How We Refactor and How We Document it? On the Use of Supervised Machine Learning Algorithms to Classify Refactoring Documentation

Eman Abdullah AlOmar[a,*], Anthony Peruma[a], Mohamed Wiem Mkaouer[a], Christian Newman[a], Ali Ouni[b], Marouane Kessentini[c]

[a]*Rochester Institute of Technology, Rochester, NY, USA*
[b]*ETS Montreal, University of Quebec, Montreal, QC, Canada*
[c]*University of Michigan, Dearborn, MI, USA*

Abstract

Refactoring is the art of improving the structural design of a software system without altering its external behavior. Today, refactoring has become a well established and disciplined software engineering practice that has attracted a significant amount of research presuming 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 especially with the emerging challenges in contemporary software engineering. 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 a large set of 111,884 commits containing refactoring activities, extracted from 800 open source Java projects. We trained a multi-class classifier to categorize these commits into three categories, namely, Internal Quality Attribute, External Quality Attribute, and Code Smell Resolution, along with the traditional Bug Fix and Functional categories. This classification 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. Refactoring is solicited for a wide variety of reasons, going beyond its traditional definition; (2) the distribution of refactoring operations differ between production and test files; (3) developers use a variety of patterns to purposefully target refactoring-

*Corresponding author
Email addresses: eman.alomar@mail.rit.edu (Eman Abdullah AlOmar), anthony.peruma@mail.rit.edu (Anthony Peruma), mwmvse@rit.edu (Mohamed Wiem Mkaouer), cdnvse@rit.edu (Christian Newman), ali.ouni@etsmtl.ca (Ali Ouni), marouane@umich.edu (Marouane Kessentini)

related activities; (4) the textual patterns, extracted from commit messages, provide a better coverage for how developers document their refactorings.

Keywords: Refactoring, Software Quality, Software Engineering, Machine Learning

---

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 (Erlikh, 2000; Barry et al., 1981). One key practice to cope with this challenge is refactoring. Refactoring is the art of remodeling the software design without altering its functionalities (Fowler et al., 1999; AlDallal & Abdin, 2017). It was popularized by (Fowler et al., 1999), who identified 72 refactoring types and provided examples of how to apply them in his catalog.

Refactoring is a critical software maintenance activity that is performed by developers for an amalgamation of reasons (Tsantalis et al., 2013; Silva et al., 2016; Palomba et al., 2017). Refactoring activities in the source code can be automatically detected (Dig et al., 2006; Tsantalis et al., 2013) providing a unique opportunity to practitioners and researchers to analyze how developers maintain their code during different phases of the development life-cycle and over large periods of time. Such valuable knowledge is vital for understanding more about the maintenance phase; the most costly phase in software development (Boehm, 2002; Erlikh, 2000). To detect refactorings, the state-of-the-art techniques (Dig et al., 2006; Tsantalis et al., 2013) typically search at the level of commits. As a result, these techniques are also able to group commit messages with their corresponding refactorings.

Commit messages are the description, in natural language, of the code-level changes. To understand the nature of the change, recent studies have been using natural language processing to process commit messages for multiple reasons, such as classification of code changes (Hindle et al., 2008), change summarization (McBurney et al., 2017), change bug-proneness (Xia et al., 2016), and developer's rationale behind their coding decisions (Alkadhi et al., 2018). That is, commit messages are a common way for researchers to study developer rationale behind different types of changes to the code. There are two primarily challenges to using commit messages to understand refactorings: 1) the commit message does not have to refer to the refactoring that took place at all, 2) developers have many ways of describing the same activity. For example, instead of explicitly stating that they are refactoring, a developer may instead state that they are *performing code clean-up* or *simplifying a method*. Developers are inconsistent in the way they discuss refactorings in commit messages. This makes it difficult to perform analysis on commit messages, since researchers may find it challenging to determine whether a commit message discusses the

refactoring(s) being performed or not. Thus, it is hard to determine when the commit message is discussing a refactoring at all and it is hard to determine how a commit message is discussing the refactoring.

To cope with the above-mentioned challenges, the purpose of this study is to augment our understanding of the development contexts that trigger refactoring activities and enable future research to take development contexts into account more effectively when studying refactorings. Thus, the advantages of analyzing the textual description of the code change that was intended to describe refactoring activities are three-fold: 1) it improves our ability to study commit message content and relate this content to refactorings; a challenging task which posed a significant hurdle in recent work on contextualizing rename refactorings (Peruma et al., 2018, 2019b), 2) it gives us a stronger understanding of commit message practices and could help us improve commit message generation by making it clear how developers prefer to express their refactoring activities, 3) it provides us with a way of relating common words and phrases used to describe refactorings with one another. Typically frameworks like WordNet, which does not recognize refactoring phrases and terminology, are used for this task. Our dataset and methodology reduces the need to rely on frameworks which are not trained for natural language found in software projects.

In this paper, we present a way to partially-automatically detect how developers document their refactorings in commit messages, and classify these into categories that reflect the type of activity that refactoring was co-located with. The goal of this work is to create a data set of terms and phrases, used by developers, to describe refactorings. Further, we group these words and phrases by maintenance-type (e.g., bug fix, external, code smell) to obtain a fine-grained and maintenance-type-specific dataset of terms and phrases. Recent studies have shown the feasibility of extracting insights of software quality from developers inline documentation. For instance, mining developer's comments has unveiled how developers knowingly commit code that is either incomplete, temporary, or faulty. Such phenomenon is known as *Self-Admitted Technical Debt* (SATD) (Potdar & Shihab, 2014). Similarly, our previous study has introduced *Self-Affirmed Refactoring* (SAR) (AlOmar et al., 2019a, 2020a), defined as developer's explicit documentation of refactoring operations intentionally introduced during a code change.

To perform this analysis, we formulate the following research questions:

- **RQ1.** To what purposes developers refactor their code?

  While previous surveys studied how developers apply refactorings in varying development contexts, none of them have measured the ubiquity of these varying contexts in practice. Therefore, it is important to *quantify* the distribution of refactoring activities performed in varying development contexts to augment our understanding of refactoring in theory versus in practice.

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

This question further explores the findings of the classification to see to what extent developers refactor production files differently from test files.

- **RQ2.** What patterns do developers use to describe their refactoring activities?

  Since there is no consensus on how to formally document refactoring changes, we intend to extract (from commit messages) words and phrases commonly used by developers in practice to document their refactorings. Such information is useful from many perspectives. First, it allows to understand the rationale behind the applied refactorings, e.g., fixing code smells or improving specific quality attributes. Moreover, it may reveal what specific refactoring operations are being documented, and whether developers explicitly mention it as part of their documentation. Such details are of crucial importance especially in modern code review the help code reviewers understand the rationale behind such refactorings. Little is known about how developers document refactoring as previous studies mainly rely on the keyword *refactor* to annotate such documentation.

- **RQ2.1** Do commits containing the label *Refactor* indicate more refactoring activity than those without the label?

  We revisit the hypothesis raised by (Murphy-Hill et al., 2008) about whether developers use a specific pattern, i.e., *"refactor"* when describing their refactoring activities.

The dataset of classified refactorings along with textual patterns are available online (AlOmar, 2020 (last accessed October 20, 2020) for replication and extension purposes.

The remainder of this paper is organized as follows. Section 2 discusses the notion of refactoring related documentation or Self-Affirmed Refactoring. Section 3 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 construction 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. Self-Affirmed Refactoring

Commit messages are the description, in natural language, of the code-level changes. In this paper, we want to automatically detect how refactoring is documented in the commit message, and classify it into categories that reflect the type of activity that refactoring was co-located with. Earlier studies were relying on developer surveys for extracting such information. But 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, mined commits' messages and their issues descriptions constitute 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 (Tan et al., 1999).

However, recent studies have shown the feasibility of extracting insights of software quality from developer's inline documentation. For instance, mining developers' comments has unveiled how developers knowingly commit code that is either incomplete, temporary, or faulty. Such phenomenon is known as *Self-Admitted Technical Debt* (SATD) (Potdar & Shihab, 2014). Similarly, our previous study has introduced *Self-Affirmed Refactoring* (SAR) (AlOmar et al., 2019a, 2020a), defined as developers' explicit documentation of refactoring operations intentionally introduced during a code change.

As explained later in the related work section, existing studies locate refactoring documentation through the localization of the keyword *"refactor"*, being the most intuitive and widely known. However, a recent study has also shown that the *"refactor"* can also be misused, and such information becomes misleading (Zhang et al., 2018). Yet, such findings are mainly taken from interviews. In this study, we leverage the existence of a large set of refactorings, extracted form a wide variety of projects, to design an empirical study to classify the context in which it was performed, for that, we start with the traditional categorization of Swanson (Swanson, 1976), and we extend its "Perfective" category to cover what has been known by existing studies as drivers to recommend refactorings. This study also further explores how developers document refactorings, and extracts a new terminology that was found to be consistently used in refactoring-related commit messages.

3. 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 documentation, along with the research on refactoring motivations.

3.1. Refactoring Documentation

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 & 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

Table 1: Existing works on refactoring identification.

Study	Year	Purpose	Approach	Source of Info.	Ref. Patterns
(Stroggylos & 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

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), 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.

(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., 2010) 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 tends 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 to locate refactoring events, but it does not guarantee the application of refactorings. We summarize these state-of-the-art approaches in Table 1.

3.2. Refactoring Motivation

(Silva et al., 2016) investigate what motivates developers when applying specific refactoring operations by surveying GitHub contributors of 124 software projects. They observe that refactoring activities are mainly caused by changes in the project requirements and much less by code smells. (Palomba et al., 2017) verify the relationship between the application of refactoring operations and different types of code changes (i.e., *Fault Repairing Modification, Feature Introduction Modification*, and *General Maintenance Modification*) over the change history of three open source systems. Their main findings are that developers apply refactoring to: 1) improve comprehensibility and maintainability when fixing bugs, 2) improve code cohesion when adding new features, and 3) improve the comprehensibility when performing general maintenance activities. On the other hand, (Kim et al., 2014) do not differentiate the motivations between different refactoring types. They surveyed 328 professional software engineers at Microsoft to investigate when and how they do refactoring. When surveyed, the developers cited the main benefits of refactoring to be: improved readability (43%), improved maintainability (30%), improved extensibility (27%) and fewer bugs (27%). When asked what provokes them to refactor, the main reason provided was poor readability (22%). Only one code smell (i.e, code duplication) was mentioned (13%).

(Murphy-Hill et al., 2012) examine how programmers perform refactoring in practice by monitoring their activity and recording all their refactorings. 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., 2006) study the impact of refactoring on reusability. They showed that refactoring increases the reusability of classes in an industrial, agile environment. In a subsequent study, (Moser et al., 2007) 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) conduct 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) extend their study by investigating whether the refactorings applied when fixing issues did improve the system's nonfunctional 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 concluded that refactoring cannot be cornered only in the context of design improvement.

(Tsantalis et al., 2013) manually inspect 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. Besides 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 focuses 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.

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. More recently, (Pantiuchina et al., 2020) present a mining-based study to investigate why developers are performing refactoring in the history of 150 open source systems. Particularly, they analyzed 551 pull requests implemented refactoring operations and reported a refactoring taxonomy that generalizes the ones existing in the literature. (Paixão et al., 2020) perform an empirical study on refactoring activities in code review in which they captured Bug Fix and Feature refactoring categories. (AlOmar et al., 2020c) studied how developers refactor their code to improve its reuse by analyzing the impact of reusability refactorings on the state-of-the-art reusability metrics. Figure 1 depicts how our classification clusters the existing refactoring taxonomy reported in the literature (Moser et al., 2006, 2007; Tsantalis et al., 2013; Kim et al., 2014; Silva et al., 2016; Palomba et al., 2017; Vassallo et al., 2019; AlOmar et al., 2019b; Pantiuchina et al., 2020; Paixão et al., 2020; AlOmar et al., 2020c). As can be seen, our classification covers these categories. 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.

All the above-mentioned studies have agreed on the existence of motivations that go beyond the basic need for improving the system's design. Refactoring activities have been solicited in scenarios that have been coined by the previous studies as follows: Functional, Bug Fix, Internal Quality Attribute, Code Smell Resolution, and External Quality Attribute. Since these categories are the main drivers for refactoring activities, we decided to cluster our mined refactoring operations according to these groups.

Our proposal differs from commit classification-related 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 Quality Attribute, External Quality Attribute and Code Smell Resolution. This division is inherited from the analysis of previous papers whose detection of refactoring opportunities rely 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 Attribute (ii) External Quality Attribute, and (iii) Code Smell Resolution. 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., (Hindle et al., 2011; Kochhar et al., 2014; Levin & Yehudai, 2017; Hönel et al., 2019; AlOmar et al., 2020a)) in the context of commit classification and challenge them using our defined set of classes. Although several studies (Hattori & Lanza, 2008; Mauczka et al., 2012; Hindle et al., 2009; Amor et al., 2006; Levin & Yehudai, 2017; Hindle et al., 2008; Mauczka et al., 2015; 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', 'Bug Fix', 'Internal Quality Attribute', 'Code Smell Resolution', and 'External Quality Attribute'.

4. Empirical Study Setup

To answer our research questions defined in Section 1, we design a five-steps approach as shown in Figure 2. Our approach consists of: (1) data collection, (2) refactoring detection, (3) automatic refactoring classification, (4) unit test files detection, and (5) refactoring documentation extraction.

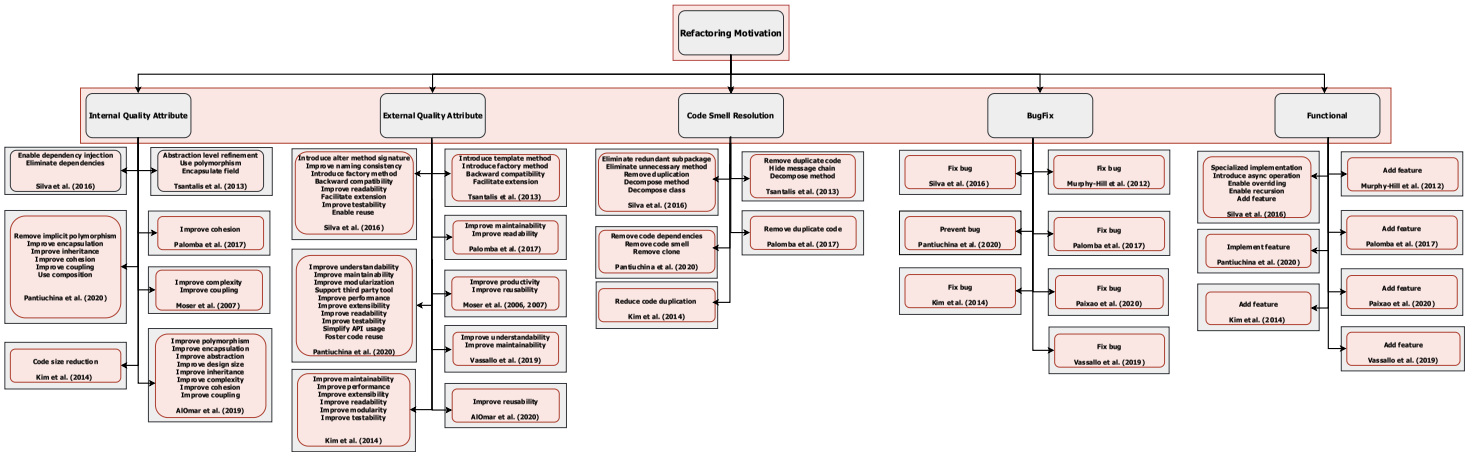


Figure 1: Refactoring motivation.

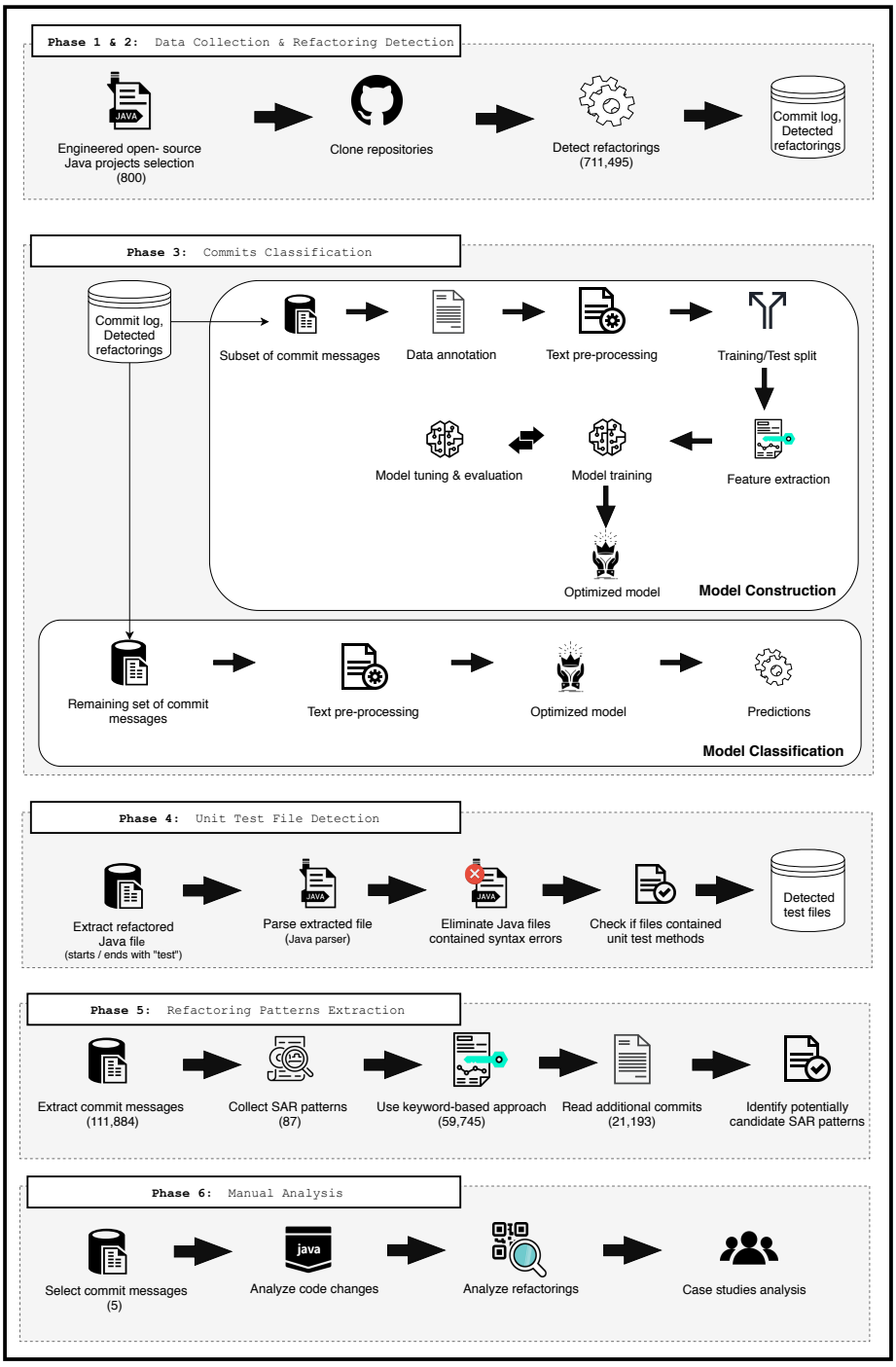


Figure 2: Empirical study design overview.

4.1. Phase 1: Data Collection

Our first step consists of randomly selecting 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 that they were Java-based, the only language supported by Refactoring Miner (Tsantalis et al., 2018). The authors of this dataset selected "well-engineered software projects" based on the projects' use of software engineering best practices such as documentation, testing, and project management. Additionally, these projects are non-forked (i.e., not cloned from other projects) as forked projects may impact our conclusions by introducing duplicate code and data. We cloned the 800 selected projects having a total of 748,001 commits, and a total of 711,495 refactoring operations from 111,884 refactoring commits. Additionally, these projects contain on average 935 commits and 19 developers. An overview of the project's statistics is provided in Table 2. This table shows the total number of Java projects used in this study (800), the total number of commits across all projects combined (748,001), the total number of refactoring commits and the associated refactoring operations respectively, 111,884 and 711,495. The table also details the number of refactoring operations per code element at different levels of granularity, including method, attribute, class, variable, parameter, package, and interface, ordered from highest down to the lowest. Additionally, the standard deviation reported in the table shows that these projects are very diverse.

Table 2: Projects overview.

Item	**Count**	**Standard Deviation**
Total of projects	800	N/A
Total commits	748,001	1233.69
Refactoring commits	111,884	195.48
Refactoring operations	711,495	2402.12
Considered Projects - Refactored Code Elements		
Code Element	**# of Refactorings**	**Standard Deviation**
Method	222,785	415.55
Attribute	201,791	1854.35
Class	121,625	273.24
Variable	115,717	383.91
Parameter	48,054	127.48
Package	2380	8.25
Interface	1742	6.01

4.2. Phase 2: Refactoring Detection

To extract the entire refactoring history of each project, we used the Refactoring Miner[1] tool introduced by (Tsantalis et al., 2018). We decided to use Refactoring Miner as it has shown promising results in detecting refactorings compared to the state-of-the-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 (e.g., Ref-Finder (Kim et al., 2010)), in contrast, require user interaction to select projects as inputs and trigger the refactoring detection, which is impractical since multiple releases of the same project have to be imported to Eclipse to identify the refactoring history.

4.3. Phase 3: Commits Classification

After all refactoring operations are collected, we need to classify them. 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. We started by collecting the different motivations that drive developers to refactor their code as reported in the literature (Kim et al., 2014; AlDallal & Abdin, 2017; Fowler et al., 1999; Lanza & Marinescu, 2007; Silva et al., 2016; Tsantalis et al., 2013; Palomba et al., 2017; Murphy-Hill et al., 2012). Then, we search for common categories among the reported motivations. The following step involves identifying categories clustering functional requirements, quality attributes and software issues under the identified categories. This process resulted in five different categories. Hence, we aim to classify the refactoring commit, into one of five main categories: '*Functional*', '*Bug Fix*', '*Internal Quality Attribute*', '*Code Smell Resolution*', and '*External Quality Attribute*'. Table 3 provides a description of each category.

In this supervised multi-class classification problem, we followed a multi-staged approach to build our model for commit messages classification. The first stage consists of the model construction. In the second stage, we utilized the built model to classify the entire dataset of commit messages. An overview of our methodology is depicted in Figure 2. In the following subsections, we describe in detail the different steps in each stage.

Model Construction

In the first stage of the experiment, our goal is to build a model from a corpus real world documented refactorings (i.e., commit message) to be utilized in the second stage to classify commit messages. The following subsections detail the different steps in the model construction phase.

[1] `https://github.com/tsantalis/RefactoringMiner`

Table 3: Classification categories.

Category	Description
Functional	Feature implementation, modification or removal
Bug Fix	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 violate the fundamentals of software design principles and decrease code quality such as duplicated code and long method
External QA	Property or feature that indicates the effectiveness of a system such as testability, understandability, and readability

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 prepare this gold set, a manual annotation (i.e., labeling) of commit messages needs to be performed by subject 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. Confidence level and interval are utilized to obtain an accurate and statistically significant sample size from a population (Brownlee, 2018). The authors of this paper performed the 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 Quality Attribute', 'Code Smell Resolution', and 'External Quality Attribute'. 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. In total, we annotated 348 commit messages as 'Functional', 'Bug Fix', 'Internal Quality Attribute', and 'Code Smell Resolution', while 310 messages were labeled as 'External Quality Attribute'.

4.3.2. Text Pre-Processing

To better support 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 & 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 non-alphabet characters, stop words, and (3) reducing each word to its lemma. The lemmatization process either replaces the suffix of a word with a different one or removes the suffix of a word to get the basic word form (lemma) (Lane et al., 2019). In our work, the lemmatization process involves sentence separation, part-of-speech identification, and generating dictionary form. We split the commit messages into sentences, since input text could constitute a long chunk of text. The part-of-speech identification helps in filtering words used as features that aid in key-phrase extraction. Lastly, since

the word could have multiple dictionary forms, only the most probable form is generated. 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. Custom stop words include 'git', 'code', 'refactor', 'svn', 'gitsvnid', 'signedoffby', 'reviewedon', 'testedby', 'us', id', 'changeid', 'lot', 'small', 'thing', 'way'. 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 & 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 & Casari, 2018). In our study, we constructed our model using the text in the commit message. Hence, the feature for this model is limited to the commit message. We utilized Term Frequency-Inverse Document Frequency (TF-IDF) (Manning et al., 2008), commonly used in the literature (Lin et al., 2013; Le et al., 2015), 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. The N-Gram technique is a set of *n-word* that occurs in a text set and could be used as a feature to represent that text (Kowsari et al., 2019). In general, the N-Gram term has more semantic than an isolated word.

Some of the keywords (e.g., "*extract*") do not provide much information when used on its own. However, when collecting N-Gram from commit message (e.g., *Refactor createOrUpdate method in MongoChannelStore to extract methods and make code more readable*), the keyword "extract" clearly indicates that this refactoring commit belongs to *Extract Method* refactoring. In our classification, we use N-Grams since it is very common to enhance the performance of text classification (Tan et al., 2002). Using TF-IDF, we can determine words that are common and rare across the documents (i.e., commit messages) in our dataset; the model utilizes these words. In other words, The value for each N-Gram is proportional to its TF score multiplied by its IDF score. Thus, each preprocessed word in the commit message is assigned a value which is the weight of the word computed using this weighting scheme. TF-IDF gives greater weight (e.g., value) to words which occur frequently in fewer documents rather than words which occur frequently in many documents.

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 (C-Support Vector Classification based on LIBSVM (Deng et al., 2012; Chang & Lin, 2011)), and Decision Tree (CART (Breiman, 2017)). We selected these classifiers since they are widely adopted in several classification problems in software engineering, as reported in Section 3. It is important to note that the library containing the classification algorithms are capable of multiclass classification. As per the Python's SKlearn documentation, Random Forest, K-Nearest Neighbors, Logistic Regression, and Multinomial Naive Bayes are inherently multiclass (SKlearn, 2007a), while SVC utilizes a one-vs-one approach to handle multiclass (SKlearn, 2007b). Moreover, 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). Our test dataset did not take part in the training process, which provides a more realistic model evaluation. This approach is also known to prevent overfitting that leads to incorrect conclusions. Grid search utilizes a brute force

technique to evaluate all combinations of hyperparameters to obtain the best performance. It is used to find the optimal hyperparameters of a model which results in the most accurate predictions. 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. We provide, in Table 4, the optimal hyperparameter values for the classification algorithms in our study.

Table 4: Optimal parameter values for the classification algorithms.

Algorithm	**Parameter**	**Value**
Random Forest	max_depth	78
	n_estimators	500
	criterion	gini
	bootstrap	false
Support Vector Classification	c	1.99
	gamma	scale
	kernel	linear
Decision Tree	criterion	gini
	max_depth	75
Logistic Regression	penalty	l1
	solver	liblinear
	c	1.0
Multinomial Naive Bayes	alpha	2.63
K-Nearest Neighbors	n_neighbors	69
	weights	uniform

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. In Section 5, we detail our classification results.

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 distinguish between refactorings applied to production and unit test files and perform comparisons against both production-file-based refactorings versus test-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 standards[2], we

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

first extracted all refactored Java source files where the filename either starts or ends with the word "test". Next, we utilized JavaParser[3] 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 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 document their refactoring activities, 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 having a list of the most insightful and unique patterns. It also helps in making more concise patterns that are usable for readers. We also analyzed the top 100 features, distilled by the classifier, for each category.

The manual analysis process took approximately 20 days in total. In the first two weeks, the authors had regular meetings to discuss top features, extracted from each category, to understand how each class was represented by its corresponding set of keywords, along with extracting any patterns that are most likely to be descriptive to refactoring, besides being another verification level of the classification accuracy. Moreover, during these meetings, the extraction of textual patterns from commit messages was also performed by the authors. Due to the subjective nature of this process, we opted to report as many keywords as possible for better coverage. When reporting keywords from top features, we kept the majority of keywords, for each category. keywords that were removed were either proper names of code elements (method names, identifiers, etc.), or languages and frameworks. For the identification of patterns from commit messages, the authors kept any keyword that can be either tightly or loosely coupled to refactoring. Such decision mitigates the selection bias, at the expenses of reporting keywords that may or may not be relevant to refactoring documentation. During the last week, two authors have finished analyzing the remaining commit messages. 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

[3] `https://javaparser.org/`

patterns. We manually read through 21,193 commit messages. Our in-depth inspection resulted in a list of 513 potential self-affirmed refactoring candidates, identified across the considered projects, as illustrated later in Tables 8 and 9.

4.6. Phase 6: Manual Analysis

To get a more qualitative sense of the classification results, we created five case studies that demonstrate GitHub developers' intentions when refactoring source code. Case study is one of the empirical methods used for studying phenomena in a real-life context (Wohlin et al., 2012). In our study, we performed a combination of manual analysis and quantitative analysis using custom-built scripts. For each case study, we provide the commit message and its corresponding refactoring operations detected by Refactoring Miner. We elaborate in detail these case studies in Section 5.2, where we report on our results.

5. Experimental Results

This section reports and discusses our experimental results and aims to answer the research questions in Section 1.

Replication package. We provide our comprehensive experiments package available in (AlOmar, 2020 (last accessed October 20, 2020) to further replicate and extend our study. The package includes the selected Java projects, the detailed refactoring and non-refactoring commits and documentation, manual commits classification, the automatic commits classification, and the JUnit file detection.

5.1. RQ1: To what purposes developers refactor their code?

To answer this research question, we present the refactoring commit messages classification results explained in Subsection 4.3. This section details the classification of 111,884 commit messages containing 711,495 refactoring operations. The complete set of scores for all the classifiers including the Precision, Recall, and F-measure scores per class for each machine learning classifier is provided in Table 5. The best performing model was used to classify the test dataset. Based on our findings, we observed that ***Random Forest achieved the best F1 score: 87%*** which is higher than its competitors. 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. In order to compare classification algorithms performance, we use the McNemar test (Dietterich, 1998). We compare the performance of Random Forest against the other five classifiers. As shown in Table 6, the McNemar's test results show that there are statistically significant differences in the performance of the classifiers except for the classifier Support Vector Classification in which the difference is not statistically significant.

Table 5: Detailed classification metrics (Precision, Recall, and F-measure) 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
Average F1	0.87	0.87	0.87	Average F1	0.87	0.86	0.86	Average F1	0.85	0.85	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
Average F1	0.83	0.83	0.83	Average F1	0.81	0.78	0.78	Average F1	0.78	0.77	0.76

Table 6: McNemar's test results.

Random Forest	
Classifier	**p-value**
Support Vector Classification	0.1
Decision Tree	0.04
Logistic Regression	0.02
Multinomial Naive Bayes	0.01
K-Nearest Neighbors	0.01

Figure 3 shows the categorization of commits, from all projects combined. We observe that all of the categories had almost a uniform distribution of refactoring classes with low variability. For instance, Bug Fix, Functional, Internal Quality Attribute, External Quality Attribute, and Code Smell Resolution had commit message distribution percentages of 24.3%, 22.3%, 20.1%, 17.5%, and 15.9%, respectively.

The first observation that we can draw from these findings is that developers do not solely refactor their code to fix code smells. They instead refactor the code for multiple purposes. Our manual analysis show 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 4 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

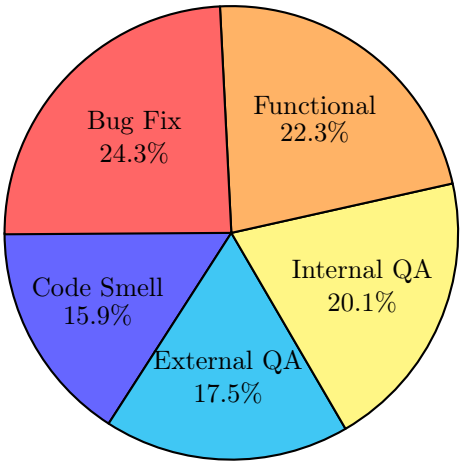
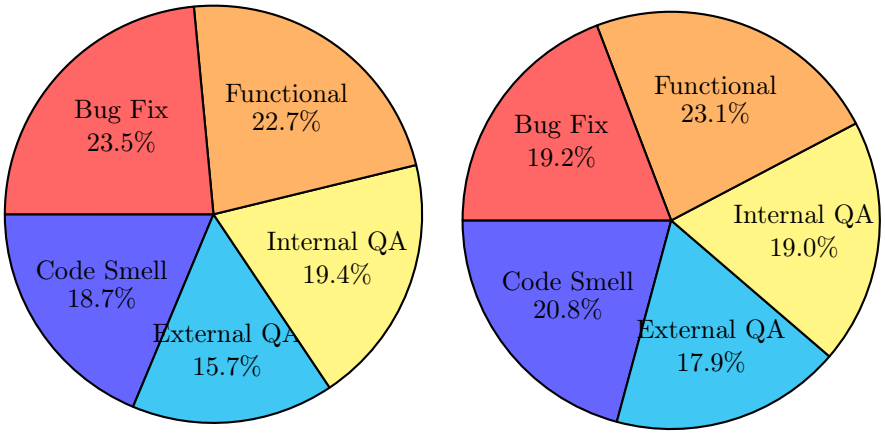


Figure 3: Percentage of classified commits per category in all projects combined.

differences.

Production code. 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 & Chatzigeorgiou, 2011). Regarding the Internal Quality Attributes category, developers move code elements for design-level changes (Stroggylos & Spinellis, 2007; Alshayeb, 2009; Bavota et al., 2015; Mkaouer et al., 2015), 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 Quality Attributes category, developers often 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 Functional category, developers implement a new feature or modify the existing ones.

Test code. With regards to test files, developers perform refactoring to improve the design of the code. An example can be shown 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. Developers explicitly mention the use of the renaming operations for the purpose of disambiguation the redundancy of methods names and enhancing their usability. Another activity to refactor test files could be moving methods,



(a) Production file

(b) Test file

Figure 4: Percentage of classified commits per category in production and test files.

or pushing code elements across hierarchies, e.g., pushing up attributes. Each of these activities are performed to support several refactoring motivations.

Summary. Our study has shown that fixing code smells is not the main driver for developers to refactor their code bases. Indeed, the percentage of commits belonging to this category account for only 15.9 % of the overall classified refactoring commits, making this class the least among all other categories. Bug Fix is found to be leading with a percentage of 24.3%, but, the sum of the design-related categories, namely Code Smell, Internal Quality Attribute, and External Quality Attribute, 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.

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 distri-

bution of its corresponding refactoring, along with our understanding of their usage context.

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

Figure 5: Commit message stating the restructuring of code to improve its structure.

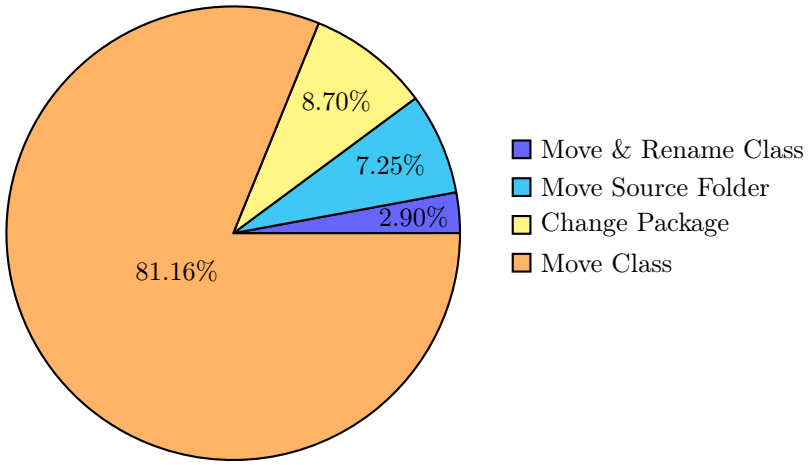


Figure 6: Distribution of refactoring operations.

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 is documented in the commit message and the actual size of refactoring operations.

Closer inspection of the nature and type of the 69 refactorings and the corresponding source code shows that the GitHub commit author massively op-

timized the package structure within existing modularizations. Particularly, as Figure 6 shows, the developer performed four refactoring types, namely, *Move Class, Change Package, Move Source Folder*, and *Move and Rename Class*. A percentage of 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[4], *Change Package* refactoring involves several package-level refactorings (i.e., Rename, Move, Split, and Merge packages).

We observe that the developer is 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 behavior 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 to identify 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 5). 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

[4] `https://github.com/tsantalis/RefactoringMiner`

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

Figure 7: Commit message stating the removal of duplicate code.

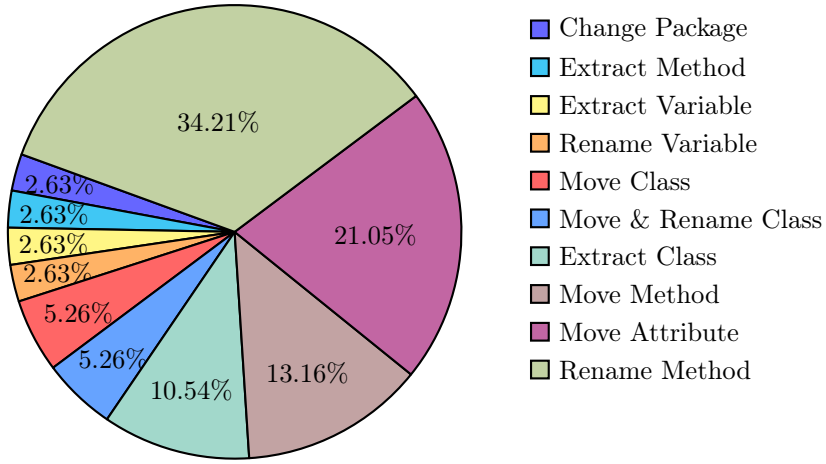


Figure 8: Distribution of refactoring operations.

This case study illustrates another developer's perception of refactoring which is mainly about code smell resolution. Figure 7 shows that the developer performed large-scale refactoring to eliminate duplicated code. Generally, code duplication belongs to the "Dispensable" code smell category, i.e., code fragments that are unneeded and whose absence would make the code cleaner and more efficient.

Figure 8 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 7: *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

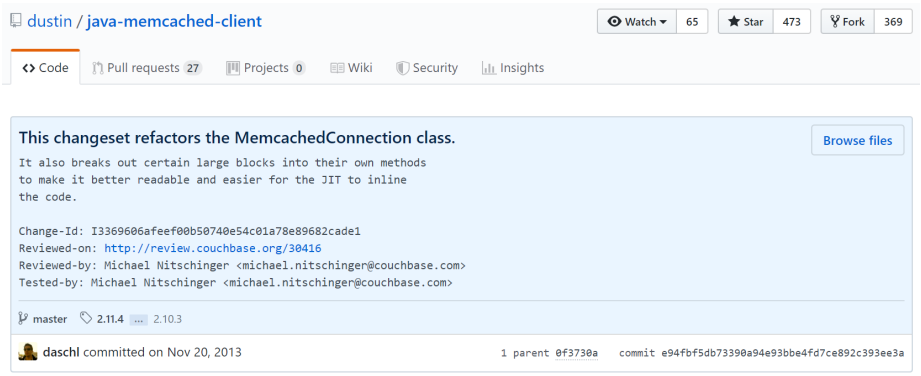


Figure 9: Commit message stating the refactoring to improve code readability.

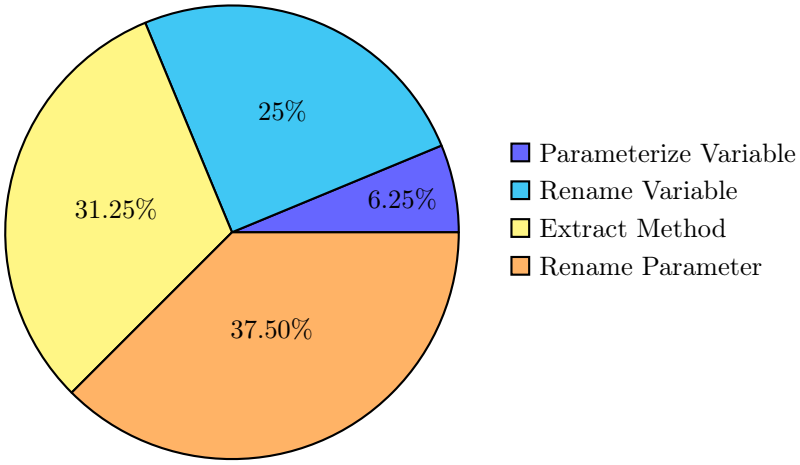


Figure 10: Distribution of refactoring operations.

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 9, 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 10 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

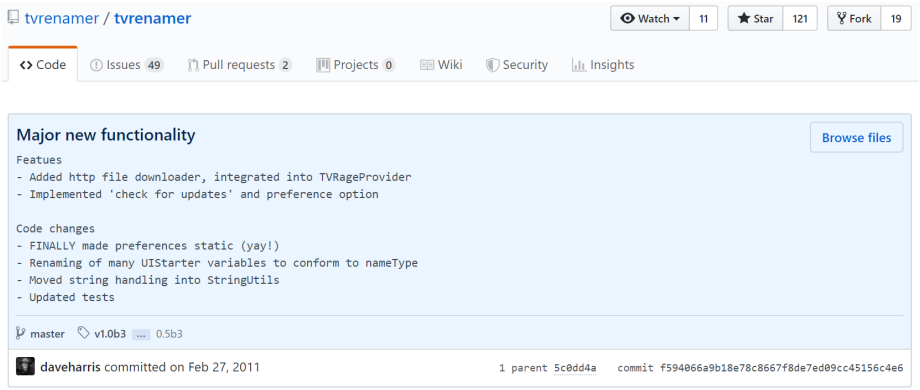


Figure 11: Commit message stating the addition of a new functionality.

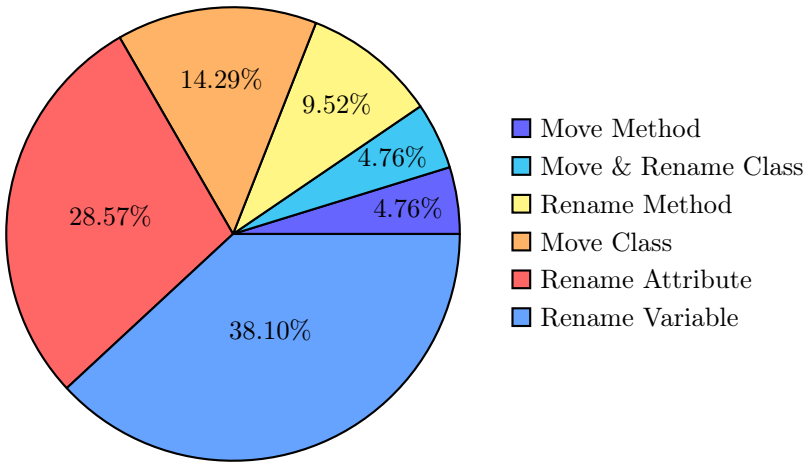


Figure 12: Distribution of refactoring operations.

This case study discusses another motivation of refactoring that is different than the traditional design improvement motivation. As shown in Figure 11, 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 12 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

Figure 13: Commit message stating the correction of user interface related bugs.

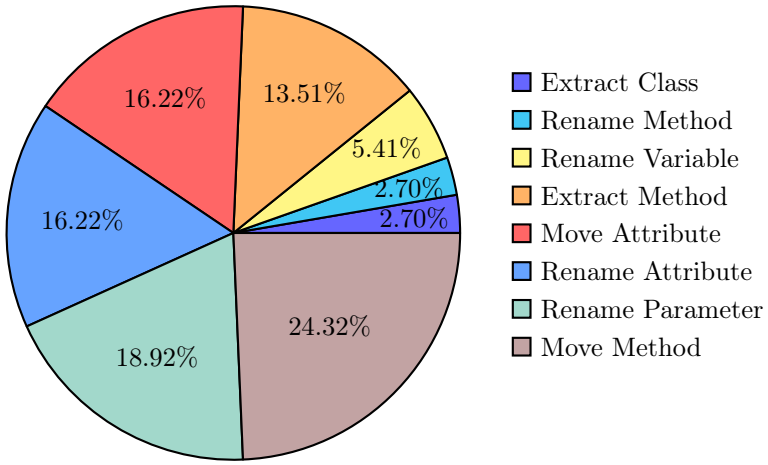


Figure 14: Distribution of refactoring operations.

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 13) 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 behavior 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 behavior 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?

In Table 7, 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 topmost refactoring operations performed across all refactoring motivations is *Move Class* refactoring, except for Bug Fix in which *Rename Attribute* is the highest performed refactoring. In the case of internal quality attribute-related motivations,

Table 7: 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%)	**120376 (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%)

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?

In this research question, we explore the set of 513 potential self-affirmed refactoring candidates, extracted by manual inspection from commits messages and categories top 100 features. We classify these SAR candidates into two tables: Table 8 contains generic candidate patterns that were found across categories; Table 9 contains candidate patterns that are specific to each category.

Table 8: 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**.

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

Table 9: 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**.

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		

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*.

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 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 is approximately close (118 for refactoring commits, and 120 for non-refactoring commits).

Once the set of non-refactoring commit messages constructed, for each keyword, we calculate its occurrence per project for both corpora. This generates vector of 800 occurrences per corpus. Each vector dimension contains a positive number representing the keyword occurrence for a project, and zero otherwise. Figure 15 illustrates occurrences violin plots of the keyword *"refactor"* in both corpora.

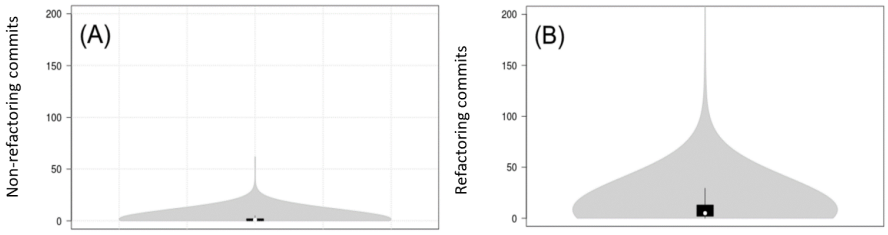


Figure 15: Violin plots representing the occurrence of *refactor* keyword in (A) non-refactoring corpus vs. (B) refactoring corpus.

While it is observed in Figure 15 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 of commits in the second group. We formulate the comparison of each keyword occurrence corpora by defining the alternative 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 vectors. We report our findings in Table 8 and 9 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**" (Stroggylos & Spinellis, 2007; Ratzinger et al., 2008; Ratzinger, 2007; Murphy-Hill et al., 2012; Kim et al., 2014), 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 also be 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 10 enumerates the top-20 keywords, sorted by the percentage of projects they were located in.

- The keyword *refactor* was also used in non-refactoring commit messages. This can be explained either by its occasional misuse, like some previous studies found, or by the existence of refactoring operations that were not identified by the tool we are using. Yet, the frequency of misuse of this popular pattern remains significant in the refactoring-related commit messages (i.e., $p < 0.05$).

- 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 the rationale behind our classification in the first research question.

- Furthermore, our classification has revealed the existence of patterns that are used in specific categories (i.e., 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 11 contains a summary of category-specific patterns that we manually identified. These keywords are found relevant based on how previous studies have been identifying refactoring opportunities (removing code smells, improving structural metrics, optimizing external quality attributes like performance etc.). Note that, in Table 11, we did not quantify the frequency of these patterns, and we plan on the future to further analyze their popularity, along with the type of refactoring operations that are mostly used with their existence, similarly to previous empirical studies (Bavota et al., 2013, 2015).

- Developers occasionally mention the refactoring operation(s) they perform. The Mann-Whitney U test accepted the alternative 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* does exist in 787 projects (98.37%, fourth 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

Table 10: Top generic refactoring 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%)

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.

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 16 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.

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

Table 11: 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	

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 were 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.

In order to compare the number of refactorings identified for each set, i.e.,

Figure 16: Sample of SAR patterns identified in our study.

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 of the occurrence of the remaining patterns).

Figure 17 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 result 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., 2014).

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 corroborates 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,

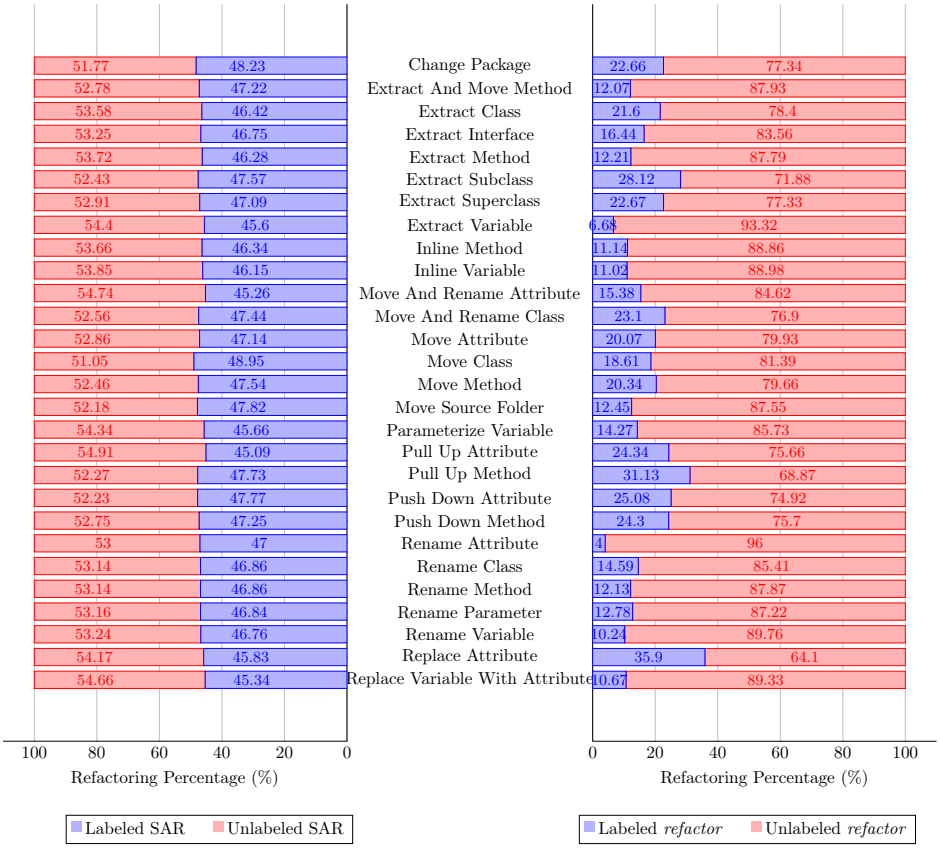


Figure 17: Distribution of refactoring operations for commits labeled and unlabeled SAR (left side) and commits labeled and unlabeled *refactor* (right side).

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. Discussions and Implications

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 findings show that renames are dominant, for test files, across all categories (Table 7). 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 (Peruma et al., 2018, 2019b; Arnaoudova et al., 2013, 2014; Liu et al., 2013).

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. (Negara et al., 2013; Murphy-Hill & Black, 2008) have shown that refactorings are frequently applied manually instead of automatically. 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 (Newman et al., 2018; Li & Thompson, 2012). 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 (Tufano et al., 2016). Like this example, 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 (Tsantalis et al., 2008; Bavota et al., 2013), or improving the design structural measurements (Mkaouer et al., 2014; Bavota 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 proposal, of 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 8, 9, and 11 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., 2017). 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 8, 9, and 11) which would represent concrete examples of common ways to document refactoring activities in commit messages. This catalog of SAR patterns can encourage developers to follow best documentation patterns and 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 & Weimer, 2010; Linares-Vásquez et al., 2015; Liu et al., 2018).

Refactoring and Developer's Experience. While refactoring is being applied by various developers (AlOmar et al., 2020b), 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., 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 experience plays a role in the introduction of SARs or not. 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).

Refactoring Automation. There have been various studies targeting the automation of refactoring (Harman & Tratt, 2007; Simons et al., 2015; Mkaouer et al., 2015; Lin et al., 2016; Mkaouer et al., 2016). They mainly rely on the correspondence between the impact of refactoring on the source code to guide the generation of code changes that will potentially improve it. Therefore, existing studies heavily rely on structural measurements to guide the search for these code changes, and so, improving quality attributes and removing anti-patterns were the main drivers for automated refactoring. Clearly, the challenge facing such approaches is applicability. Performing large-scale code changes, impacting various components in the source code, may be catchy for its quality, but it also drastically disturbs the existing software design. Although developers are in favor for optimizing the quality of their software, they still want to recognize their own design.

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. Additionally, 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 3) 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 (Yan et al., 2016; Kirinuki et al., 2014) 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. In this context, statistical tests have been used to test the significance of the results gained. Specifically, we applied the Wilcoxon test and the Mann-Whitney U test, widely used non-parametric tests, to test whether refactoring patterns are significant or not, and to test the occurrence of *refactor* in refactoring commits and non-refactoring commits, respectively. These tests make 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 pose 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

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, and (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 also refactoring motivators for developers, researchers are encouraged to adopt a maintenance-related refactoring beside design-related refactoring when building a refactoring tool in the future.

References

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

AlDallal, J., & Abdin, A. (2017). Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, *PP*, 1–1. doi:10.1109/TSE.2017.2658573.

Alkadhi, R., Nonnenmacher, M., Guzman, E., & Bruegge, B. (2018). How do developers discuss rationale? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 357–369). IEEE.

AlOmar, E. A. (2020 (last accessed October 20, 2020)). *self-affirmed-refactoring repository*. URL: https://smilevo.github.io/self-affirmed-refactoring/.

AlOmar, E. A., Mkaouer, M. W., & Ouni, A. (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*.

AlOmar, E. A., Mkaouer, M. W., & Ouni, A. (2020a). Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, (p. 110821).

AlOmar, E. A., Mkaouer, M. W., Ouni, A., & Kessentini, M. (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)* (pp. 1–11). IEEE.

AlOmar, E. A., Peruma, A., Newman, C. D., Mkaouer, M. W., & Ouni, A. (2020b). 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. New York, NY, USA: Association for Computing Machinery.

AlOmar, E. A., Rodriguez, P. T., , J., Bowman, Wang, T., Adepoju, B., Lopez, K., Newman, C. D., Ouni, A., & Mkaouer, M. W. (2020c). How do developers refactor code to improve code reusability? In *International Conference on Software and Systems Reuse*. Springer.

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

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

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

Arnaoudova, V., Eshkevari, L. M., Di Penta, M., Oliveto, R., Antoniol, G., & Gueheneuc, Y.-G. (2014). Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, *40*, 502–532.

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

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

Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2013). An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 692–701). IEEE Press.

Bavota, G., Panichella, S., Tsantalis, N., Di Penta, M., Oliveto, R., & Canfora, G. (2014). Recommending refactorings based on team co-maintenance patterns. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (pp. 337–342). ACM.

Bird, S. (2002). Nltk: The natural language toolkit. *ArXiv*, *cs.CL/0205028*.

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

Breiman, L. (2017). *Classification and Regression Trees*. CRC Press.

Brownlee, J. (2018). *Statistical Methods for Machine Learning: Discover how to Transform Data into Knowledge with Python*. Machine Learning Mastery. URL: https://books.google.com/books?id=386nDwAAQBAJ.

Buse, R. P., & Weimer, W. (2010). Automatically documenting program changes. In *ASE* (pp. 33–42). volume 10.

Cedrim, D., Sousa, L., Garcia, A., & Gheyi, R. (2016). Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering* (pp. 73–82). ACM.

Chang, C.-C., & Lin, C.-J. (2011). Libsvm: A library for support vector machines, . *2*. URL: https://doi.org/10.1145/1961189.1961199. doi:10.1145/1961189.1961199.

Dangeti, P. (2017). *Statistics for Machine Learning*. Packt Publishing.

Deng, N., Tian, Y., & Zhang, C. (2012). *Support Vector Machines: Optimization Based Theory, Algorithms, and Extensions*. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series. Taylor & Francis.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, *10*, 1895–1923.

Dig, D., Comertoglu, C., Marinov, D., & Johnson, R. (2006). Automated detection of refactorings in evolving components. In D. Thomas (Ed.), *ECOOP 2006 – Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings* (pp. 404–428). Berlin, Heidelberg: Springer Berlin Heidelberg. URL: https://doi.org/10.1007/11785477_24. doi:10.1007/11785477_24.

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

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

Harman, M., & Tratt, L. (2007). Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (pp. 1106–1113). ACM.

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

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

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

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

Hönel, S., Ericsson, M., Löwe, W., & Wingkvist, A. (2019). Importance and aptitude of source code density for commit classification into maintenance activities. In *The 19th IEEE International Conference on Software Quality, Reliability, and Security*.

Jurafsky, D., & Martin, J. H. (2019). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. *Prentic e Hall*, .

Kim, M., Gee, M., Loh, A., & Rachatasumrit, N. (2010). Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (pp. 371–372). ACM.

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

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

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

Kowsari, K., Jafari Meimandi, K., Heidarysafa, M., Mendu, S., Barnes, L., & Brown, D. (2019). Text classification algorithms: A survey. *Information*, *10*, 150.

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

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

Le, T.-D. B., Linares-Vásquez, M., Lo, D., & Poshyvanyk, D. (2015). Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *2015 IEEE 23rd International Conference on Program Comprehension* (pp. 36–47). IEEE.

Levin, S., & Yehudai, A. (2017). Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering* PROMISE (pp. 97–106). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/3127005.3127016. doi:10.1145/3127005.3127016.

Li, H., & Thompson, S. (2012). Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools* WRT '12 (pp. 32–39). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2328876.2328881. doi:10.1145/2328876.2328881.

Lin, S., Ma, Y., & Chen, J. (2013). Empirical evidence on developer's commit activity for open-source software projects. In *SEKE* (pp. 455–460). volume 13.

Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., & Zhao, W. (2016). Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 535–546). ACM.

Linares-Vásquez, M., Cortés-Coy, L. F., Aponte, J., & Poshyvanyk, D. (2015). Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (pp. 709–712). IEEE volume 2.

Liu, H., Guo, X., & Shao, W. (2013). Monitor-based instant software refactoring. *IEEE Transactions on Software Engineering*, *39*, 1112–1126. doi:10.1109/TSE.2013.4.

Liu, Z., Xia, X., Hassan, A. E., Lo, D., Xing, Z., & Wang, X. (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* (pp. 373–384). ACM.

Manning, C., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

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

Mauczka, A., Huber, M., Schanes, C., Schramm, W., Bernhart, M., & Grechenig, T. (2012). Tracing your maintenance work – a cross-project validation of an automated classification dictionary for commit messages. In J. de Lara, & A. Zisman (Eds.), *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (pp. 301–315). Berlin, Heidelberg: Springer Berlin Heidelberg. URL: https://doi.org/10.1007/978-3-642-28872-2_21. doi:10.1007/978-3-642-28872-2_21.

McBurney, P. W., Jiang, S., Kessentini, M., Kraft, N. A., Armaly, A., Mkaouer, M. W., & McMillan, C. (2017). Towards prioritizing documentation effort. *IEEE Transactions on Software Engineering*, *44*, 897–913.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Cinnéide, M. Ó., & Deb, K. (2016). On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, *21*, 2503–2545.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., & Ó Cinnéide, M. (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* (pp. 331–336). ACM.

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

Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., & Succi, G. (2007). A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering* (pp. 252–266). Springer.

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does refactoring improve reusability? In *International Conference on Software Reuse* (pp. 287–297). Springer.

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

Murphy-Hill, E., & Black, A. P. (2008). Refactoring tools: Fitness for purpose. *IEEE Software*, *25*, 38–44. doi:10.1109/MS.2008.123.

Murphy-Hill, E., Black, A. P., Dig, D., & Parnin, C. (2008). Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools* (p. 7). ACM.

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

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., & Dig, D. (2013). A comparative study of manual and automated refactorings. In G. Castagna (Ed.), *ECOOP 2013 – Object-Oriented Programming* (pp. 552–576). Berlin, Heidelberg: Springer Berlin Heidelberg.

Newman, C. D., Mkaouer, M. W., Collard, M. L., & Maletic, J. I. (2018). A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring* (pp. 34–41). ACM.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., & Arvonio, E. (2020). Behind the intents: An in-depth empirical study on software refactoring in modern code review. *17th MSR*, .

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

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., & Di Penta, M. (2020). Why developers refactor source code: A mining-based study, .

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., & Palomba, F. (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 (p. 193–202). USA: IBM Corp.

Peruma, A., Mkaouer, M. W., Decker, M. J., & Newman, C. D. (2018). An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring* (pp. 26–33). ACM.

Peruma, A., Mkaouer, M. W., Decker, M. J., & Newman, C. D. (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*.

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

Ratzinger, J. (2007). *sPACE: Software Project Assessment in the Course of Evolution*. Ph.D. thesis. URL: http://www.infosys.tuwien.ac.at/Staff/ratzinger/publications/ratzinger_phd-thesis_space.pdf.

Ratzinger, J., Sigmund, T., & Gall, H. C. (2008). On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* MSR '08 (pp. 35–38). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1370750.1370759. doi:10.1145/1370750.1370759.

Silva, D., Tsantalis, N., & Valente, M. T. (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 (pp. 858–870). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2950290.2950305. doi:10.1145/2950290.2950305.

Simons, C., Singer, J., & White, D. R. (2015). Search-based refactoring: Metrics are not enough. In *International Symposium on Search Based Software Engineering* (pp. 47–61). Springer.

Singh, R., & Mangat, N. (2013). *Elements of Survey Sampling*. Texts in the Mathematical Sciences. Springer Netherlands.

SKlearn (2007a). 1.12. multiclass and multilabel algorithms — scikit-learn 0.23.2 documentation. https://scikit-learn.org/stable/modules/multiclass.html.

SKlearn (2007b). sklearn.svm.svc — scikit-learn 0.23.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklear.svm.SVC.

Soares, G., Cavalcanti, D., Gheyi, R., Massoni, T., Serey, D., & Cornélio, M. (2009). Saferefactor-tool for checking refactoring safety. *Tools Session at SBES*, (pp. 49–54).

Soares, G., Gheyi, R., Murphy-Hill, E., & Johnson, B. (2013). Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, *86*, 1006–1022.

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

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

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

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

Tan, A.-H. et al. (1999). Text mining: The state of the art and the challenges. In *Proceedings of the PAKDD 1999 Workshop on Knowledge Disocovery from Advanced Databases* (pp. 65–70). sn volume 8.

Tan, C.-M., Wang, Y.-F., & Lee, C.-D. (2002). The use of bigrams to enhance text categorization. *Information processing & management*, *38*, 529–546.

Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering* (pp. 329–331). IEEE.

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

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

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 483–494). ACM.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (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 (pp. 4–15). New York, NY, USA: ACM. doi:10.1145/2970276.2970340.

Vassallo, C., Grano, G., Palomba, F., Gall, H. C., & Bacchelli, A. (2019). A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, *180*, 1–15.

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

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Xia, X., Lo, D., Wang, X., & Yang, X. (2016). Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, *65*, 1810–1829.

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

Zhang, D., Li, B., Li, Z., & Liang, P. (2018). A preliminary investigation of self-admitted refactorings in open source software. doi:10.18293/SEKE2018-081.

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