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

EMAN ABDULLAH ALOMAR, Rochester Institute of Technology, USA
ANTHONY PERUMA, Rochester Institute of Technology, USA
MOHAMED WIEM MKAOUER, Rochester Institute of Technology, USA
CHRISTIAN NEWMAN, Rochester Institute of Technology, USA
MAROUANE KESSENTINI, University of Michigan, USA
NASIR SAFDARI, Rochester Institute of Technology, USA

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 software design. Since then, a significant amount of research in refactoring has presumed that refactoring is primarily motivated 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.

To cope with the above-mentioned limitations, we aim to better understand what motivates developers to apply a refactoring by mining and automatically classifying 111,884 commits containing refactoring activities, extracted from 800 open source Java projects. We trained a multi-class classifier to categorize these commits into 3 novel categories, namely, Internal Quality Attributes, External Quality Attributes, and Code Smells, along with the traditional Bug Fix and Functional categories. Such classification enables to quantification of how much each of these categories trigger refactoring activities in general, and challenges the original definition of refactoring, being exclusive to improving software design and fixing code smells. Furthermore, to better understand our classification results, we qualitatively analyzed commit messages to extract textual patterns that developers regularly use to describe their refactoring activities.

The results of our empirical investigation show that (1) 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 for a wide variety of reasons, going beyond its traditional definition, such as reducing the software's proneness to bugs, easing the addition of functionality, resolving lexical ambiguity, enforcing code styling, and improving the design's testability and reusability; (2) the distribution of refactoring operations differ between production and test files. Operations undertaking production is significantly larger than operations applied to test files; (3) developers use a variety of patterns to purposefully target refactoring-related activities; (4) developers occasionally explicitly mention the motivation behind their refactoring strategies; (5) the textual patterns, which we extracted in this paper, provide a better coverage for how developers document their refactorings.

Authors' addresses: Eman Abdullah AlOmar, eman.alomar@mail.rit.edu, Rochester Institute of Technology, Rochester, Newyork, USA; Anthony Peruma, Rochester Institute of Technology, Rochester, Newyork, USA, anthony.peruma@mail.rit.edu; Mohamed Wiem Mkaouer, Rochester Institute of Technology, Rochester, USA, mwmvse@rit.edu; Christian Newman, Rochester Institute of Technology, Rochester, Newyork, USA, cnewman@se.rit.edu; Marouane Kessentini, University of Michigan, Dearborn, Michigan, USA, marouane@umich.edu; Nasir Safdari, Rochester Institute of Technology, Rochester, USA, nsafdari@rit.edu.

Manuscript submitted to ACM

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: Refactoring, Software Quality, Empirical study, Software Engineering

ACM Reference Format:
Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Marouane Kessentini, and Nasir Safdari. 2018. How We Refactor and How We Affirm it? A Large-Scale Empirical Study on Refactoring Activities in Open Source Systems. 1, 1 (April 2018), 49 pages. https://doi.org/10.1145/1122445.1122456

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 [Barry et al. 1981; Erlikh 2000a]. One key practice to cope with this challenge is refactoring. Refactoring is the art of remodeling the software design without altering its functionalities [Dallal and Abdin 2017; Fowler et al. 1999]. 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 an amalgamation of reasons [Palomba et al. 2017b; Peruma 2019; Silva et al. 2016; Tsantalis et al. 2013]. 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 2000b].

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. 2010a; Murphy-Hill et al. 2012; Newman et al. 2018; Peruma et al. 2018; Tsantalis et al. 2013] have investigated development contexts in which refactorings are applied; finding, in part, that there are many facets to refactoring beyond simply improving the design and removal of code smells. Because these are conducted using a smaller sample of developers, due to being case studies, generalizing their findings is challenging.

Therefore, recent studies have taken a developer-centric strategy by detecting how developers refactor their source code [Silva et al. 2016; Silva and Valente 2017; Tsantalis et al. 2018] and how they document their refactoring strategies [AlOmar et al. 2019a; Zhang et al. 2018]. The detection of refactoring operations has revealed another dimension of how we should perceive refactoring: Instead of *dictating* how the code should be refactored in the presence of code smells or design pattern opportunities, we can reverse engineer the impetus which caused the developer to refactor their code and analyze how it relates to the type of refactoring applied. While automating the detection of refactoring

operations has been recently reaching a significant precision [Tsantalis et al. 2018], there is a critical need for a deeper analysis of how such refactoring activities are being documented. Therefore, recent studies [AlOmar et al. 2019a; Zhang et al. 2018] have introduced a taxonomy on how developers actually document their refactoring strategies in commit messages. Such documentation is known as *Self-Admitted* or *Self-Affirmed* refactoring. However, the detection of such refactoring documentation was primarily manual and limited. Furthermore, developer's motivations to refactor and the context in which refactoring is performed is under-researched.

Despite the fact that research has shown that refactorings occur in many different development contexts, there are currently no approaches which automatically categorize refactorings by development activity (e.g., bug fix, feature addition). Our work provides an automated approach to categorize refactorings and discusses the significant impact which different development activities have on how refactorings are used and applied by developers. Furthermore, existing studies on understanding developers motivation behind applied refactorings mainly rely on developers surveys and formal interviews [Ge et al. 2012; Kim et al. 2014]. To cope with the above-mentioned limitations, the purpose of this study is to augment our understanding of the development contexts which trigger refactorings and to enable future research to more effectively take development context into account when studying refactorings. As the existing refactoring detectors offer an abundant source of commits containing refactoring operations, this paper aims at exploring how developers document their refactoring activities during the software life-cycle: we text-mine the developer's messages in refactoring-related commits to detect any potentially relevant information regarding the applied refactorings. Commit messages represent an atomic documentation of a code change, written by the change author, and thus are a reliable and rich source of information to describe why a change has been made. Thus, we contribute a technique which is critically needed by the research community to properly support both the development community and our own understanding of refactorings as a vital technique for software evolution. We do this by addressing the following research questions:

- **RQ1.** What is the purpose of the applied refactorings?
  Rationale. While previous surveys analyze how developers apply refactorings in varying development contexts, none of them have measured the ubiquity of these varying contexts in practice. Therefore, we *quantify* the distribution of refactorings performed in varying development contexts to augment our understanding of refactorings in theory vs. practice.
  Methodology. We determine the motivation of the developer through the automatic classification of commits containing these refactoring operations. Our approach identifies the type of development tasks that refactorings were interleaved with, e.g., updating a feature, debugging, etc. Practically, we labeled 1,702 commit messages, selected from 111,884 commits, based on the category they belong to, i.e., Functional, BugFix, Internal Quality Attribute, Code Smell Resolution, and External Quality Attribute. We deploy several models and we report their performance in terms of classification accuracy. The goal is to identify the developer's motivation behind every application of a refactoring; what caused the developer to refactor? No prior studies have automatically classified refactoring activities, previously applied by a diverse set of developers, belonging to a large set of varied projects. We also randomly sampled an example, from each category, to analyze how developers interleave its corresponding refactorings to support various programming activities.
- **RQ2.** What patterns do developers use to describe their refactoring activities?
  Rationale. Since there is no consensus on how to formally document the act of refactoring code, we intend to extract (from commit messages) words and phrases commonly used to document refactoring. Such information

is useful from many perspectives: it allows the understanding of the rationale behind the applied refactorings, e.g., fixing code smells or improving specific quality attributes. Also, it reveals what refactoring operations tend to be typically documented, and whether developers explicitly mention them as part of their documentation. Little is known about how developers document refactoring as previous studies mainly rely on the keyword *refactor* to annotate such documentation.

Methodology. We mine, in 111,884 commit messages, textual patterns, used by developers to describe their refactoring activities. We extract the most common text patterns used in across all categories, along with specific patterns that were distinguished per category. Since the extracted keywords can be generic and so used in several contexts, we perform the necessary statistical analysis to further extract patterns that are most likely to occur more frequently on commits containing refactorings.

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, and shows how we extracted the categories used for the classification. In Section 4, we give the design of our empirical study, mainly with regard to the constructing of the dataset and classification. Section 5 presents the study results while further discussing our findings in Section 6. The next Section 7 reports threats to the validity of our experiments, before concluding the paper in Section 8.

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 refactoring documentation and 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 mining tools to identify refactoring operations between two versions of a software system. Our technique takes advantage of these tools to discover refactorings in large bodies of software. [Dig et al. 2006] developed a tool called Refactoring Crawler, which uses syntax and graph analysis to detect refactorings. [Prete et al. 2010]

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
[Kim et al. 2010a; Prete et al. 2010]	2010	Ref-Finder	Template-based rules reconstruction technique	63
[Silva and Valente 2017]	2017	RefDiff	Static analysis & code similarity	13
[Tsantalis et al. 2018]	2018	RefactoringMiner	Design Evolution analysis	28

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 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. [Tsantalis et al. 2018] extended their tool to enhance the accuracy of the 28 refactoring types that can be detected through structural constraints. A recent survey [Tan and Bockisch 2019] compares several refactoring detection tools and shows that Refactoring Miner is currently the most accurate refactoring detection tool. Table 1 summarizes the detection tools cited in this study.

Our choice of the mining tool is driven by accuracy, therefore we use Refactoring Miner for this because it is found to be the most accurate [Tsantalis et al. 2018] in terms of correctly identifying refactoring operations (precision of 98% and recall of 87%).

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. 2017b] 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. They 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 a 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.

As refactoring is intended to improve design, several studies have performed a fine-grained analysis to see the impact of individual refactoring types, on various code quality metrics [Chaparro et al. 2014], their interesting findings show that not all refactoring operations significantly improve structural metrics, even some operations may negatively impact some quality metrics. Developers need to pay closer attention to the type of operations performed when targeting specific quality metrics. However, several studies by [Bavota et al. 2014a, 2013b] induced that design issues are not the only drivers for recommending refactoring. For instance, semantics and relational topic models can better mimic the human thinking when extracting classes and moving methods.

[Palomba and Zaidman 2017] have demonstrated that refactoring can be used to repair flaky tests, i.e., tests with random output for the same given input, under the same configuration. They have experienced the correction of test flakiness in 18 open-source Java projects and shown the effectiveness of refactoring as an automated tool for test repair.

[Vassallo et al. 2018] questioned whether the rise of continuous integration has affected the developer's refactoring practices. Their key findings highlight that (i) continuous refactoring is becoming a good practice in CI, (ii) promptly correct design issues as they appear in order to pass CI enforced quality reviews (iii) there is a need for *developer-centric* static analysis tools and refactoring prioritization techniques.

[Lin et al. 2019] investigated the relationship between refactoring and code naturalness by measuring the impact of 10 refactoring types on code cross-entropy. Their findings show that naturalness can be an indicator that is worth analyzing, when recommending better refactoring operations.

Another study relevant to our work is by [Vassallo et al. 2019]. They performed an exploratory study on refactoring activities in 200 projects, by mining their performed refactoring operations. Their findings show the need for better understanding the rationale behind these operations, and so our study focuses on contextualizing refactoring activities within typical software engineering activities and questions whether such difference in developers' intentions would infer different refactorings strategies. Such investigation has not been investigated before in the literature. Furthermore, previous studies have shown that refactoring can be used outside of the *design box*, e.g., correction flaky tests, code naturalness, etc., therefore, our study is the first to engage the automated classification of commit messages in order to cluster the refactoring effort that has been performed in non-design circumstances.

Existing studies have shown that refactoring does not necessarily always improve design quality [Bavota et al. 2015; Chaparro et al. 2014] and can be used out of its traditional context, e.g., dealing with flaky tests, supporting CI [Palomba and Zaidman 2017; Vassallo et al. 2018], etc. Since, 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:

- **Functional.** Refactoring the existing system to account for the upcoming functionalities.
- **BugFix.** Refactoring the design is a part of the debugging process.
- **Internal Quality Attribute.** Refactoring is still the de-facto for increasing the system's modeling quality and design defect's correction.
- **Code Smell Resolution.** Refactoring to remove design anti-patterns.
- **External Quality Attribute.** refactoring helps in increasing the system's visibility to the developers, through the optimization of non-functional attributes such as testability, understandability, and readability.

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 Refactoring Documentation

Table 2. Refactoring identification related work

Study	Year	Purpose	Approach	Source of Info.	Ref. Patterns
[Stroggylos and Spinellis 2007]	2007	Identify refactoring commits	Mining commit logs	General commits	1 keyword
[Ratzinger et al. 2008][Ratzinger 2007]	2007 & 2008	Identify refactoring commits	Mining commit logs	General commits	13 keywords
[Murphy-Hill et al. 2012]	2012	Identify refactoring commits	Ratzinger's approach	General commits	13 keywords
[Soares et al. 2013]	2013	Analyze refactoring activity	Ratzinger's approach Manual analysis Dynamic analysis	General commits	13 keywords
[Kim et al. 2014]	2014	Identify refactoring commits	Identifying refactoring branches Mining commit logs	Refactoring branch	Top 10 keywords
[Zhang et al. 2018]	2018	Identify refactoring commits	Mining commit logs	General commits	22 keywords
[AlOmar et al. 2019a]	2019	Identify refactoring patterns	Detecting refactorings Extracting commit messages	Refactoring commits	87 keywords & phrases

A number of studies have focused recently on the identification and detection of refactoring activities during the software life-cycle. One of the common approaches to identify refactoring activities is to analyze the commit messages

in versioned repositories. [Stroggylos and Spinellis 2007] opted for searching words stemming from the verb *"refactor"* such as "refactoring" or "refactored" to identify refactoring-related commits. [Ratzinger 2007; Ratzinger et al. 2008] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions to identify whether a transformation contains refactoring. The authors identified refactorings based on a set of keywords detected in commit messages, and focusing, in particular, on the following 13 terms in their search approach: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move.*

Later, [Murphy-Hill et al. 2012] replicated Ratzinger's experiment in two open source systems using Ratzinger's 13 keywords. They conclude that commit messages in version histories are unreliable indicators of refactoring activities. This is due to the fact that developers do not consistently report/document refactoring activities in the commit messages. In another study, [Soares et al. 2013] compared and evaluated three approaches, namely, manual analysis, commit message (Ratzinger et al.'s approach [Ratzinger 2007; Ratzinger et al. 2008]), and dynamic analysis (SafeRefactor approach [Soares et al. 2009]) to analyze refactorings in open source repositories, in terms of behavioral preservation. The authors found, in their experiment, that manual analysis achieves the best results in this comparative study and is considered as the most reliable approach in detecting behavior-preserving transformations.

In another study, [Kim et al. 2014] surveyed 328 professional software engineers at Microsoft to investigate when and how they do refactoring. They first identified refactoring branches and then asked developers about the keywords that are usually used to mark refactoring events in change commit messages. When surveyed, the developers mentioned several keywords to mark refactoring activities. Kim et al. matched the top ten refactoring-related keywords identified from the survey (*refactor, clean-up, rewrite, restructure, redesign, move, extract, improve, split, reorganize, rename*) against the commit messages to identify refactoring commits from version histories. Using this approach, they found 94.29% of commits do not have any of the keywords, and only 5.76% of commits included refactoring-related keywords.

More recently, [Zhang et al. 2018] performed a preliminary investigation of Self-Admitted Refactoring (SAR) in three open source systems. They first extracted 22 keywords from a list of refactoring operations defined in the Fowler's book [Fowler et al. 1999] as a basis for SAR identification. After identifying candidate SARs, they used Ref-Finder [Kim et al. 2010b] to validate whether refactorings have been applied. In their work, they used code smells to assess the impact of SAR on the structural quality of the source code. Their main findings are the following (1) SAR tend to enhance the software quality although there is a small percentage of SAR that have introduced code smells, and (2) the most frequent code smells that are introduced or reduced depend highly on the nature of the studied projects. They concluded that SAR is a signal that helps finding refactoring events, but it does not guarantee the application of refactorings. We summarize these state-of-the-art approaches in Table 2.

2.4 Commits Classification

A wide variety of approaches to categorize commits have been presented in the literature. The approaches vary between performing manual classification [Cedrim et al. 2017; Chávez et al. 2017; Hindle et al. 2008; Mauczka et al. 2015; Silva et al. 2016], to developing an automatic classifier [Hassan 2008; Mauczka et al. 2012], to using machine learning techniques [Amor et al. 2006; Hindle et al. 2011, 2009; Levin and Yehudai 2017; Mahmoodian et al. 2010; McMillan et al. 2011] and developing discriminative topic modeling [Yan et al. 2016] to classify software changes. These approaches are summarized in Table 3.

[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

Table 3. 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 (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 (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 (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
[Cedrim et al. 2017]	2017	Yes/No	Yes/No	Systematic Labeling	Floss Refactoring Root-canal Refactoring	part of sample of 2584	Manual
[Levin and Yehudai 2017]	2017	Yes/No	No/Yes	Machine Learning Classifier	Swanson's category	1151 Commits	Accuracy: 73%

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, [Cedrim et al. 2017; 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". They automatically assess 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 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.

Our proposal differs from previous studies, as their classification targeted general maintenance activities (perfective, adaptive) and was not specific to commits containing messages describing refactoring activities. In this study, we subdivide what would have been considered "perfective" in previous studies, into three separate categories, namely, Internal, External and Code smell. This division is inherited from the analysis of previous papers whose detection of refactoring opportunities relies on the optimization of high-level design principles, structural metrics, and reduction of code smells. Thus, this is not a typical commit classification since refactoring related commit messages contain a strong overlap in their terminology and so their classification is challenging. Moreover, as we previously stated, existing studies in recommending refactoring are based on (i) internal quality attributes (ii) external quality attributes, and (iii) code smells. The classification of commits according to these categories, will be an empirical evidence of whether and to what extent these factors are being used in practice. To perform the classification, we use existing classifiers (e.g., Random Forest, Naive Bayes Multinominal, etc) that have been used by several studies (e.g., [Kochhar et al. 2014; Levin and Yehudai 2017]) in the context of commit classification and challenge them using our defined set of classes. Although several studies [Amor et al. 2006; Hattori and Lanza 2008; Hindle et al. 2009, 2008; Levin and Yehudai 2017; Mauczka et al. 2015, 2012; Yan et al. 2016] have discussed how to automatically classify change messages into Swanson's general maintenance categories (i.e., Corrective, Adaptive, Perfective), refactoring, in general, has been classified as a sub-type of "Perfective" in these maintenance categories. While we are motivated by the above-mentioned studies, our work is still different from them since we apply the machine learning technique to automatically classify commit messages into five main refactoring motivations defined in this study, i.e., functional, bugfix, internal QA, code smell resolution and external QA.

3 REFACTORING DOCUMENTATION CHALLENGES

Commit messages represent the human translation, in natural language, of the code-level changes. Therefore, with the raise of version control systems and mining software repositories, several studies have been analyzing commit messages for various purposes including change classification [??], change bug-proneness [?], and developers rationale behind their coding decisions [?].

In this context, we aim to extract how developers express their nonfunctional activities, namely improving software design, renaming semantically ambiguous identifiers, removing code redundancies etc. Multiple studies have been detecting the performed refactoring operations, *e.g.*, rename class, move method etc. within committed changes to better understand how developers cope with bad design decisions, also known as design antipatterns, and to extract their removal strategy through the selection of the appropriate set of refactoring operations [Tsantalis et al. 2018]. As the accuracy of refactoring detectors has reached a relatively high rate, the mined commits represent a rich space to understand how developers describe, in natural language, their refactoring activities. Yet, such information retrieval can be challenging since there are no common standards on how developers should be formally documenting their refactorings, besides inheriting all the challenges related to natural language processing [?].

However, using the developer inline documentation has added another dimension to better understanding software quality, as mining developers comments, for instance, has unveiled how developers knowingly commit code that is either incomplete, temporary, error-prone. These situations have been coined as *Self-Admitted Technical Debt* (SATD) [?], as they are extracted from text-mining developers' messages, which represents a reliable source, instead of measuring technical debt only by the deviations from ideal code, *i.e.*, code smells. Inspired by the study of Potdar and Shihab [?], we analyze commits which are known to contain refactoring operations and we extract how developers describe, in plain unformal text, their refactoring activities. However, we were skeptical about using the term self-admitted refactoring, as self-admission is defined as confessing a specific charge or accusation, which makes it appropriate for technical debt, but not for what may reduce it *i.e.*, refactoring. Therefore, we label developer's documented refactorings as *Self-Affirmed Refactoring* (SAR), since self-affirmation is defined as the assertion of self-existence [?], which is, in the context of refactoring, refers to developers recognition of the value of refactoring activities as means to improve their code and more specifically cope with technical debt, if we refer to its original definition [?].

Similarly to the early studies in SATD, we perform our study on several various open source projects, to capture a wider variety of potential expressions and to have a better quantification of the amount of SAR, along with identifying the rationale behind refactoring, if mentioned. This study represents our initial exploration of the existence of SAR, and we plan on extending it to investigate whether SAR is an indicator of lesser technical debt in the code. The next section discusses the related work and then details the methodology of our empirical study and how we collected the data used in our experiments.

4 EMPIRICAL STUDY SETUP

To answer our research questions, we conducted a five phased approach that consisted of: (1) data collection, (2) refactoring detection, (3) automatic classification, (4) unit test file detection and (5) refactoring patterns extraction.

4.1 Phase 1: Data Collection

To perform this study, we randomly selected 800 projects, which were curated open-source Java projects hosted on GitHub. These curated projects were selected from a dataset made available by [Munaiah et al. 2017], while verifying

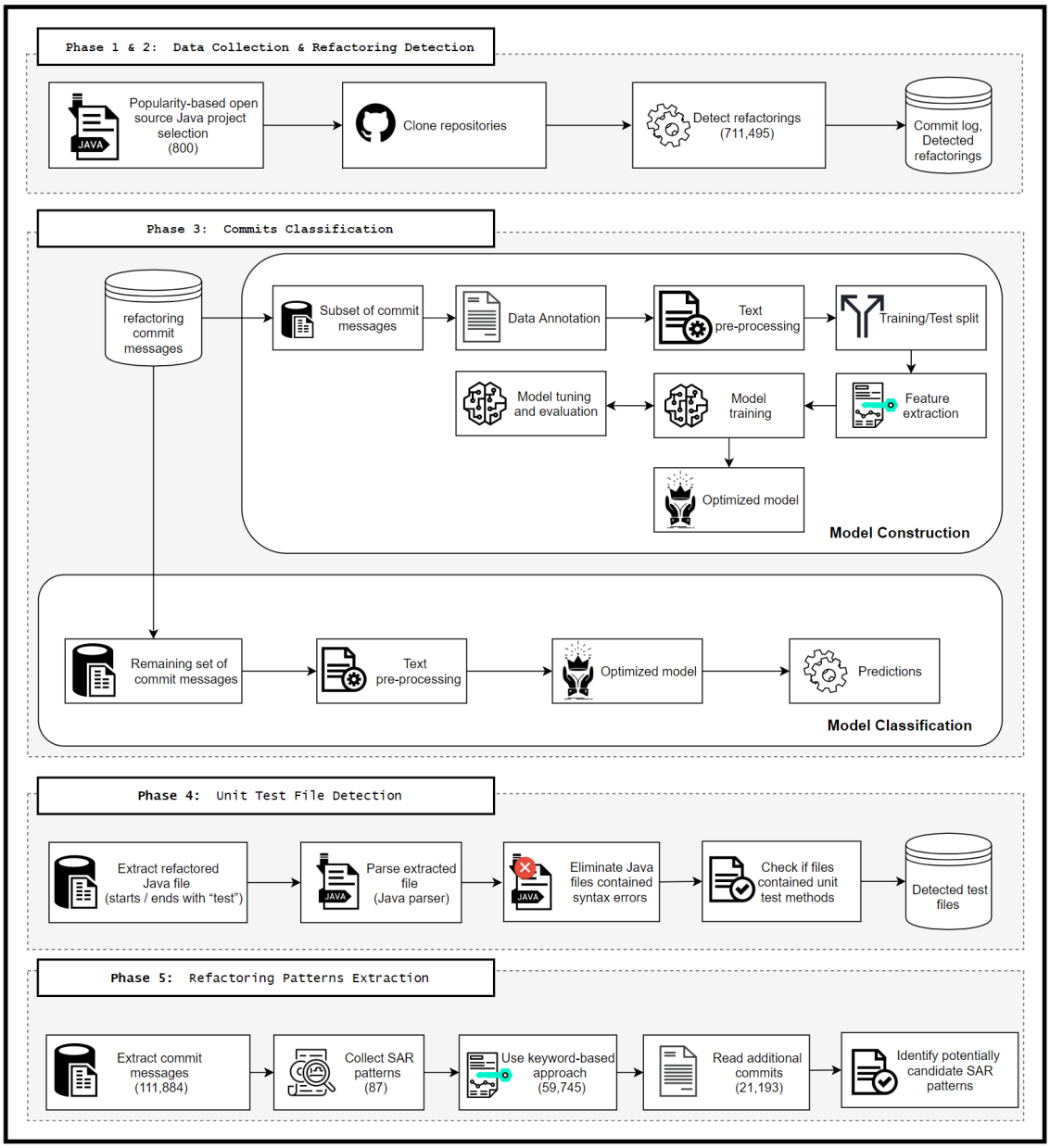


Fig. 1. Empirical study design overview

that they were Java-based; the only language supported by Refactoring Miner. The authors of this dataset classified "well-engineered software projects" based on the projects' use of software engineering practices such as documentation, testing, and project management. Additionally, these projects are non-forked (i.e., not cloned from other projects); forked projects may impact our conclusions by introducing duplicate code and documents. We cloned these projects

in early 2019, and 74.6% of the projects had their most recent commit within the last four years. The 800 selected projects analyzed in this study have a total of 748,001 commits, and a total of 711,495 refactoring operations from 111,884 refactoring commits. Additionally, these projects contain 732 commits and involve 19 developers on average. An overview of the projects is provided in Table 4.

Table 4. Projects overview

Item	**Count**
Total of projects	800
Total commits	748,001
Refactoring commits	111,884
Refactoring operations	711,495
Considered Projects - Refactored Code Elements	
Code Element	**# of Refactorings**
Method	302,929
Class	228,974
Attribute	80,509
Parameter	42,992
Variable	28,765
Package	2380
Interface	1742

4.2 Phase 2: Refactoring Detection

For the purpose of extracting the entire refactoring history of each project, we used the Refactoring Miner tool proposed by [Tsantalis et al. 2018]. 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 is described in Table 5. 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 [Tsantalis et al. 2018] 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,

Table 5. Refactoring Types (Extracted from [Fowler et al. 1999])

Refactoring Type	Description
(1) Change Package	Change package involves several package-level operations (Move, Rename, Split, Merge).
(2) Extract & Move Method	Turn the code fragment into a method whose name explains the purpose of the method and move it to class on which it is defined.
(3) Extract Class	A class does a work of two classes. Create a new class and move relevant fields and methods to the new class.
(4) Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. Extract the subset into an interface.
(5) Extract Method	A code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.
(6) Extract Subclass	A class has features used only in some instances. Create subclass for that subset of features.
(7) Extract Superclass	There are two classes with similar features. Create a superclass and move the common features to the superclass.
(8) Extract Variable	Turn the fragment into a method whose name explains the purpose of the method.
(9) Inline Method	When method's body is just as clear as its name. Put the method's body into the body of its callers and remove the method.
(10) Inline Variable	When a variable is just as clear as its name. Put the variable into the body of its callers and remove the variable.
(11) Move & Rename Attribute	Move attribute into another class and change the name of the attribute.
(12) Move & Rename Class	Move all class features into another class and change the name of the class.
(13) Move Attribute	A field is, or will be, used by another class more than the class on which it is defined.
(14) Move Class	A class isn't doing very much. Move all its features into another class and delete it.
(15) Move Method	A method is, or will be, using or used by more features of another class than the class on which it is defined.
(16) Move Source Folder	Move a source folder to a different folder.
(17) Parameterize Variable	When a variable is changed to be a parameter of a method.
(18) Pull Up Attribute	Two subclasses have the same field. Move the field to the superclass.
(19) Pull Up Method	If at least two subclasses share an identical method, move it to the superclass.
(20) Push Down Attribute	A field is used only by some subclasses. Move the field to those subclasses.
(21) Push Down Method	The Behavior on a superclass is relevant only for some of its subclasses. Move it to those subclasses.
(22) Rename Attribute	The name of an attribute does not reveal its purpose. Change the name of the attribute.
(23) Rename Class	The name of a class does not reveal its purpose. Change the name of the class.
(24) Rename Method	The name of a method does not reveal its purpose. Change the name of the method.
(25) Rename Parameter	The name of a parameter does not reveal its purpose. Change the name of the parameter.
(26) Rename Variable	The name of a variable does not reveal its purpose. Change the name of the variable.
(27) Replace Attribute	An attribute is replaced with another attribute.
(28) Replace Variable with Attribute	A variable is replaced with an attribute.

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. A recent survey [Tan and Bockisch 2019] shows that Refactoring Miner is currently the most accurate refactoring detection tool. It is important to note that, while we were conducting this study, a more recent tool named *refDiff* [Silva and Valente 2017] 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.

4.3 Phase 3: Commits Classification

As part of the development workflow, developers associate a message with each commit they make to the project repository. These commit messages are usually written using natural language, and generally convey some information about the commit they represent. In this study, we aim to determine the type of refactoring activity performed by the developer based on the message associated with a refactoring based commit. Hence, we aim to classify the refactoring commit, into one of five categories – 'Functional', 'Bug Fix', 'Internal QA', 'Code Smell Resolution', and 'External QA'. We start by collecting motivation driving refactoring reported in the literature [Dallal and Abdin 2017; Fowler et al. 1999; Kim et al. 2014; Lanza and Marinescu 2007; Murphy-Hill et al. 2012; Palomba et al. 2017a; Silva et al. 2016; Tsantalis et al. 2013]. Then, we search for common categories among the reported motivations. The following step involves identifying categories clustering feature requests, quality attributes and software issues under the identified categories. This process resulted in five different categories mentioned earlier. Table 6 provides a description for each category.

In this supervised multi-class classification problem, we followed a multi-staged approach in model constructing and commit message classification. The first stage involved the construction of the model. In the second stage, we utilized the constructed model to classify the entire dataset of commit messages. An overview of our methodology is depicted in Figure 1. In the following subsections, we describe in detail the activities involved in each stage of the construction and classification process.

Table 6. Classification categories

Category	Description
Functional	Feature implementation, modification or removal
BugFix	Tagging, debugging, and application of bug fixes
Internal QA	Restructuring and repackaging the system's code elements to improve its internal design such as coupling and cohesion
Code Smell Resolution	Removal of design defects that might indicate a problem such as duplicated code and long method
External QA	Enhancement of non-functional attributes such as testability, understandability, and readability

Model Construction

4.3.1 Data Annotation. In order to construct a machine learning model, a gold set of labeled data is needed to train and test the model. To construct this gold set, a manual annotation (i.e., labeling) of commit messages needs to be performed by subject matter experts. To this end, we annotated 1,702 commit messages. This quantity roughly equates to a sample size with a confidence level of 95% and a confidence interval of 2. The authors of this paper performed annotation of the commit messages. Provided to each author was a random set of commit messages along with details defining the annotation labels. Each annotator had to label each provided commit message with a label of either 'Functional', 'Bug Fix', 'Internal QA', 'Code Smell Resolution', and 'External QA'. To mitigate bias in the annotation process, the annotated commit messages were peer-reviewed by the same group. All decisions made during the review had to be unanimous; discordant commit messages were discarded and replaced. The results of our annotation resulted in an moderately balanced dataset. In total, we annotated 348 commit messages as 'Functional', 'Bug Fix', 'Internal QA', and 'Code Smell Resolution', while 310 messages were labeled as 'External QA'.

4.3.2 Text Pre-Processing. To better aide the model in correctly classifying commit messages, we performed a series of text normalization activities. Normalization is a process of transforming non-standard words into a standard and convenient format [Jurafsky and Martin 2019]. Similar to [Kochhar et al. 2014; Le et al. 2015], the activities involved in our pre-processing stage included: (1) expansion of word contractions (e.g., 'I'm' → 'I am'), (2) removal of URLs, single-character words, numbers, punctuation (and other non-alphabet) characters, stop words, and (3) reducing each word to its lemma. We opted to use lemmatization over stemming, as the lemma of a word is a valid English word [Lane et al. 2019]. In relation to stopwords - we used the default set of stopwords supplied by NLTK [Bird 2002] and also added our own set of custom stop words. To derive the set of custom stop words, we generated and manually analyzed the set of frequently occurring words in our corpus. Examples of custom stop words include 'git', 'code', 'refactor', etc. Additionally, for more effective pre-processing, we tokenized each commit message. Tokenization is the process of dividing the text into its constituent set of words.

4.3.3 Training/Test Split. To gauge the accuracy of a machine learning model, the implemented model must be evaluated on a never-seen-before set of observations with known labels. To construct this set of observations, the set of annotated commit messages were divided into two sub-datasets - a training set and a test set. The training set was utilized to construct the model while the test set was utilized to evaluate the classification ability of the model. For our experiment, we performed a shuffled stratified split of the annotated dataset. Our test dataset contained 25% of the annotated

commit messages, while the training dataset contained the remaining 75% of annotated commit messages. This split results in the training dataset containing a total of 1,276 commit messages, which breaks down to 246 'Functional', 271 'BugFix', 255 'Internal', 276 'CodeSmell', and 228 'External' labeled commit messages. The stratification was performed based on the class (i.e, annotated label) of the commit messages. The use of a random stratified split ensures a better representation of the different types (i.e., labels) of commit messages and helps reduce the variability within the strata [Singh and Mangat 2013].

4.3.4 Feature Extraction. In order to create a model, we need to provide the classifier with a set of properties or features that are associated with the observations (i.e., commit messages) in our dataset. However, not all features associated with each observation will be useful in improving the prediction abilities of the model. Hence, a feature engineering task is required to determine the set of optimum features [Zheng and Casari 2018]. In our study, we evaluated the model using the text of the commit message. We utilized Term Frequency-Inverse Document Frequency (TF-IDF) [Manning et al. 2008] to convert the textual data into a vector space model that can be passed into the classifier. In our experiments, we evaluate the accuracy of the model by constructing the TF-IDF vectors using different types of N-Grams and feature sizes.

4.3.5 Model Training. For our study, we evaluated the accuracy of six machine learning classifiers – Random Forest, Logistic Regression, Multinomial Naive Bayes, K-Nearest Neighbors, Support Vector Classification, and Decision Tree. Additionally, to ensure consistency, we ran each classifier with the same set of test and training data each time we updated the input features.

4.3.6 Model Tuning & Evaluation. The purpose of this stage in the model construction process is to obtain the optimal set of classifier parameters that provide the highest performance; in other words, the objective of this task is to tune the hyperparameters. For example, for the K-Nearest Neighbors classifier, we tuned the number of neighbors hyperparameter (i.e., 'k') by evaluating the accuracy of the model as we increased the value of 'k' from 1 to 50 in increments of one. We tuned at least one hyperparameter associated with each classifier in our list. For numeric-based hyperparameters, we determined the bounds/range for testing through continuously running the classifier with a different range of values to identify the appropriate minimum and maximum value.

We performed our hyperparameter tuning on the training dataset using a combination of 10-fold cross-validation and an exhaustive grid search [Dangeti 2017]. Grid search utilizes a brute force technique to evaluate all combinations of hyperparameters to obtain the best performance. Since our classification is multiclass, we relied on the Micro-F1 score. The combination of hyperparameters that resulted in the highest Micro-F1 score was selected to construct the model.

4.3.7 Optimized Model. In this stage, the optimized model produced by the training phase is utilized to predict the labels of the test dataset. Based on the predictions, we measure the precision and recall for each label as well as the overall F1-Score of the model. We observed that ***Random Forest resulted in the best F1 score: 87%***. Random Forest belongs to the family of ensemble learning machines, and has typically yielded superior predictive performance mainly due to the fact that it aggregates several learners. Hence, we utilized this machine learning algorithm (and its optimal set of hyperparameters) as the optimum model for our study. The complete set of scores for all the classifiers is provided in Table 7. Additionally, Table 8 provides a more detailed report – the Precision, Recall, and F-measure scores per class for each machine learning classifier.

Table 7. Precision, Recall, and F-measure of studied classifiers

Classifier	Precision	Recall	F1
Random Forest	0.87	0.87	0.87
Support Vector Classification	0.87	0.86	0.86
Decision Tree	0.85	0.85	0.85
Logistic Regression	0.83	0.83	0.83
Multinomial Naive Bayes	0.81	0.78	0.78
K-Nearest Neighbors	0.78	0.77	0.76

Table 8. Detailed classification metrics of each classifier

Random Forest				*Support Vector Classification*				*Decision Tree*			
Category	**Precision**	**Recall**	**F1**	**Category**	**Precision**	**Recall**	**F1**	**Category**	**Precision**	**Recall**	**F1**
Bug Fix	0.83	0.79	0.81	Bug Fix	0.75	0.78	0.77	Bug Fix	0.77	0.80	0.78
Code Smell	0.93	0.95	0.94	Code Smell	0.93	0.94	0.93	Code Smell	0.89	0.91	0.90
External QA	0.85	0.91	0.88	External QA	0.92	0.89	0.90	External QA	0.77	0.90	0.83
Functional	0.81	0.91	0.86	Functional	0.77	0.88	0.82	Functional	0.92	0.83	0.87
Internal QA	0.95	0.81	0.87	Internal QA	0.95	0.84	0.89	Internal QA	0.91	0.80	0.85

Logistic Regression				*Multinomial Naive Bayes*				*K-Nearest Neighbors*			
Category	**Precision**	**Recall**	**F1**	**Category**	**Precision**	**Recall**	**F1**	**Category**	**Precision**	**Recall**	**F1**
Bug Fix	0.66	0.70	0.68	Bug Fix	0.63	0.77	0.69	Bug Fix	0.62	0.71	0.66
Code Smell	0.89	0.94	0.91	Code Smell	0.82	0.94	0.87	Code Smell	0.76	0.93	0.84
External QA	0.88	0.88	0.88	External QA	0.97	0.71	0.82	External QA	0.85	0.75	0.79
Functional	0.77	0.87	0.82	Functional	0.66	0.83	0.74	Functional	0.68	0.73	0.71
Internal QA	0.96	0.78	0.86	Internal QA	0.99	0.67	0.80	Internal QA	0.97	0.71	0.82

Model Classification

In this stage of our experiment, we utilized the optimized model that we created in the prior stage. However, to be consistent, before classifying each commit message, we performed the same text pre-processing activities, as in the prior stage, on the commit message. The result of this stage is the classification of each refactoring commit into one of the five categories. The output of this classification process was utilized in our experiments in order to answer our research questions.

4.4 Phase 4: Unit Test File Detection

As part of our study, we look at the refactoring applied to unit test files and perform comparisons against production-file-based refactorings. To identify all test files that were refactored, we followed the same detection approach as [Peruma et al. 2019a]. In this approach, following JUnit's file naming recommendation[1], we first extracted all refactored java source files where the filename either starts or ends with the word "test". Next, we utilized JavaParser[2] to parse each extracted file. By parsing the files, we were able to eliminate Java files that contained syntax errors and were able to detect if the file contained JUnit-based unit test methods accurately, thereby cutting down on false positives. Finally, to ensure that the files were indeed unit test files, we checked if the files contained unit test methods. As per JUnit

[1] https://junit.org/junit4/faq.html#running_15
[2] https://javaparser.org/

specifications, a test method should have a public access modifier, and either has an annotation called @Test (JUnit 4), or the method name should start with "test" (JUnit 3).

4.5 Phase 5: Refactoring Patterns Extraction

To identify self-affirmed refactoring patterns, we perform manual analysis similar to our previous work [AlOmar et al. 2019a]. Since commit messages are written in natural language and we need to understand how developers express refactoring, we manually analyzed commit messages by reading through each message to identify self-affirmed refactorings. We then extracted these commit comments to specific patterns (i.e., a keyword or phrase). To avoid redundancy of any kind of patterns, we only considered one phrase if we found different forms of patterns that have the same meaning. For example, if we find patterns such as "Simplifying the code", "Code simplification", and "simplify code", we add only one of these similar phrases in the list of patterns. This enables us to have a list of the most insightful and unique patterns. It also helps in making more concise patterns that are usable for readers. The manual analysis process took approximately 20 days in total, and was performed by the authors of the paper. This step resulted in analyzing 59,745 commit messages. Then, we iterated over the set again while excluding the terms identified in our previous work, to identify additional self-affirmed refactoring patterns. We manually read through 21,193 commit messages. Our in-depth inspection resulted in a list of 513 self-affirmed refactoring patterns identified across the considered projects, as illustrated in Table 12 and 13.

To mitigate the risk of a biased dataset, we extract stratified sample from each category, which were annotated by the first author, and have these sample commits independently labeled again by the second author. We used Cohen's Kappa coefficient [Cohen 1960] to evaluate the inter-rater agreement level for the categorical classes. We achieved an agreement level 0.82. According to Fleiss et al. [Fleiss et al. 1981], these agreement values are considered to have an almost *perfect agreement* (i.e., 0.81˜1.00).

5 EXPERIMENTAL RESULTS

This section reports our experimental results and aims to answer the research questions in Section 1. The dataset of classified refactorings along with textual patterns are available online[3] for replication and extension purposes. A summary of all figures and tables which help address our research questions is illustrated in Table 9.

Table 9. Summary of the empirical study design.

Research question	Reference
RQ1: Motivation behind refactoring	Figure 2, 3, 4, 6, 8, 10, 12, Table 10, 11
RQ2: Refactoring documentation	Table 12, 13, 14, 15, Figure 14, 15, 16

5.1 RQ1: 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 4.3. This section details the classification of 111,884 commit messages containing 711,495 refactoring operations.

[3]https://refactorings.github.io/empirical-refactoring/

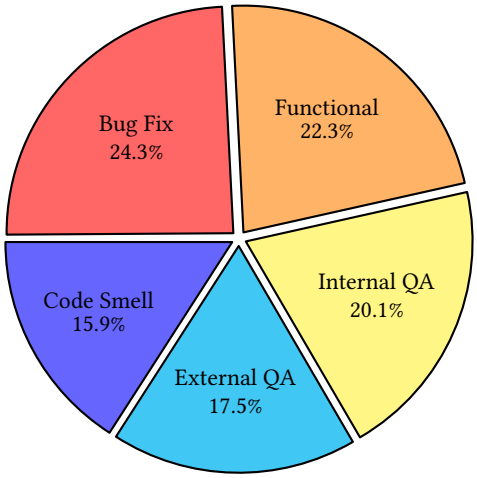
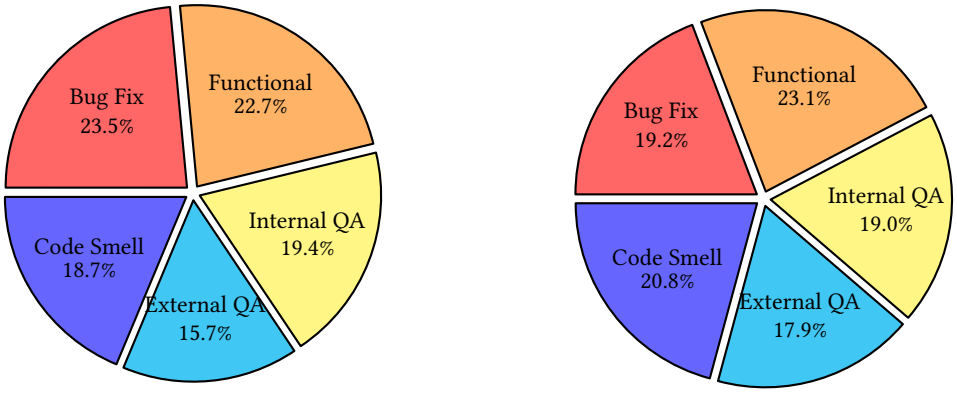


Fig. 2. Percentage of classified commits per category in all projects combined



(a) Production File

(b) Test File

Fig. 3. Percentage of classified commits per category in production and test files

Figure 2 shows the categorization of commits, from all projects combined. All of the categories had a uniform distribution of refactoring classes. For instance, Bug Fix, Functional, Internal QA, External QA, and Code Smell Resolution had respectively commit message distribution percentages of 24.3%, 22.3%, 20.1%, 17.5%, and 15.9%.

The first observation that we can draw is that developers do not solely refactor their code to fix code smells. They instead refactor the code for multiple purposes. This can be explained by the fact that developers tend to make design-improvement decisions that include re-modularizing packages by moving classes, reducing class-level coupling, increasing cohesion by moving methods, and renaming elements to increase naming quality in the refactored design. Developers also 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 the system's non-functional attributes such as extensibility and maintainability.

Figure 3 depicts the distribution of refactoring commits for all production and test files for each refactoring motivation. As can be seen, developers tend to refactor these two types of source files for several refactoring intentions, and they care about refactoring the logic of the application and refactoring the test code that verifies if the application works as expected. Although developers usually handle production and test code differently, the similarity of the patterns shows that they refactor these source files for the same reasons with unnoticeable differences.

Concerning refactorings applied in the production files, developers perform refactoring for several motivations. For the Bug Fix category, an interpretation for this comes from the nature of the debugging process that includes the disambiguation of identifier naming that may not reflect the appropriate code semantics or that may be infected with lexicon bad smells (i.e., 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 reduces its proneness to errors [Tsantalis and Chatzigeorgiou 2011]. Regarding the Internal category, developers move code elements for design-level changes [Alshayeb 2009; Bavota et al. 2015; Mkaouer et al. 2015; Stroggylos and Spinellis 2007], e.g., developers tend to re-modularize classes to make packages more cohesive, and extract methods to reduce coupling between classes. As for the External category, developers optimize the code to improve the non-functional quality attributes such as readability, understandability, and maintainability of the production files. For the Code Smell Resolution category, developers eliminate any bad practices and adhere to object-oriented design principles. Finally, for the Feature category, Table 10 not only illustrates how diverse the operations used while updating the system's functionality can be, but also 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 fault at the same time.

With regards to test files, developers perform refactoring to improve the design of the code. A good example can be demonstrated by renaming a given code element such as a class, a package or an attribute. Finding better names for code identifiers serves the purpose of increasing the software's comprehensibility. As in Table 10, developers explicitly mention the use of the renaming operations for the purpose of disambiguating the redundancy of methods names and enhancing their usability. Another activity to refactor test files could be moving methods, or pushing code elements across hierarchies, e.g., pushing up attributes. Each of these activities are performed to support several refactoring motivations.

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 10 shows the five main motivations driving refactoring, which we can also divide into fine-grained subcategories. For instance, we subcategorize *Functional*-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 papers we surveyed in the related work. So, the corresponding subcategories for *Design* are *code smell resolution*, *duplicate 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; some of which go beyond

Martin Fowler's traditional definition of associating refactoring with improving design by removing code smells. These subcategories are not exhaustive, there are many others. These are just examples to illustrate what commits look like.

- For functional-focused refactoring, developers are refactoring the code to introduce, modify, or delete features. 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 the 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 addition).
- 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).
- For code smell-focused refactoring, it is clear that developers performed Extract Method refactoring to remove a code smell which corresponds to a long method bad smell. Further, developers primarily refactor the code to improve the dominant modularization driving forces (i.e., cohesion and coupling) to maximize intra-class connectivity and minimize inter-class connectivity.
- To improve the internal design, it is apparent from Table 10 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.*
- 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 making changes such as 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 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 the long method code smell, developers extracted a method to reduce the length of the original method body. Also, to remove the feature envy code smell, move method refactoring operation was performed in order to place the fields and methods in 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 was noted for some of the refactoring commit messages reviewed for this paper. For instance, to eliminate a long method, Fowler

Table 10. Commits message examples classified by category

Category	Subcategory	Refactoring Operation	Commit Messages
Functional	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.
Code Smell	Long Method	Extract Method	MongoStore: Refactor to avoid long methods
	Duplicate Code	Extract Method	Minor refactoring to remove duplicate code
	Data Class	Move Class	Refactoring of core data classes. Implementation of RefactoringRegistry.
	Feature Envy	Move Method	Introduced adjuncts from stapler - fixed the feature envy problem in the version computation.
Internal	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.
External	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).

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.

> *Summary.* Our classification has shown that fixing code smells is not the main driver for developers to refactoring their code bases. Indeed, the percentage of commits belonging to this category account for only 15.9 % of the overall classified commits, making this class the least among all other categories. BugFix is found to be leading with a percentage of 24.3%, but, the sum of the design-related categories, namely code smell, internal, and external, represent the majority with a 53.5%. As explicitly mentioned by the developers in their commits messages, refactoring is solicited for a wide variety of reasons, going beyond its traditional definition, such as reducing the software's proneness to bugs, easing the addition of functionality, resolving lexical ambiguity, enforcing code styling, and improving the design's testability and reusability.

To further analyze our classification, this following subsection also discusses five case studies that demonstrate GitHub developers intentions when refactoring source code. For each case study, we provide the commit message and its corresponding refactoring operations detected by Refactoring Miner. The corresponding source code for each case study is available online. Additional case studies can also be found online.

5.2 Case Studies

This subsection reveals more details with respect to our classified commits. As we validate our classification results, we have selected an example from each category. For each example, we checkout the corresponding commit to obtain the source code, then two authors manually analyze the code changes. The purpose is not to verify the consistency between the commit message and its corresponding changes, but to capture the context in which refactorings were applied. In each analyzed commit, we report its class, its message, the distribution of its corresponding refactoring, along with our understanding of their usage context.

5.2.1 *Case Study 1. Refactoring to improve internal quality attributes.*

Fig. 4. Commit message stating the restructuring of code to improve its understandability

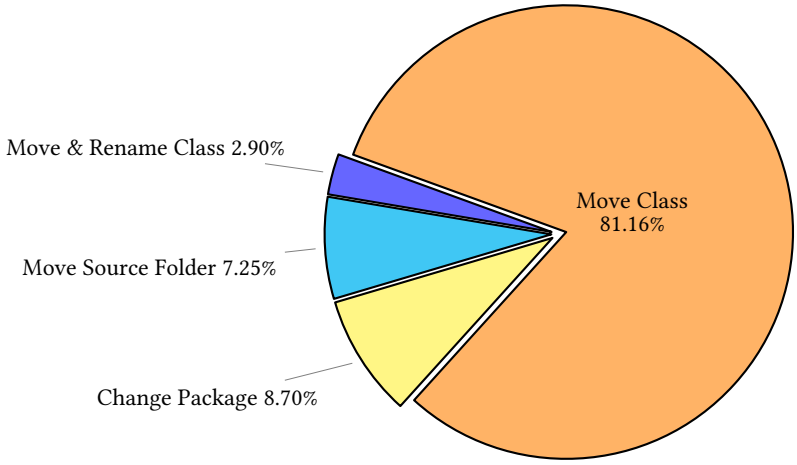


Fig. 5. Distribution of Refactoring Operations

This case study aims to demonstrate one of the five refactoring motivations reported in this study. The commit message mainly discussed two refactoring practices: (1) performing large refactorings, and (2) optimizing the structure of the codebase. It is apparent from this commit that the main intention behind refactoring the code is to improve the design. Specifically, Refactoring Miner detected 69 refactoring operations associated with this commit message. We observe there is consistency between what documented in the commit message and the actual size of refactoring operations.

Closer inspection of the nature and type of 69 refactorings and the corresponding source code shows that the GitHub commit author massively optimized the package structure within existing modularizations. Particularly, as Figure 5 shows, the developer performed four refactoring types, namely, *Move Class, Change Package, Move Source Folder*, and *Move and Rename Class*. 80.16% of these refactorings were *Move Class* refactorings, 8.70% were *Change Package*, 7.25% were *Move Source Folder*, and 2.90% were composite refactorings (*Move and Rename Class*). As pointed out in Refactoring Miner documentation, *Change Package* refactoring involves several package-level refactorings (i.e., Rename, Move, Split, and Merge packages).

We observe that the developer optimizing the design by performing repackaging, i.e., extracting packages and moving the classes between these packages, merging packages that have classes strongly related to each other, and renaming packages to reflect the actual behaviour of the package. The present observations are significant in at least two major respects: (1) improving the quality of packages structure when optimizing intra-package (i.e., cohesion) and inter-package (i.e., coupling) dependencies and minimizing package cycles and (2) avoiding increasing the size of the large packages and/or merging packages into larger ones. Developer intention to distribute classes over packages, however, might depend on other design factors than package cohesion and coupling. This remodularization activity helps identifying packages containing classes poorly related to each other.

In order to confirm the main refactoring intention when performing this refactoring, we emailed the GitHub contributor and asked about the main motivation behind performing this massive refactorings in the commit message (Figure 4). The GitHub contributor confirmed that the intention was to improve the design and this motivation is best illustrated in the following response about the commit we examined: "*there are a few reasons for large refactorings: (1) the codebase is becoming increasingly difficult to evolve. Sometimes relatively small conceptual changes can make a huge difference, but requires a lot of changes in many places.*" and "*(2) analysis of codebase dependencies, call sequences and so on reveal that the codebase is a mess and needs to be fixed to avoid current or future bugs.*"

The most striking observation to emerge from the response was that as a program evolves in size it is vital to design it by splitting it into modules, so that developer does not need to understand all of it to make a small modification. Generally, refactoring to improve the design at different levels of granularity is crucial. This case study sheds light on the importance of refactoring at package-level of granularity and how it plays a crucial role in the quality and maintainability of the software. In future investigations, it might be possible to extend this work by learning from existing remodularization process and then recommending the right package for a given class taking into account the design quality (e.g., coupling, cohesion, and complexity). Future studies on remodularization topic can develop refactoring tool which can refactor software systems at different levels of granularity.

5.2.2 *Case Study 2. Refactoring to remove code smells.*

Fig. 6. Commit message stating the removal of duplicate code

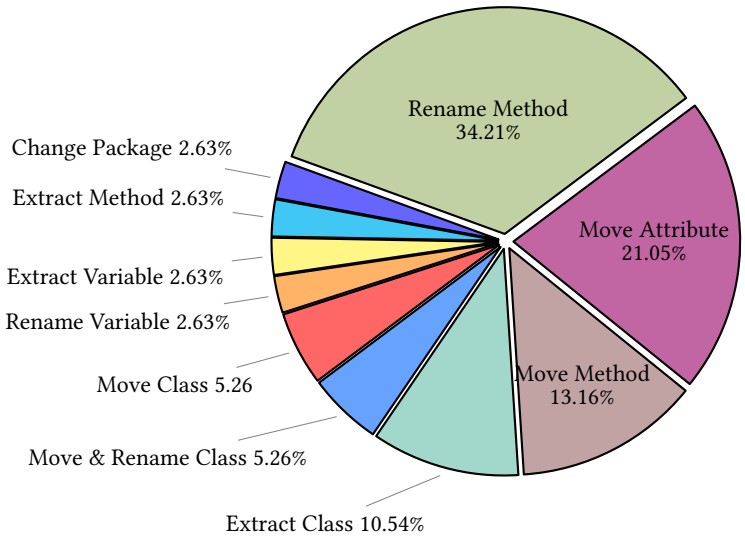


Fig. 7. Distribution of Refactoring Operations

This case study illustrates another developer's perception of refactoring which is mainly about code smell resolution. Figure 6 shows that the developer performed large-scale refactoring to eliminate duplicated code. Generally, code duplication belongs to the "Dispensible" code smell category, i.e., code fragments that are unneeded and whose absence would make the code cleaner and more efficient.

Figure 7 depicts the 38 refactoring operations performed in which the developer removed duplicated code. The developer performed 10 different types of refactorings associated with the commit message shown in Figure 6: *Rename Method, Move Attribute, Move Method, Extract Class, Move and Rename Class, Move Class, Rename Variable, Extract Variable, Extract Method*, and *Change Package*.

From the pie chart, it is clear that the majority of refactorings performed were *Rename Method and Move Attribute* with 34.21% and 21.05% respectively, followed by *Move Method* with 13.16% and *Extract Class* refactorings with 10.54%. Nearly 5% were *Move Class* and *Move and Rename Class* refactorings and only a small percentage of refactoring commits were *Change Package, Extract Method, Extract Variable*, and *Rename Variable*.

On further examination of the source code and the corresponding refactorings detected by the tool, we notice that there are a variety of cases in which the code fragments are considered duplicate. One case is when the same code structure is found in more than one place in the same class, and the other one is when the same code expression is written in two different and unrelated classes. The developer treated the former case by using *Extract Method* refactoring followed by the necessarily naming and moving operations and then invoked the code from both places. As for the latter case, the developer solved it by using *Extract Class* refactoring and the corresponding renaming and moving operations for the class and/or attribute that maintained the common functionalities. The developer also performed *Change Package* refactorings when removing code duplication as a complementary step of refactoring, which could indicate motivations outside of those described in the commit message.

It appears to us that composite refactorings have been performed for resolving this code smell. These activities help eliminate code duplication since merging duplicate code simplifies the design of the code. Additionally, these activities could help improving many code metrics such as the lines of code (LOC), the cyclomatic complexity (CC), and coupling between objects (CBO).

5.2.3 *Case Study 3. Refactoring to improve external quality Attributes.*

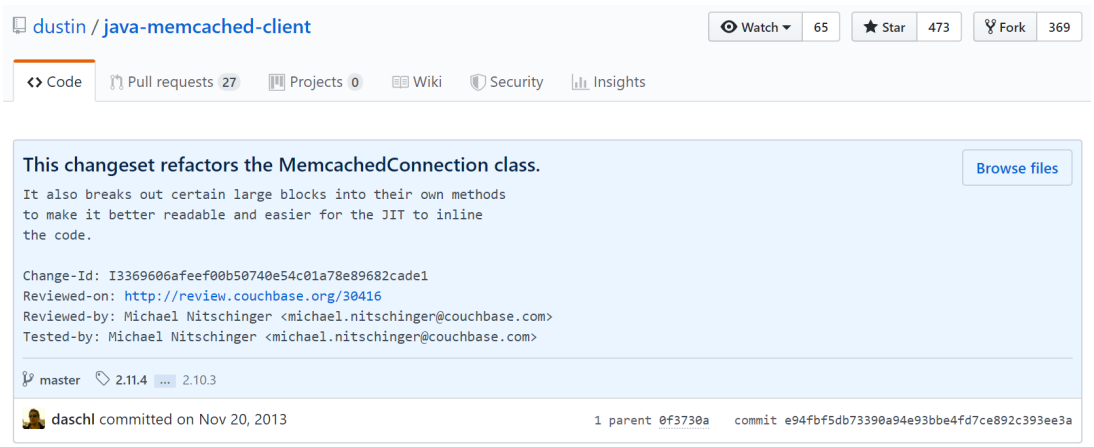


Fig. 8. Commit message stating the refactoring to improve code readability

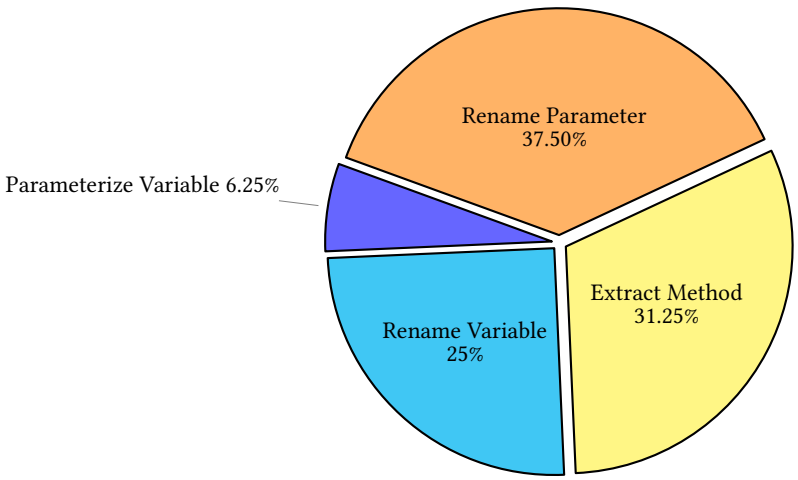


Fig. 9. Distribution of Refactoring Operations

This case study demonstrates another refactoring intention which is related to improving external quality attributes (i.e., indication of the enhancement of non-functional attributes such as readability and understandability of the source code). As shown in Figure 8, the developer stated that the purpose of performing this refactoring is to improve the readability of the source code by breaking large blocks of code into separate methods.

Figure 9 illustrates the breakdown of 16 refactoring operations related to readability associated with this commit message. It can be seen that *Rename Parameter* and *Extract Method* refactorings have the highest refactoring-related commits with 37.50% and 31.25%, respectively. *Rename Variable* is the third most performed refactoring with 25%, in front of *Parameterize Variable* refactorings at 6.25%. By analyzing the corresponding source code, it is clear that developer decomposed four methods for better readability, namely, `handleIO()`, `handleIO(sk SelectionKey)`, `handleReads(sk SelectionKey, qa MemcachedNode)`, and `attemptReconnects()`. The name could also change for a reason (e.g., when *Extract Method* is applied to a method, the method name and its parameters or variables also update as a result).

To improve code readability, developer used *Extract Method* refactorings as a treatment for this case study to reduce the length of the method body. Additionally, renaming operations were used to improve naming quality in the refactored code and reflect the actual purpose of the parameters and variables. Converting variables to parameters could also make the methods more readable and understandable. To develop a full picture of how to create readable code, future studies will be needed to focus on code readability guidelines or rules for developers.

5.2.4 *Case Study 4. Refactoring to add feature.*

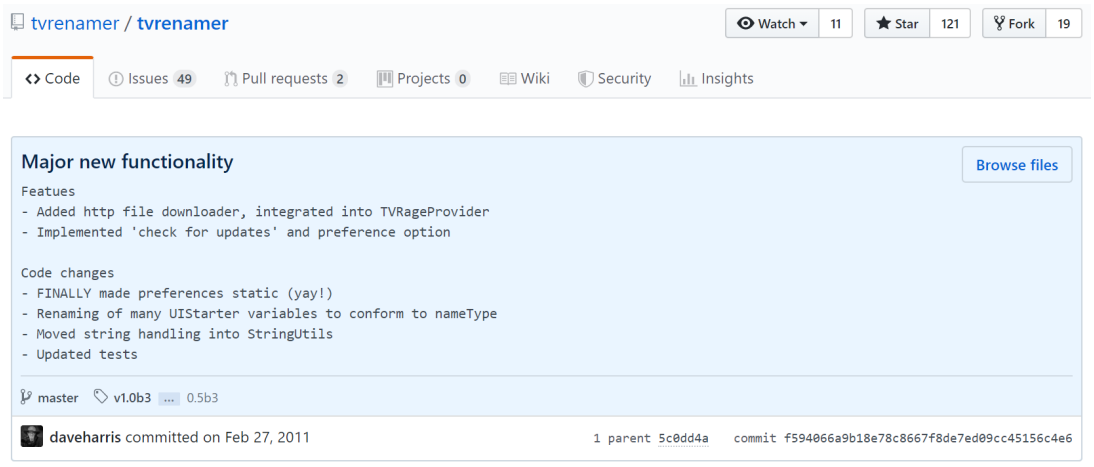


Fig. 10. Commit message stating the addition of a new functionality

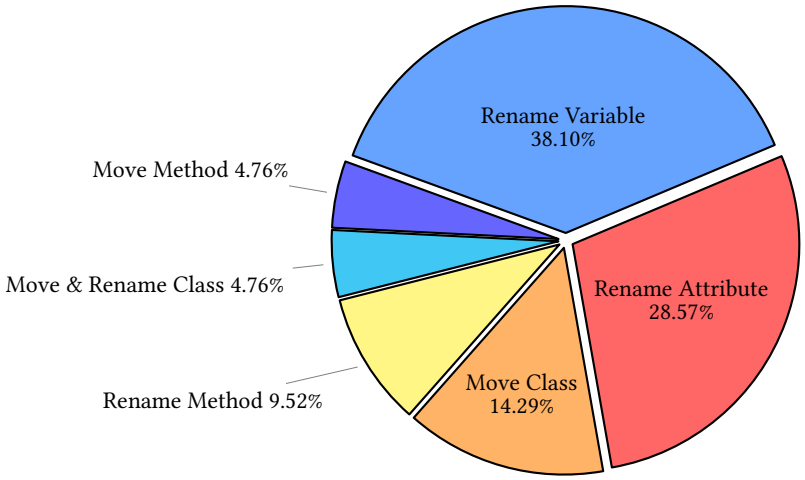


Fig. 11. Distribution of Refactoring Operations

This case study discusses another motivation of refactoring that is different than the traditional design improvement motivation. As shown in Figure 10, developers interleaved refactoring practices with other development-related tasks, i.e., adding feature. Specifically, the developer implemented two new functionalities (i.e., allow the user to download files, and "check for updates" and "preference option" features. Developers also performed other code changes which involved renaming, moving, etc.

Figure 11 portrays the 21 refactoring operations performed in which the developer added features and made other related code changes. With regards to the type of refactoring operations used to perform these implementations, the developer mainly performed moving and renaming related operations that are associated with code elements related to that implementation. Overall, *Rename Variable* and *Rename Attribute* constitute the main refactoring operations performed accounting for 38.10% and 28.57% respectively, followed by *Move Class* with 14.29% and *Rename Method* with 9.52% . The percentage of *Move Method* and *Move and Rename Class* refactorings, by contrast, made up a mere 4.76%.

Upon exploring the source code, it appears to us that the developer performed moving-related refactorings when adding features (e.g., update checker functionality and activate user preference option) to the system, and renaming-related operations have been performed for several enhancements related to the UI (e.g., renaming buttons, task bar, progress bar, etc). These observations may explain that adding feature is one type of development task that refactorings were interleaved with and the refactoring definition in practice seems to deviate from the rigorous academic definition of refactoring, i.e., refactoring to improve the design of the code.

5.2.5 *Case Study 5. Refactoring to fix bug.*

Fig. 12. Commit message stating the correction of user interface related bugs

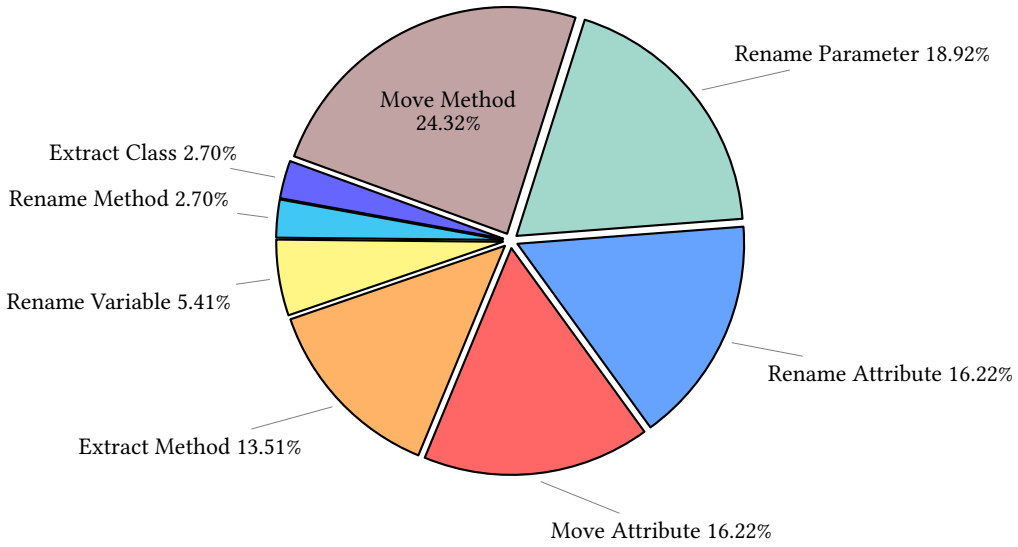


Fig. 13. Distribution of Refactoring Operations

This case study presents another refactoring intention, i.e., refactoring to fix bugs that differs from the academic definition of refactoring. It can be seen from the above commit message (Figure 12) that several UI-related bugs have been solved while performing refactorings. Similar to the commit in case study 4, the developer interleaved these changes with other types of refactoring.

The pie chart above shows 7 distinct refactoring operations performed that constituted 37 refactoring instances for bug fixing-related process. The type of refactorings involved in this activity are mainly focused on extracting, moving, and renaming-related operations.

From the graph above we can see that roughly a quarter of refactorings were *Move Method. Rename Parameter, Rename Attribute, and Move Attribute* constituted almost the same percentage with slight advantage to *Rename*

Parameter. Extract Method was comprised of 13,51%, whereas *Rename Variable, Rename Method*, and *Extract Class* combined just constituted under a fifth. The present results are significant in at least two major respects: (1) developers flossly refactor the code to reach a specific goal, i.e., fix bugs, and (2) developers did not separate refactoring techniques from bug fixing-related activities. Interleaving these activities may not guarantee behaviour preserving transformation as reported by [Fowler et al. 1999]. Developers are encouraged to frequently refactor the code to make finding and debugging bugs much easier. Fowler et al. pointed out that developers should stop refactoring if they notice a bug that needs to be fixed since mixing both tasks may lead to changing the behaviour of the system. Testing the impact of these changes is a topic beyond the scope of this paper, but it is an interesting research direction that we can take into account in the future.

Analyzing the distributions of refactoring operations in the case studies, and observing how they vary due to the context of refactoring and due to the difference between production and test files, has raised our curiosity about whether we can observe similar difference if we analyze distributions of refactorings across classification categories. In the next subsection, we define the following research question to investigate the frequency of refactorings, spit by target refactored element (production vs. test) per category.

5.3 RQ1.1: Do software developers perform different types of refactoring operations on test code and production code between categories?

Table 11. Refactoring frequency in production and test files in all projects combined

Refactoring	Internal QA		Code Smell		External QA		Functional		BugFix	
	Prod.	Test	Prod.	Test	Prod.	Test	Prod.	Test	Prod.	Test
Change Package	4626 (0.46%)	0 (0.0%)	12693 (0.56%)	0 (0.0%)	3760 (0.57%)	0 (0.0%)	5120 (0.50%)	0 (0.0%)	3541 (0.29%)	0 (0.0%)
Extract & Move Method	20822 (2.08%)	32 (2.50%)	8643 (0.38%)	30 (2.74%)	8369 (1.27%)	35 (2.22%)	27327 (2.67%)	111 (7.86%)	24576 (2.08%)	51 (2.84%)
Extract Class	13156 (1.31%)	13 (1.01%)	6785 (0.30%)	18 (1.64%)	6019 (0.91%)	68 (4.33%)	23625 (2.31%)	13 (0.92%)	19510 (1.65%)	21 (1.16%)
Extract Interface	1700 (0.17%)	0 (0.0%)	2522 (0.11%)	0 (0.0%)	1874 (0.28%)	0 (0.0%)	2104 (0.20%)	0 (0.0%)	3658 (0.30%)	2 (0.11%)
Extract Method	31357 (3.13%)	246 (19.19%)	18177 (0.80%)	52 (4.75%)	22772 (3.47%)	148 (9.42%)	40286 (3.94%)	218 (15.43%)	113416 (9.60%)	287 (15.98%)
Extract Subclass	1578 (0.16%)	0 (0.00%)	1109 (0.05%)	1 (0.09%)	1084 (0.16%)	0 (0.0%)	1171 (0.11%)	0 (0.0%)	2391 (0.20%)	9 (0.50%)
Extract Superclass	7262 (0.72%)	8 (0.62%)	7380 (0.32%)	4 (0.36%)	6917 (1.05%)	11 (0.70%)	10290 (1.00%)	13 (0.92%)	14939 (1.26%)	0 (0.0%)
Extract Variable	9922 (0.99%)	30 (2.34%)	8893 (0.39%)	40 (3.65%)	8233 (1.25%)	74 (4.71%)	15694 (1.53%)	43 (3.04%)	59323 (5.02%)	71 (3.95%)
Inline Method	6139 (0.61%)	22 (1.72%)	5100 (0.22%)	19 (1.73%)	3717 (0.56%)	14 (0.89%)	7219 (0.70%)	11 (0.77%)	12662 (1.07%)	9 (0.50%)
Inline Variable	3107 (0.31%)	4 (0.31%)	4290 (0.19%)	3 (0.27%)	1473 (0.22%)	12 (0.76%)	2443 (0.23%)	13 (0.92%)	10570 (0.89%)	10 (0.55%)
Move & Rename Attribute	171 (0.02%)	1 (0.08%)	60 (0.0%)	0 (0.0%)	318 (0.04%)	0 (0.0%)	204 (0.01%)	0 (0.0%)	120 (0.01%)	0 (0.0%)
Move & Rename Class	14426 (1.44%)	4 (0.31%)	12493 (0.55%)	14 (1.27%)	14958 (2.27%)	5 (0.31%)	16054 (1.57%)	7 (0.49%)	11192 (0.94%)	0 (0.0%)
Move Attribute	112542 (11.23%)	51 (3.98%)	43865 (1.92%)	61 (5.57%)	22716 (3.46%)	16 (1.01%)	45025 (4.41%)	33 (2.33%)	66400 (5.62%)	54 (3.00%)
Move Class	**213609 (21.32%)**	26 (2.03%)	**1202376 (52.73%)**	42 (3.83%)	**129816 (19.78%)**	32 (2.03%)	**175675 (17.21%)**	12 (0.84%)	94787 (8.02%)	41 (2.28%)
Move Method	61349 (6.12%)	120 (9.36%)	47268 (2.07%)	202 (18.46%)	21830 (3.32%)	168 (10.70%)	101096 (9.90%)	185 (13.10%)	87099 (7.37%)	89 (4.95%)
Move Source Folder	6219 (0.62%)	0 (0.0%)	4636 (0.20%)	0 (0.0%)	8087 (1.23%)	0 (0.0%)	9293 (0.91%)	0 (0.0%)	5906 (0.50%)	0 (0.0%)
Parameterize Variable	2595 (0.26%)	12 (0.94%)	1623 (0.07%)	2 (0.18%)	1572 (0.23%)	26 (1.65%)	3548 (0.34%)	6 (0.42%)	5474 (0.46%)	10 (0.55%)
Pull Up Attribute	8803 (0.88%)	52 (4.06%)	53171 (2.33%)	8 (0.73%)	8781 (1.33%)	40 (2.54%)	15023 (0.34%)	26 (1.84%)	29810 (2.52%)	69 (3.84%)
Pull Up Method	71906 (7.18%)	133 (10.37%)	26439 (1.17%)	39 (3.56%)	24539 (3.74%)	55 (3.50%)	31181 (1.47%)	39 (2.76%)	81997 (6.94%)	204 (11.36%)
Push Down Attribute	5186 (0.52%)	0 (0.0%)	5688 (0.25%)	5 (0.45%)	6476 (0.98%)	2 (0.12%)	4167 (3.05%)	0 (0.0%)	6778 (0.57%)	0 (0.0%)
Push Down Method	14215 (1.42%)	1 (0.08%)	11874 (0.52%)	8 (0.73%)	14689 (2.23%)	1 (0.06%)	9222 (0.90%)	(0.0%)	14581 (1.23%)	0 (0.0%)
Rename Attribute	68893 (6.88%)	43 (3.35%)	229670 (10.07%)	20 (1.82%)	112477 (17.14%)	67 (4.26%)	142835 (14.00%)	26 (1.84%)	109286 (9.25%)	29 (1.61%)
Rename Class	28254 (2.82%)	16 (1.25%)	56894 (2.50%)	9 (0.82%)	24241 (3.69%)	15 (0.95%)	27010 (2.64%)	18 (1.27%)	37555 (3.17%)	10 (0.55%)
Rename Method	90809 (9.06%)	**314 (24.49%)**	393385 (17.25%)	**393 (35.92%)**	97245 (14.82%)	**461** (29.36%)	120188 (11.77%)	**371 (26.27%)**	125897 (10.65%)	**532 (29.63%)**
Rename Parameter	89514 (8.93%)	12 (0.94%)	41900 (1.84%)	16 (1.46%)	41483 (6.32%)	17 (1.08%)	72275 (7.08%)	14 (0.99%)	**138436 (11.72%)**	16 (0.89%)
Rename Variable	104621 (10.44%)	127 (9.91%)	70507 (3.09%)	100 (9.14%)	60788 (9.26%)	286 (18.21%)	108056 (10.59%)	211 (14.94%)	95289 (8.06%)	249 (13.87%)
Replace Attribute	356 (0.04%)	5 (0.39%)	207 (0.01%)	0 (0.0%)	32 (0.004%)	0 (0.0%)	212 (0.02%)	0 (0.0%)	102 (0.008%)	1 (0.05%)
Replace Variable with Attribute	8703 (0.87%)	10 (0.78%)	2546 (0.11%)	8 (0.73%)	1813 (0.27%)	17 (1.08%)	3931 (0.38%)	42 (2.97%)	5889 (0.49%)	31 (1.72%)

In Table 11, we show the volume of operations for each refactoring operation applied to the refactored test and production files grouped by the classification category associated with the file. Values in bold indicate the most common applied refactoring operation – *Move Class* and *Rename Parameter* for production files, and *Rename Method* for test files.

Concerning production file-related refactoring motivations, the top most refactoring operations performed across all refactoring motivations is *Move Class* refactoring, except for BugFix in which *Rename Attribute* is the highest performed refactoring. In the case of internal quality attribute-related motivations, developers performed *Move Class* refactoring

to move the relevant classes to the right package if there are many dependencies for the class between two packages. This could eliminate undesired dependencies between modules. Another possibility for the reason to perform such refactoring is to introduce a sub-package and move a group of related classes to a new subpackage. With respect to code smell resolution motivation, developers eliminate a redundant sub-package and nesting level in the package structure when performing *Move Class* refactoring operations. With regards to external quality attribute-related motivation, developers can target improving the understandability of the code by repackaging and moving the classes between these packages. Hence, the structure of the code becomes more understandable. Developers could also maintain code compatibility by moving a class back to its original package to maintain backward compatibility. For feature addition or modification, *Move Class* refactoring is performed when adding new or modifying the implemented features. This could be done by moving the class to appropriate containers or moving a class to a package that is more functionally or conceptually relevant. Lastly, for bug fixing-related motivations, developers mainly improve parameter and method names; they rename a parameter or method to better represent its purpose and to enforce naming consistency and to conform to the project's naming conventions. Developers need to change the semantics of the code to improve the readability of the code. For test files-related refactoring motivations, the most frequently applied refactoring is *Rename Method*. This can be explained by the fact that test methods are the fundamental elements in a test suite. Test methods are utilized to test the production source code; hence, the high occurrence of method based refactorings in unit test files. The observed difference in the distribution of refactorings in production/test files between our study and the related work [Tsantalis et al. 2013] is also due to the size (number of projects) effect of the two groups under comparison.

> *Summary.* Our findings are aligned with the previous work [Tsantalis et al. 2013]. 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.

5.4 RQ2: What patterns do developers use to describe their refactoring activities?

Upon a closer inspection of these refactoring patterns, we have made several observations: we noticed that developers document refactoring activities at different levels of granularity, e.g., package, class, and method level. Furthermore, we observe that developers state the motivation behind refactoring, and some of these patterns are not restricted only to fixing code smells, as in the original definition of refactoring in Fowler's book, i.e., improving the structure of the code. For instance, developers tend often to improve certain non-functional attributes such as the readability and testability of the source code. Additionally, developers occasionally apply the "Don't Repeat Yourself" principle by removing excessive code duplication. A few patterns indicated that developers refactor the code to improve internal quality attributes such as inheritance, polymorphism, and abstraction. We also noticed the application of a single responsibility principle which is meant to improve the cohesion and coupling of the class when developers explicitly mentioned a few patterns related to dependency removal.

Further, we observe that developers tend to report the executed refactoring operations by explicitly using terms from Fowler's taxonomy; terms such as *inline class/method*, *Extract Class/Superclass/Method* or *Push Up Field/Method* and *Push Down Field/Method*.

Table 12. Patterns detected across all classes. Patterns whose occurrence in refactoring commits is significantly higher than non-refactoring commits (i.e., $p < 0.05$) are in **bold**

Potential Candidate Self-Affirmed Refactoring Patterns				
(1) Add*	(47) Chang*	(93) **Cleaned out**	(139) **CleanUp**	(185) **CleaningUp**
(2) **Clean* up**	(48) **Clean-up**	(94) Creat*	(140) **Decompos***	(186) **Encapsulat***
(3) **Enhanc***	(49) **Extend***	(95) **Extract***	(141) **Factor* Out**	(187) Fix*
(4) **Improv***	(50) **Inlin***	(96) **Introduc***	(142) Merg*	(188) **Migrat***
(5) Modif*	(51) **Modulariz***	(97) Mov*	(143) **Organiz***	(189) **Polish***
(6) **Pull* Up**	(52) **PullUp**	(98) **Push Down**	(144) PushDown	(190) **Repackag***
(7) **Re packag***	(53) **Re-packag***	(99) **Redesign***	(145) **Re-design***	(191) **Reduc***
(8) **Refactor***	(54) **Refin***	(100) Reformat*	(146) Remov*	(192) **Renam***
(9) Reorder*	(55) **Reorganiz***	(101) **Re-organiz***	(147) Repackag*	(193) **Replac***
(10) **Restructur***	(56) **Rework***	(102) **Rewrit***	(148) **Re-writ***	(194) **Rewrot***
(11) **Simplif***	(57) **Split***	(103) **TidyUp**	(149) **Tid*-up**	(195) **Tid* Up**
(12) A bit of refactor*	(58) Basic code clean up	(104) Chang* code style	(150) Ease maintenance moving forward	(196) Replace it with
(13) **Big refactor***	(59) **Big cleanup**	(105) Clean* up the code style	(151) Ease of code maintenance	(197) Extracted out code
(14) Better factored code	(60) Cleanliness	(106) Code style improv*	(152) **Easier to maintain**	(198) Reduced code dependency
(15) **Code refactor***	(61) Clean* up unnecessary code	(107) Code style unification	(153) Simplify future maintenance	(199) Pushed down dependencies
(16) Code has been refactored extensively	(62) Cleanup formatting	(108) Fix code style	(154) Improve quality	(200) **Simplify the code**
(17) **Extensive refactor***	(63) **Code clean**	(109) Improv* code style	(155) Improvement of code quality	(201) Less code
(18) Refactoring towards nicer name analysis	(64) Code cleanup	(110) Minor adjustments to code style	(156) Improved style and code quality	**(202) Change package**
(19) Heavily refactored code	(65) Code cleanliness	(111) Modifications to code style	(157) Maintain quality	(203) Cosmetic changes
(20) **Heavy refactor***	(66) **Code clean up**	(112) Lots of modifications to code style	(158) More quality cleanup	(204) Full customization
(21) **Little refactor***	(67) **Massive cleanup**	(113) Makes the code easier to program	(159) **Better name**	(205) **Structure change**
(22) **Lot of refactor***	(68) Minor cleaning of the code	(114) **Code review**	(160) Chang* name	(206) Module structure change
(23) **Major refactor***	(69) **Housekeeping**	(115) Code rewrite	(161) **Chang* the name**	(207) Module organization structure change
(24) **Massive refactor***	(70) Major rewrite and simplification	(116) Code cosmetic	(162) Chang* the package name	(208) Polishing code
(25) **Huge refactor***	(71) **Improv* consistency**	(117) Code revision	(163) **Chang* method name**	(209) **Improv* code quality**
(26) **Minor refactor***	(72) Some fix* and optimization	(118) Code optimization	(164) Chang* method parameter names for consistency	(210) **Chang* package structure**
(27) **More refactor***	(73) Minors fix* and tweak	(119) Code reformatting	(165) Enables condensed naming	(211) Fix quality flaws
(28) Refactor* code	(74) Fix* annoying typo	(120) Code organization	(166) Fix* naming convention	(212) **Get rid of**
(29) Refactor* existing schema	(75) Fix* some formatting	(156) Code rearrangement	(167) **Fix nam***	(213) **Chang* design**
(30) **Refactor out**	(76) Fix* formatting	(122) Code formatting	(168) **Typo in method name**	(214) Improv* naming consistency
(31) **Small refactor***	(77) Modifications to make it work better	(123) Code polishing	(169) Maintain convention	(215) Remov* unused classes
(32) **Some refactor***	(78) Make it simpler to extend	(124) **Code simplification**	(170) Maintain naming consistency	(216) Minor simplification
(33) Tactical refactor*	(79) Fix* Regression	(125) Code adjustment	(171) Major renam*	(217) Fix* quality issue
(34) Moved a lot of stuff	(80) Remov* the useless	(126) **Code improvement**	(172) **Name cleanup**	(218) **Naming improvement**
(35) Fix this tidily	(81) Remov* unneeded variables	(127) Code style	(173) **Renam* for consistency**	(219) Packaging improvement
(36) **Further tidying**	(82) Remov* unneeded code	(128) Code restructur*	(174) Renam* according to java naming conventions	(220) **Structural chang***
(37) Tidied up and tweaked	(83) Remov* redundant	(129) Code beautifying	(175) Renam* classes for consistency	(221) Hierarchy clean*
(38) Tidied up some code	(84) Remov* dependency	(130) **Code tidying**	(176) **Renam* package**	(222) Hierarchy reduction
(39) **Restructur* package**	(85) Remov* unused dependencies	(131) Code enhancement	(177) Resolv* naming inconsistency	(223) Enhanc* architecture
(40) **Restructur* code**	(86) Remov* unused	(132) Code reshuffling	(178) Simpler name	(224) Architecture enhanc*
(41) Aggregat* code	(87) Remov* unnecessary else blocks	(133) Code modification	(179) Us* appropriate variable names	(225) Trim unneeded code
(42) Beautif* code	(88) Remov* needless loop	(134) Code unification	(180) Us* more consistent variable names	(226) Remov* unneeded code
(43) Tidy code	(89) Maintain consistency	(135) Code quality	(181) Neaten up	(227) More consistent formatting
(44) Beautify*	(90) **Customiz***	(136) Make code clearer	(182) Moved more code out of	(228) More easily extended
(45) Moved all integration code to separate package	(91) Improve code clarity	(137) Code clarity	(183) Fix bad merge	(229) Makes it more extensible-friendly
(46) **Improve code**	(92) **Simplify code**	(138) Clean* code	(184) **Cleanup code**	(230) Clean* up code

The generic nature of some of these patterns was a critical observation that we encountered, i.e., many of these patterns are context specific and can be subject to many interpretations, depending on the meaning the developer is trying to convey. For instance, the pattern *fixed a problem* is descriptive of any anomaly developer encountered and it can be either functional or non-functional. Since in our study, we are interested in textual patterns related to refactoring, we decided to filter this list down by reporting patterns whose frequency in commit messages containing refactoring is significantly higher than in messages of commits without refactoring. The rationale behind this idea is to identify patterns that are repeatedly used in the context of refactoring, and less often in other contexts. Since the patterns were

Table 13. Patterns detected by class. Patterns whose occurrence in refactoring commits is significantly higher than non-refactoring commits (i.e., $p < 0.05$) are in **bold**

Potential Candidate Self-Affirmed Refactoring Patterns per Refactoring Motivation				
BugFix	**Code Smell**	**External**	**Functional**	**Internal**
Minor fixes	**Avoid code duplication**	Reusable structure	Add* feature	**Decoupling**
Bug* fix*	**Avoid duplicate code**	Improv* code reuse	Add new feature	Enhance loose coupling
Fix* bug*	Avoid redundant method	Add* flexibility	Added a bunch of features	Reduced coupling
Bug hunting	**Code duplication removed**	Increased flexibility	**New module**	Reduce coupling and scope of responsibility
Correction of bug	**Delet* duplicate code**	**More flexibility**	**Fix some GUI**	Prevent the tight coupling
Improv* error handling	Remove unnecessary else blocks	Provide flexibility	Added interesting feature	Reduced the code size
Fix further thread safety issues	**Eliminate duplicate code**	A bit more readable	Added more features	Complexity has been reduced
Fixed major bug	Fix for duplicate method	**Better readability**	Adding features to support	**Reduce complexity**
Fix numerous bug	Filter duplicate	Better readability and testability	Adding new features	Reduced greatly the complexity
Fix several bug	Joining duplicate code	Code readability optimization	Addition of a new feature	Removed unneeded complexity
Fixed a minor bug	Reduce a ton of code duplication	Easier readability	Feature added	Removes much of the complexity
Fixed a tricky bug	**Reduce code duplication**	**Improve readability**	Implement one of the batch features	Add inheritance
Fix* small bug	Reduced code repetition	**Increase readability**	Implementation of feature	Added support to the inheritance
Fixed nasty bug	**Refactored duplicate code**	Make it better readable	Implemented the experimental feature	Avoid using inheritance and using composition instead
Fix* some bug*	Clear up a small design flaw	Make it more readable	Introduced possibility to erase features	Better support for specification inheritance
Fixed some minor bugs	TemporalField has been refactored	Readability enhancement	New feature	**Change* inheritance**
bugfix*	Remove commented out code	Readability and supportability improvement	Remove the default feature	Extend the generated classes using inheritance
Fix* typo*	Removed a lot of code duplication	**Readability improvements**	Removed incomplete features	Improved support for inheritance
Fix* broken	**Remov* code duplication**	Reformatted for readability	Renamed many features	Perform deep inheritance
Fix* incorrect	Remove some code duplication	Simplify readability	Renamed some of the features for consistency	Remove inheritance
Fix* issue*	Removed the big code duplication	Improve* testability	Small feature addition	Inheritance management
Fix* several issue*	Removed some dead and duplicate code	Update the performance	Support of optional feature	Loosened the module dependency
Fix* concurrency issue* with	**Remov* duplicate code**	Add* performance	Supporting for derived features	Prevents circular inheritance
Fixes several issues	Resolved duplicate code	Scalability improvement	Added functionality	Avoid using inheritance
Solved some minor bugs	Sort out horrendous code duplication	Better performance	Added functionality for merge	Add composition
Working on a bug	Remove duplicated field	Huge performance improvement	Adding new functionality	Composition better than inheritance
Get rid of	Remov* dead code	**Improv* performance**	Adds two new pieces of functionality to	Us* composition
A bit of a simple solution to the issue	Remove some dead-code	**More manageable**	Consolidate common functionality	Separates concerns
A fix to the issue	Removed all dead code	**More efficient***	Development of this functionality	Better handling of polymorphism
Fix a couple of issue	Removed apparently dead code	Make it reusable for other	Export functionality	Makes polymorphism easier
Issue management	This is a bit of dead code	Increase efficiency	Extend functionality of	**Better encapsulation**
Fix* minor issue	Removed more dead code	Verify correctness	Extract common functionality	Better encapsulation and less dependencies
Correct issue	Fix* code smell	Massive performance improvement	Functionality added	Pushed down dependencies
Additional fixes	Fix* some code smell	Increase performance	House common functionality	**Remov* dependency**
Resolv* problem	Remov* some 'code smells'	Largely resolved performance issues	Improved functionality	Split out each module into
Correct* test failure*	Update data classes	Lots of performance improvement	**Move functionality**	
Fix* all failed test*	Remove useless class	Measuring the performance	Moved shared functionality	
Fix* compile failure	Removed obviously unused/faulty code	Improv* stability	Feature/code improvements	
A fix for the errors	Lots of modifications to code style	Usability improvements	Pulling-up common functionality	
Better error handling	Antipattern bad for performances	Noticeable performance improvement	Push more functionality	
Better error message handling	Killed really old comments	Optimizing the speed	Re-implemented missing functions	
Cleanup error message	Less long methods	Performance boost	Refactored functionality	
Error fix*	Removed some unnecessary fields	**Performance enhancement**	Refactoring existing functionality	
Fixed wrong		Performance improvement	Add functionality to	
Fix* error*		Performance much improved	Remov* function*	
Fix* some error*		Performance optimization	Merging its functionality with	
Fix small error		Performance speed-up	Remove* unnecessary function*	
Fix some errors		Refactor performance test	Reworked functionality	
Fix compile error		Renamed performance test	Removing obsolete functionality	
Fix test error		Speed up performance	Replicating existing functionality with	
Fixed more compilation errors		Backward compatible with	Split out the GUI function	
Fixed some compile errors		Fix backward compatibility	Add cosmetic changes	
Fixes all build errors		Fixing migration compatibility	Add* support	
Fixed Failing tests		Fully compatible with	Implement* new architecture	
Handle		Keep backwards compatible	Update	
Handling error*		Maintain compatibility	Additional changes for	
Error* fix*		Make it compatible with	UI layout has changed	
Tweaking error handling		**More compatible**	GUI: Small changes	
Various fix*		Should be backward-compatible	New UI layout	
Fix* problem*		Retains backward compatibility	**UI changes**	
Got rid of deprecated code		Stay compatible with	UI enhancements	
Delet* deprecated code		Added some robustness		
Remov* deprecated code		Improve robustness		
		Improve usability		
		Robustness improvement		
		To be more robust		
		Better understanding		
		Bit to be easier to understand		
		Easier to understand		
		Increases understandability		
		Adding checks on accessibility		
		Easier accessible and evolvable		
		Make class more extendable		
		Allow extensibility		
		For future extensibility		
		Improved modularity and extensibility		
		Increase testability		
		More testable design		
		Accuracy improvement		

extracted from 111,884 messages of commits containing refactoring (we call them refactoring commits), we need to build another corpus of messages from commits that do not contain refactorings (we call them non-refactoring commits). As we plan on comparing the frequency of keywords between the two corpora, i.e., refactoring and non-refactoring commit messages, it is important to adequately choose the non-refactoring messages to ensure fairness. To do so, we follow the following heuristics: we randomly select a statistically significant sample of commits (confidence level of 95%), 1) chosen from the same set of 800 projects that issued the refactoring commits; 2) whose authors are from the same authors of the refactoring commits; 3) whose timestamps are in the same interval of refactoring commits timestamps; 4) and finally, the average length of commit messages are approximately close (118 for refactoring commits, and 120 for non-refactoring commits).

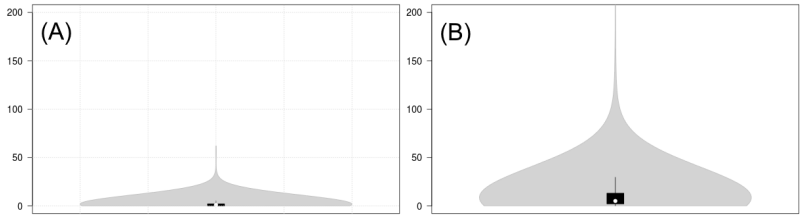


Fig. 14. Violin plots representing the occurrence of *refactor* keyword in (A) non-refactoring corpus vs. (B) refactoring corpus

Once the set of non-refactoring commit messages constructed, for each keyword, we calculate its occurrence per project for both corpora. This generates an vector of 800 occurrences per corpus. Each vector dimension contains a positive number representing the keyword occurrence for a project, and zero otherwise. Figure 14 illustrates occurrences violin plots of the keyword *refactor* in both corpora. While it is observed in Figure 14 that the occurrence of *refactor* in refactoring commits is higher, we need a statistical test to prove it. So, we perform such comparison using the Mann-Whitney U test, a non-parametric test that checks continuous or ordinal data for a significant difference between two independent groups. This applies to our case, since the commits, in the first group, are independent from commits in the second group. We formulate the comparison of each keyword occurrence corpora by defining the alternate hypothesis as follows:

Hypothesis 1. *The occurrence vector of refactoring commits is strictly higher than the occurrence vector of non-refactoring commits.*

And so, the null hypothesis is defined as follows:

Null Hypothesis 1. *The occurrence vector of non-refactoring commits is equal or smaller than the occurrence vector of refactoring commits.*

We start with generating occurrence vectors for each keyword, then we perform the statistical test for each pair of vector. We report our findings in Table 12 and 13 where keywords in **bold** are the ones rejecting the null hypothesis (i.e., $p < 0.05$).

With the analysis of these tables results, we observe the following:

- While previous studies have been relying on the detection of refactoring activity in software artifacts using the keyword "*refactor**" [Kim et al. 2014; Murphy-Hill et al. 2012; Ratzinger 2007; Ratzinger et al. 2008; Stroggylos and

Table 14. Top generic refacoring patterns

Patterns			
Refactor* (89.00%)	Renam* (83.63%)	Improv* (78.75%)	CleanUp (67.38%)
Replac* (66.88%)	Introduc (53.00%)	Extend (52.63%)	Simplif (52.50%)
Extract (49.00%)	Added support (47.38%)	Split (45.50%)	Reduc* (45.00%)
Chang* name (44.88%)	Migrat (32.88%)	Enhanc (32.63%)	Organiz* (32.25%)
Rework (27.25%)	Rewrit* (27.25%)	Code clean* (25.63%)	Remov* dependency (25.00%)

Spinellis 2007], our findings demonstrate that developers use a variety of keywords to describe their refactoring activities. For instance, keywords such as *clean up*, *repackage*, *restructure*, *re-design*, and *modularize* has been used without the mention of the *refactoring* keyword, to imply the existence of refactorings in the committed code. While these keywords are not exclusive to refactoring, and could be also used for general usage, their existence in commits containing refactoring operations has been more significant (i.e., $p < 0.05$), which qualifies them to be close synonyms to *refactoring*. Table 14 enumerates the top-20 keywords, sorted by the percentage of projects they were located in.

- We notice that developers document refactoring activities at different levels of granularity, *e.g.*, package, class, and method level. We also observe that developers occasionally state the motivation behind refactoring, which is not restricted only to fixing code smells, as in the original definition of refactoring in the Fowler's book [Fowler et al. 1999], and so, this supports rationale behind our classification in the first research question.
- Furthermore, our classification has revealed the existence of patterns that are used in specific categories (motivations). For instance, the traditional code smell category is mainly populated with keywords related to removing duplicate code. Interestingly, all patterns whose existence in refactoring commit messages is statistically significant, were related to duplicate code deletion. Although patterns related to removing code smells exist, e.g., *Clear up a small design flaw* or *fix code smell* or *Antipattern bad for performances*, these patterns occurrence was not large enough to reject the null hypothesis. Nevertheless, Table 15 contains a summary of category-specific patterns that we manually identified.
- Developers occasionally mention the refactoring operation(s) they perform. The Mann-Whitney test accepted the alternate hypothesis for all patterns linked to refactoring operations i.e., *Pull Up*, *Push Down*, *Inline*, *Extract*, *Rename*, *Encapsulate*, *Split*, *Extend*, except for the famous *move*. Unlike code smell patterns, *move* do exist in 787 projects (98.37%, forth most used keyword, after respectively *Fix*, *Add*, and *Merge*) and it is heavily used by both refactoring and non-refactoring commit messages.
- Similarly, to move, keywords like *merge*, *reformat*, *remove redundant*, *performance improvement*, *code style*, were popular across many projects, and typically invoked by both refactoring and non-refactoring commits. So, although they do serve in documenting refactoring activities, their generic nature makes them also used in several other contexts. For example, merge is typically used when developers combine classes or methods, as well as describing the resolution of merge conflicts. Similarly, performance improvement is not restricted to non-functional changes, as several performance optimization techniques and genetic improvements are not necessarily linked to refactoring.

Table 15. Summary of refactoring patterns, clustered by refactoring related categories

Internal	External	Code Smell
Inheritance	Functionality	Duplicate Code
Abstraction	Performance	Dead Code
Complexity	Compatibility	Data Class
Composition	Readability	Long Method
Coupling	Stability	Switch Statement
Encapsulation	Usability	Lazy Class
Design Size	Flexibility	Too Many Parameters
Polymorphism	Extensibility	Primitive Obsession
Cohesion	Efficiency	Feature Envy
Messaging	Accuracy	Blob Class
Concern Separation	Accessibility	Blob Operation
Dependency	Robustness	Redundancy
	Testability	Useless class
	Correctness	Code style
	Scalability	Antipattern
	Configurability	Design Flaw
	Simplicity	Code Smell
	Reusability	Temporary Field
	Reliability	Old Comment
	Modularity	
	Maintainability	
	Traceability	
	Interoperability	
	Fault-tolerance	
	Repeatability	
	Understandability	
	Effectiveness	
	Productivity	
	Modifiability	
	Reproducibility	
	Adaptability	
	Manageability	

Summary 1. Developers tend to use a variety of textual patterns to document their refactoring activities, besides '*refactor*', such as '*re-package*', '*redesign*', '*reorganize*', and '*polish*'.
Summary 2. These patterns can be either (1) generic, providing high-level description of the refactoring, *e.g.,* 'Clean up unnecessary code', 'Ease maintenance moving forward', or (2) specific by explicitly mentioning the rationale behind the refactoring, *e.g.,* 'reduce the code coupling' (internal), 'improving code *readability*' (external), and 'fix *long method*' (code smell).
Summary 3. Developers occasionally express their refactoring strategy. We detected several refactoring operations, known from the refactoring catalog, such as *'extract method'*, *'extract class'*, and *'extract interface'*.

The extraction of these patterns raised our curiosity about the extent to which they can represent an alternative to the keyword *refactor*, being the de facto keyword to document refactoring activities. Figure 15 reveals examples of these patterns. In the next subsection, we challenge the hypothesis raised by [Murphy-Hill et al. 2008] about whether developers use a specific pattern, i.e., *"refactor"* when describing their refactoring activities. We quantify the messages with the label *"refactor"* and without to compare between them.

Fig. 15. Sample of SAR Patterns Identified in this work

5.5 RQ2.1: Do Commits containing the label *Refactor* indicate more refactoring activity than those without the label?

[Murphy-Hill et al. 2008] proposed several hypotheses related to four methods that gather refactoring data and outlined experiments for testing those hypotheses. One of these methods concerns mining the commit log. [Murphy-Hill et al. 2008] hypothesize that commits labeled with the keyword *"refactor"* do not indicate more refactoring instances than unlabeled commits. In an empirical context, we test this hypothesis in two steps. In the first steps, we used the keyword *"refactor"*, exactly as dictated by the authors. Thereafter, we quantified the proportion of commits including the searched label across all the considered projects in our benchmark. In the second step, we re-tested the hypothesis using the subset of 230 SAR patterns, whose occurrence in refactoring commits was found to be significant with respect to non-refactoring commits. We counted the percentage of commits containing any of our SAR labels. The result of the two rounds resides in a strict set of commits containing the label *"refactor"*, which is included in a larger set containing all patterns, and finally a remaining set of commits which does not contain any patterns. For each of the sets, we count the number of refactoring operations identified in the commits. Then, we break down the set per operation type.

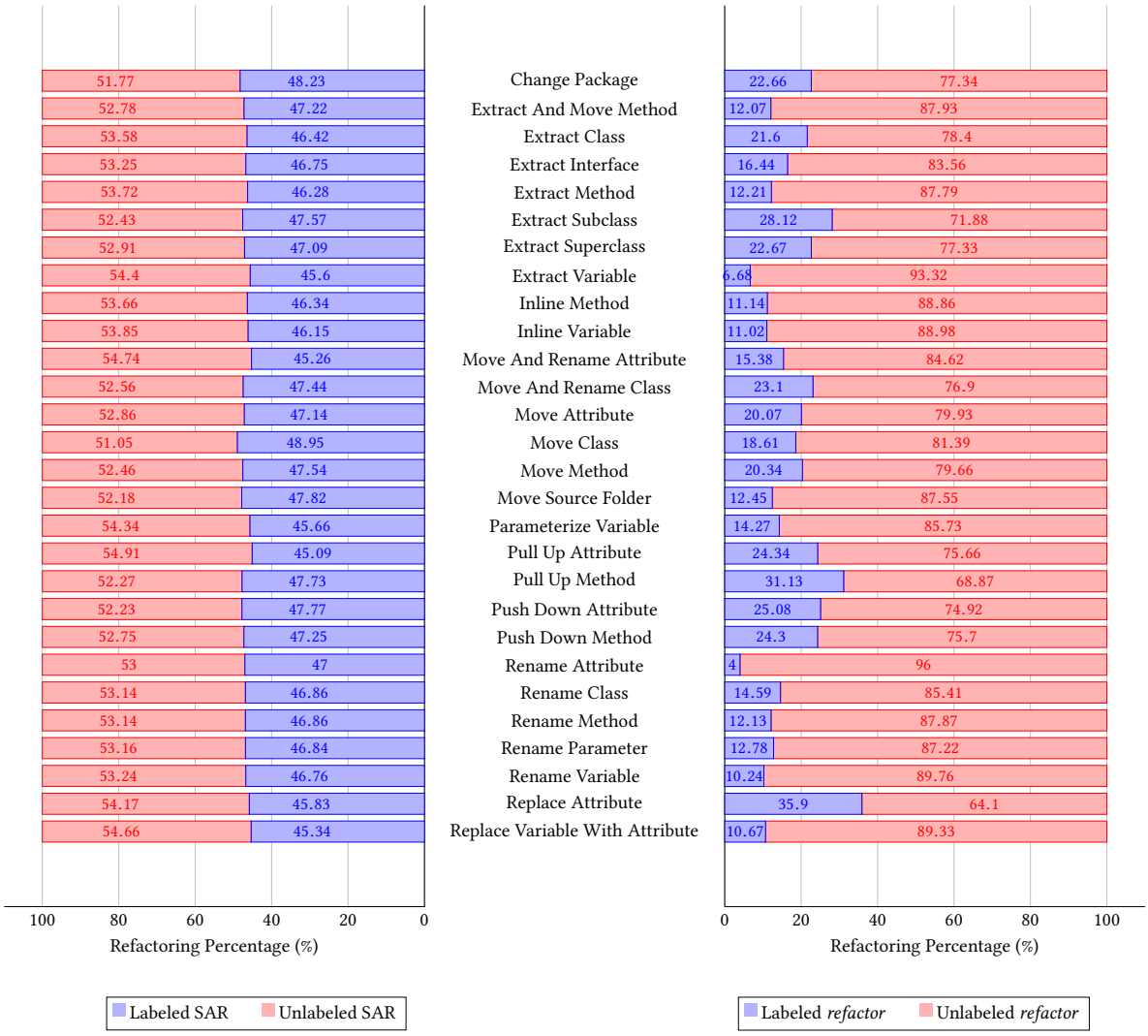


Fig. 16. Distribution of Refactoring Operations for Commits Labeled and Unlabeled SAR (left side) and Commits Labeled and Unlabeled *refactor* (right side)

In order to compare the quantity of refactorings identified for each set, i.e., labeled and unlabeled commits with the keyword *"refactor"*, along with labeled and unlabeled commits with SAR patterns. We used the Wilcoxon test, as suggested by [Murphy-Hill et al. 2008] for the purpose of testing the hypothesis. We then applied the non-parametric Wilcoxon rank-sum test to estimate the significance of differences between the numbers of the sets. The choice of Wilcoxon rank-sum test is motivated by the independence of sets from each other (the occurrence of *refactor* is independent from the occurrence of the remaining patterns).

Figure 16 shows the distribution of refactorings in labeled and unlabeled commits with SAR patterns (group 1 on the left) and labeled and unlabeled commits with the keyword refactor (group 2 on the right). The first observation we

can draw is that *"Replace Attribute"* stands as most labeled refactoring with a percentage of 35.9% for group 2, while the difference between operations percentages, in group 1, is not significant, with *"Move Class"* having the highest percentage of 48.95%. Another observation is that *"Pull Up Attribute"* turns out to be the most unlabeled refactoring with a score of 54.91% for group 1, whereas *"Rename Attribute"* tends to be the most unlabeled refactoring for group 2. This results is consistent with one of the previous studies stating that renames are rarely labeled, as they detected explicit documentation of renames in less than 1% of their dataset [Arnaoudova et al. 2014a]. For both tests, we notice that developers tend to label more refactorings applied to code elements with higher granularity level, i.e., at the package level. Conversely, refactorings that are implemented at method level and at attribute level tend to have the lowest percentage with commits labeled *"refactor"*. That sheds light on the variety of ways to express refactorings, which depend on the levels of granularity. While it is difficult to find a rationale for such observation, it seems that, making changes at the package and class level requires more exposure to the code design, in comparison with identifier or parameter changes, besides being less frequently to occur, and so when it happens, it is most likely to be documented.

By comparing the different commits that are labeled and unlabeled with SAR patterns, we observe a significant number of labeled refactoring commits for each refactoring operation supported by the tool Refactoring Miner (p-value = 0.0005). This implies that there is a strong trend of developers in using these phrases in refactoring commits. The results for commits labeled and unlabeled *"refactor"* with a p-value = 0.0005 engender an opposite observation, which corroborate the expected outcome of Murphy-Hill et al.'s hypothesis. Thus, the use of *"refactor"* is not a great indication of refactoring activities. The difference between the two tests indicates the usefulness of the list of SAR patterns that we identified.

It is to note that we did not perform any correspondence between the mentioned patterns and the corresponding refactoring operation(s). In other terms, if an operation is explicitly mentioned in a commit message, we have not checked whether it was among the applied refactoring at the source code level. We opted for such verification to be outside of the scope of the current study, while it would be an interesting direction we can consider in our future investigations.

> *Summary.* In consistency with the previous findings of [Murphy-Hill et al. 2008], our findings confirm that developers do not exclusively rely on the pattern *"refactor"* to describe refactoring activities. However, we found that developers do document their refactoring activities in commit messages with a variety of patterns that we identified in this study.

6 RESEARCH DISCUSSIONS

In this section, we want to further discuss our findings and outline their implications on future research directions in refactoring.

Developer's motivation behind Refactoring. One of main findings show that developers are not only driven by design improvement and code smell removal when taking decisions about refactoring. According to our RQ1 findings, fixing bugs, and feature implementation play a major role in triggering various refactoring activities. Traditional refactoring tools are still leading their refactoring effort based on how it is needed to cope with design antipatterns, which is acceptable to the extent where it is indeed the developer's intention, otherwise, they have not been designed or tested in different circumstances. So, an interesting future direction is to study how we can augment existing refactoring tools to better frame the developer's perception of refactoring, and then their corresponding objectives to achieve

(reducing coupling, improve code readability, renaming to remove ambiguity etc.). This will automatically induce the search for more adequate refactoring operations, to achieve each objective.

Refactoring Support. Classifying refactoring commits by message is an important activity because it allows us to contextualize these refactoring activities with information about the development activities that led to them. This contextualization is critical and will augment our ability to study the reasoning behind decisions to apply different types of refactoring. This will lead to better support for informing developers of when to apply a refactoring and what refactoring to apply. For example, recent studies try to understand how the development context which motivated a rename refactoring affects the way the words in a name changes when the refactoring is applied [Peruma et al. 2018, 2019b] for the purpose of modeling, more formally, how names evolve given a development context. Without approaches such as the one proposed in this work, these studies will be missing critical data. In particular, our studies shows that renames are dominant across all contexts except for design (Table 11). This indicates that renames occur in many, many different development contexts and, with our tool, studies such as these could be extended to study how names change *given each individual context* instead of assuming they are indistinguishable. This extends to other work as well; there is a critical need for assisting developers in determining when to apply a refactoring; what refactoring to apply; and in some cases how to apply the refactoring [Arnaoudova et al. 2013, 2014b; Liu et al. 2013; Peruma et al. 2018, 2019b].

Additionally, there is a demonstrated need to further automate refactoring support. Prior research by [Kim et al. 2014] has investigated the way developers interact with IDEs when applying refactorings. Murphy-Hill et al. and Nigara et al. have shown that refactorings are frequently applied manually instead of automatically [Murphy-Hill and Black 2008; Negara et al. 2013]. This indicates that current support for refactoring is not enough; the benefit of automated application is outweighed by the cost, which other researchers have highlighted [Li and Thompson 2012; Newman et al. 2018]. Finally, we theorize that it will be beneficial to study how refactorings are applied to solve different types of problems (i.e., in this case, different maintenance tasks). This is supported by research that isolates certain types of code or code changes, such as isolating test from production code [Peruma et al. 2020; Tufano et al. 2016] or isolating API changes [Alrubaye and Mkaouer 2018]. Like these examples, future research must understand the context surrounding refactorings by identifying the reasoning (i.e., development context) behind refactoring operations. The results from this work directly impact research in this area by providing a methodology to categorize refactoring commit messages and providing an exploratory discussion of the motivation behind different types of refactorings. We plan to explore this question in greater detail in future research.

Refactoring Documentation. One of the main purposes of the automatic detection of refactoring is to better understand how developers cope with their software decay by extracting any refactoring strategies that can be associated with removing code smells [Bavota et al. 2013a; Tsantalis et al. 2008], or improving the design structural measurements [Bavota et al. 2014b; Mkaouer et al. 2014]. However, these techniques only analyze the changes at the source code level, and provide the operations performed, without associating it with any textual description, which may infer the rationale behind the refactoring application. Our proposed textual patterns is the first step towards complementing the existing effort in detecting refactorings, by augmenting it with any description that was intended to describe the refactoring activity. As previously shown in Tables 12, 13, and 15 developers tend to add a high-level description of their refactoring activity, and occasionally mention their intention behind refactoring (remove duplicate code, improve readability, etc.), along with mentioning the refactoring operations they apply (type migration, inline methods, etc.). This paper proposes, combined with the detection of refactoring operations, a solid background for future empirical investigations. For instance, previous studies have analyzed the impact of refactoring operations on

structural metrics [Bavota et al. 2015; Cedrim et al. 2016; Palomba et al. 2017a]. One of the main limitations of these studies is the absence of any context related to the application of refactorings, i.e., it is not clear whether developers did apply these refactoring with the intention of improving design metrics. Therefore, it is important to consider commits whose commit messages specifically express the refactoring for the purpose of optimizing structural metrics, such as coupling, and complexity, and so, many empirical studies can be revisited with a more adequate dataset.

Furthermore, our study provides software practitioners with a catalog of common refactoring documentation patterns (cf. Tables 12, 13, and 15) which would represent concrete examples of common ways to document refactoring activities in commit messages. This catalog of SAR patterns can encourage developers follow best documentation patterns and also to further extend these patterns to improve refactoring changes documentation in particular and code changes in general. Indeed, reliable and accurate documentation is always of crucial importance in any software project. The presence of documentation for low level changes such as refactoring operations and commit changes helps to keep track of all aspects of software development and it improves on the quality of the end product. Its main focuses are learning and knowledge transfer to other developers.

Another important research direction that requires further attention concerns the documentation of refactoring. It has been known that there is a general shortage of refactoring documentation, as developers typically focus on describing their functional updates and bug patches. Also, there is no consensus about how refactoring should be documented, which makes it subjective and developer specific. Moreover, the fine-grained description of refactoring can be time consuming, as typical description should contain indication about the operations performed, refactored code elements, and a hint about the intention behind the refactoring. In addition, the developer specification can be ambiguous as it reflects the developer's understanding of what has been improved in the source code, which can be different in reality, as the developer may not necessarily adequately estimate the refactoring impact on the quality improvement. Therefore, our model can help to build a corpus of refactoring descriptions, and so many studies can better analyze the typical syntax used by developers in order to develop better natural language models to improve it, and potentially automate it, just like existing studies related to other types of code changes [Buse and Weimer 2010; Linares-Vásquez et al. 2015; Liu et al. 2018].

Further, a potential research direction is to use the current findings (see Tables 12, and 13) to build a tool that supports the automatic identification and detection of self-affirmed refactoring commits. We also plan to conduct different user studies with our industrial partner predict the refactoring intention of the developers and further assess whether it aligns with what happened to his source code after applying refactoring.

Refactoring and Developer's Experience. While refactoring is being applied by various developers [AlOmar et al. 2020], it would be interesting to evaluate their refactoring practices. We would like to capture and better understand the code refactoring best practices and learn from these developers so that we can recommend them for other developers. Previous work [AlOmar et al. 2019a] performed an exploratory study on how developers document their refactoring activities in commit messages, this activity is called Self-Affirmed Refactoring (SAR). They found that developers tend to use a variety of textual patterns to document their refactoring activities, such as *"refactor"*, *"move"* and *"extract"*. In follow-up work, AlOmar et al. [AlOmar et al. 2019b] identified which quality models are more in-line with the developer's vision of quality optimization when they explicitly mention in the commit messages that they refactor to improve these quality attributes. Since we noticed that various developers are responsible for performing refactorings, one potential research direction is to investigate which developers are responsible for the introduction of SARs in order to examine whether or not experience plays a role in the introduction of SARs. Another potential research direction is to study if developer experience is one of the factors that might contribute to the significant improvement of the quality

metrics that are aligned with developer description in the commit message. In other words, we would like to evaluate the top contributors refactoring practice against all the rest of refactoring contributors by assessing their contributions on the main internal quality attributes improvement (e.g., cohesion, coupling, and complexity).

7 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 only the 28 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 types are 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] report that Refactoring Miner has a precision of 98% and a recall of 87% which significantly outperforms the previous state-of-the-art tools, which gives us confidence in using the tool.

Further, the set of commit messages used in this study may represent a threat to validity, because not all of the messages it may indicate refactoring activities. To mitigate this risk, we manually inspected a subset of change messages and ensured that projects selected are well-commented and use meaningful commit messages. Another potential threat to validity relates to our findings regarding counting the reported quality attributes and code smells. Due to the large number of commit messages, we have not performed a manual validation to remove false positive commit messages. Thus, this may have an impact on our findings. Further, since extracting refactoring patterns heavily depends on the content of commit messages, our results may be impacted by the quantity and quality of commits in a software project. To alleviate this threat, we examined multiple projects. Moreover, our manual analysis is a time consuming and an error prone task, which we tried to mitigate by focusing mainly on commits known to contain refactorings. Also, since our keywords largely overlap with keywords used in previous studies, this raised our confidence about the found set but does not guarantee that we did not miss any patterns.

Another threat relates to the detection of JUnit test files. The task of associating a unit test file with its production file was an automated process (performed based on filename/string matching associations). If developers deviate from JUnit guidelines on file naming, false positives may be triggered. However, our manual verification of random associations and the extensiveness of our dataset acts as a means of countering this risk.

External Validity. The first threat is that the analysis was restricted to only open source, Java-based, Git-based repositories. However, we were still able to analyze 800 projects that are highly varied in size, contributors, number of commits and refactorings. Another threat concerns the generalization of the identified recurring patterns in the refactoring commits. Our choice of patterns may have an impact on our findings and may not generalize to other open source or commercial projects since the identified refactoring patterns may be different for another set of projects (e.g., outside the Java developers community or projects that have a low number of or no commit messages). Consequently, we cannot claim that the results of refactoring motivation (see Figure 2) can be generalized to other programming languages in which different refactoring tools have been used, projects with a significantly larger number of commits, and different software systems where the need for improving the design might be less important.

Construct validity. The classification of refactorings heavily relies on commit messages. Even when projects are well-commented, they might not contain SAR, as developers might not document refactoring activities in the commit messages. We mitigate this risk by choosing projects that are appropriate for our analysis. Another potential threat relates to manual classification. Since the manual classification of training commit messages is a human intensive task

and it is subject to personal bias, we mitigate manual classification related errors by discarding short and ambiguous commits from our dataset and replacing them with other commits. Another important limitation concerns the size of the dataset used for training and evaluation. The size of the used dataset was determined similarly to previous commit classification studies, but we are not certain that this number is optimal for our problem. It is better to use a systematic technique for choosing the size of the evaluation set.

To mitigate the impact of different commit message 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. An additional threat to validity relates to the construction of our set of refactoring patterns. One pattern could be used as an umbrella term for lots of different types of activity (e.g., "Cleaning" might mean totally different things to different developers). However, we mitigate this threat by focusing mainly on commits known to contain refactorings. Further, recent studies [Guinan et al. 1998; Kirinuki et al. 2014; Yan et al. 2016] 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.

Conclusion Validity. The refactoring documentation research question has been provided along with the corresponding hypotheses in order to aid in drawing a conclusion. To accompany this, a statistical test has been used to test the significance of the results gained. Specifically, we applied the Wilcoxon test, a widely used non-parametric test, to test whether refactoring patterns are significant or not. This test makes no assumption that the data is normally distributed. Refactoring motivation categories and the way we grouped the refactoring concepts described in previous papers and established relations between them poses a threat to the conclusion validity of our study. If some information was not described in the literature, it may affect our conclusions.

8 CONCLUSION & FUTURE WORK

In this paper, we performed a large scale empirical study to explore the motivation driving refactorings, the documentation of refactoring activities, and the proportion of refactoring operations performed on production and test code. In summary, the main conclusions are:

(1) Our study shows that code smell resolution is not the only driver for developers to factor out their code. Refactoring activity is also driven by changes in requirements, correction of errors, structural design optimization and nonfunctional quality attributes enhancement. Developers are using wide variety of refactoring operations to refactor production and test files.
(2) A wide variety of textual patterns is used to document refactoring activities in the commit messages. These patterns could demonstrate developer perception of refactoring or report a specific refactoring operation name following Fowler's names.

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

Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 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, 125–134.

Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019a. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3nd International Workshop on Refactoring-accepted. IEEE.*

Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019b. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

Eman Abdullah AlOmar, Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results. In *Proceedings of the 4th International Workshop on Refactoring (IWoR 2020)*. Association for Computing Machinery, New York, NY, USA.

Hussein Alrubaye and Mohamed Wiem Mkaouer. 2018. Automating the Detection of Third-party Java Library Migration at the Function Level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Riverton, NJ, USA, 60–71. http://dl.acm.org/citation.cfm?id=3291291.3291299

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

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

J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. C. Khoo. 2012. Semantic patch inference. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 382–385. https://doi.org/10.1145/2351676.2351753

Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2013. A new family of software anti-patterns: Linguistic anti-patterns. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 187–196.

Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2014a. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

Venera Arnaoudova, Laleh Mousavi Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014b. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Trans. Software Eng.* 40, 5 (2014), 502–532.

Boehm Barry et al. 1981. Software engineering economics. *New York* 197 (1981).

Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014a. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.

Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013a. An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 692–701.

Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2013b. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2013), 671–694.

Gabriele Bavota, Sebastiano Panichella, Nikolaos Tsantalis, Massimiliano Di Penta, Rocco Oliveto, and Gerardo Canfora. 2014b. Recommending refactorings based on team co-maintenance patterns. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 337–342.

Steven Bird. 2002. NLTK: The Natural Language Toolkit. *ArXiv* cs.CL/0205028 (2002).

Barry W. Boehm. 2002. Software Pioneers. Springer-Verlag, Berlin, Heidelberg, Chapter Software Engineering Economics, 641–686. http://dl.acm.org/citation.cfm?id=944331.944370

Raymond PL Buse and Westley Weimer. 2010. Automatically documenting program changes.. In *ASE*, Vol. 10. 33–42.

Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 465–475.

Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*. ACM, 73–82.

Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 456–460.

Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3131151.3131171

Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

J. Al Dallal and A. Abdin. 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 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2658573

P. Dangeti. 2017. *Statistics for Machine Learning*. Packt Publishing.

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. *Automated Detection of Refactorings in Evolving Components*. Springer Berlin Heidelberg, Berlin, Heidelberg, 404–428. https://doi.org/10.1007/11785477_24

Len Erlikh. 2000a. Leveraging legacy system dollars for e-business. *IT professional* 2, 3 (2000), 17–23.

L. Erlikh. 2000b. Leveraging legacy system dollars for e-business. *IT Professional* 2, 3 (May 2000), 17–23. https://doi.org/10.1109/6294.846201

Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. 1981. The measurement of interrater agreement. *Statistical methods for rates and proportions* 2, 212-236 (1981), 22–23.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. http://dl.acm.org/citation.cfm?id=311424

Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 211–221. http://dl.acm.org/citation.cfm?id=2337223.2337249

Patricia J. Guinan, Jay G. Cooprider, and Samer Faraj. 1998. Enabling Software Development Team Performance During Requirements Definition: A Behavioral Versus Technical Approach. *Information Systems Research* 9, 2 (1998), 101–125. https://doi.org/10.1287/isre.9.2.101 arXiv:https://doi.org/10.1287/isre.9.2.101

Ahmed E. Hassan. 2008. Automated Classification of Change Messages in Open Source Projects. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*. ACM, New York, NY, USA, 837–841. https://doi.org/10.1145/1363686.1363876

L. P. Hattori and M. Lanza. 2008. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*. 63–71. https://doi.org/10.1109/ASEW.2008.4686322

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

Abram Hindle, Neil A. Ernst, Michael W. Godfrey, and John Mylopoulos. 2011. Automated Topic Naming to Support Cross-project Analysis of Software Maintenance Activities. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 163–172. https://doi.org/10.1145/1985441.1985466

A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. 2009. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*. 30–39. https://doi.org/10.1109/ICPC.2009.5090025

Abram Hindle, Daniel M. German, and Ric Holt. 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 (MSR '08)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773

Daniel Jurafsky and James H Martin. 2019. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. *Prentic e Hall* (2019).

Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010a. Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 371–372. https://doi.org/10.1145/1882291.1882353

Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010b. 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, 371–372.

M. Kim, T. Zimmermann, and N. Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (July 2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! Are You Committing Tangled Changes?. In *Proceedings of the 22Nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 262–265. https://doi.org/10.1145/2597008.2597798

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

H. Lane, H. Hapke, and C. Howard. 2019. *Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python*. Manning Publications Company.

Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

T. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk. 2015. RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information. In *2015 IEEE 23rd International Conference on Program Comprehension*. 36–47. https://doi.org/10.1109/ICPC.2015.13

Stanislav Levin and Amiram Yehudai. 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 (PROMISE)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/3127005.3127016

Huiqing Li and Simon Thompson. 2012. Let's Make Refactoring Tools User-extensible!. In *Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12)*. ACM, New York, NY, USA, 32–39. https://doi.org/10.1145/2328876.2328881

Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. On the Impact of Refactoring Operations on Code Naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 594–598.

Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

H. Liu, X. Guo, and W. Shao. 2013. Monitor-Based Instant Software Refactoring. *IEEE Transactions on Software Engineering* 39, 8 (Aug 2013), 1112–1126. https://doi.org/10.1109/TSE.2013.4

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 373–384.

N. Mahmoodian, R. Abdullah, and M. A. A. Murad. 2010. Text-based classification incoming maintenance requests to maintenance type. In *2010 International Symposium on Information Technology*, Vol. 2. 693–697. https://doi.org/10.1109/ITSIM.2010.5561540

C.D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig. 2015. Dataset of Developer-Labeled Commit Messages. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 490–493. https://doi.org/10.1109/MSR.2015.71

Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. 2012. *Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 301–315. https://doi.org/10.1007/978-3-642-28872-2_21

Collin McMillan, Mario Linares-Vasquez, Denys Poshyvanyk, and Mark Grechanik. 2011. Categorizing Software Applications for Maintenance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM '11)*. IEEE Computer Society, Washington, DC, USA, 343–352. https://doi.org/10.1109/ICSM.2011.6080801

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 502–511. http://dl.acm.org/citation.cfm?id=2486788.2486855

Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 331–336.

Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 17.

Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 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, 252–266.

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

Emerson Murphy-Hill and Andrew P Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (Sept 2008), 38–44. https://doi.org/10.1109/MS.2008.123

Emerson Murphy-Hill, Andrew P Black, Danny Dig, and Chris Parnin. 2008. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools*. ACM, 7.

Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. https://doi.org/10.1109/TSE.2011.41

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 552–576.

Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. 2018. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*. ACM, 34–41.

Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017a. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 176–185.

F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. 2017b. An Exploratory Study on the Relationship between Changes and Refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 176–185. https://doi.org/10.1109/ICPC.2017.38

A. Peruma. 2019. A Preliminary Study of Android Refactorings. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 148–149. https://doi.org/10.1109/MOBILESoft.2019.00030

Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019a. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., USA, 193–202.

Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. 2018. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*. ACM, 26–33.

Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. 2019b. Contextualizing Rename Decisions using Refactorings and Commit Messages. In *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE*.

Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali. Ouni, and Fabio Palomba. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In *Proceedings of the 4th International Workshop on Refactoring (IWoR 2020)*. Association for Computing Machinery,

New York, NY, USA.

K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. 2010. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

Jacek Ratzinger. 2007. *sPACE: Software Project Assessment in the Course of Evolution*. Ph.D. Dissertation. http://www.infosys.tuwien.ac.at/Staff/ratzinger/publications/ratzinger_phd-thesis_space.pdf

Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. ACM, New York, NY, USA, 35–38. https://doi.org/10.1145/1370750.1370759

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

D. Silva and M. T. Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279. https://doi.org/10.1109/MSR.2017.14

R. Singh and N.S. Mangat. 2013. *Elements of Survey Sampling*. Springer Netherlands.

Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Márcio Cornélio. 2009. Saferefactor-tool for checking refactoring safety. (01 2009).

Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. 2013. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software* 86, 4 (2013), 1006–1022.

Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*. IEEE, 10–10.

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

Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2017. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software* 129 (2017), 107–126.

Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 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, 524–540.

Liang Tan and Christoph Bockisch. 2019. A Survey of Refactoring Detection Tools. In *Software Engineering*.

Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 329–331.

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

Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Riverton, NJ, USA, 132–146. http://dl.acm.org/citation.cfm?id=2555523.2555539

Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 483–494.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1–15.

Carmine Vassallo, Fabio Palomba, and Harald C Gall. 2018. Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 564–568.

Y. Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *2009 IEEE International Conference on Software Maintenance*. 413–416. https://doi.org/10.1109/ICSM.2009.5306290

P. Weissgerber and S. Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 231–240. https://doi.org/10.1109/ASE.2006.41

Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1101908.1101919

Zhenchang Xing and Eleni Stroulia. 2008. The JDEvAn Tool Suite in Support of Object-oriented Evolutionary Development. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 951–952. https://doi.org/10.1145/1370175.1370203

Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D. Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113, Supplement C (2016), 296 – 308. https://doi.org/10.1016/j.jss.2015.12.019

Di Zhang, Bing Li, Zengyang Li, and Peng Liang. 2018. A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software. https://doi.org/10.18293/SEKE2018-081

A. Zheng and A. Casari. 2018. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists.* O'Reilly Media.