Noname manuscript No.
(will be inserted by the editor)

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

Eman Abdullah AlOmar · Mohamed Wiem Mkaouer · Nasir Safdari · Anthony Peruma · Christian Newman · Marouane Kessentini

Received: date / Accepted: date

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 defects that negatively impact the software's design. Since then, the research in refactoring has been driven by the need to improve system structures. However, recent studies have shown that developers may incorporate refactoring strategies in other development-related activities that go beyond improving the design. Unfortunately, these studies are limited to developer interviews and a reduced set of projects. In this context, we aim to better understand the developer's perception of refactoring by mining and automatically classifying refactoring activities in 1,706 open source Java projects. Our study is driven by several research questions that go beyond the classical measurement of refactoring impact on software design. We combine text and refactoring mining techniques to extract developer's intention behind specific types of refactorings, investigate whether various types of refactorings are applied to specific source types (i.e., production vs. test), identify the degree to which experienced developers contribute to refactoring efforts, and finally the chronological collocation of refactoring with release and testing periods. Our approach relies on mining refactoring instances executed throughout several releases of each project we studied. We also mine several properties related to these projects; namely their commits, contributors, issues, test files, etc. Our results confirm some of the findings of previous surveys, 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 significant correlation between the type of refactoring activities taking place and whether the source code is undergoing a release or a test period.

Keywords Refactoring · Software Quality · Empirical study · Software Engineering

Eman Abdullah AlOmar
Rochester Institute of Technology E-mail: eaa6167@rit.edu

Mohamed Wiem Mkaouer
Rochester Institute of Technology E-mail: mwmvse@rit.edu

Nasir Safdari
Rochester Institute of Technology E-mail: nsafdari@rit.edu

Anthony Peruma
Rochester Institute of Technology E-mail: anthony.peruma@mail.rit.edu

Christian Newman
Rochester Institute of Technology E-mail: cdnvse@rit.edu

Marouane Kessentini
University of Michigan E-mail: marouane@umich.edu

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 (Fowler et al, 1999; Dallal and Abdin, 2017). It was popularized by Fowler et al (1999), who identified 72 refactoring types and provided examples of how to apply them in his catalog.

Refactoring is a critical activity in software maintenance and is regularly performed by developers for multiple reasons (Tsantalis et al, 2013; Silva et al, 2016; Palomba et al, 2017). Because refactoring is both a common activity and can be automatically detected (Dig et al, 2006; Prete et al, 2010; Tsantalis et al, 2013), 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 (Boehm, 2002; Erlikh, 2000).

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 (Murphy-Hill et al, 2012). For instance, Murphy-Hill and Black (2008) 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 data about 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). Furthermore, recent case studies (Kim et al, 2010; Murphy-Hill et al, 2012; Tsantalis et al, 2013; Newman et al, 2018; Peruma et al, 2018) that have observed various usage contexts of refactorings, which go beyond improving the design and the removal of code smells, were conducted with a reduced sample of developers and a limited set of refactoring operations, thus generalizing their findings is challenging.

The purpose of this study is to extend further our basic definition of refactoring and to challenge the existing case studies; advocating a need for research on refactorings that takes into account the fact that refactorings are not always solicited in the same context of improving the internal design by fixing smells.Our work intends to *extract* the developer's perception of refactoring by *mining* its refactoring operations, and positioning their context of usage within the development process. We go beyond analyzing whether refactoring have been intended to fix code smells, by adding dimensions related to verifying whether production files are refactored similarly to test files, identifying whether the application of refactoring is randomly distributed among developers, analyzing the frequency of refactorings during the release and testing periods, and detecting the type of activities that refactorings are interleaved with. We propose to achieve it 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. The goal is to identify 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.

Our goal of understanding how refactoring is performed in real-world scenarios is driven by the following research questions:

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

The rationale behind this research question is to investigate whether the refactoring strategies differ by the nature of the target file. We answer it by seeking any patterns that developers specifically apply to either production or test codes.

- **RQ2.** Is there a subset of developers, within the development team, who are responsible for refactoring the code?
  This research question challenges the scalability of several papers that have performed developer interviews and demonstrated that only a subset of developers perform major refactoring activities. In our study, we verify whether the task of refactoring is equally distributed among all developers or if it is the responsibility of a subset of developers. We also consider whether it is proportional to each developer's overall contribution to the project in terms of commits.
- **RQ3.** Are there more refactoring activities before project releases than after?
  Refactoring is known to be solicited around major releases, yet there is not enough information on the nature of activities applied. 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.
- **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 creation or updates of a large set of test code.
- **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 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. For example, one recent area of research that is growing in popularity is automating the construction of transformations from examples (Andersen et al, 2012; Meng et al, 2013; Rolim et al, 2017), which may be used for refactoring. The results of our approach could help support 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 (2013) 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.

2 Related Work

This paper focuses on mining commits to initially detect refactorings and then to classify them. Thus, in this section, we are interested in exploring refactoring detection tools, along with the recent research on commit classification in general. Finally, we report studies that analyze the human aspect in the refactoring-based decision making.

2.1 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

Table 1: Characteristics of refactoring detection studies

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

bodies of software. Dig et al (2006) developed a tool called Refactoring Crawler, which uses syntax and graph analysis to detect refactorings. Prete et al (2010) proposed Ref-Finder, which identifies complex refactorings using a template-based approach. Hayashi et al (2010) 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 (2005) proposed JDevAn, which is a UMLDiff based, design-level analyzer for detecting refactorings in the history Object-Oriented systems. Tsantalis et al (2013) presented Refactoring Miner, which is a lightweight, UMLDiff based algorithm that mines refactorings within git commits. Silva et al (2016) extended their tool to enhance the accuracy of the 14 refactoring types that can be detected through structural constraints. Silva and Valente (2017) extended Refactoring Miner by combining the heuristics-based static analysis with code similarity (TF-IDF weighting scheme) to identify 13 refactoring types. Table 1 summarizes the detection tools cited in this study.

2.2 Refactoring Motivation

Silva et al (2016) 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 (2017) 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 developers apply refactoring to: 1) 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 (2014) do not differentiate the motivations between different refactoring types. They surveyed 328 professional software engineers at 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 (2012) 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 observation 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.

Moser et al (2008) question the effectiveness of refactoring on increasing the productivity in agile environments. They performed an comparative study of developers coding effort before and after refactoring their code. They measured the developer's effort in terms of added lines of code and time. Their findings show that not only does the refactored system improve in terms of coupling and complexity, but also that the coding effort was reduced and the difference is statistically significant.

Szőke et al (2014) conducted 5 large-scale industrial case studies on the application of refactoring while fixing coding issues, they have shown that developers tend to apply refactorings manually at the expense of a large time overhead. Szőke et al (2017) extended their study by investigating whether the refactorings applied when fixing issues did improve the system's non functional requirements with regard to maintainability. They noticed that refactorings performed manually by developers do not significantly improve the system's maintainability like those generated using fully automated tools. They conclude that refactoring cannot be cornered only in the context of design improvement.

Tsantalis et al (2013) 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 (2009) generally focused on the human and social factors affecting the refactoring practice rather than on the technical motivations. He interviewed 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/delete: 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.

2.3 Commits Classification

A wide variety of approaches to categorize commits have been presented in the literature. The approaches vary between performing manual classification (Hindle et al, 2008; Silva et al, 2016; Mauczka et al, 2015; Chávez et al, 2017), to developing an automatic classifier (Hassan, 2008; Mauczka et al, 2012), to using machine learning techniques (Hindle et al, 2011, 2009; Amor et al, 2006; McMillan et al, 2011; Mahmoodian et al, 2010; Levin and Yehudai, 2017) and developing discriminative topic modeling (Yan et al, 2016) to classify software changes. These approaches are summarized in Table 2.

Hattori and Lanza (2008) 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 (2015) have addressed the multi-category changes in a manual way using three classification schemes from existing literature. Silva et al (2016) applied thematic analysis process to reveal the actual motivation behind refactoring instances after collecting all developers' responses. Further, Chávez et al (2017) propose the classification of refactoring instances as root-canal or floss refactoring through the use of manual inspection. An automatic classifier is proposed by Hassan (2008) to classify commits messages as a bug fix, introduction of a feature, or a general maintenance change. Mauczka et al (2012) developed an Eclipse plug-in named Subcat to classify the change messages into Swanson's original category set (i.e., Corrective, Adaptive and Perfective Swanson (1976)), 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 (2009) 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 Swanson (1976) 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 (2011). Their

Table 2: Characteristics of commit classification studies

Study	Year	Single/Multi-labeled	Manual/Automatic	Classification Method	Category	Training Size	Result
Amor et al (2006)	2006	Yes/No	No/Yes	Machine Learning Classifier	Swanson's category Administrative	400 commits (1 participant)	Accuracy = 70%
Hattori and Lanza (2008)	2008	Yes/No	No/Yes	Keywords-based Search	Forward Engineering Reengineering Corrective Engineering Management	1088 commits	F-Measure = 76%
Hassan (2008)	2008	Yes/No	No/Yes	Automated Classifier	Bug Fixing General Maintenance Feature Introduction	18 commits per participant (6 participants)	Agreement = 70%
Hindle et al (2008)	2008	Yes/Yes	Yes/No	Systematic Labeling	Swanson's category Feature Addition Non-Functional	2000 commits	Not mentioned
Hindle et al (2009)	2009	Yes/No	No/Yes	Machine Learning Classifier	Swanson's category Feature Addition Non-Functional	2000 commits	F-Measure: 50%
Hindle et al (2011)	2011	Yes/Yes	No/Yes	Machine Learning Classifier	Non-Functional	Not Mentioned	Receiver Operating Characteristic up to 80%
Mauczka et al (2012)	2012	Yes/No	No/Yes	Subcat tool	Swanson's category Blacklist	21 commits per participants (5 participants)	Precision: 92% Recall: 85%
Tsantalis et al (2013)	2013	Yes/No	Yes/No	Systematic Labeling	Code Smell Resolution Extension Backward Compatibility Abstraction Level Refinement	Not Mentioned	Manual
Mauczka et al (2015)	2015	Yes/Yes	Yes/No	Systematic Labeling	Swanson's category Hattori and Lanza (2008) category Non-Functional	967 commits	Manual
Yan et al (2016)	2016	Yes/Yes	No/Yes	Topic Modeling	Swanson's category	80 commits per participant (5 participants)	F-Measure: 0.76%
Chávez et al (2017)	2017	Yes/No	Yes/No	Systematic Labeling	Floss Refactoring Root-canal Refactoring	sample of 2119	Manual
Levin and Yehudai (2017)	2017	Yes/No	No/Yes	Machine Learning Classifier	Swanson's category	1151 Commits	Accuracy: 73%

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 (2006) had a similar idea to Hindle et al (2009) 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 Mahmoodian et al (2010). McMillan et al (2011) 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. Levin and Yehudai (2017) 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.

3 Empirical Study Setup

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.

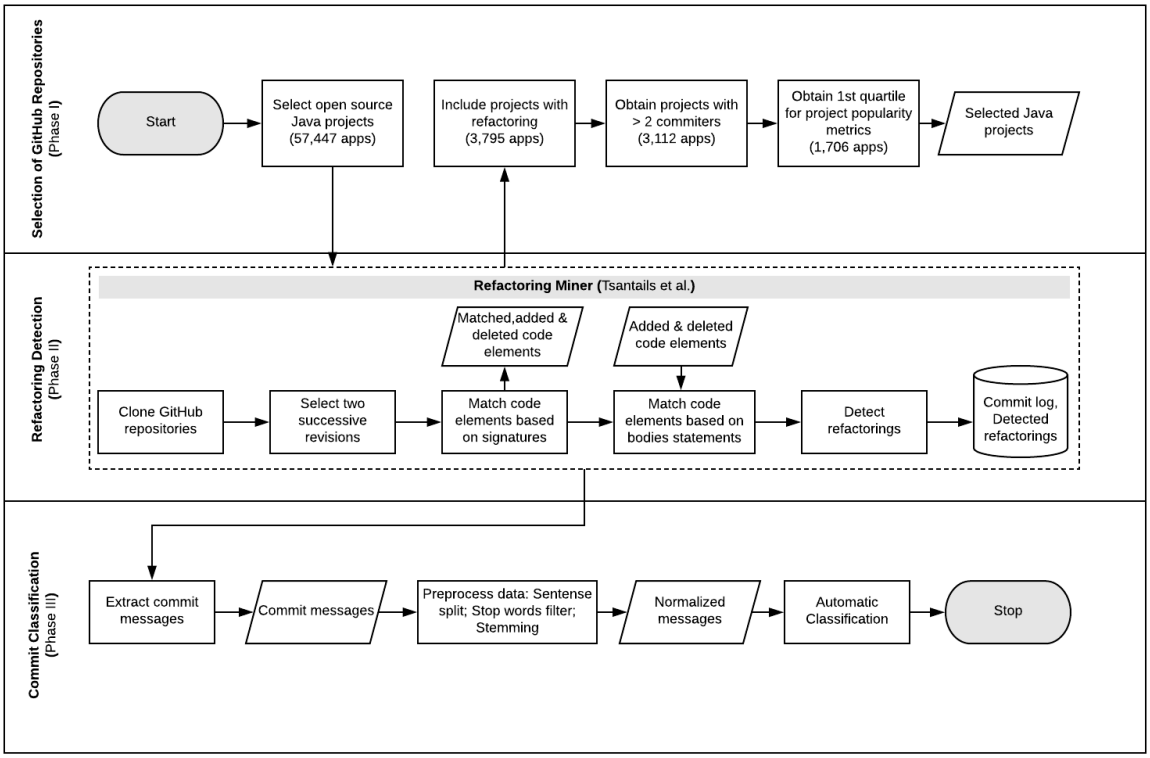


Fig. 1: Empirical study design overview

3.1 Phase 1: Selection of GitHub Repositories

To perform this study, we randomly selected 57,447 projects from a list of GitHub repositories (Allamanis and Sutton, 2013), 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 are of the quality required to answer our research questions, we apply further selection criteria using *Reaper* (Munaiah et al, 2017), an open

Table 3: Refactoring Types (Extracted from Fowler et al (1999))

Refactoring Type	level	Description
Extract Method	Medium	A code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.
Move Class	High	A class isn't doing very much. Move all its features into another class and delete it
Move Attribute	Low	A field is, or will be, used by another class more than the class on which it is defined
Rename Package	High	The name of a package does not reveal its purpose. Change the name of the package
Rename Method	High	The name of a method does not reveal its purpose. Change the name of the method
Rename Class	High	The name of a class does not reveal its purpose. Change the name of the class
Move Method	High	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	Medium	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	High	If at least two subclasses share an identical method, move it to the superclass
Pull Up Attribute	High	Two subclasses have the same field. Move the field to the superclass
Extract Superclass	Medium	There are two classes with similar features. Create a superclass and move the common features to the superclass
Push Down Method	High	The Behavior on a superclass is relevant only for some of its subclasses. Move it to those subclasses
Push Down Attribute	Low	A field is used only by some subclasses. Move the field to those subclasses
Extract Interface	Medium	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

source tool for selecting curated, and well engineered Java projects 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 Ray et al (2014), and Silva et al (2016), we filter out the projects that have less than the 1st quartile for all of these engagement metrics to ensure that they have significant maintenance activity and were 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 4.

Table 4: Projects overview

Item	**Count**
Total projects	57,447
Projects with identified refactorings	26,899
Considered projects	1,706
Refactoring commits	322,479
Refactoring operations	1,208,970
Production-based refactoring operations	950,739
Test-based refactoring operations	258,231
Considered Projects - Refactored Code Elements	
Code Element	**# of Refactorings**
Class	329,378
Method	718,335
Attribute	97,516
Package	18,334
Interface	8,096

3.2 Phase 2: Refactoring Detection

For the purpose of extracting the entire refactoring history of each project, we used the Refactoring Miner tool proposed by Silva et al (2016). Refactoring Miner is designed to analyze code changes (i.e., commits) in git repositories to detect any applied refactoring. The types of detected refactorings can be clustered into three levels: (1) high-level operations, which are those changing the signature of the code element without changing its body. The family of rename refactorings fall into this category; (2) medium-level operations, which are those that make changes to the signature along with the body. Extraction and inlining are good examples of this; finally (3) low-level operations which are those that only affect the body of elements without changing their signatures. Moving attributes is an example of this. The list of detected refactoring types and their corresponding levels is described in Table 3. Two phases are involved when detecting refactorings using the Refactoring Miner tool. In the first phase, the tool matches two code elements (e.g., packages, classes, methods) 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 (2018).

We decided to use Refactoring Miner as it has shown promising results in detecting refactorings compared to the state-of-art available tools (Silva et al, 2016) 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* (Silva et al, 2016) 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 5: Classification categories

Category	**Description**
Feature	Implementation of a introduced/updated/removed feature(s).
BugFix	Tagging, debugging and 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.

3.3 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. An overview of the approach is depicted in Figure 2.

3.3.1 Choice of the classifier

Selecting the proper classifier for optimal classification of the commits is rather a challenging task (Fernández-Delgado et al, 2014). 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 5. Because we have predefined set of categories, our approach relies on supervised machine learning algorithms to assign each commit comment to one category. 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 (Levin and Yehudai, 2017; Kochhar et al, 2014). These studies found that Random Forest gives satisfactory results. Beside a good overall accuracy of Random Forest classifier, it is also known to be robust to outliers as well as noisy data (Thongtanunam et al, 2018). 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 6 shows the performance comparison of different classifiers we evaluated in this study. Below is a brief description of each of the classification algorithms used in this study.

- K-Nearest Neighbor (KNN) is an instance-based algorithm that starts by searching for its first k nearest neighbors and then assigns the most frequent label to each unlabeled instance.
- Naive Bayes Multinomial is an extension of Naive Bayes machine learner. Naive Bayes considers the presence or absence of a feature whereas Naive Bayes Multinomial considers the actual value of the feature.
- Linear Support Vector Classifier (LinearSVC) is a learner that uses One-vs-Rest scheme for multi-class classification.
- Random Forest (RF) is a tree-based learner that builds many classification trees. A specific classification is associated with each tree produces. To classify a new object, RF chooses the classification that has the most votes over all other trees.

Table 6: Performance of different classifiers

Classifier	**Precision**	**Recall**	**F1**
Random Forest	0.85	0.84	0.84
Linear SVC	0.83	0.80	0.81
Naive Bayes Multinomial	0.80	0.78	0.78
K-Nearest Neighbor	0.73	0.70	0.71

3.3.2 Data Preprocessing

We applied a similar methodology explained in Kochhar et al (2014) 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, numbers and stop words (i.e., words include "is", "are", "if", etc) 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 Natural Language Toolkit (NLTK) package Bird and Loper (2004). 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 Term Frequency-Inverse Document Frequency (TF-IDF) vectorizer in Python's SKlearn (Pedregosa et al, 2011) package to extract the features. We use TF-IDF to give greater weight (e.g., value) to words which occur frequently in fewer documents rather than words which occur frequently in many documents.

3.3.3 Building the Training Set

Our goal is to provide the classifier with sufficient commits that represent the categories analyzed in this study. To do so, we used an existing dataset by Mauczka et al (2015), manually labeled by developers and containing many classes, which our categories are included among them. Since these commits do not necessarily contain refactoring operations, and it is important to capture the taxonomy used by developers when describing any activity containing refactoring, we extended the training set with commits that we manually labeled. Since the number of candidate commits to classify is large, we cannot manually process them all and so we need to randomly sample a subset while making sure it equitably represent

the featured classes i.e., categories. The classification process has been performed by the 6 authors of the paper, the authors have been regular users of commits in both academic and industrial setting, the authors experience with committing changes ranges from 3 to 9 years. To approximate the needed number of commits to add, we reviewed the thresholds used in the studies summarized in Table 2. The highest number of commits used in three studies (Hindle et al, 2008, 2009; Chávez et al, 2017) was around 2000 commits. As for the classification process, we were inspired by the manual classification of Hindle et al (2009). We started by indexing all the commits on an SQL database and we developed a web application that randomly selects and sequentially displays commits and the possible classification options. As commits are classified, the application keeps track of the number of commits per each category. To reduce the effect of personal bias, each commit is being classified by two subjects. In case of an agreement over the classification, the commits is automatically saved, otherwise it is automatically discarded and another commit is randomly selected. As we analyze the discarded commits, we notice, aside from cases where the disagreement is due different interpretations, that discarded commits were either too short, or ambiguous. Some examples of the hard-to-classify commit messages are: "*Solr Indexer ready*"[1], "*allow multiple collections*"[2], "*Auto configuration of AgiScripts*"[3].

The classification process has ended when a needed threshold per category is reached. Because of the random nature of the process, some classes were saturated faster than others and when all classes had the target number, the evaluation covered around 3500 commits.

Also, as another step in mitigating the subjectivity involved in the manual classification, we followed the method explained in Levin and Yehudai (2017). We randomly selected commit messages from three different categories and had those commits evaluated again by one of the authors who did not evaluate those commits from those three categories during the first pass. The process has taken around 56 hours of collective work. The result of this classification is available in the reproduceability package of this work, thus, it can be reused and extended. Finally, we combined our commits with those form Mauczka et al (2015), then we divided the resulting set randomly into 75% of training set and 25% of testing set to evaluate the performance of different classifiers introduced at the beginning of this section.

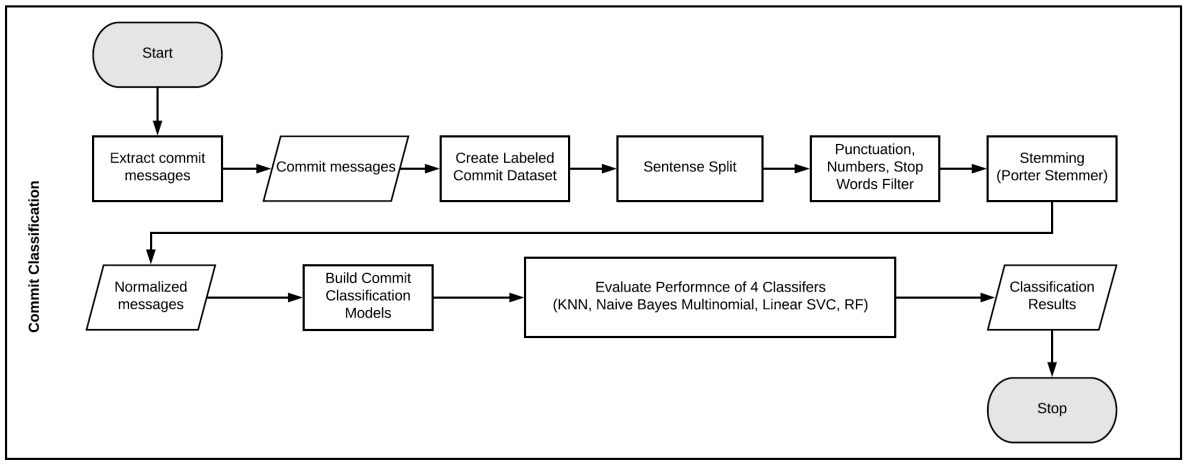


Fig. 2: Commit classification approach stages

4 Experimental Results

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[4] for replication and extension

[1] https://github.com/01org/graphbuilder
[2] https://github.com/0install/java-model
[3] https://github.com/1and1/attach-qar-maven-plugin
[4] https://refactorings.github.io/empirical-refactoring/

purposes. The summary of all figures and tables, addressing the research questions of the empirical study, is illustrated in Table 7.

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 Levin and Yehudai (2017); system characteristics are illustrated in Table 8. 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 7: Summary of the empirical study design

Research question	**Reference**	
	Automated	**Manual**
RQ1: Refactoring operations perform on production and test code	Table 9	Table 10
RQ2: Refactoring contributors	Fig 3,	Fig 4
RQ3: Correlation between refactoring and project releases	Fig 5, 6, 7, 8, Table 11	Fig 9, 10, 11
RQ4: Correlation between refactoring and test hoptspots	Fig 12, 13, 14	Fig 15, 16, 17
RQ5: Motivation behind refactoring	Fig 18, 19, Table 13	Fig 20, 21, 22, 23

Table 8: Characteristics of three software projects under study

Software Project	**All Commits**	**Ref. Commits**	**Refactorings**	**Contributors**	**Ref. Contributors**
apache/hadoop[5]	17,510	1,462	4,207	114	73
orientechnologies/orientdb[6]	16,246	1,737	5,063	113	35
apache/camel[7]	30,781	2,863	8,422	368	73

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 operation performed on test files, are considered test refactorings. Then, we cluster the refactorings we detected per type and we report, for each target production and test file, the percentage of each type.

We compare the distribution of refactoring refactorings identified for each category using the Wilcoxon (1945) signed-rank test; a pairwise statistical test verifying whether two sets have a similar distribution. If the p-value of smaller than 0.05, the distribution difference between the two sets is considered statistically significant. The choice of Wilcoxon comes from its non-parametric nature with no assumption of a normal data distribution. The magnitude of the observed difference is controlled using Cliff's delta (d), another non-parametric measure for effect size, in terms of the probability of a distribution values being larger than those of another distribution. The effect size, ranging between [-1; 1], is interpreted as follows: **small:** 0:148 $<$ d $<$ 0:33; **medium:** 0:33 $<$ d $<$ 0:474; **large:** d $<$ 0:474.

4.1.1 Automated

An overview of the detected refactorings is provided in Table 9. 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 only study, which also clustered refactorings based on the target type of files, found that, on average,

Table 9: 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%**

73% of refactorings were applied to production files while 27% were applied to test files Tsantalis et al (2013). Furthermore, we found that the topmost applied refactorings (shown in **bold** in Table 9) are respectively *Rename Method, Extract Method, and Move Class*. Our topmost used refactoring types overlaps with the findings of Silva et al (2016), since they ranked *Extract Method, and Move Class* as the 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.

The observed difference in the distribution of refactorings in production/test files between our study and the related work 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 8.

4.1.2 Manual

Table 10: 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

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).

Our findings are aligned with the previous studies. The distribution of refactoring operations differ between production and test files. Operations undertaking production is significantly larger than operations applied to test files. *Rename Method* and *Move Class* are the most solicited operations for both production and test files. Yet, we could not confirm that developers uniformly apply the same set of refactoring types when refactoring either production or test files.

4.2 RQ2: Which developers are responsible for refactorings?

In this research question, we investigate whether specific developers are significantly contributing in the overall refactoring of the system or it is randomly distributed among all developers. Similarly to the first research question, we tag all changed files in the set of analyzed commits containing refactorings. Then, we identify all committers using their GithHub ID and we track their contributions in terms of commits on the tagged files. We only consider tagged files to ignore all commits related to non-refactored files, also, all non-source related changes were automatically discarded. Afterward, for each committer, we calculate his/her number of performed code changes in terms of commits along with the number of corresponding refactorings, per project. In the *automated* subsection, we perform a comparative analysis between top X% contributors and the remaining contributors. In this study, we selected X=1% (labeled TOP-1) to consider develper(s) with highest refactoring activity with comparison to the remaining development team. For the *manual* analysis, we identify all developers in the 3 projects and we plot the percentage of their refactoring contributions in both production and test files.

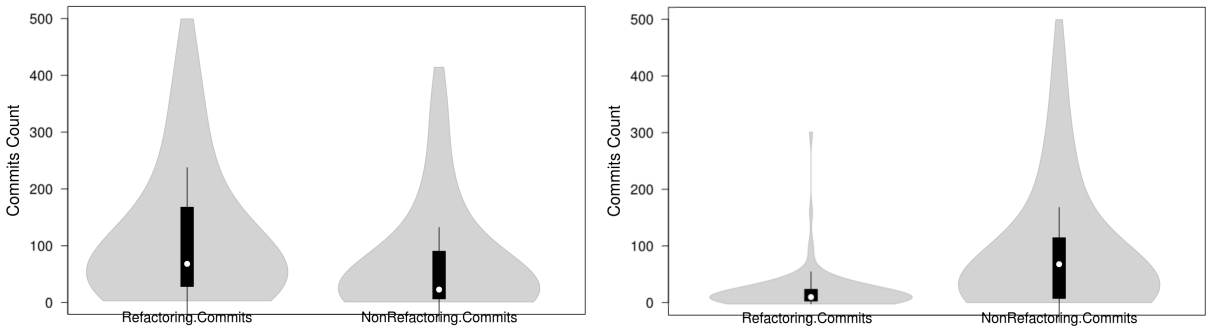
4.2.1 Automated

Figure 3a and 3b are violin plots of the cumulative commit distribution of both the top refactoring contributor(s) (i.e., Figure 3a) and the rest of the refactoring contributors (i.e., Figure 3b) on refactoring and non-refactoring commits for all 1,706 systems. We observe in Figure 3a that the number of commits containing refactoring is significantly higher than the number of commits without detected refactorings as the median of the first set is 68 with (9595% confidence level over 200 commits), while the second's set median is 19 (95% confidence level over 100 commits). This implies that TOP-1 frequently refactor the code when introducing changes. Their contribution in terms of refactoring is significantly higher than the remaining committers as the median of the latter is 15 (95% confidence level under 100 commits). This observation can be influenced by the hypothesis that the developers with the highest number of refactorings are also the developers with the highest number of commits. However, Figure 3b, denies that hypothesis since the non-refactoring commits median of TOP-1 is significantly smaller than the median of the remaining set of developers.

To better understand the key role of the TOP-1 contributors in the development team, we extract refactorings from Hadoop, OrientDB, and Camel, then we cluster them, in both production and test files, by developer ID. Finally, we closely examine the TOP contributors professional profiles to identify their role in the organization hosting the software project. Our findings are detailed in the next subsection.

4.2.2 Manual

Figure 4 portrays the distribution of the refactoring activities on production code and test code performed by project contributors for each software system we examined. The Hadoop project has a total of 114 developers. Among them are 73 (64%) refactoring contributors. As we observe in Figures 4a and 4b, not all of the developers are major refactoring contributors. The main refactoring contributor has a refactoring



(a) Top 1 Refactoring Contributors

(b) All the Rest of Refactoring Contributors

Fig. 3: Top 1 and remaining refactoring contributors for all projects combined (Outliers ommitted)

ratio of 25% on production code and 10% on test code. Figure 4c and 4d 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 production and test code respectively. For Camel, in Figures 4e and 4f, 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. We also note that very few developers applied refactorings exclusively on either production code or test code for the three projects under study.

The manual analysis aligns with the findings of the previous section in distinguishing a subset of developers that monopolize the refactoring activity across the three projects. To identify their key role in the development of the project, we searched, using their GiHub IDs, their professional profiles in LinkedIn[8]. We were successful in locating all the contributors for the 3 projects, and we found, through their public affiliation to the project, that they were either development leads or senior developers.

While refactoring is being applied by various developers, only a reduced set is responsible for performing the majority of these activities, in both production and test files. This set takes over refactoring activities without necessarily being dominant in the other programming activities. As we examine the top contributors public professional profiles, we identify their positions to be advanced in the development team, advocating for their extensive knowledge about the software design. Our results are in agreement with Tsantalis et al (2013), which also found that the top refactoring contributors had a management role within the project.

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

The empirical verification of the impact of releases on refactoring activities was previously studied by Tsantalis et al. Tsantalis et al (2013). They monitored the refactoring activity in a time span centered by a release for three projects, namely JUnit, HTTPCore and HTTPClient. 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. In our study, for the automated analysis, we have considered all major 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. For the manual analysis, we kept the same settings used by Tsantalis et al. Tsantalis et al (2013) for Hadoop, OrientDB and Camel.

4.3.1 Automated

We report in Table11 the results of analyzing the refactoring density in three situations: (1) when pre-release refactorings are significantly greater than post-release refactorings (labeled *Before » After*), (2) when post-release refactorings are significantly greater than pre-release refactorings (labeled *Before «*

[8] Used in previous studies as a source to identify developers skills and experience.

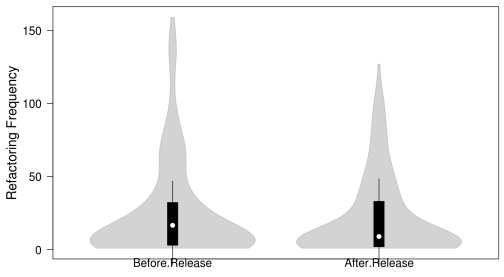
Fig. 4: Refactoring contributors in production and test files in Hadoop, OrientDB and Camel

After), and (3) when there is no significant different between pre-release and post-release refactorings (labeled *Before* ≈ *After*). We found that the percentage of pre-release refactorings being significantly greater than post-release is 66% and 62% respectively for production and test files. While post-release refactoring dominate the pre-release ones in only 31% and 34% respectively for production and test files. The percentages of 3% and 4% represent the remaining percentage of releases where there is no significant difference between the two refactoring sets.

Table 11: Percentage of Releases Clustered by the Significance of Refactoring Density in a pre- and post-Release, on Production and Test Files in all Projects Combined

Comparison Type	**Production Code**	**Test Code**
Before » After	66%	62%
Before « After	31%	34%
Before ≈ After	3%	4%

Figure 5a and 5b present the density of refactoring activity before and after all project releases respectively in production and test files. We break down these refactoring per category, where Figure 6b and Figure 7b are associated with refactoring density for *Before » After* while Figure 6a and Figure 7a represent *Before « After*.

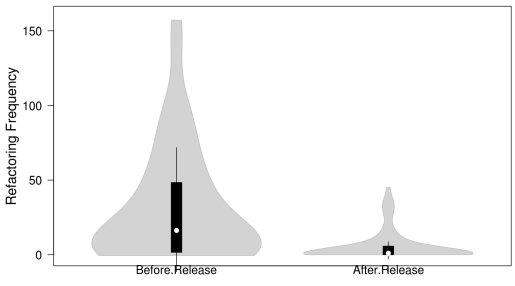


(a) Production Files - Release Point Comparison

(b) Test Files - Release Point Comparison.

Fig. 5: Refactoring Density in both production and test files - Release point comparison (Outliers ommitted)

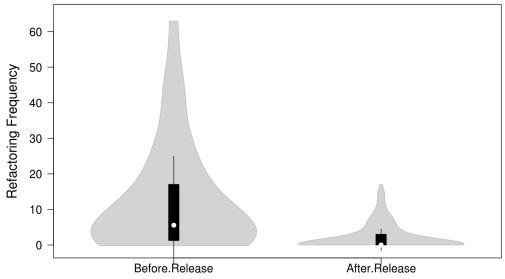
(a) Before « After - Release Point Comparison.



(b) Before » After - Release Point Comparison.

Fig. 6: Refactoring Density in production files - Release point comparison (outliers ommitted)

(a) Before « After - Release Point Comparison.



(b) Before » After - Release Point Comparison.

Fig. 7: Refactoring density in test files - Release point comparison (Outliers ommitted)

4.3.2 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 (Tsantalis et al, 2013). 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 9 shows, there are significant peaks in refactoring activity on the same day of the release and two days after the release day. 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 10 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 11 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.

Aligned with Tsantalis et al (2013), we found that refactoring activity is higher before the major release dates than after. As for the type of applied refactorings, we could not extract any significant, generalizable patterns in refactoring operations before and after projects major release dates.

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

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.

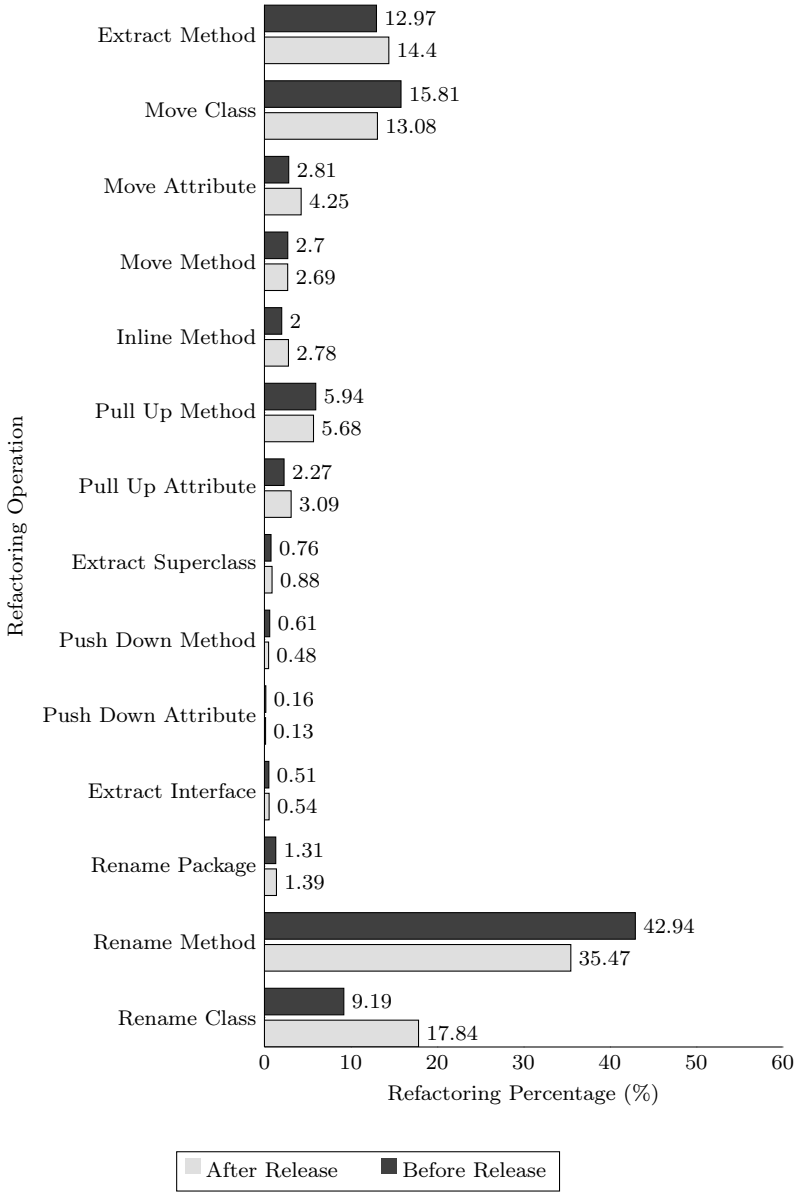


Fig. 8: Frequency of pre-reslease and post-release refactorings per type

4.4.1 Automated

Similarly to the previous research question, Table 12 contains the results of analyzing the refactoring density in three situations: (1) when pre-hotspot refactorings are significantly greater than post-hotspot refactorings (labeled *Before » After*), (2) when post-hotspot refactorings are significantly greater than pre-hotspot refactorings (labeled *Before « After*), and (3) when there is no significant different between pre-hotspot and post-hotspot refactorings (labeled *Before* $\approx$ *After*).

According to Table 12, the distribution of refactorings surrounding hotspots tend to be random as the percentages of hotspots experiencing no significant difference in the refactoring activities are 64% and 56% respectively for production and test files. In contrast with the release point comparison, the percentage of hotspots experiencing the domination of pre-hotspot refactorings are 29% and 40%, which is higher than the case where post-hotspots refactorings are dominant.

The equitable distribution of refactorings before and after hotspots can be also seen in Figure 12a where the medians of refactoring frequency in production files are similar even if the frequency of post-

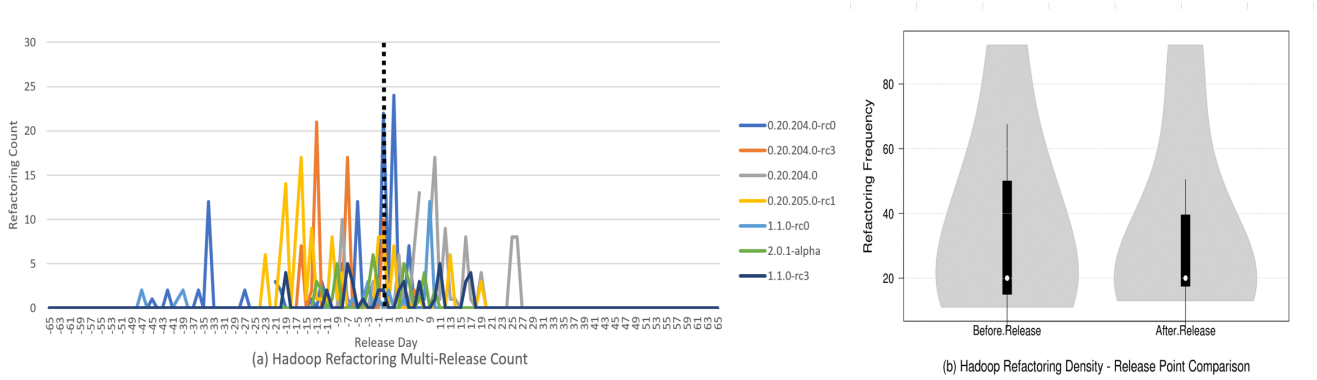


Fig. 9: Refactoring Activity Comparison (Release)- Hadoop

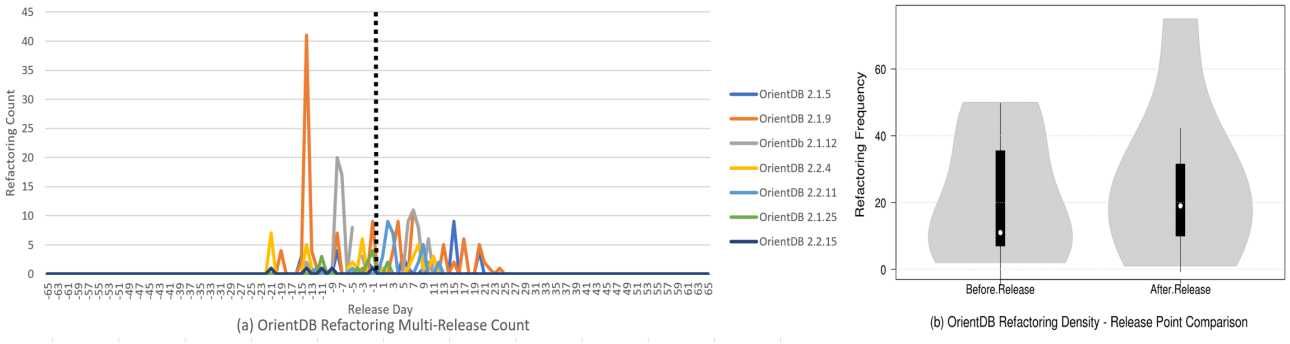


Fig. 10: Refactoring Activity Comparison (Release)- OrientDB

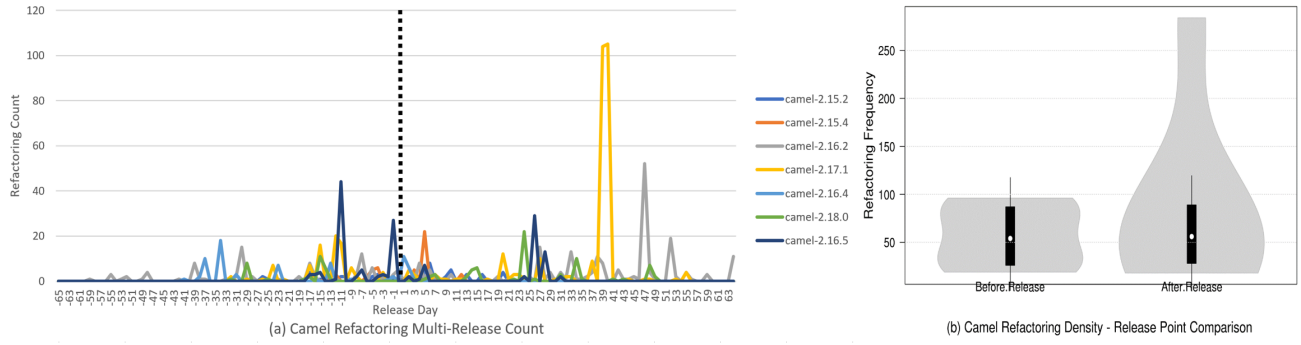


Fig. 11: Refactoring Activity Comparison (Release)- Camel

hotspot refactorings is relatively higher. As for Figure 12b, the median of post-hotspot refactorings on test files is relatively higher than the median of pre-hotspot refactorings.

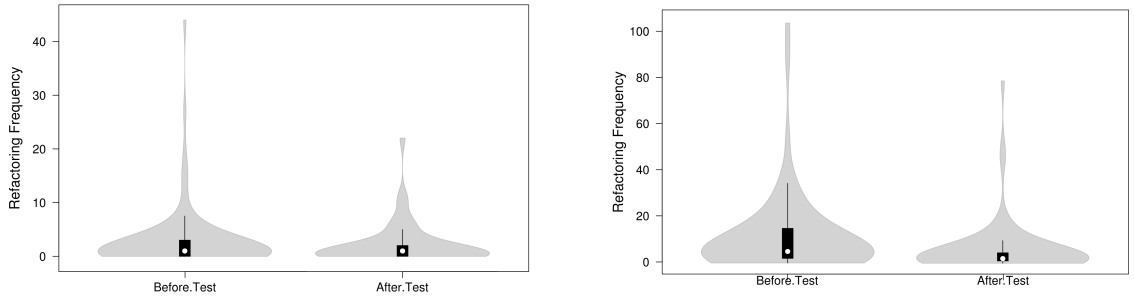
A similar interpretation can be drawn from Figure 13b and Figure 14b where medians, associated with refactoring density for *Before » After*, are not significantly different, while in Figure 13a and Figure 14a, representing *Before « After*, show that refactorings a-posteiori in test files are relatively higher.

4.4.2 Manual

We begin with the Hadoop project. Figure 15 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

Table 12: Percentage of testing periods clustered by the significance of refactoring density in a pre- and post-hotspot, on production and test files in all projects combined

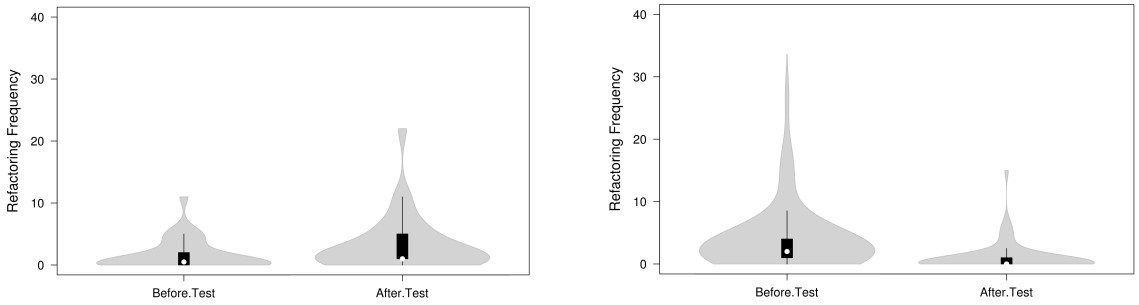
Comparison Type	Production Code	Test Code
Before » After	29%	40%
Before « After	7%	4%
Before ≈ After	64%	56%



(a) Production Files - Test Point Comparison

(b) Test Files - Test Point Comparison

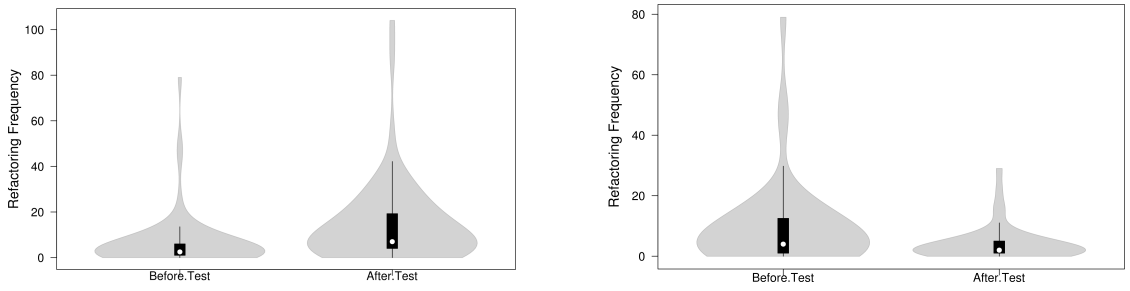
Fig. 12: Refactoring density - Test point comparison (Outliers ommitted)



(a) Before « After - Test point comparison

(b) Before » After - Test point comparison

Fig. 13: Refactoring density in production files - Test point comparison (Outliers ommitted)



(a) Before « After - Test Point Comparison

(b) Before » After - Test Point Comparison

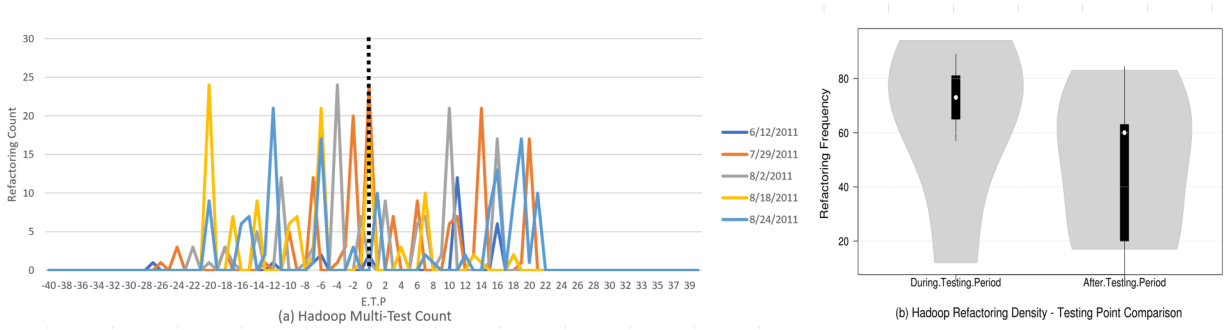
Fig. 14: Refactoring density in test files - Release point comparison (Outliers ommitted)

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 16 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.

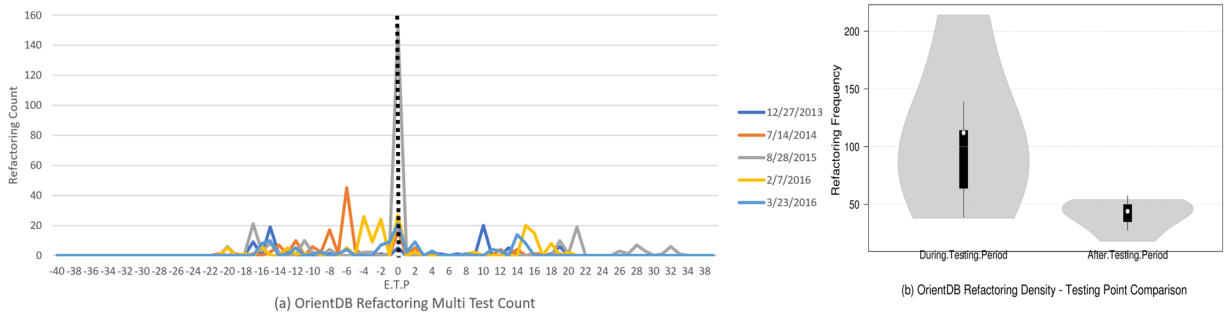
Finally, we examine the Camel project. There is common pattern between the Hadoop and the Camel projects in terms of active refactoring activity around testing periods. Comparatively, refactoring activity in Camel peaks earlier and higher than Hadoop (about 32 and 24 refactoring events, respectively).



(a) Hadoop Multi-Test Count

(b) Hadoop Refactoring Density - Testing Point Comparison

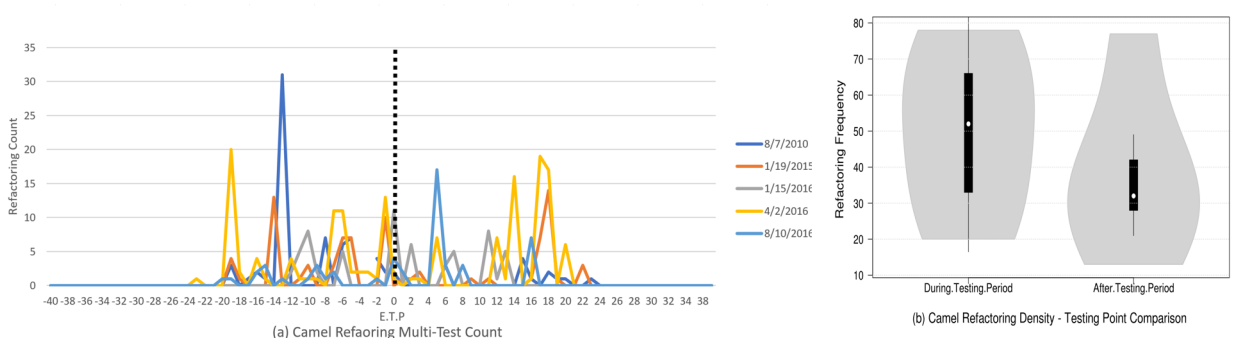
Fig. 15: Refactoring activity comparison (Test)- Hadoop



(a) OrientDB Refactoring Multi Test Count

(b) OrientDB Refactoring Density - Testing Point Comparison

Fig. 16: Refactoring activity comparison (Test)- OrientDB



(a) Camel Refaoring Multi-Test Count

(b) Camel Refactoring Density - Testing Point Comparison

Fig. 17: Refactoring activity comparison (Test)- Camel

The majority of testing hotspots experience no substantial difference between refactorings applied a-priori or a-posteriori fashion. The qualitative analysis of three projects also confirm the random distribution of refactorings across testing hotspots. Still, there is up to 40% of hotspots experiencing significantly higher pre-hotspot refactorings. This outcome slightly aligns with Tsantalis et al. Tsantalis et al (2013) as they found, in their qualitative analysis, that there are high refactoring activity before testing period rather than after.

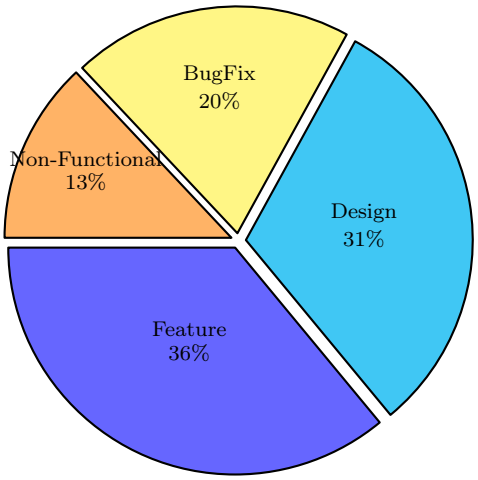


Fig. 18: Percentage of classified commits per category in all projects combined

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

To answer this research question, we present the results of classifying commit messages using the random forest classifier, as explained in Subsection 3.3. The *automated* analysis details the classification of 322,479 commit messages containing 1,208,97 refactoring operations. The manual subsection represents commit classification of Hadoop, OrientDB and Camel.

4.5.1 Automated

Figure 18 shows the categorization of commits, from all projects combined. The *Feature* and *Design* categories had the highest number of commits with a slight advantage to the first category, since its ratio was 36%, while the *Design* category had a ratio of 31%. Interestingly, *bugFix* was the third most popular category for refactoring-related commits with 20%, in front of the *Non-functional* category, which had a ratio of 13%.

Although there was limited evidence about which refactoring tactic is more common (Murphy-Hill and Black, 2008), 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. Murphy-Hill et al (2012), 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 (2016) 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 19 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.

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 13, 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

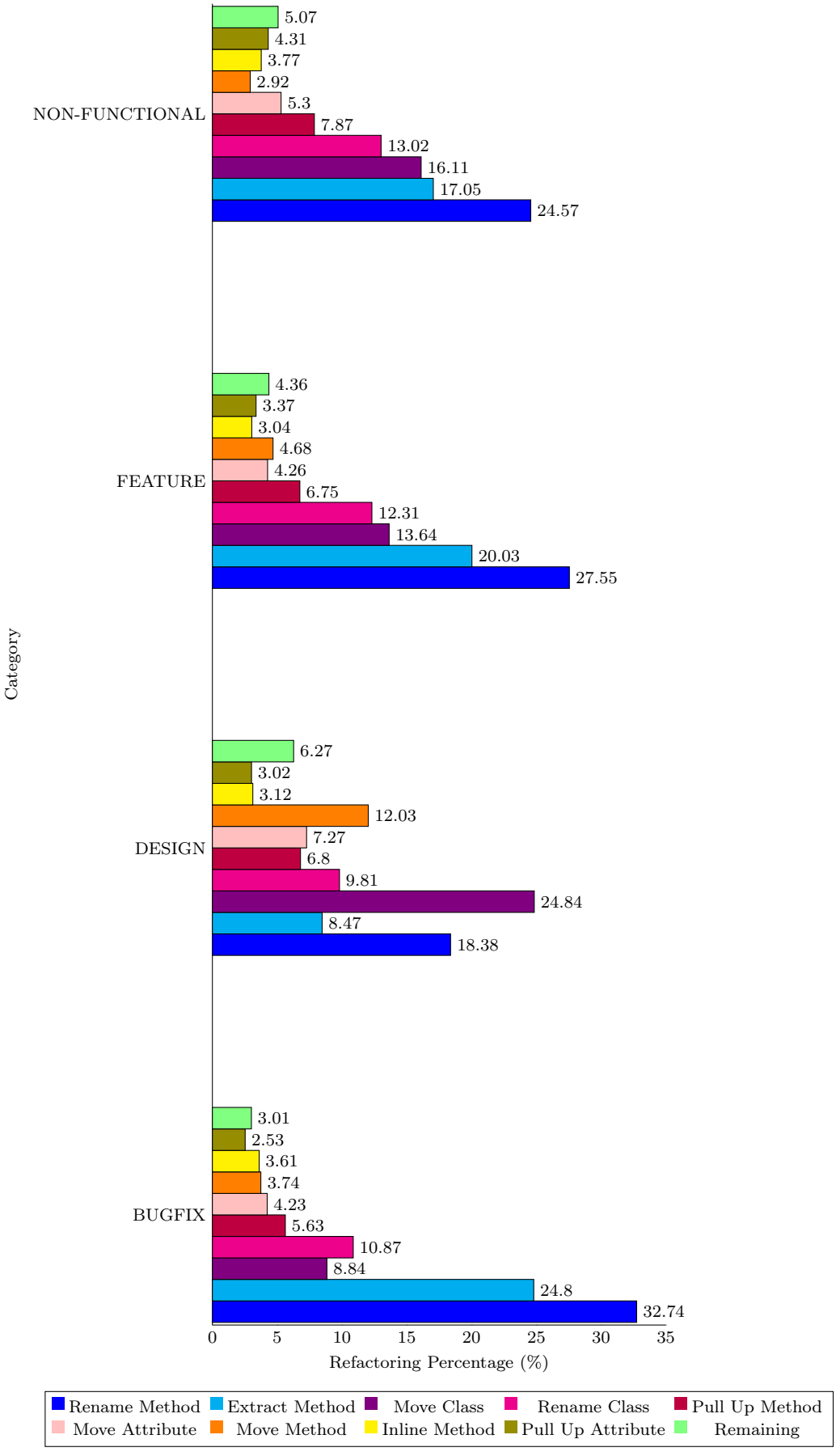


Fig. 19: Distribution of refactoring types per category

Table 13: 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%

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 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.

To better understand the nature of classified commits, we randomly sampled examples from each class to illustrate the type of information contained in these messages, and how it infers the use of refactorings in specific contexts. Table 14 shows the four main motivations driving refactoring, which we can also divide into fine-grained subcategories. For instance, we subcategorize *Feature* classified commits into *addition*, *update* and *deletion*. Similarly, *BugFix* is decomposed into *Localization*, *debugging* and *correction*. As for *Design*, there are various possible subcategories to consider, so we have restricted our subcategories to the ones extracted from the paper we surveyed in the related work. So, the corresponding subcategories for *Design* are *code smell resolution*, *duplicates removal* and *Object Oriented design improvement*. The subcategories of *Non-functional* are more straightforward to extract from the commit messages since developers tend to explicitly mention which quality attribute they are trying to optimize. For this study, the sub-categories we found in our mined commits include *testability*, *usability*, *performance*, *reusability* and *readability*. Then, for each subcategory, we provide an illustrative commit message as an example. We also indicate refactoring operations that are associated with each of the example commit messages. One important observation that can be drawn from these messages, is that developers may have multiple reasons to refactor the code, which go beyond Martin Fowler's traditional definition of associating refactoring with improving design by removing code smells.

For feature-focused refactoring, developers are refactoring the code to either introduce new functionalities, or modify or delete existing ones. For instance, the commit description in one of the analyzed commits was: *refactoring of album view page (adding some string templates) (modifying homepage) (modifying the view of a photo) (fixing search result display problems)*. It is clear that the intention of the refactoring was to enhance existing functionalities related to user interface, which include modifying the view of some pages, improving the design of the homepage of a website, and fixing the display resulted from the search engine feature. As can be seen, developers incorporate refactoring activities in development-related task (i.e., feature requests).

In the second category, developers perform refactoring to facilitate bug fix-related activities: resolution, debugging, and localization. As described in the following commit comment: *Refactor the code to fix bug 256238, Shouldn't update figure before inserting extended item such as Chart in library editor.* , the developer who performed refactoring explained that the purpose for the refactoring was to resolve certain bug that required adding a specific item before updating the chart in library editor. Thus, it is clear that developers frequently floss refactor since they intersperse refactoring with other programming activity (i.e., bug fixing).

To improve code design, it is apparent from Table 14 that developers either introduce good practices (e.g., use inheritance, polymorphism, and enhance the main modularization quality drivers) or remove certain code smells such as feature envy, duplicated code, and long methods. This traditional design improvement refactoring motivation is best illustrated in the following change messages: (1) *MongoStore: Refactor to avoid long methods*, (2) *Addresses issue #11 where a general refactor is needed to improve the cohesion of the creation of property value mergers.*, and (3) *Removes tight coupling between managed service and heartbeat notification.* It is clear that developers performed *Extract Method* refactoring to remove code smell which corresponds to a Long Method bad smell. Further, developers purely refactor

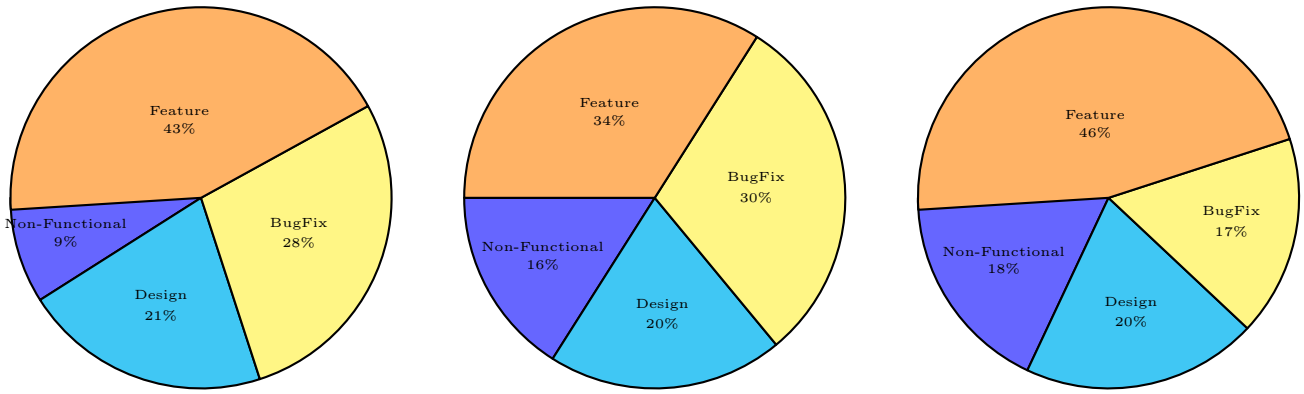


Fig. 20: Percentage of classified commits per category in respectively Hadoop, OrientDB and Camel

the code to improve the dominant modularization driving forces (i.e., cohesion and coupling) to maximize intra classes connectivity and minimize inter classes connectivity.

For the last category, refactorings are performed to enhance nonfunctional attributes. For example, developers refactor the code to improve its testability, usability, performance, reusability, and readability. Developers are particularly extracting a method for improving testability as they test parts of the code separately. They are also extracting a method to improve code readability. This is illustrated by the following commit messages: (1) *Minor refactoring for better testability without the need to generate intermediary files.* and (2) *Refactoring mostly for readability (and small performance improvement).*. Closer inspection of the these commit comments show that developers intended to apply nonfunctional-related development topics while performing refactorings.

From the refactoring operation usage perspective, we notice that some commit messages describe the method of refactoring identified by Refactoring Miner. For instance, in order to remove long method code smell, developers extracted a method to reduce the length of the original method body. Also, to remove feature envy code smell, move method refactoring operation was performed in order to place the fields and methods to their preferred class. In both cases, developers explained these changes in the commit messages. A similar pattern can be seen in the other examples in the table. It is worth noting that a singular refactoring is almost never performed on its own, as can be seen for some of the refactorings that are associated with commit messages. For instance, to eliminate a long method, Fowler suggests using several refactorings (e.g., Replace Temp with Query, Introduce Parameter Object, and Preserve Whole Object) depending on the complexity of the transformation. Due to the limited refactoring operations supported by Refactoring Miner, we could not demonstrate such composite refactorings. However, this is an interesting research direction that can be investigated in the future.

4.5.2 Manual

Overall, as can be seen from Figure 20, *Feature* category constitutes the primary driver for applying refactorings in all of the three projects. the *Feature* class made up 43% in Hadoop, 34% in OrientDB and 46% in Camel. In contrast, we have found that nonfunctional attribute enhancement is the lowest motivation across all of the three projects. *Bugfix* and *Design* categories are the second refactoring motivations, that constitute slightly above a quarter of the motivation behind refactorings for Hadoop and OrientDB projects. These findings shed light on how refactorings are applied in real settings. The results indicate that developers intersperse refactoring with other programming activity (e.g., feature addition or modification, bug fixing) more frequently than performing refactorings on it pure definition of improving the software design.

The manual validation agrees mostly with the automated classification as the *Feature* was primarily dominant in all projects, while *Design* had the second highest number of commits in Camel but third behind *BugFix* in both Hadoop and OrientDB.

To understand the nature of refactorings applied in each category, we have classified the operations into *renaming*, *extracting* and *moving*. The *renaming* class contains all refactorings that rename a given

Table 14: Commits message examples classified by category

Category	Subcategory	Refactoring Operation	Commit Messages
Feature	Feature Addition	Rename & Extract Method	Implementation of the feature to allow user to add custome exception. Code refactoring for various modules. Incorporated Fabios old review comment.
	Feature Modification	Rename Method	Feature #5157 - refactoring of album view page (adding some string templates) (modifying homepage) (modifying the view of a photo) (fixing search result display problems)
	Feature Deletion	Pull Up Attribute	Refactored the sauce test class to a base and subclass with test content. Reused sauce base test class to run session id and job-name printing test and checking output by capturing stdout. The nature of the feature being tested forced some unorthodox testing. Also removed build id update from the SauceOnDemandTestListener class to give user the freedom to set their own. In the earlier setup the Jenkins tag was overwriting the user set one causing odd behaviour.
BugFix	Bug Resolution	Extract Method	Refactoring of SlidingWindow class in RR to reduce complexity and fix important bug
	Bug Resolution	Rename Method	Fixed a bug in Extract Function refactoring.
	Bug Resolution	Extract Method	Refactor the code to fix bug 256238,Shouldn't update figure before inserting extended item such as Chart in library editor.
	Debugging	Move & Rename Class	Renaming and refactoring towards debugging functionality.
	Bug Localization	Move Method	Moved generateObjects() from AbstractFunction to FunctionHelper. More natural. Updated Wiki docs to reflect this. Found a bug with using variables in predicates. commented out the test case that breaks. Added some test cases that are more complicated uses of ScriptEngine.
Design	Code Smell Resolution	Extract Method	MongoStore: Refactor to avoid long methods
	Duplicates removal	Extract Method	Minor refactoring to remove duplicate code
	Code Smell Resolution	Move Class	Refactoring of core data classes. Implementation of RefactoringRegistry.
	Code Smell Resolution	Move Method	Introduced adjuncts from stapler - fixed the feature envy problem in the version computation.
	Inheritance	Move Attribute, Rename & Inline Method	Refactoring of filters to make better use of inheritance hierarchy.
	Polymorphism	Move Attribute, Rename & Inline Method	Refactored to remove Classifier factory pattern and allow for model polymorphism.
	Coupling Improvement	Extract Interface	Removes tight coupling between managed service and heartbeat notification.
	Cohesion Improvement	Extract Method	Addresses issue #11 where a general refactor is needed to improve the cohesion of the creation of property value mergers.
Non-Functional	Testability	Extract Method	Minor refactoring for better testability without the need to generate intermediary files.
	Usability	Rename Method	Refactor: Remove redundant method names, improve usability.
	Performance	Extract Method	Improve performance of MessageSource condition.
	Reusability	Move Method	Refactored for better reusability.
	Readability	Extract Method	Refactoring mostly for readability (and small performance improvement).

code element such as a class, a package or an attribute. The extraction of classes and methods are included in the *extracting* class. As for, *Moving*, it gathers all the movement of code elements, e.g., moving methods, or pushing code elements across hierarchies e.g., pushing up attributes. The following Figures 21, 22, and 23 contain the percentage of refactorings per class for each category in Hadoop, OrientDB and Camel.

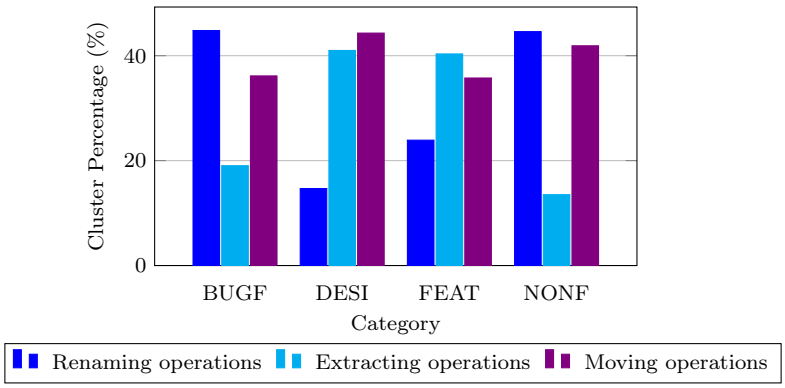


Fig. 21: Percentage of refactorings, clustered by operation class, per category in Hadoop

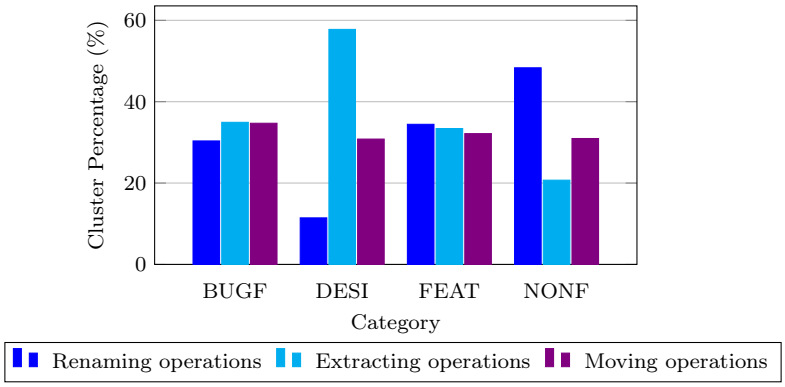


Fig. 22: Percentage of refactorings, clustered by operation class, per category in OrientDB

For the *BugFix* category, the *renaming* operations are constantly relied on across the three projects, and it reached the percentage of 45% in Hadoop, higher than *extracting* and *moving* whose percentage is respectively 19% and 36%. For OrientDB and Camel, *extracting* was also mostly used and its percentage is respectively 35% and 44%. An interpretation for this comes from the nature of the debugging process that includes the disambiguation of identifiers naming that may not relfect the appropriate code semantics or that may infected with lexicon bad smells or linguistic anti-patterns (Abebe et al, 2011; Arnaoudova et al, 2013). Another debugging practice would be the separation of concerns which helps in Reducing the core complexity of a larger module and reduce its proneness to errors (Tsantalis and Chatzigeorgiou, 2011).

Extracting is also very popular refactoring for the *Design* category, achieving a percentage of 41%, 58% and 52% respectively in Hadoop, OrientDB and Camel. On the other hand, *renaming* was least used in the *Design* category, this can be explained by the fact that moving code elements is a popular practice for design-level changes (Stroggylos and Spinellis, 2007; Alshayeb, 2009; Bavota et al, 2015; Mkaouer et al, 2015) e.g., developers tend to remodularize classes to make packages more cohesive, extract methods to reduce coupling between classes.

Similarly, *renaming* has scored its highest peaks with a percentage of of 44%, 48% and 42% respectively in Hadoop, OrientDB and Camel, for the *Non-Functional* category. Finding better namings for

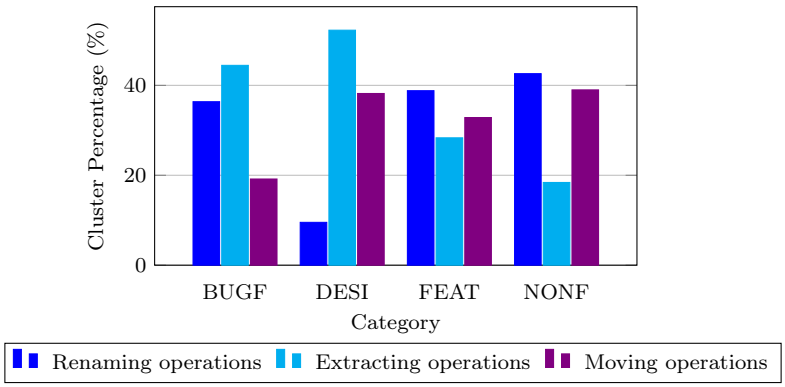


Fig. 23: Percentage of refactorings, clustered by operation class, per category in Camel

code identifiers serves the purpose of increasing the software's comprehension. As in Table 14, developers explicitly mention the use of the renaming operations for the purpose of disambiguating the redundancy of methods names and enhancing their usability.

Finally, the *Feature* category had a uniform distribution of refactoring classes where no specific class has been drastically deployed at the expense of another. For instance in OrientDB, *reaming*, extracting and *moving* had respectively percentages of 34%, 33% and 32%. Table 14 does not only illustrate how diverse can be the operations used while updating the system's functionality, but also it demonstrates how condensed the developers actions can be i.e., developers may perform many actions that may belong to more than one category. For instance, the third commit shows how refactoring can be interleaved with updating a feature and fixing a misbehavior at the same time. Since these commit messages can be classified to more than one class, we plan, in our future work, to extend our classification to support multi-labeling in order to precisely capture the refactoring activities associated with these scenarios.

Our classification has shown that improving the design through fixing code smells is not the main driver for developers to refactoring their code bases. As explicitly mentioned by the developers in their commits messages, refactoring is solicited to reduce the software's prone to bugs, resolve lexical ambiguity and improve the design's testability and reusability. Thus, we empirically confirm the results of the previous studies that highlight the popularity of floss refactoring (Murphy-Hill and Black, 2008; Murphy-Hill et al, 2012). We also found that high-level refactorings are still widely used in all categories.

5 Threats to Validity

We identify, in this section, potential threats to the validity of our approach and our experiments.

Internal Validity. In this paper, 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 (1999). However, in a previous study, Murphy-Hill et al (2012) 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. So our study is mainly threatened by the accuracy of the detection tool. Yet, Tsantalis et al (2018) reports that Refactoring Miner has a precision of 98% and significantly outperforms the previous state-of-the-art tools, which gives us a confidence in using the tool.

Moreover, 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.

Construct validity. It gathers all the threats linked to the relationship between theory and observation. The classification of refactorings heavily rely on classified commits, which represents the main threat to the reliability of our empirical study. Since the manual classification is a human intensive task,

it is subject to personal bias. Thus, we constructed the training set using an existing dataset classified by developers, and we expanded it by including our manual classified commits. To mitigate the impact of commit styles and auto-generated messages, we diversified the set of projects to extract commits from, we also randomly sampled from the two commits clusters, those containing detected refactorings and those without. Also, recent studies (Yan et al, 2016; Kirinuki et al, 2014; Guinan et al, 1998) 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.

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. Another threat concerns the difference in the number of versions that each software undergoes. This is an unavoidable fact but we have considered projects with a relatively long lifespan and significant number of releases when answering the research questions.

6 Differential Replication

Biegel et al (2011) replicated Weissgerber and Diehl (2006) work, which considers a signature-based refactoring detection technique using CCFinder to determine the similarity between the body of old and new code elements. The authors extended afterwards the replicated experiment by plugging in the two similarity metrics, namely JCCD (Biegel and Diehl, 2010) and Shingles (Broder, 1997), to investigate their influence on the signature-based detection technique. Bavota and Russo (2016) presented a large-scale empirical study as a differentiated replication of the study by Potdar and Shihab (2014) 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 the work of Potdar and Shihab (2014) 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 Potdar and Shihab (2014), 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. Table 15 summarizes the main differences between the original studies and the replication.

Our work represents a replication and an extension of the work by Tsantalis et al (2013). 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 (Levin and Yehudai, 2017). Revisiting the qualitative analysis allows a direct comparison with the results of the previous studies (Murphy-Hill et al, 2012; Tsantalis et al, 2013; Silva et al, 2016). 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 Tsantalis et al (2013).
- *Refactoring detection tool:* Since the approach relies on mining refactorings, we use the Refactoring Miner tool, which detects 14 refactoring types. Tsantalis et al (2013) 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 (2013), 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 (2013) 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 for the quantitative analysis, 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. (Tsantalis et al, 2013) performs a systematic labeling of the refactoring instances by manually inspecting the text diff reports along with their commit logs.

Table 15: Replicated studies in software refactoring

Study	Attribute	Original Study	Replication	Conclusion
WeiBgerber & Diehl Biegel et al.	Similarity Metrics	CCFinder	CCFinder, Shingles, JCCD	Replication experiment helps in showing comparable quality and providing new insights on how metrics differ.
Potdar & Shihab Bavota & Russo	Size of studies	4	159	Replication experiment helps in improving the generalizability of the achieved results and performing a finer granularity level of the performed analysis
	Granularity of the analysis	Compare detected SATD at a release-level	Compare detected SATD in each commit.	

7 Conclusion & Future Work

In this paper, 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 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. Since we observed that feature requests and fix bugs are strong refactoring motivators for developers, researchers are encouraged to adopt a maintenance-related refactoring beside design-related refactoring when building a refactoring tool in the future.

References

Abebe SL, Haiduc S, Tonella P, Marcus A (2011) The effect of lexicon bad smells on concept location in source code. In: Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on, Ieee, pp 125–134

Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, pp 207–216

Alshayeb M (2009) Empirical investigation of refactoring effect on software quality. Information and software technology 51(9):1319–1326

Amor J, Robles G, Gonzalez-Barahona J, Navarro Gsyc A, Carlos J, Madrid S (2006) Discriminating development activities in versioning systems: A case study

Andersen J, Nguyen AC, Lo D, Lawall JL, Khoo SC (2012) Semantic patch inference. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp 382–385, DOI 10.1145/2351676.2351753

Arnaoudova V, Di Penta M, Antoniol G, Gueheneuc YG (2013) A new family of software anti-patterns: Linguistic anti-patterns. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, pp 187–196

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp 315–326, DOI 10.1109/MSR.2016.040

Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107:1–14

Biegel B, Diehl S (2010) Highly configurable and extensible code clone detection. In: 2010 17th Working Conference on Reverse Engineering, pp 237–241, DOI 10.1109/WCRE.2010.34

Biegel B, Soetens QD, Hornig W, Diehl S, Demeyer S (2011) Comparison of similarity metrics for refactoring detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '11, pp 53–62, DOI 10.1145/1985441.1985452, URL `http://doi.acm.org/10.1145/1985441.1985452`

Bird S, Loper E (2004) Nltk: the natural language toolkit. In: Proceedings of the ACL 2004 on Interactive poster and demonstration sessions, Association for Computational Linguistics, p 31

Boehm BW (2002) Software pioneers. Springer-Verlag, Berlin, Heidelberg, chap Software Engineering Economics, pp 641–686, URL `http://dl.acm.org/citation.cfm?id=944331.944370`

Broder A (1997) On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences 1997, IEEE Computer Society, Washington, DC, USA, SEQUENCES '97, pp 21–, URL `http://dl.acm.org/citation.cfm?id=829502.830043`

Chávez A, Ferreira I, Fernandes E, Cedrim D, Garcia A (2017) How does refactoring affect internal quality attributes?: A multi-project study. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, ACM, New York, NY, USA, SBES'17, pp 74–83, DOI 10.1145/3131151.3131171, URL `http://doi.acm.org/10.1145/3131151.3131171`

Dallal JA, Abdin A (2017) 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, DOI 10.1109/TSE.2017.2658573

Dig D, Comertoglu C, Marinov D, Johnson R (2006) Automated Detection of Refactorings in Evolving Components, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 404–428. DOI 10.1007/11785477_24, URL `https://doi.org/10.1007/11785477_24`

Erlikh L (2000) Leveraging legacy system dollars for e-business. IT Professional 2(3):17–23, DOI 10.1109/6294.846201

Fernández-Delgado M, Cernadas E, Barro S, Amorim D (2014) Do we need hundreds of classifiers to solve real world classification problems. J Mach Learn Res 15(1):3133–3181

Fowler M, Beck K, Brant J, Opdyke W, Roberts d (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, URL `http://dl.acm.org/citation.cfm?id=311424`

Guinan PJ, Cooprider JG, Faraj S (1998) Enabling software development team performance during requirements definition: A behavioral versus technical approach. Information Systems Research 9(2):101–125, DOI 10.1287/isre.9.2.101, URL `https://doi.org/10.1287/isre.9.2.101`, `https://doi.org/`

10.1287/isre.9.2.101

Hassan AE (2008) Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '08, pp 837–841, DOI 10.1145/1363686.1363876, URL http://doi.acm.org/10.1145/1363686.1363876

Hattori LP, Lanza M (2008) On the nature of commits. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, pp 63–71, DOI 10.1109/ASEW.2008.4686322

Hayashi S, Tsuda Y, Saeki M (2010) Search-based refactoring detection from source code revisions. IEICE TRANSACTIONS on Information and Systems 93(4):754–762

Hindle A, German DM, Holt R (2008) What do large commits tell us?: A taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '08, pp 99–108, DOI 10.1145/1370750.1370773, URL http://doi.acm.org/10.1145/1370750.1370773

Hindle A, German DM, Godfrey MW, Holt RC (2009) Automatic classication of large changes into maintenance categories. In: 2009 IEEE 17th International Conference on Program Comprehension, pp 30–39, DOI 10.1109/ICPC.2009.5090025

Hindle A, Ernst NA, Godfrey MW, Mylopoulos J (2011) Automated topic naming to support cross-project analysis of software maintenance activities. In: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '11, pp 163–172, DOI 10.1145/1985441.1985466, URL http://doi.acm.org/10.1145/1985441.1985466

Kim M, Gee M, Loh A, Rachatasumrit N (2010) 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, ACM, New York, NY, USA, FSE '10, pp 371–372, DOI 10.1145/1882291.1882353, URL http://doi.acm.org/10.1145/1882291.1882353

Kim M, Zimmermann T, Nagappan N (2014) An empirical study of refactoringchallenges and benefits at microsoft. IEEE Transactions on Software Engineering 40(7):633–649, DOI 10.1109/TSE.2014.2318734

Kirinuki H, Higo Y, Hotta K, Kusumoto S (2014) Hey! are you committing tangled changes? In: Proceedings of the 22Nd International Conference on Program Comprehension, ACM, New York, NY, USA, ICPC 2014, pp 262–265, DOI 10.1145/2597008.2597798, URL http://doi.acm.org/10.1145/2597008.2597798

Kochhar PS, Thung F, Lo D (2014) Automatic fine-grained issue report reclassification. In: Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on, IEEE, pp 126–135

Levin S, Yehudai A (2017) 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, ACM, New York, NY, USA, PROMISE, pp 97–106, DOI 10.1145/3127005.3127016, URL http://doi.acm.org/10.1145/3127005.3127016

Mahmoodian N, Abdullah R, Murad MAA (2010) Text-based classification incoming maintenance requests to maintenance type. In: 2010 International Symposium on Information Technology, vol 2, pp 693–697, DOI 10.1109/ITSIM.2010.5561540

Mauczka A, Huber M, Schanes C, Schramm W, Bernhart M, Grechenig T (2012) Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 301–315. DOI 10.1007/978-3-642-28872-2_21, URL https://doi.org/10.1007/978-3-642-28872-2_21

Mauczka A, Brosch F, Schanes C, Grechenig T (2015) Dataset of developer-labeled commit messages. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp 490–493, DOI 10.1109/MSR.2015.71

McMillan C, Linares-Vasquez M, Poshyvanyk D, Grechanik M (2011) Categorizing software applications for maintenance. In: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '11, pp 343–352, DOI 10.1109/ICSM.2011.6080801, URL http://dx.doi.org/10.1109/ICSM.2011.6080801

Meng N, Kim M, McKinley KS (2013) Lase: Locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 502–511, URL http://dl.acm.org/citation.cfm?id=2486788.2486855

Mkaouer W, Kessentini M, Shaout A, Koligheu P, Bechikh S, Deb K, Ouni A (2015) Many-objective software remodularization using nsga-iii. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(3):17

Moser R, Abrahamsson P, Pedrycz W, Sillitti A, Succi G (2008) A case study on the impact of refactoring on quality and productivity in an agile team. In: Balancing Agility and Formalism in Software Engineering, Springer, pp 252–266

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empirical Software Engineering 22(6):3219–3253

Murphy-Hill E, Black AP (2008) Refactoring tools: Fitness for purpose. IEEE Software 25(5):38–44, DOI 10.1109/MS.2008.123

Murphy-Hill E, Parnin C, Black AP (2012) How we refactor, and how we know it. IEEE Transactions on Software Engineering 38(1):5–18, DOI 10.1109/TSE.2011.41

Newman CD, Mkaouer MW, Collard ML, Maletic JI (2018) A study on developer perception of transformation languages for refactoring. In: Proceedings of the 2nd International Workshop on Refactoring, ACM, pp 34–41

Palomba F, Zaidman A, Oliveto R, Lucia AD (2017) An exploratory study on the relationship between changes and refactoring. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp 176–185, DOI 10.1109/ICPC.2017.38

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al (2011) Scikit-learn: Machine learning in python. Journal of machine learning research 12(Oct):2825–2830

Peruma A, Mkaouer MW, Decker MJ, Newman CD (2018) An empirical investigation of how and why developers rename identifiers. In: Proceedings of the 2nd International Workshop on Refactoring, ACM, pp 26–33

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp 91–100, DOI 10.1109/ICSME.2014.31

Prete K, Rachatasumrit N, Sudan N, Kim M (2010) Template-based reconstruction of complex refactorings. In: 2010 IEEE International Conference on Software Maintenance, pp 1–10, DOI 10.1109/ICSM.2010.5609577

Ray B, Posnett D, Filkov V, Devanbu P (2014) 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, ACM, New York, NY, USA, FSE 2014, pp 155–165, DOI 10.1145/2635868.2635922, URL `http://doi.acm.org/10.1145/2635868.2635922`

Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B (2017) Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '17, pp 404–415, DOI 10.1109/ICSE.2017.44, URL `https://doi.org/10.1109/ICSE.2017.44`

Silva D, Valente MT (2017) Refdiff: Detecting refactorings in version histories. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 269–279, DOI 10.1109/MSR.2017.14

Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2016, pp 858–870, DOI 10.1145/2950290.2950305, URL `http://doi.acm.org/10.1145/2950290.2950305`

Stroggylos K, Spinellis D (2007) Refactoring–does it improve software quality? In: Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on, IEEE, pp 10–10

Swanson EB (1976) The dimensions of maintenance. In: Proceedings of the 2Nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE '76, pp 492–497, URL `http://dl.acm.org/citation.cfm?id=800253.807723`

Szőke G, Nagy C, Ferenc R, Gyimóthy T (2014) A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In: International Conference on Computational Science and Its Applications, Springer, pp 524–540

Szőke G, Antal G, Nagy C, Ferenc R, Gyimóthy T (2017) Empirical study on refactoring large-scale industrial systems and its effects on maintainability. Journal of Systems and Software 129:107–126

Thongtanunam P, Shang W, Hassan AE (2018) Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones. Empirical Software Engineering pp 1–36

Tsantalis N, Chatzigeorgiou A (2011) Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software 84(10):1757–1782

Tsantalis N, Guana V, Stroulia E, Hindle A (2013) A multidimensional empirical study on refactoring activity. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp., Riverton, NJ, USA, CASCON '13, pp 132–146, URL http://dl.acm.org/citation.cfm?id=2555523.2555539

Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history

Wang Y (2009) What motivate software engineers to refactor source code? evidences from professional developers. In: 2009 IEEE International Conference on Software Maintenance, pp 413–416, DOI 10.1109/ICSM.2009.5306290

Weissgerber P, Diehl S (2006) Identifying refactorings from source-code changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp 231–240, DOI 10.1109/ASE.2006.41

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics bulletin 1(6):80–83

Xing Z, Stroulia E (2005) Umldiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE '05, pp 54–65, DOI 10.1145/1101908.1101919, URL http://doi.acm.org/10.1145/1101908.1101919

Xing Z, Stroulia E (2008) The jdevan tool suite in support of object-oriented evolutionary development. In: Companion of the 30th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE Companion '08, pp 951–952, DOI 10.1145/1370175.1370203, URL http://doi.acm.org/10.1145/1370175.1370203

Yan M, Fu Y, Zhang X, Yang D, Xu L, Kymer JD (2016) Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. Journal of Systems and Software 113(Supplement C):296 – 308, DOI https://doi.org/10.1016/j.jss.2015.12.019, URL http://www.sciencedirect.com/science/article/pii/S016412121500285X