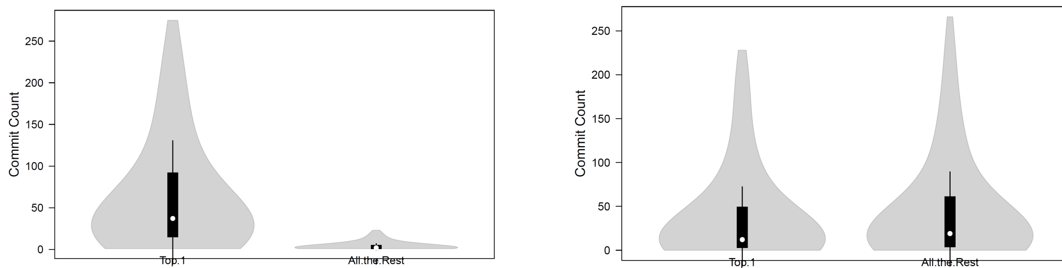
(b) All the Rest of Refactoring Contributors

Figure 4.1: Top 1 and all the Rest of Refactoring Contributors for all Projects Combined

Manual

Figure 4.3 portrays the distribution of the refactoring activities on production code and test code performed by project contributers for each system we examined.

The Hadoop project has a total of 114 developers. Among them are 73 (64%) refactoring contributors. As we observe in Figures 4.3a and 4.3b, not all of the developers are major



(a) Refactoring Commits for Top 1 and all the Rest

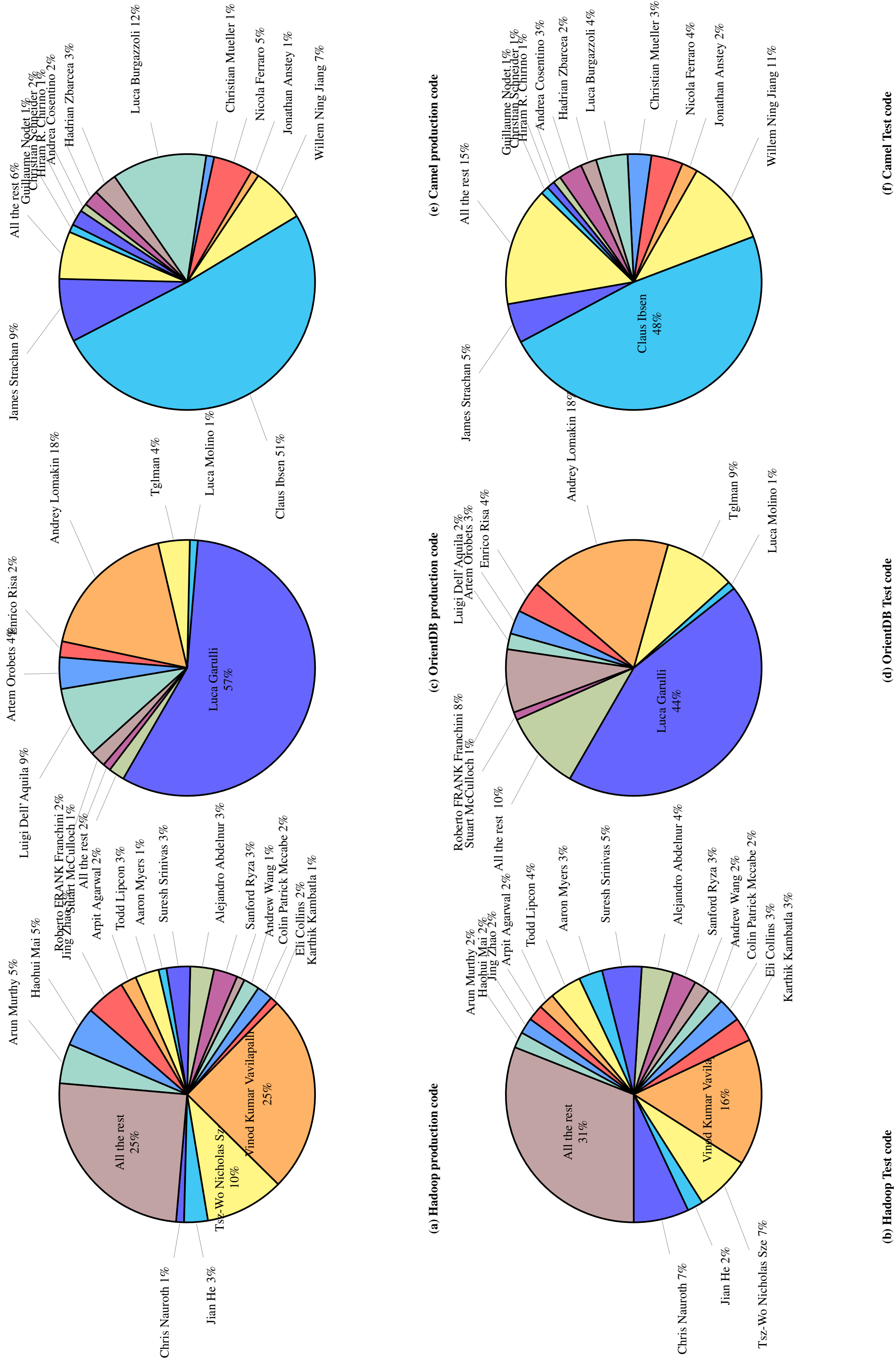
(b) Non-Refactoring Commits for Top 1 and all the Rest

Figure 4.2: Total Refactoring and Non-Refactoring Commits for Refactoring Contributors for all Projects Combined

refactoring contributors. The main refactoring contributor has a refactoring ratio of 25% on production code and 10% on test code. Figure 4.3c and 4.3d present the percentage of the refactorings for the OrientDB production code and test code. Out of the total 113 developers, 35 (31%) were involved refactoring. The top contributor has a refactoring ratio of 57% and 44% on respectively production and test code. For Camel, in Figures 4.3e and 4.3f, 73 (20%) developers were on the refactoring list out of 368 total committers. The most active refactoring contributor has high ratios of 51% and 48% respectively in production and test code. It is important to note that very few developers applied refactorings exclusively on either production code or test code for the three projects under study. Interestingly, we found that the most active refactoring contributors of each project were either development lead or experienced developers.

Refactoring is performed by a subset of developers, in both production and test files. Their public profiles show they they are senior developers that are experienced with the project. Our results are in agreement with [42], which also found that the top refactoring contributors had a management role within the project.

Figure 4.3: Refactoring Contributors in Production and Test Files in Hadoop, OrientDB and Camel.

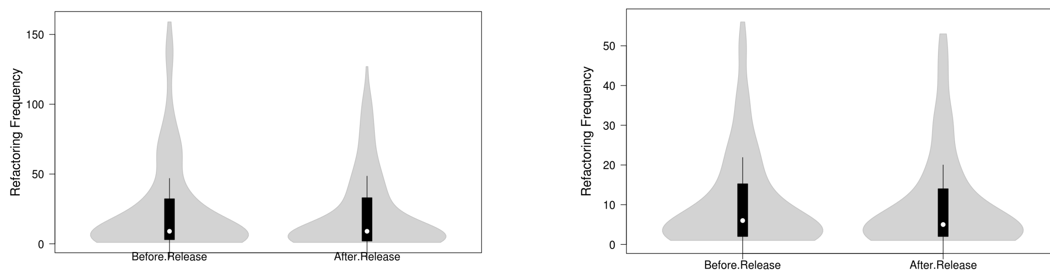


4.3 RQ3: Is there more refactoring activity before project releases than after?

Automated

In the previous study [42], they manually filtered minor releases and selected only major project releases within an 80-day window, divided into two groups of 40 days before and after each release. To perform our study, we have considered all release dates of the projects. In order to avoid any overlap between refactoring operations, we split the analysis into the periods before and after the release at the midpoint between two release points because the time between two releases is not evenly spaced.

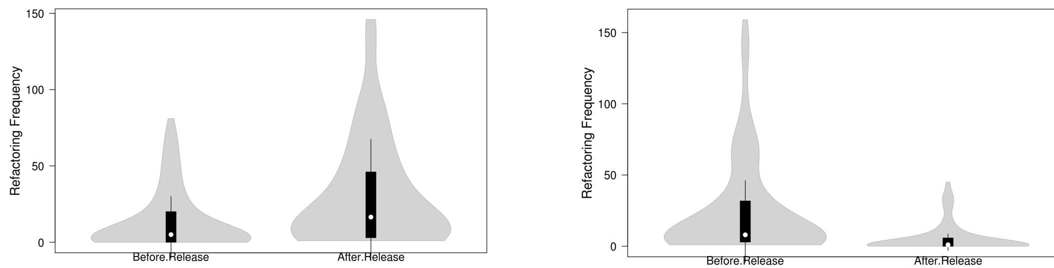
Figure 4.4a and 4.4b present the distribution of refactoring activity before and after all project releases on production and test code. As shown in Figure 4.4a, production-based refactorings are more frequent before release than after. Similarly, in Figure 4.4b, test-based refactorings are more frequent before release dates. We are also interested in reporting the results in detail by comparing refactoring density in three situations: (1) when refactorings after the releases are greater than before, (2) when refactorings after the releases are fewer than before, and (3) when refactorings after the releases are equal to the one before. We found that the first situation is more frequent than the other two. Additionally, frequency distribution of refactoring types before and after the release for both production and test files is depicted in Figure 4.7



(a) Production Files - Release Point Comparison.

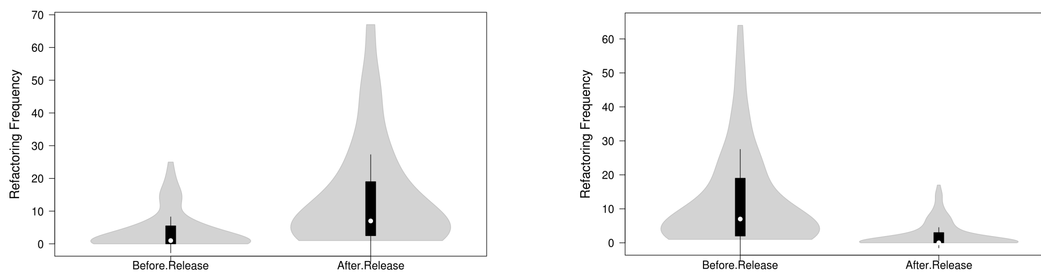
(b) Test Files - Release Point Comparison.

Figure 4.4: Refactoring Density - Release Point Comparison.



(a) After > Before - Release Point Comparison. (b) After < Before - Release Point Comparison.

Figure 4.5: Refactoring Density - Release Point Comparison (Production Files).



(a) After > Before - Release Point Comparison. (b) After < Before - Release Point Comparison.

Figure 4.6: Refactoring Density - Release Point Comparison (Test Files).

Manual

In order to study refactoring activities along the lifetime of the three projects, our approach involved two steps: (1) Selecting release date, and (2) identifying refactorings chronologically executed around the selected release dates. In the first step, we randomly selected seven public release dates for each project under study, making sure that they did not overlap or only slightly overlap, on a 20 day window that can reach up to 40 days; similar to previous work [42]. In the second step, we select the commits that are collocated in the 40 day window, centered by each release date. We extract the refactorings using Refactoring Miner and examine refactoring events for each version supported in the release; counting the number of refactoring operations performed before and after a release.

Starting with the Hadoop project, as Figure 4.8 shows, there are significant peaks in refactoring activity on the same day of the release and two days after the release day.

Table 4.4: Percentage of Refactorings on Production and Test Files in all Projects Combined.

Comparison Type	Production Code	Test Code
After > Before	31%	34%
After < Before	66%	62%
After = Before	3%	4%

Further analysis showed constant refactoring activities in the 20 days window before and 20 days window after the release day in multiple release points. In comparison with the previous study results, no significant refactoring patterns were identified around the period of release dates.

Next, we examine the OrientDB project. It is apparent from Figure 4.9 that there are constant refactoring activities within a three week period before and after the release day. A significant increase to a peak of 41 refactoring events can be observed 14 days before the release day, but it is only observed for one release and it cannot be generalized.

Finally, the Camel project observation was not different from the two previous ones. Figure 4.10 demonstrates a similar behavior of constant refactoring activity in the 40 days leading up to the release day and throughout the 53 days after the release day. What is different about Figure 4 is the significant peak of about 105 refactoring events on the 40th day after the release point. From there, refactoring activity returned to normal with no significant peaks. Moreover, between 41 and 60 days before the project release day, there is no notable observation about refactoring activity.

Tsantails et al [42] showed that refactoring activity is high before the major release dates than after. This is consistent with the findings presented here for large-scale projects, but we noticed no significant, generalizable patterns in refactoring activity before and after projects major release dates for small-scale projects.

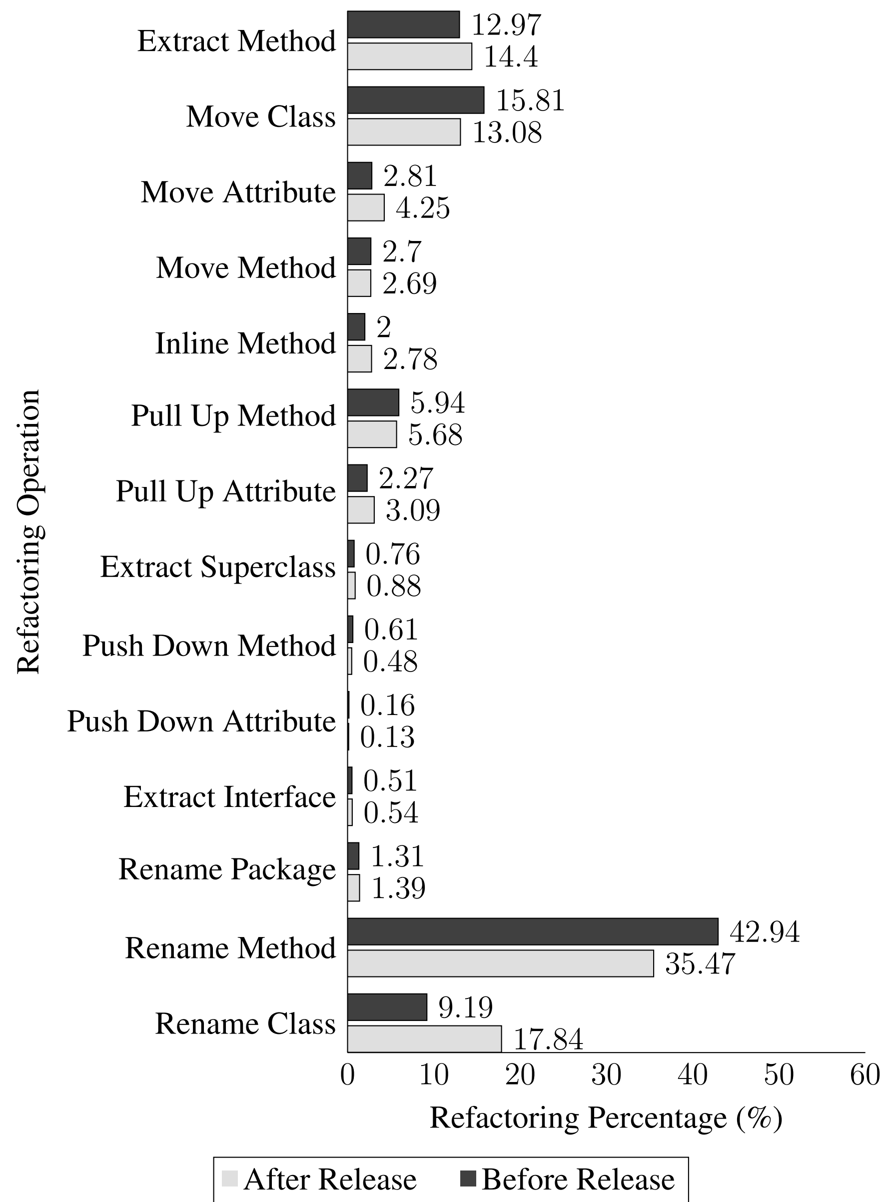


Figure 4.7: Frequency Occurrence of each Refactoring Operation before and after Release

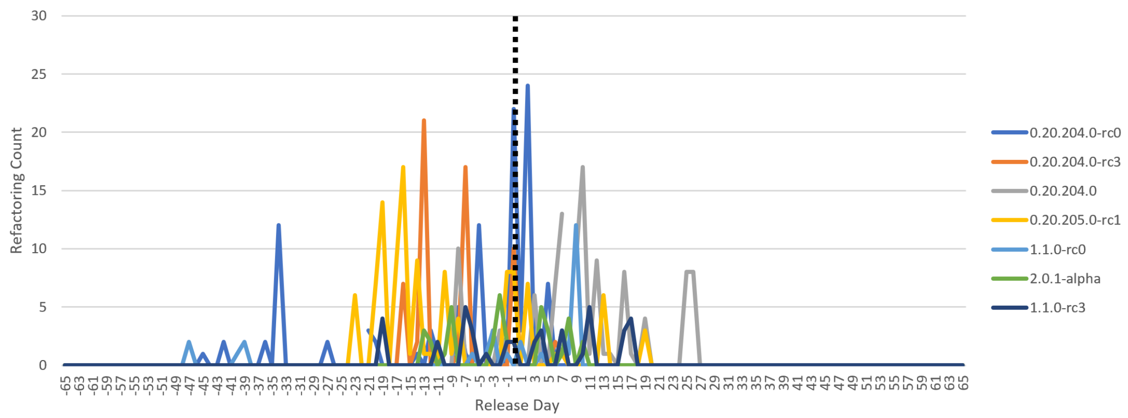


Figure 4.8: Refactoring Activity Comparison (Release)- Hadoop.

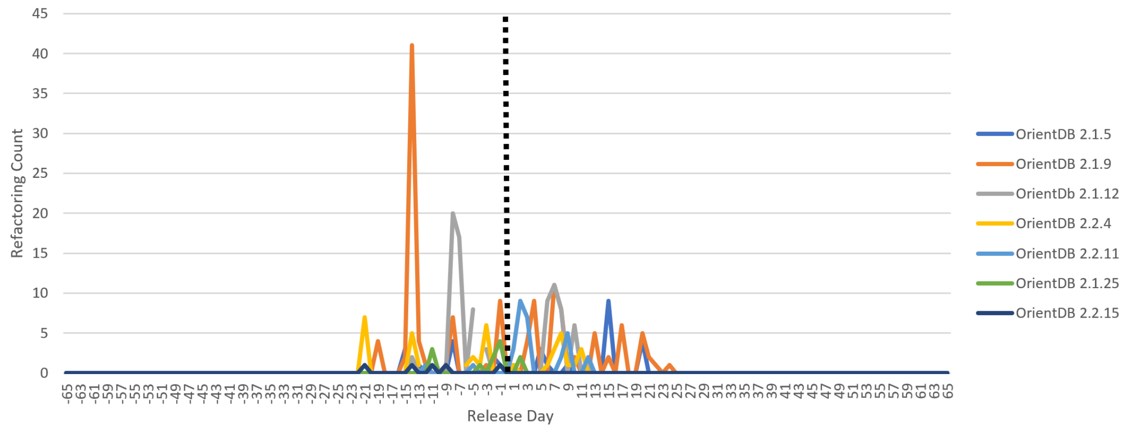


Figure 4.9: Refactoring Activity Comparison (Release)- OrientDB.

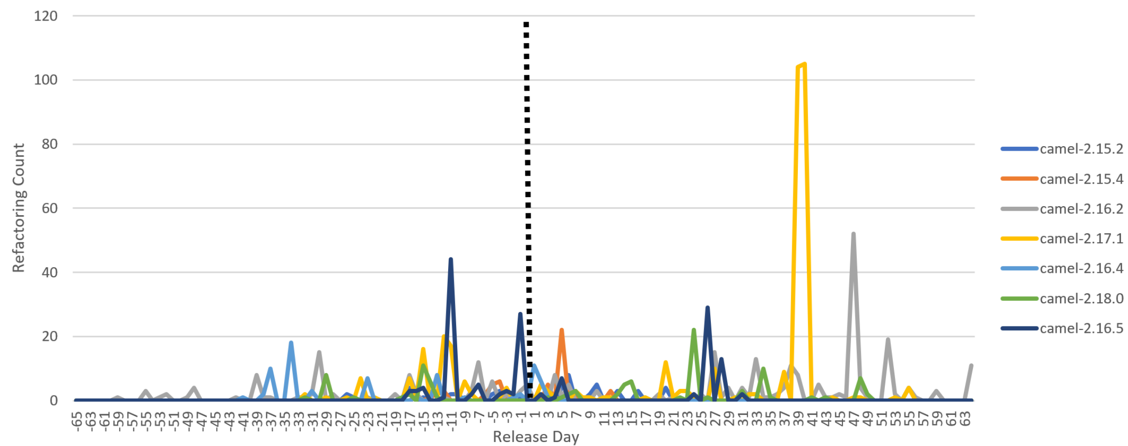


Figure 4.10: Refactoring Activity Comparison (Release)- Camel.

4.4 RQ4: Is refactoring activity on production code preceded by the addition or modification of test code?

Automated

Since identifying the end of testing periods for each project require manual work, we could not perform the analysis to conclude the relationship between refactoring and testing for large-scale projects.

Manual

Our methodology to study the relationship between refactorings and the test code changes is composed of three steps: (1) detecting test activity peaks, (2) selecting testing periods based on testing peaks, and (3) identifying refactorings chronologically executed during the testing periods. For each of the three projects, we first apply the same procedure for identifying test files as in our methodology for RQ1. We then rank commits having the highest number of added/changed test files. By monitoring commits that have high volume of testing activity, we identified commits that were *hotspots*; commits with significant code churn. We selected a window of 40 days around the end of testing periods for each system, splitting the window into groups of 20 days before and after the testing point. Lastly, we counted refactoring occurrences around each of the testing periods.

We begin with the Hadoop project. Figure 4.11 presents the experimental data and depicts the refactoring events around the five testing periods. From the graph, we can see high refactoring activity around the testing periods(including the same day of testing periods). There is a clear trend of increasing the number of refactoring events one week before the end of testing periods.

Next, we examine OrientDB. Figure 4.12 provides the results. From the chart, we can observe that, by far, the greatest refactoring activity happened the same day the test period ended. Comparing the two results of Hadoop and OrientDB, we see that there is a substantial increase in refactoring events before and on the same day of testing periods.

Finally, we examine the Camel project. There is common pattern between the Hadoop

and the Camel projects in terms of active refactoring activity around testing periods. Comparatively, refactoring activity in Camel peaks earlier and higher than Hadoop (about 32 and 24 refactoring events, respectively).

This outcome is contrary to that of Tsantalis et al. [42] who found that there are high refactoring activity during testing period than after. In our study, we found that there is a substantial refactoring activity before and after the end of testing periods.

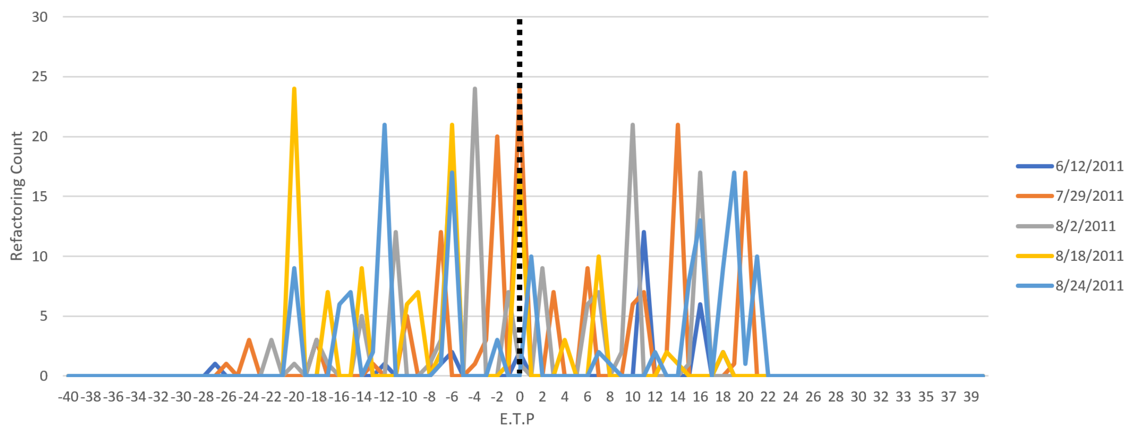


Figure 4.11: Refactoring Activity Comparison (Test)- Hadoop.

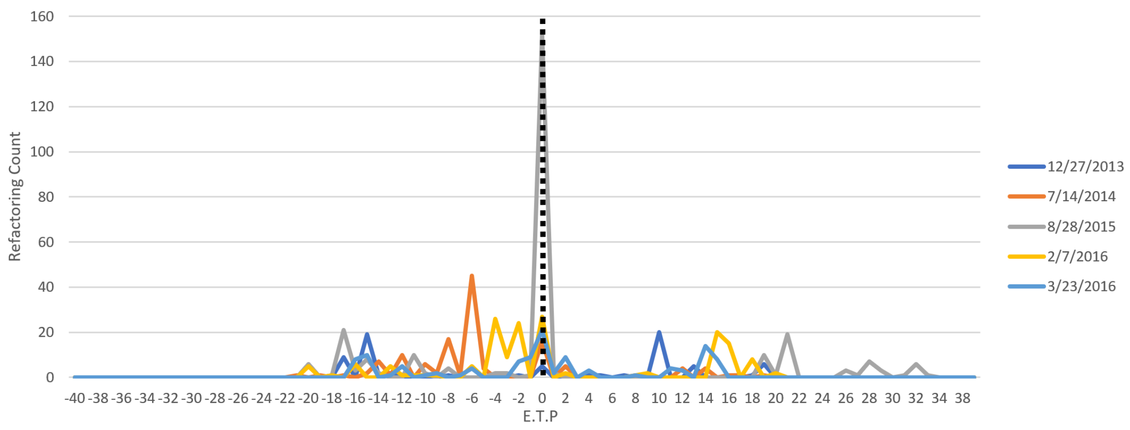


Figure 4.12: Refactoring Activity Comparison (Test)- OrientDB.

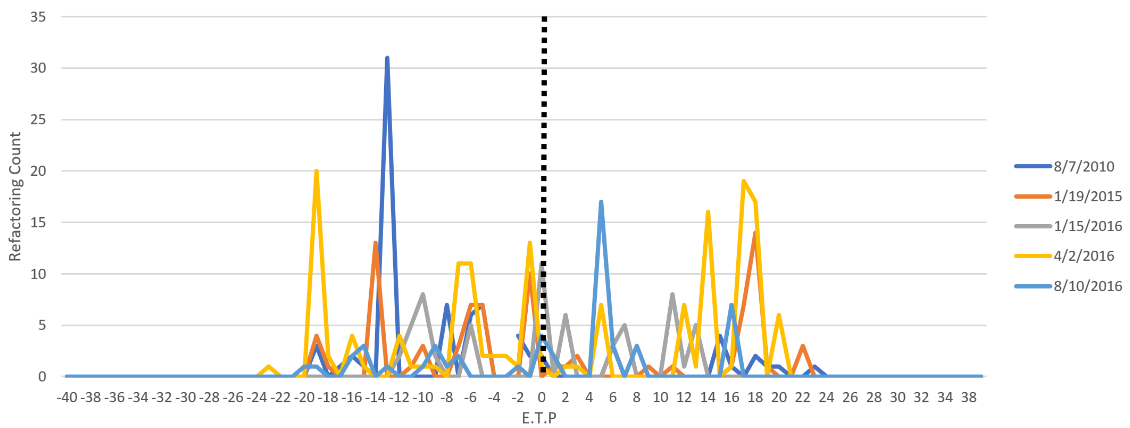


Figure 4.13: Refactoring Activity Comparison (Test)- Camel.

4.5 RQ5:What is the purpose of the applied refactorings?

Automated

To answer this research question, Figure 4.14 shows the categorization of commits, from all projects combined. Interestingly, the feature and design categories had the highest number of commits with a slight advantage to the first category, since its ratio was 27%, while the design category had a ratio of 25%. Surprisingly, bugFix was the third most popular category for refactoring-related commits with 14%, in front of the non-functional category, which had a ratio of 8%. It is important to note that the unclassified commits are those gathered by the unknown category that we have explained earlier, in Section 3.0.3.

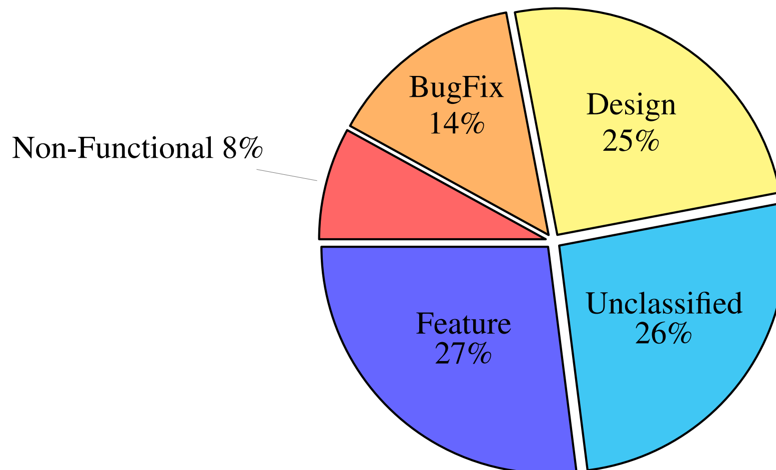


Figure 4.14: Percentage of Classified Commits per Category in all Projects Combined.

Although there was a limited evidence which refactoring tactic is more common [32], if we consider feature and bugFix categories to be containers for floss refactorings, while design and non-functional commits are labeled root-canal, then the percentage of floss operations is higher than the percentage of root-canal operations. This aligns with the survey results of Murphy-Hill et al. [33], stating that floss refactoring is a more frequent refactoring tactic than pure refactoring (i.e, root-canal refactoring). This observation agrees with the findings of Silva et al. [40] as they also conclude that *refactoring activity is mainly driven by changes in the requirements (new feature and fix bugs requests) and much less by*

code smell resolution.

To better analyze the existence of any patterns on the types of refactorings applied in each category, Figure 4.15 presents the distribution of refactoring types in every category. The Rename Method type was dominant in all categories except for the design. Extract Method was also popular among all categories. Next, two class-level types, namely Rename Class and Move Class, were ranked respectively third and fourth generally in all categories. Then the remaining refactoring types were applied with similar frequency. Surprisingly, the popular Move Method type was not among the topmost used refactoring except for the design category.

Table 4.5: Refactoring Level Percentages per Category.

Category	**High**	**Medium**	**Low**
BugFix	56.97%	35.31%	7.72%
Feature	59.57%	31.85%	8.58%
Design	69.17%	19.71%	11.12%
Non-Funtional	57.89%	32.01%	10.09%

The first observation that we can draw is that method-level refactorings are employed more than package, class, and attribute-level refactorings, but without any statistical significance (bugFix p-value=0.1356, design p-value=0.2388, feature p-value=0.20045, non-functional p-value=0.2004). Another important observation is the dominance of the high level refactorings compared to medium and low-level refactorings. As shown in table 4.5, the high-level refactorings percentages was the highest among all categories, its highest percentage was in the design category; this can be explained by the fact that developers tend to make a lot of design-improvement decisions that include remodularizing packages by moving classes, reducing class-level coupling, increasing cohesion by moving methods, and renaming elements to increase naming quality in the refactored design. Medium-level refactorings are typically used in bugFix, feature, and non-functional, as developers tend to split classes and extract methods for 1) separation of concerns, 2) helping in

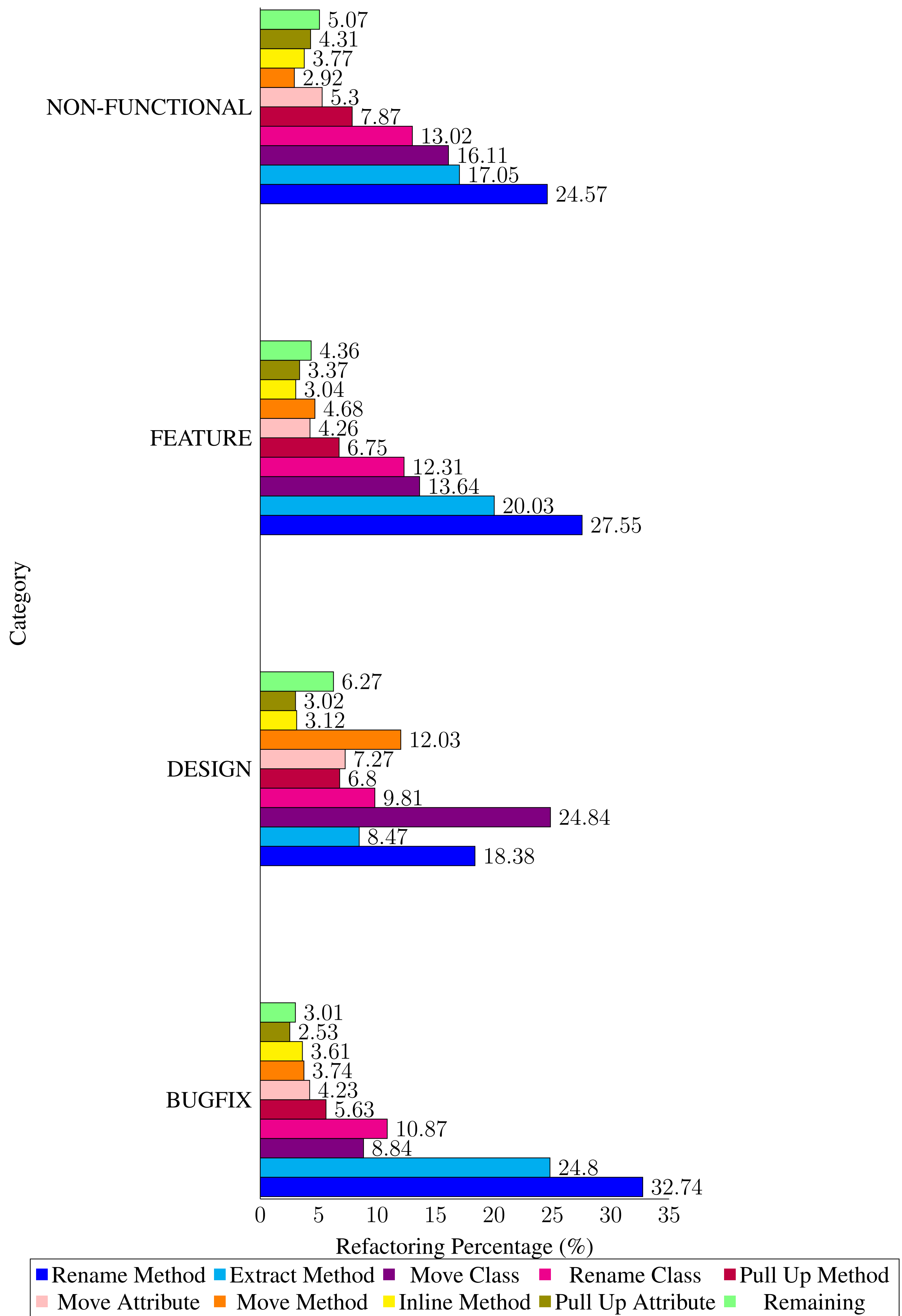


Figure 4.15: Distribution of Refactoring Types per Category

easily adding new features, 3) reducing bug propagation, and 4) improving system's non-functional attributes such as extensibility and maintainability. Low-level refactorings that mainly change class/method code-blocks are the least utilized since moving and extracting attributes are intuitive tasks that developers tend to manually perform.

Manual

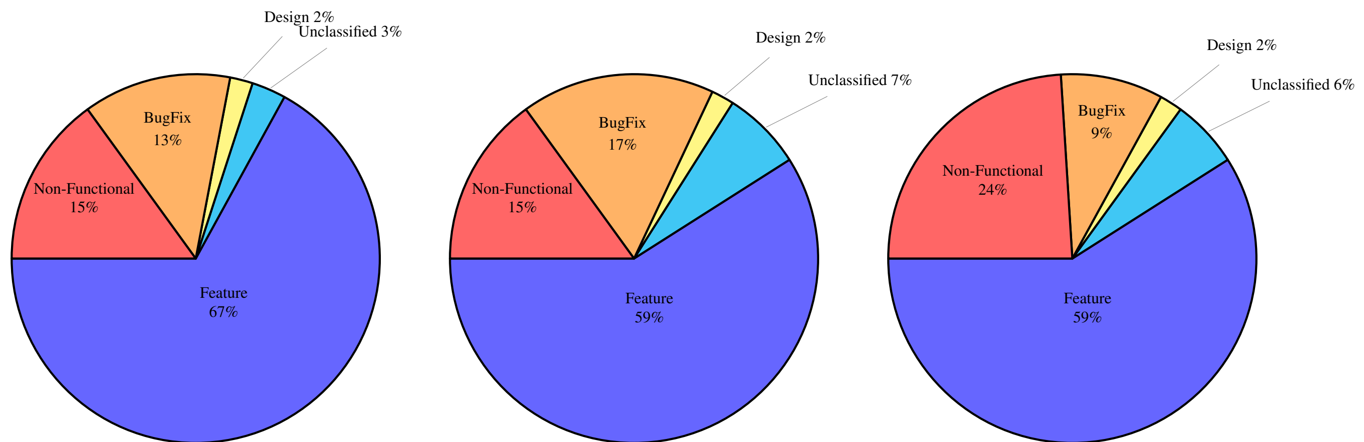


Figure 4.16: Percentage of Classified Commits per Category in Hadoop, OrientDB and Camel.

Overall, as can be seen from Figure 4.16, feature category constitutes the primary driver for applying refactorings in all of the three projects. Feature addition motivation made up 67% in Hadoop and 59% in both OrientDB and Camel. In contrast, we have found that design is the lowest motivation, accounting for only 2%, across all of the three projects. Bugfix and non-functional attribute enhancement categories are the second refactoring motivations, that constitute slightly above a quarter of the motivation behind refactorings. These findings shed light on how refactorings are applied in real settings. Results indicate that developers intersperse refactoring with other programming activity (e.g., feature addition or modification, bug fixing) more frequently than performing pure refactorings.

Our classification has shown that developers not only apply a variety of refactorings to improve the design, like in the previous study, but are driven by preparing the system's design to accommodate features or make it less prone to bugs. Thus, we empirically confirm the results of the previous studies that highlight the popularity of floss refactoring. We also found that high-level refactorings are still widely used in all categories.

Chapter 5

Threats to Validity

External Validity. The first threat is that the analysis was restricted to only open source, Java-based, Git based repositories. However, we were still be able to analyze 1,706 projects that are highly varied in size, contributors, number of commits and refactorings. Also, recent studies [49] [24][15] indicate that commit comments could capture more than one type of classification (i.e. mixed maintenance activity). In this work, we only consider single-labeled classification, but this is an interesting direction that we can take into account in our future work.

Internal Validity. In this thesis, we analyzed 14 refactoring operations detected by Refactoring Miner which can be viewed as a validity threat because the tool did not consider all refactoring types mentioned by Fowler et al. [14]. However, a previous study [33] reported that these 14 types amongst the most common refactoring types. Moreover, we did not perform a manual validation of refactoring types detected by Refactoring Miner to assess its accuracy. However, a previous study [43] report that this tool has a precision of 98% and significantly outperforms the previous state-of-the-art tool, which gives us a confidence in using the tool. Refactoring Miner does not support the detection of the refactoring patterns (e.g., developers applied sequences of refactoring operations to the same part of code) and also the tool may miss the detection of some refactorings in large scale software projects. Further, the set of commit messages used in this study may represent a threat to validity, because it may not indicate refactoring activities. To mitigate this risk, we manually inspect a subset of change messages and ensure that projects selected are well-commented and use meaningful commit messages.

Chapter 6

Conclusion & Future Work

In this thesis, we revisited five research questions that explore different types of refactoring activity and applied them to larger scale software projects. The empirical study we conducted included: the proportion of refactoring operations performed on production and test code, the most active refactoring contributors, the inspection of refactoring activity along the lifetime of 1,706 Java projects, the relationship between refactorings and testing activity, and the main motivations of refactoring. In summary, the main conclusions are:

1. Developers are using wide variety of refactoring operations to refactor production and test files.

2. Specific developers are responsible for performing refactoring, and they are either technical managers or experienced developers.

3. Significant refactoring activity is detected before and after major project releases. There is a strong correlation between refactoring activity and active testing periods.

4. Refactoring activity is mainly driven by changes in requirements and design improvement. The rename method is a key refactoring type that serves multiple purposes.

As future work, we aim to investigate the effect of refactoring on both change and fault-proneness in large-scale open source systems. Specifically, we would like to investigate commit-labeled refactoring to determine if certain refactoring motivations lead to decreased change and fault-prone classes. Further, since a commit message could potentially belong to multiple categories (e.g., improve the design and fix a bug), future research could

usefully explore how to automatically classify commits into this kind of hybrid categories. Another potentially interesting future direction will be to conduct additional studies using other refactoring detection tools to analyze open source and industrial software projects and compare findings.

Bibliography

[1] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[2] Juan Amor, Gregorio Robles, Jesus Gonzalez-Barahona, Alvaro Navarro Gsyc, Juan Carlos, and Spain Madrid. Discriminating development activities in versioning systems: A case study. 01 2006.

[3] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. C. Khoo. Semantic patch inference. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 382–385, Sept 2012.

[4] G. Bavota and B. Russo. A large-scale empirical study on self-admitted technical debt. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 315–326, May 2016.

[5] B. Biegel and S. Diehl. Highly configurable and extensible code clone detection. In *2010 17th Working Conference on Reverse Engineering*, pages 237–241, Oct 2010.

[6] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 53–62, New York, NY, USA, 2011. ACM.

[7] Barry W. Boehm. Software pioneers. chapter Software Engineering Economics, pages 641–686. Springer-Verlag, Berlin, Heidelberg, 2002.

[8] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM.

[10] J. Al Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[11] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. *Automated Detection of Refactorings in Evolving Components*, pages 404–428. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[12] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[13] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res*, 15(1):3133–3181, 2014.

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] Patricia J. Guinan, Jay G. Cooprider, and Samer Faraj. Enabling software development team performance during requirements definition: A behavioral versus technical approach. *Information Systems Research*, 9(2):101–125, 1998.

[16] Ahmed E. Hassan. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 837–841, New York, NY, USA, 2008. ACM.

[17] L. P. Hattori and M. Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71, Sept 2008.

[18] Shinpei Hayashi, Yasuyuki Tsuda, and Motoshi Saeki. Search-based refactoring detection from source code revisions. *IEICE TRANSACTIONS on Information and Systems*, 93(4):754–762, 2010.

[19] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39, May 2009.

[20] Abram Hindle, Neil A. Ernst, Michael W. Godfrey, and John Mylopoulos. Automated topic naming to support cross-project analysis of software maintenance activities. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 163–172, New York, NY, USA, 2011. ACM.

[21] Abram Hindle, Daniel M. German, and Ric 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*, MSR '08, pages 99–108, New York, NY, USA, 2008. ACM.

[22] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.

[23] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of*

the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM.

[24] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 262–265, New York, NY, USA, 2014. ACM.

[25] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Automatic fine-grained issue report reclassification. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 126–135. IEEE, 2014.

[26] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, pages 97–106, New York, NY, USA, 2017. ACM.

[27] N. Mahmoodian, R. Abdullah, and M. A. A. Murad. Text-based classification incoming maintenance requests to maintenance type. In *2010 International Symposium on Information Technology*, volume 2, pages 693–697, June 2010.

[28] A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig. Dataset of developer-labeled commit messages. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 490–493, May 2015.

[29] Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. *Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages*, pages 301–315. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] Collin McMillan, Mario Linares-Vasquez, Denys Poshyvanyk, and Mark Grechanik. Categorizing software applications for maintenance. In *Proceedings of the 2011 27th*

IEEE International Conference on Software Maintenance, ICSM '11, pages 343–352, Washington, DC, USA, 2011. IEEE Computer Society.

[31] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[32] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, Sept 2008.

[33] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[34] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, May 2017.

[35] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, Sept 2014.

[36] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[37] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.

[38] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program

transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.

[39] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, May 2017.

[40] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.

[41] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[42] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[43] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. 2018.

[44] Y. Wang. What motivate software engineers to refactor source code? evidences from professional developers. In *2009 IEEE International Conference on Software Maintenance*, pages 413–416, Sept 2009.

[45] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Sept 2006.

[46] C Wohlin, P Runeson, M Host, MC Ohlsson, B Regnell, and A Wesslen. Experimentation in software engineering: an introduction., 2000.

[47] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[48] Zhenchang Xing and Eleni Stroulia. The jdevan tool suite in support of object-oriented evolutionary development. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 951–952, New York, NY, USA, 2008. ACM.

[49] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D. Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software*, 113(Supplement C):296 – 308, 2016.