Can Refactoring be Self-Affirmed? An Exploratory Study on How Developers Document their Refactoring Activities in Commit Messages

Eman Abdullah AlOmar*, Mohamed Wiem Mkaouer*, Ali Ouni†
*Software Engineering Department, Rochester Institute of Technology, NY, USA
†ETS Montreal, University of Quebec, Montreal, QC, Canada
eman.alomar@mail.rit.edu, mwmvse@rit.edu, ali.ouni@etsmtl.ca

***Abstract*—Refactoring is a critical task in software maintenance and is usually performed to enforce best design practices, or to cope with design defects. Previous studies heavily rely on defining a set of keywords to identify refactoring commits from a list of general commits extracted from a small set of software systems. All approaches thus far consider all commits without checking whether refactorings had actually happened or not. In this paper, we aim at exploring how developers document their refactoring activities during the software life cycle. We call such activity Self-Affirmed Refactoring, which is an indication of the developer-related refactoring events in the commit messages. Our approach relies on text mining refactoring-related change messages and identifying refactoring patterns by only considering refactoring commits. We found that (1) developers use a variety of patterns to purposefully target refactoring-related activities; (2) developers tend to explicitly mention the improvement of specific quality attributes and code smells; and (3) commit messages with self-affirmed refactoring patterns tend to have more significant refactoring activity than those without.**

***Index Terms*—Self-Affirmed Refactoring, Software Quality, Mining Software Repositories.**

I. Introduction

Refactoring is the art of improving the quality of software design without altering its behavior. With the rise of agile methodologies that encourage developers to interleave refactoring within their other development activities, and with the incorporation of refactoring operations in modern Integrated Development Environments (IDEs), there is a lot of growing research to better understand how developers practically refactor their codebases [13]–[15]. Thus, several studies focus on detecting refactoring operations, performed by developers, by mining their commit changes and extracting the refactoring history [10], [19], [21]. These refactoring detectors rely mainly on analyzing code changes to identify refactorings strategies, previously performed by developers in various development contexts.

In order to learn from these refactoring strategies, it is essential to also understand the developer's rationale and intent behind applying them, *i.e.*, the context in which the refactoring operations were executed. Existing studies on understanding developers perception of refactorings mainly rely on developers surveys and formal interviews [7], [11]. As the existing refactoring detectors offer an abundant source of commits containing refactoring operations, this paper aims at exploring how developers document their refactoring activities during the software life-cycle.

Inspired by various studies in analyzing the developer's internal documentations to extract their perception of their own code, *e.g.*, self-admitted technical debt [5], [16], we text-mine the developer's messages in refactoring-related commits to detect any potentially relevant information regarding the applied refactorings. Indeed, commit messages represent an atomic documentation of a code change, written by the change author, and thus represents a reliable and rich source of information to describe their intention behind the performed changes. Therefore, we conduct this empirical study to identify how developers describe their refactoring activities. Then we extract the rationale behind the applied refactorings, *e.g.*, fixing code smells or improving specific quality attributes. To perform this analysis, we formulate the following research questions:

RQ1. *What patterns do developers use to describe their refactoring activities?* Since there is no consensus on how to formally document the act of refactoring code, we mine in this research questions, patterns, using which developers have described their refactoring activities. We explore 322,479 commit messages, belonging to a large variety of projects. The outcome of this research question enumerates the most popular text patterns used in the analyzed commit messages.

RQ2. *What are the quality issues that drive developers to refactor?* Various studies have explored the bad programming practices that trigger refactoring and the potential quality attributes that are optimized when restructuring the code. In this research question, we investigate whether developers explicitly mention the purpose of their refactoring activity, *e.g.,* improving structural metrics of fixing code smells.

RQ3. *What are the top-10 patterns developers use to describe quality issues in their commits?* In this research questions, we link between patterns extracted from the first research question and the quality issues found in the second question. We explore how developers express combining them to express their refactoring activity.

RQ4. *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. [13] about whether developers use a specific pattern, *i.e., "refactor"*

978-1-7281-2270-0/19/$31.00 ©2019 IEEE
DOI 10.1109/IWoR.2019.00017

when describing their refactoring activities. We quantify the messages with the label *"refactor"* and without to compare between them.

The remainder of this paper is organized as follows. Section II discusses the rationale of our study. Section III reviews the relevant studies to refactoring detection and documentation. Section IV outlines our experimental methodology in collecting the necessary refactoring data for the experiments that are discussed afterward in Section V. Section VI gathers potential limitations to the validity of our empirical analysis before concluding and describing our work directions in Section VII.

II. SELF-AFFIRMED REFACTORING

Commit messages represent the human translation, in natural language, of the code-level changes. Therefore, with the raise of version control systems and mining software repositories, several studies have been analyzing commit messages for various purposes including change classification [8], [9], change bug-proneness [26], and developers rationale behind their coding decisions [2].

In this context, we aim to extract how developers express their nonfunctional activities, namely improving software design, renaming semantically ambiguous identifiers, removing code redundancies etc. Multiple studies have been detecting the performed refactoring operations, *e.g.*, rename class, move method etc. within committed changes to better understand how developers cope with bad design decisions, also known as design antipatterns, and to extract their removal strategy through the selection of the appropriate set of refactoring operations [25]. As the accuracy of refactoring detectors has reached a relatively high rate, the mined commits represent a rich space to understand how developers describe, in natural language, their refactoring activities. Yet, such information retrieval can be challenging since there are no common standards on how developers should be formally documenting their refactorings, besides inheriting all the challenges related to natural language processing [24].

However, using the developer inline documentation has added another dimension to better understanding software quality, as mining developers comments, for instance, has unveiled how developers knowingly commit code that is either incomplete, temporary, error-prone. These situations have been coined as *Self-Admitted Technical Debt* (SATD) [16], as they are extracted from text-mining developers messages, which represents a reliable source, instead of measuring technical debt only by the deviations from ideal code, *i.e.*, code smells. Inspired by the study of Potdar and Shihab [16], we analyze commits which are known to contain refactoring operations and we extract how developers describe, in plain unformal text, their refactoring activities. However, we were skeptical about using the term self-admitted refactoring, as self-admission is defined as confessing a specific charge or accusation, which makes it appropriate for technical debt, but not for what may reduce it *i.e.*, refactoring. Therefore, we label developers documented refactorings as *Self-Affirmed Refactoring* (SAR), since self-affirmation is defined as the assertion of self-existence [22], which is, in the context of refactoring, refers to developers recognition of the value of refactoring activities as means to improve their code and more specifically cope with technical debt, if we refer to its original definition [4].

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

III. RELATED WORK

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 [23] opted for searching words stemming from the verb *"refactor"* such as "refactoring" or "refactored" to identify refactoring-related commits. Ratzinger et al. [17], [18] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions to identify whether a transformation contains refactoring. The authors identified refactorings based on a set of keywords detected in commit messages, and focusing, in particular, on the following 13 terms in their search approach: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move*.

Later, Murphy-Hill et al. [14] 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. [21] compared and evaluated three approaches, namely, manual analysis, commit message (Ratzinger et al.'s approach [17], [18]), and dynamic analysis (SafeRefactor approach [20]) 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. [11] 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

Table I: Refactoring Identification in Related Work.

Study	Year	Purpose	Approach	Source of Info.	Ref. Patterns
Stroggylos & Spinellis [23]	2007	Identify refactoring commits	Mining commit logs	General commits	1 keyword
Ratzinger, Ratzinger et al. [18] [17]	2007 & 2008	Identify refactoring commits	Mining commit logs	General commits	13 keywords
Murphy-Hill et al. [14]	2012	Identify refactoring commits	Ratzinger's approach	General commits	13 keywords
Soares et al. [21]	2013	Analyze refactoring activity	Ratzinger's approach Manual analysis Dynamic analysis	General commits	13 keywords
Kim et al. [11]	2014	Identify refactoring commits	Identifying refactoring branches Mining commit logs	Refactoring branch	Top 10 keywords
Zhang et al. [27]	2018	Identify refactoring commits	Mining commit logs	General commits	22 keywords
This work		**Identify refactoring patterns**	**Detecting refactorings** **Extracting commit messages**	**Refactoring commits**	**87 keywords & phrases**

refactoring-related keywords identified from the survey (*refactor, clean-up, rewrite, restructure, redesign, move, extract, improve, split, reorganize, rename*) against the commit messages to identify refactoring commits from version histories. Using this approach, they found 94.29% of commits do not have any of the keywords, and only 5.76% of commits included refactoring-related keywords.

More recently, Zhang et al. [27] 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 [6] as a basis for SAR identification. After identifying candidate SARs, they used Ref-Finder [10] to validate whether refactorings have been applied. In their work, they used code smells to assess the impact of SAR on the structural quality of the source code. Their main findings are the following (1) SAR tend to enhance the software quality although there is a small percentage of SAR that have introduced code smells, and (2) the most frequent code smells that are introduced or reduced depend highly on the nature of the studied projects. They concluded that SAR is a signal that helps finding refactoring events, but it does not guarantee the application of refactorings. We summarize these state-of-the-art approaches in Table I.

As seen in the table, except for [11], all of the above-mentioned studies heavily rely on defining a set of keywords to identify refactoring commits from a list of general commits extracted from a small set of software systems.

Our work is different from these papers since we include a large set of projects and consider only commits identified by Refactoring Miner [25] to contain at least one refactoring operation. Moreover, our work does not only rely on a single keyword to identify refactoring activities, instead we identify reported keywords and phrases, which we call patterns, to capture better developers' refactoring taxonomy. The restricted set of commits has significantly narrowed our search space and constrain us to extract expressions that are most likely to describe refactoring activities, which increases our confidence as we identify them. Unlike our study, prior works merely identified refactoring commits by considering all commits without applying any form of verification. A keyword-based identification of refactoring commits, using predefined keywords, may thus miss certain keywords, as developers may use various expression to annotate how they refactor.

IV. Empirical Setup

To answer our research questions, we conducted a two-phased approach. The initial phase consists of selecting a large number open-source Java projects and detecting refactorings that occur throughout the development history, *i.e.*, code changes, of each considered project. The second phase consists of analyzing the commit messages as a mean of identifying self-affirmed refactoring patterns. An overview of the experiment methodology is depicted in Figure 1.

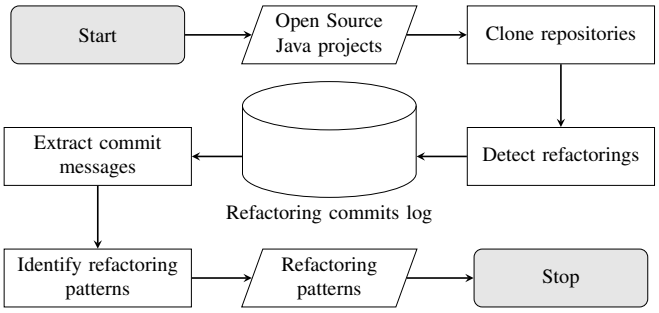

Figure 1: Approach Overview.

A. Data Collection & Refactoring Detection

To perform our experimental study, we utilize an existing benchmark of GitHub repositories by Allamanis [3]. To extract the entire refactoring history in each project, we use the Refactoring Miner tool, developed by Tsantalis et al. [25]. Refactoring Miner is designed to analyze code changes in git repositories to detect applied refactoring. Our choice to use Refactoring Miner is justified by the fact that it achieved accurate results in detecting refactorings compared to the state-of-the-art available tools, with a precision of 98% and recall of 87% [19], [25]. Refactoring Miner seems suitable for a study that requires a high degree of automation since it can be used through its external API. In this phase, Refactoring Miner detected 1,208,970 refactoring operations in 3,795 projects. An overview of the studied benchmark is provided in Table II.

B. Self-Affirmed Refactoring Analysis

After extracting all refactoring commit messages detected by Refactoring Miner, our next step consists of analyzing each of the commit messages. As for pattern identification, we were

inspired by the manual analysis of Potdar and Shihab [16] when analyzing comments containing self-admitted technical debt. Similarly, since commit messages are written in natural language and we need to understand how developers express refactoring, we manually analyzed commit messages by reading through each message to identify self-affirmed refactorings. We then extracted these commit comments to specific patterns (*i.e.*, a keyword or phrase). To avoid redundancy of any kind of patterns, we only considered one phrase if we found different forms of patterns that have the same meaning. For example, if we find patterns such as "Simplifying the code", "Code simplification", and "simplify code", we add only one of these similar phrases in the list of patterns. This enables us to have a list of the most insightful and unique patterns. It also helps in making more concise patterns that are usable for readers. The manual analysis process took approximately 7 days in total, and was performed by the authors of the paper. In total, we read through 58,131 commit messages and ended up with a set of 87 recurring patterns identified across 3,795 projects.

Table II: Studied Benchmark Statistics.

Item	**Count**
Studied projects	3,795
Refactoring commits	322,479
Refactoring operations	1,208,970
Analyzed Projects - Refactored Code Elements	
Code Element	**# of Refactorings**
Class	329,378
Method	718,335
Attribute	97,516
Package	18,334
Interface	8,096

V. Results & Discussion

In this section, we report and discuss our findings for analyzing the identified refactoring-related patterns to answer our four research questions *RQ1-4*.

A. RQ1. What patterns do developers use to describe their refactoring activities?

To identify self-affirmed refactoring patterns, we manually inspect a subset of change messages, *i.e.*, commits, and categorize these change messages into lexically or semantically similar patterns. These patterns are represented in the form of a keyword or phrase that frequently occur in the comments of all refactoring-related commits. The extraction of our approach has been carried through few iterations. We start our first iteration by searching for the term *"refactor*"* (we use * to capture extensions like *refactors*, *refactoring* etc.). The choice of *"refactor"*, besides being used by all related studies, is intuitively the first term to identify ideal commit messages. In this iteration, we obtained 33,301 refactoring commit messages. Then, we started a manual inspection of each commit message that are associated with the term *"refactor"* to the set of patterns that are also used to describe the refactoring activity. As developers may not always use the term *"refactor"* explicitly to document/describe their refactoring activities in their commit messages. Thus, to alleviate this issue, we reiterate again, using the extracted patterns in the first iteration, while excluding the term *"refactor"*, to identify additional self-affirmed refactoring patterns. We kept iterating by extracting new patterns while excluding the previously identified ones until we are no longer able to find any relevant patterns. Our in-depth inspection resulted into a list of 87 self-affirmed refactoring patterns identified across the considered projects, as illustrated in Table III.

Table III: List of Self-Affirmed Refacoring (SAR) Patterns.

Patterns		
(1) Refactor*	(30) Removed poor coding practice	(59) Change design
(2) Mov*	(31) Improve naming consistency	(60) Modularize the code
(3) Split*	(32) Removing unused classes	(61) Code cosmetics
(4) Fix*	(33) Pull some code up	(62) Moved more code out of
(5) Introduc*	(34) Use better name	(63) Remove dependency
(6) Decompos*	(35) Replace it with	(64) Enhanced code beauty
(7) Reorganiz*	(36) Make maintenance easier	(65) Simplify internal design
(8) Extract*	(37) Code cleanup	(66) Change package structure
(9) Merg*	(38) Minor Simplification	(67) Use a safer method
(10) Renam*	(39) Reorganize project structures	(68) Code improvements
(11) Chang*	(40) Code maintenance for refactoring	(69) Minor enhancement
(12) Restructur*	(41) Remove redundant code	(70) Get rid of unused code
(13) Reformat*	(42) Moved and gave clearer names to	(71) Fixing naming convention
(14) Extend*	(43) Refactor bad designed code	(72) Fix module structure
(15) Remov*	(44) Getting code out of	(73) Code optimization
(16) Replac*	(45) Deleting a lot of old stuff	(74) Fix a design flaw
(17) Rewrit*	(46) Code revision	(75) Nonfunctional code cleanup
(18) Simplify*	(47) Fix technical debt	(76) Improve code quality
(19) Creat*	(48) Fix quality issue	(77) Fix code smell
(20) Improv*	(49) Antipattern bad for performances	(78) Use less code
(21) Add*	(50) Major/Minor structural changes	(79) Avoid future confusion
(22) Modify*	(51) Clean up unnecessary code	(80) More easily extended
(23) Enhanc*	(52) Code reformatting & reordering	(81) Polishing code
(24) Rework*	(53) Nicer code / formatted / structure	(82) Move unused file away
(25) Inlin*	(54) Simplify code redundancies	(83) Many cosmetic changes
(26) Redesign*	(55) Added more checks for quality factors	(84) Inlined unnecessary classes
(27) Cleanup	(56) Naming improvements	(85) Code cleansing
(28) Reduc*	(57) Renamed for consistency	(86) Fix quality flaws
(29) Encapsulat*	(58) Refactoring towards nicer name analysis	(87) Simplify the code

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 occasionally state the motivation behind refactoring, and in some of these patterns that are not restricted only to fixing code smells, as in the original definition of refactoring in the Fowler's book [6].For instance, developers tend often to improve certain non-functional attributes such as the readability and testability of the source code. These observations are aligned with state-of-the-art studies by Kim et al. [11] and Silva et al. [19]. We also observe that developers tend to report the executed refactoring operations using keywords such as *"move"*, *"rename"* or *"extract"* as shown in Table III.

Furthermore, we found that our identified patterns include all of the keywords identified by Kim et al. [11] and mostly matched keywords introduced by Ratzinger [18] (cf. Table III).

Table IV: Quality Issues (Quality Attribute(s) & Code Smell(s)) Extracted from SAR Commits.

Internal QA (%)	External QA (%)	Code Smell (%)
Inheritance (31.04%)	Functionality (34.03%)	Duplicate Code (43.52%)
Abstraction (30.63%)	Performance (31.37%)	Dead Code (24.84%)
Complexity (14.30%)	Compatibility (13.61%)	Data Class (22.93%)
Composition (12.53%)	Readability (3.60%)	Long Method (3.82%)
Coupling (3.81%)	Stability (2.64%)	Switch Statement (3.18%)
Encapsulation (3.61%)	Usability (1.60%)	Lazy Class (0.42%)
Design Size (2.11%)	Flexibility (1.58%)	Too Many Parameters (0.42%)
Polymorphism (1.50%)	Extensibility (1.54%)	Primitive Obsession (0.21%)
Cohesion (0.48%)	Efficiency (1.51%)	Feature Envy (0.21%)
	Accuracy (1.05%)	Blob Class (0.21%)
	Accessibility (1.04%)	Blob Operation (0.21%)
	Robustness (0.78%)	
	Testability (0.75%)	
	Correctness (0.65%)	
	Scalability (0.62%)	
	Configurability (0.56%)	
	Simplicity (0.55%)	
	Reusability (0.45%)	
	Reliability (0.43%)	
	Modularity (0.37%)	
	Maintainability (0.26%)	
	Traceability (0.26%)	
	Interoperability (0.24%)	
	Fault-tolerance (0.16%)	
	Repeatability (0.07%)	
	Understandability (0.06%)	
	Effectiveness (0.06%)	
	Productivity (0.06%)	
	Modifiability (0.03%)	
	Reproducibility (0.03%)	
	Adaptability (0.03%)	
	Manageability (0.01%)	

Summary. Developers tend to use a variety of textual patterns to document their refactoring activities, such as '*refactor*', '*move*', '*extract*', '*reorganize*', and '*fix*'. These patterns could provide either (1) a generic description/motivation of the refactoring activity such as 'optimize the code design', 'improving code *readability*', and 'fix *long method*', or (2) a specific refactoring operation name following Fowler's names such as *'extract method'*, *'extract class'*, and *'extract interface'*.

B. RQ2: What are the quality issues that drive developers to refactor?

After identifying the different SAR patterns, we identify and categorize the patterns used to describe the motivation behind the refactoring operations into three main categories: (1) internal quality attributes, (2) external quality attributes, and (3) code smells. We perform five sequential steps to answer this research question. We start by collecting software issues *i.e.,* quality attributes and code smells reported in the literature [1], [6], [12]. Then, we search for common categories among the reported quality attributes and code smells. The following step involves identifying categories clustering quality attributes under the identified categories. This process resulted in three different categories, namely, internal quality attribute, external quality attribute, and code smell. For each of the collected quality attributes and code smells, we search in our database for any potential refactoring commit that contains any of the collected quality attributes and code smells. To further enrich our results, we randomly select a subset of commit messages that do not contain any of the identified quality issues and we inspect whether these messages contain any quality attribute or code smell that is not present in three categories.

Table IV reports each of these categories ranked based on their frequency. From these results, we notice that developers frequently mention the main internal quality attributes such as *abstraction*, *inheritance*, and *cohesion*, etc. and a wide range of external quality attributes such as *compatibility*, *readability*, *extensibility*, and *functionality*. Developers occasionally mention fixing code smells but rarely state the name of the code smell under correction. Instead, developers tend to use specific phrases to indicate this activity as shown in table III. For example, we found the following patterns to indicate code smell fixings *"fix quality flaws"*, *"fix quality issue"*, or *"antipatterns bad for performance"*.

Internal quality attributes-focused SAR indicate that there is an improvement in the structural design of the code, which could be performed in code commits related to fixing abstraction or inheritance issues, increasing the cohesion or reducing the coupling/complexity of the source code. As examples of the internal quality attribute SARs, we refer to the commit descriptions in two of the analyzed SAR commits as follows : (1) *Small refactoring to reduce cyclomatic complexity of CheckStyleTask.execute()*, from the project Checkstyle[1], and (2) *Reduce coupling between packages*, from the project Visualwas[2]. Thus, we observe that developers do mention their strategy to cope with the flaw, *e.g.,* reduce the complexity of a method.

On the other hand, SAR commits that focus on external quality attributes indicate the enhancement of non-functional attributes such as *readability*, *understandability*, and *testability*. As examples of such patterns, let us consider the three following commit comments : (1) *Refactoring mostly for readability (and small performance improvement)*, from the Gatk[3], (2) *renamed EditorPage to ContextEditorPage for better understandability*, from the project Openengsb[4], and (3) *Refactor plugin management for better maintainability*, from the project Sonar[5]. In these examples, developers explicitly state what non-functional quality attributes are in needed of improvement.

Finally, for code smell-focused SARs, we observe that developers do perform refactorings to their code to eliminate certain code smells (*e.g.*, long method, feature envy, etc.) that are known to deteriorate the quality of the source code. This type of SAR is illustrated in the following change messages : (1) *[CLEANUP] - Split overly long method into smaller chunks*, from the project Pentaho-reporting[6], (2) *refactoring of Abstract*DataSet to delete duplicate code* from Cassandra-

[1]https://github.com/isopov/checkstyle
[2]https://github.com/veithen/visualwas
[3]https://github.com/broadgsa/gatk
[4]https://github.com/openengsb/openengsb-framework
[5]https://github.com/SonarSource/sonar
[6]https://github.com/pentaho/pentaho-reporting

Table V: Top-10 Patterns across Quality Issues.

Rank	Internal QA (%)	External QA (%)	Code Smell (%)
1	Fix (34.66%)	Improve (33.36%)	Remove (39.71%)
2	Refactor (17.05%)	Fix (20.27%)	Refactor (23.28%)
3	Reduce (15.68%)	Refactor (15.72%)	Fix (10.05%)
4	Remove (14.32%)	Add (7.54%)	Move (6.62%)
5	Improve (7.50%)	Better (6.56%)	Rename (4.66%)
6	Modify (3.98 %)	Optimize (4.88%)	Reduce (4.17%)
7	Cleanup (2.50%)	Enhance (4.82%)	Cleanup (3.92%)
8	Simplify (1.93%)	Cleanup (3.34%)	Replace (3.19%)
9	Enhance (1.70%)	Introduce (2.16%)	Avoid (2.45%)
10	Restructure (0.68%)	Simplify (1.36%)	Extract (1.96%)

unit[7], and (3) *Moved data classes to a more suitable package*, from Cdk[8]. A Closer inspection of these commit messages shows that developers intentionally apply refactoring to remove antipatterns that violate design principles and good programming practices.

Figure 2: Top-10 Popular Patterns.

Summary. Our findings indicate that developers frequently state their intention behind the application of refactorings. They address quality issues that can be related to : (1) internal quality attributes, (2) external quality attributes, or (3) code smells. Furthermore, developers occasionally mention their refactoring strategy or operation performed with regard of addressing the quality issue.

C. RQ3: What are the top-10 patterns developers use to describe quality issues in their commits?

Based on the three categories defined in RQ2, we investigate what are the top common keywords, *i.e.*, patterns, that developers use when expressing SAR commits. Table V shows the top 10 keywords used to identify SARs across the studied projects, that are ranked according to their number of occurrences. Based on the obtained mining results, we found that the top ranked keywords for internal quality attribute SARs include *"fix"*, *"refactor"*, *"reduce"*, and *"remove"*, which show the improvement of the structural design/code such as cohesion and coupling. For external quality attribute SARs, the top ranked keywords include *"improve"*, *"fix"*, *"refactor"*.

This indicate the enhancement of non-functional attributes such as **testability** and **readability**. Finally, for the code smells category, the word *"remove"* was ranked first, which indicates the elimination of certain anti-pattern instances. We notice that refactoring operation-related keywords (*e.g.*, move and extract) are mostly used in code smell-focused category. This is due to the fact that the elimination process of some of the design defects requires certain level of movement and extraction of the associated code elements. We also observe that it is possible for a single keyword to serve different purposes of SARs, but with different ranking in some cases. For instance, the keyword *"improve"* is ranked first for non-functional quality attributes, whereas it is ranked fifth for internal quality attributes, as shown in Table V.

Summary. Our findings indicate that while there are no patterns that are restricted to a specific quality issue, developers occasionally do link the refactoring of a quality issue with a specific operation *e.g.,* associating the correction of a *feature envy* by the operation *move method*.

D. RQ4: Do Commits containing the label Refactor indicate more refactoring activity than those without the label?

Murphy-Hill et al. [13] 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. 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 rounds. In the first round, we used the keyword *"refactor"*, exactly as dictated by the authors. Thereafter, we quantified the proportion of commits including the searched label across all the considered projects in our benchmark. In the second round, we re-tested the hypothesis using the 87 SAR patterns listed in Table IV, *i.e.,* 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 quantity of refactorings identified for each set, *i.e.*, labeled and unlabeled commits with the keyword *refactor*, along with labeled and unlabeled commits with SAR patterns. We used the Wilcoxon test, as suggested by Murphy-Hill et al. [13] for the purpose of testing the hypothesis. We then applied the non-parametric Wilcoxon rank-

[7]https://github.com/jsevellec/cassandra-unit
[8]https://github.com/egonw/cdk

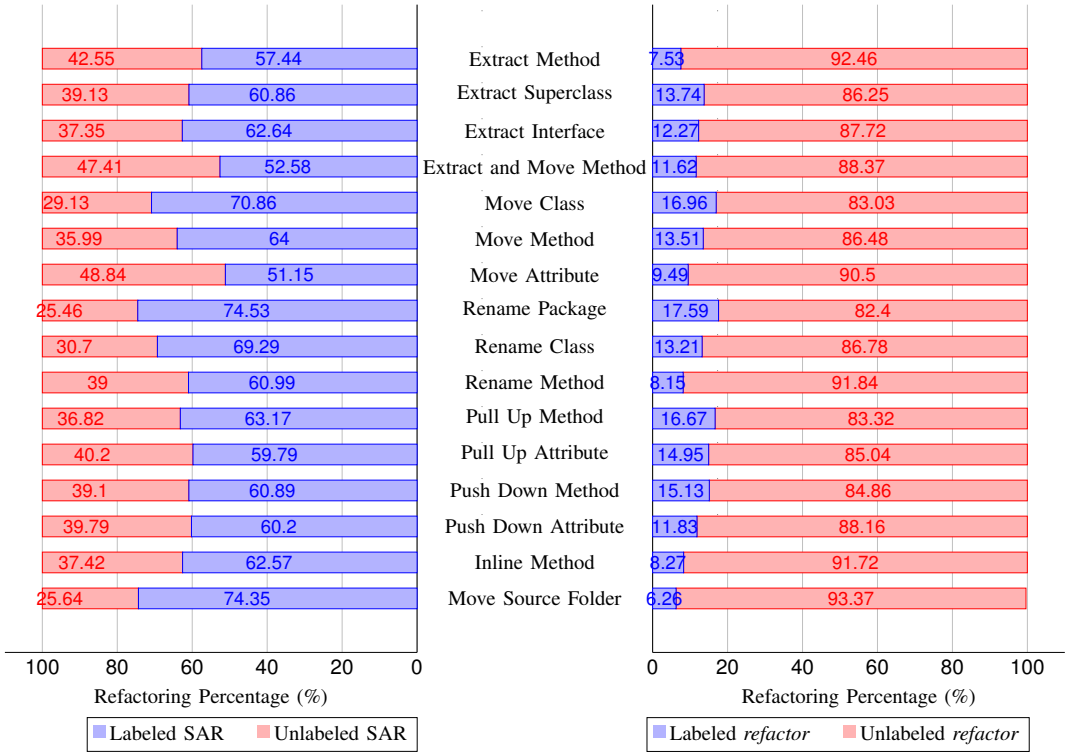

Figure 3: Distribution of Refactoring Operations for Commits Labeled and Unlabeled SAR (at left) and Commits Labeled and Unlabeled *refactor* (at right).

sum test to estimate the significance of differences between the numbers of the sets.

Figure 3 shows the distribution of refactorings in labeled and unlabeled commits with SAR patterns (group 1 at left) and labeled and unlabeled commits with the keyword refactor (group 2 at right). The first observation we can draw is that *"Rename Package"* is the most labeled refactoring with a score of 74.53% and 17.59% for group 1 and group 2 respectively. Another interesting observation is that *"Move Attribute"* turns out to be the most unlabeled refactoring with a score of 51.15% for group 1, whereas *"Move Source Folder"* tends to be the most unlabeled refactoring for group 2. For both tests, we notice that developers tend to label more refactorings applied to code elements with higher granularity level, *i.e.*, at the package level. Conversely, refactorings that are implemented at method level and at attribute level tend to have the lowest percentage with commits labeled *"refactor"*. That sheds light on the variety of ways to express refactorings, which depend on the levels of granularity.

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

> *Summary.* In consistency with the previous findings of Murphy-Hill [13], 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.

VI. Threats to Validity

Internal Validity. Our analysis is mainly threatened by the accuracy of the Refactoring Miner tool because the tool may miss the detection of some refactorings. However, previous studies [19], [25] report that Refactoring Miner has high precision and recall scores compared to other state-of-the-art refactoring detection tools, which gives us confidence in using the tool. Another potential threat to validity relates to our findings regarding counting the reported quality attributes and code smells. Due to the large number of commit messages, we have not performed a manual validation to remove false positive commit messages. Thus, this may have an impact on our findings. 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 pattern.

External Validity. The first threat is that the analysis was limited to only open source Java projects. However, we were still able to analyze 3,795 projects that are well-commented, and 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 projects since the identified refactoring patterns may be different for another set of projects (e.g., outside the Java developers community).

VII. Conclusion

Software developers may explicitly report refactoring activities in the commit messages of versioned repositories. We call such activity *Self-Affirmed Refactoring* (SAR), which is an indication of the developer-reported refactoring events in the change messages. In this work, we performed an exploratory study to identify SAR patterns, study possible SAR types, and determine how much SAR exists. Our findings show that developers use a variety of patterns to purposefully target refactoring events, developers tend to explicitly mention the improvement of certain quality attributes and code smells, and refactoring commit messages with SAR patterns tend to have more significant refactoring activity than those without. Our findings shed light on the existence of SAR. As future work, we plan to investigate which developers are responsible for SAR. Since SAR is considered a good practice, we would like to examine whether developers with higher experience are responsible for the introduction of SAR. Further, for commit messages that contain internal quality attributes (*e.g.*, cohesion and coupling), we plan to empirically assess the quality improvement as reported by developers in their commit messages.

VIII. Acknowledgments

We would like to thank the authors of Refactoring Miner for publicly providing it.

References

[1] J. Al Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

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

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

[4] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[5] E. da Silva Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and d. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, pages 211–221. IEEE Press, 2012.

[8] S. Gharbi, M. W. Mkaouer, M. Ben Messaoud, and I. Jenhani. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th Annual ACM Symposium on Applied Computing*, pages 1746–1754. ACM, 2019.

[9] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

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

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

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

[13] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 7. ACM, 2008.

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

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

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

[17] J. Ratzinger. *sPACE: Software Project Assessment in the Course of Evolution*. PhD thesis, 2007.

[18] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[19] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016.

[20] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, and M. Cornlio. Saferefactor-tool for checking refactoring safety. 01 2009.

[21] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, 2013.

[22] C. M. Steele. The psychology of self-affirmation: Sustaining the integrity of the self. In *Advances in experimental social psychology*, volume 21, pages 261–302. Elsevier, 1988.

[23] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, pages 10–10. IEEE, 2007.

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

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

[26] X. Xia, D. Lo, X. Wang, and X. Yang. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, 65(4):1810–1829, 2016.

[27] D. Zhang, B. Li, Z. Li, and P. Liang. A preliminary investigation of self-admitted refactorings in open source software. 07 2018.