How Do Developers Refactor Code to Improve Code Reusability?

Eman Abdullah AlOmar[1] (✉), Philip T. Rodriguez[1], Jordan Bowman[1], Tianjia Wang[1], Benjamin Adepoju[1], Kevin Lopez[2], Christian Newman[1], Ali Ouni[3], and Mohamed Wiem Mkaouer[1]

[1] Rochester Institute of Technology, Rochester, NY, USA
{eaa6167,ptr5201,jeb4905,tw7205,ba1724,cdnvse,mwmvse}@rit.edu
[2] California State University, Turlock, CA, USA
klopez43@csustan.edu
[3] ETS Montreal, University of Quebec, Montreal, QC, Canada
ali.ouni@etsmtl.ca

Abstract. Refactoring is the *de-facto* practice to optimize software health. While there has been several studies proposing refactoring strategies to optimize software design through applying design patterns and removing design defects, little is known about how developers actually refactor their code to improve its reuse. Therefore, we extract, from 1,828 open source projects, a set of refactorings which were intended to improve the software reusability. We analyze the impact of reusability refactorings on state-of-the-art reusability metrics, and we compare the distribution of reusability refactoring types, with the distribution of the remaining mainstream refactorings. Overall, we found that the distribution of refactoring types, applied in the context of reusability, is different from the distribution of refactoring types in mainstream development. In the refactorings performed to improve reusability, source files are subject to more design level types of refactorings. Reusability refactorings significantly impact, high-level code elements, such as packages, classes, and methods, while typical refactorings, impact all code elements, including identifiers, and parameters.

Keywords: Refactoring · Reusability · Software Metrics · Quality.

1 Introduction

Refactoring is defined as the process of changing software system in such way that changes improve software quality and do not alter the software behaviour [13,7]. Refactoring is one of the commonly-used techniques to improve software quality [18,7]. There are different refactoring operations that could be used to improve software quality such as a change in parameter types, move attributes/methods, rename variables/parameters/attributes/methods/classes, extract methods, extract classes, etc [7].

Refactoring plays an important role in software engineering, as its purpose is to improve software quality. Without refactoring, software quality would continue to deteriorate and make development more difficult. Researchers conducted many studies on refactoring in different areas, such as finding the approach to effectively refactor code and determining the impact of refactoring on software quality. One particular aspect of refactoring is increasing the reusability of software components, which provides developers a more efficient way to utilize existing code to create new functionality. Creating reusable software components facilitates development and maintenance since less work is needed to accomplish additional functionality.

While it is usually true that refactoring improves software quality, it is not known how reusability refactoring impacts metrics. Moser et al. [10] has found that the appropriate refactoring can make the necessary design level changes to improve the software reusability, however, there is no practical evidence on how developers refactor code to improve reusability in practice.

The purpose of this paper is to investigate how developers use refactoring when they state they are improving code reusability. Therefore, we have mined commits from 1,828 well-engineered project, were we have identified 1,957 reusability commits. We refer to a commit as a *reusability commit* where its developer explicitly mentions, in the commit message, that a refactoring is performed to improve reusability. Then we extract all refactorings executed in these reusability commits, and we label them as *reusability refactorings.* To better understand how developers perceive reusability and apply it in real-world scenarios, we examine how these refactorings manifest in the code by examining their impact on code quality. Furthermore, to check if there are some refactoring patterns that are specific to reusability, we report the distribution of reusability refactorings compared to other refactorings and the distribution of the different types of refactored code elements in reusability refactorings. To perform this analysis, we formulate the following research questions:

RQ1. *Do developers refactor code differently for the purpose of improving reusability?*

To answer this research question, we execute Refactoring Miner [19] to extract the type of refactorings that are chosen by developers to improve reusability. We also investigate if there are any refactoring patterns that are specific to reusability, by comparing the distribution of reusability-related refactorings, with the distribution of refactorings for other mainstream development tasks. Then, we identify any significant differences between the distribution values in the two populations.

RQ2. *What is the impact of reusability refactorings on structural metrics?*

To answer this research question, we consider the state-of-the-art reusability structural metrics, extracted from previous studies [10,4]. We calculate these metrics on files before and after they were refactored for improving reusability. Then we analyze the impact of refactorings on the variation of these metrics, to see if they were capturing the improvement.

The results of our study indicate that when developers make reusability changes, they seem to significantly impact metrics related to methods and attributes, but not parameters or interfaces. Additionally, developers perform reusability changes much less than regular refactoring changes. Aid from our empirical analysis, we provide the software reuse community with a replication package, containing the dataset we crawled, the files containing all the metric values, for the purpose of replication and extension[4].

The remainder of this paper is organized as follows: Section 2 includes some existing studies related to our work. Section 3 presents the design of our empirical study, Section 4 shows the results of our experiments, Section 5 describes the threats the validity to our study and any mitigation we took to minimize those threats, and Section 6 summarizes the contributions and results of our study.

2 Related Work

Research in refactoring software has covered a variety of aspects, including tools and methods to facilitate refactoring and accurately assess the impact of refactoring on software quality. Pantiuchina et al. [14] talked about determining if there was a difference in how developers perceive refactorings will be helpful, and how the metrics say the refactorings were. That study determined that even if a developer reports that there was a refactoring done it might not be reflected in the metrics. This study focuses on comparing specific refactorings relating to certain metrics, specifically "cohesion", "coupling", "readability", and "complexity", to metrics that measure those attributes, while we focused on using metrics to determine if there was a quantifiable difference, and if so, what that difference was, during self-proclaimed reusability refactorings. Even then, something to take away from this study is that measuring refactoring code changes focusing on quality of life, rather than strictly functional, can have many moving parts not measured by metrics. Metrics do not tell the whole story, and while it is good to see what metrics are affected when developers improve reusability, it could also be helpful to include information and narratives from actual developers alongside the pure metrics.

Fakhoury et al. [6] have shown that the existing readability models are not able to capture the readability improvement with minor changes in the code, and some metrics which can effectively measure the readability improvement are currently not used by readability models. The authors also studied the distribution of different types of changes in readability improvements, which is similar to our research question, which examines the distribution of the different types of refactored code elements in reusability refactorings.

Prior works [1,15,2] have explored how developers document their refactoring activities in commit messages; this activity is called Self-Admitted Refactoring or Self-Affirmed Refactoring (SAR). In particular, SAR indicates developers' explicit documentation of refactoring operations intentionally introduced during a code change.

[4] https://smilevo.github.io/self-affirmed-refactoring/

AlOmar et al. [3] showed that there is a misperception between the state-of-the-art structural metrics widely used as indicators for refactoring and what developers consider to be an improvement in their source code. The research aims to identify (among software quality models) the metrics that align with the vision of developers on the quality attribute they explicitly state they want to improve. Their approach entailed mining 322,479 commits from 3,795 open source projects, from which they identified about 1,245 commits based on commit messages that explicitly informed the refactoring towards improving quality attributes. Thereafter, they processed the identified commits by measuring structural metrics before and after the changes. The variations in values were then compared to distinguish metrics that are significantly impacted by the refactoring, towards better reflecting the intention of developers to improve the corresponding quality attribute. Our study also utilized software quality metrics to evaluate the impact of refactoring on reusability.

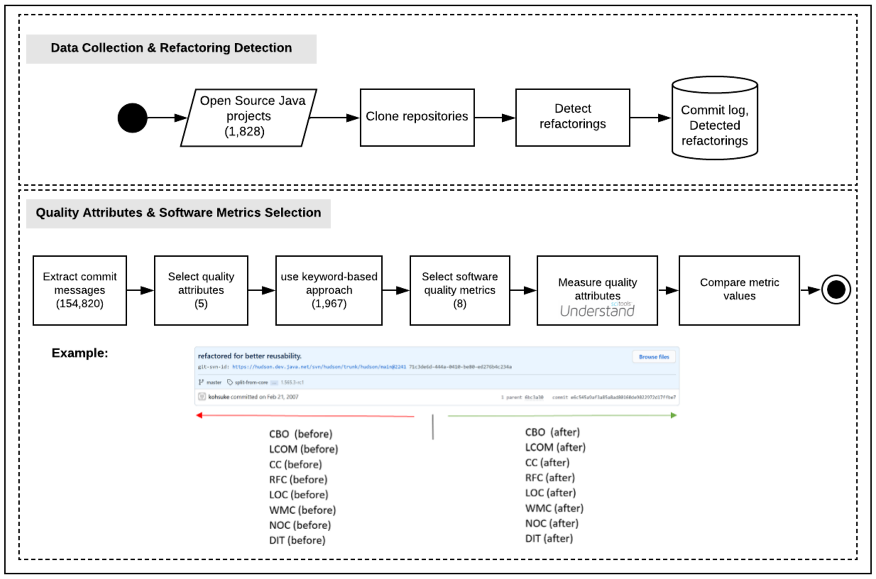
Research particularly in reusability refactoring by Moser et al. [10] showed that refactoring increases the quality and reusability of classes in an industrial, agile environment. Similar to our paper, their study examines the impact of refactoring on quality metrics related to reusability on the method and class levels, such as Weighted Method per Class (WMC) and Coupling Between Object (CBO), respectively. The results of their experiment revealed that refactoring significantly improved the metrics Response for Class (RFC) and Coupling Between Object classes (CBO) related to reusability. However, the limitations of their study involved a small project consisting of 30 Java classes and 1,770 Lines of Code (LOC) developed by two pairs of programmers over the course of 8 weeks. In addition, the authors considered how general refactoring operations impact metrics related to reusability, rather than specifically reusability refactorings. In our study, we examined 1,828 projects and 154,820 commits that modified Java files. We also considered how reusability changes affect software quality metrics and how what kinds of refactoring operations were performed during reusability changes. Table 1 shows the summary of each study related to our work.

Table 1: Summary of related studies.

Study	**Year**	**Focus**	**Dataset Size**	**Quality Attribute**	**Software Metric**
Moser et al. [10]	2006	Reusability measurement over time.	30 Java classes	Reusability	LCOM / RFC / CC CBO / WMC /LOC DIT /NOC
Pantiuchina et al. [14]	2018	Developer's perception & quality	1,282 commits.	Cohesion / Coupling Complexity / Readability	LOCM / C3 / CBO RFC / WMC / B&W Sread
Fakhoury et al. [6]	2019	Developer's perception & quality	548 commits	Readability	B&W / Sread / Dorn
AlOmar et al. [3]	2019	Developer's perception & quality	1,245 commits	Coupling / Cohesion Complexity / Inheritance Polymorphism / Encapsulation Abstraction / Size	LCOM / CBO / FANIN FANOUT / RFC /CC WMC / Evg / NPATH MaxNest / DIT / NOC IFANIN / LOC / CLOC STMTC / CDL / NIV NIM
This work	2020	Developer's perception & quality	1,967 commits	Reusability	LCOM / CBO / RFC CC / WMC / LOC DIT / NOC

3 Experimental Design

Depicted in Figure 1 is an overview of our experiment methodology. We detail each activity of our methodology in the subsequent subsections. The dataset utilized in this study is available for extension and replication purpose [5].



refactored for better reusability.
git-svn-id: https://hudson.dev.java.net/svn/hudson/trunk/hudson/main@2241 71c3de6d-444a-0410-be80-ed276b4c234a
kohsuke committed on Feb 21, 2007
1 parent 6bc3a30 commit e6c545a9af3a85a8ad80168de9822972d17ffbe7

Fig. 1: Empirical study design overview.

3.1 Selection of Quality Attributes and Structural Metrics

We started by conducting a literature review on existing and well-known software quality metrics [5,8,9]. Next, we extracted metrics that are used to assess several object-oriented design aspects in general, and software reusability in particular. For example, the RFC (Response for Class) metric is typically used to measure visibility of a given class in the project, the more a class is responsive, the more it can be accessed and its functionality can be reused by other objects in the system.

The process left us with 8 object-oriented metrics as shown in Table 2. The list of metrics is (1) well-known and defined in the literature, and (2) can assess on different code-level elements, i.e., method, class, package, and (3) can

[5] https://smilevo.github.io/self-affirmed-refactoring/

be calculated by the tool we considered. All metrics values are automatically computed using the tool UNDERSTAND[6], a software quality assurance framework.

Table 2: Reusability and its corresponding structural metrics used in this study.

Quality Attribute	**Study**	**Software Metric**
Cohesion	[4,10]	Lack of Cohesion of Methods (LCOM)
Complexity	[10]	Response for Class (RFC)
	[10]	Cyclomatic Complexity (CC)
Coupling	[4,10]	Coupling Between Objects (CBO)
Design Size	[4,10]	Weighted Method per Class (WMC)
	[4,10]	Line of Code (LOC)
Inheritance	[4,10]	Depth of Inheritance Tree (DIT)
	[4,10]	Number of Children (NOC)

3.2 Refactoring Detection

The projects in our study consist of 1,828 open-source Java projects, which were curated projects hosted on GitHub. These projects were selected from a dataset made available by Munaiah et al. [11], while verifying that these are Java-based projects since this is the only language the Refactoring Miner [19] supports. These projects utilize software engineering practices such as documentation and testing.

We utilize Refactoring Miner [19] for mining refactorings from each project in our dataset. Refactoring Miner is designed to analyze code changes (i.e., commits) in Git repositories to detect applied refactorings. Our choice of the mining tool is driven by its accuracy (precision of 98% and a recall of 87%) and is suitable for a study that requires a high degree of automation since it can be used through its external API.

In this phase, we collect a total of 862,888 refactoring operations from 154,820 commits. An overview of the studied benchmark is provided in Table 3.

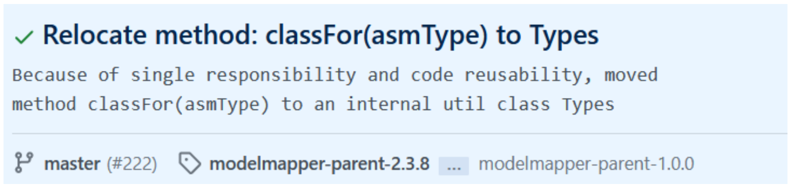
Table 3: Studied dataset statistics.

Item	**Count**
Studied projects	1,828
Commits with refactorings	154,820
Commits with *reus*/reusability* Keywords	1,967
Reusability refactoring operations	3,065

[6] https://scitools.com/

3.3 Reusability Commits Extraction

After extracting all refactoring commit messages detected by Refactoring Miner, our next step consists of analyzing each of the commit messages as we want to only keep commits where refactoring is documented, i.e., self-affirmed refactoring (SAR) [1]. As for the commit message selection, we initially use a keyword-based approach to find those commits that contain the keywords *reus**[7] and *reusability*. We have chosen these two keywords because of their popularity in the development community as being used by developers to describe software reusability [17]. We then kept commits whose messages contained the two keywords. We performed a manual analysis of all the commits, and we ended up removing any duplicates and false positives. This process resulted in selecting 1,967 commits, containing 3,065 refactorings, as our dataset for this study. Each dataset instance is a commit, along with its corresponding refactorings.


✓ Relocate method: classFor(asmType) to Types

```
Because of single responsibility and code reusability, moved
method classFor(asmType) to an internal util class Types
```

master (#222) modelmapper-parent-2.3.8 ... modelmapper-parent-1.0.0

Fig. 2: A sample instance of our dataset.

As an illustrative example, Figure 2 details a commit whose message states the relocation of the method *classFor(asmType)* to an internal class utility class for the purpose of applying the single responsibility principle and code reusability[8]. After running Refactoring Miner, we detected the existence of a *Move method* refactoring from the class *ExplicitMappingVisitor* to the class *Types*. The detected refactoring matches the description of the commit message, and gives more insights about the old placement of the method, which was absent in the textual description. As we explain in the following subsection, we need to locate all the code elements involved in the refactoring (source class, target class, etc.) for the purpose of evaluating the quality of the relocation in terms of impact of structural metrics, such as coupling and cohesion.

3.4 Metrics Calculation

To generate the metric values for reusability commits, we ran code evaluation tools, specifically using Understand[9]. The metrics we used to evaluate the code quality are summarized in Table 2.

[7] Regular expression was used to capture all expansions of reus such as reuses, reusing, reuse, etc.
[8] link to the commit: https://github.com/modelmapper/modelmapper/commit/6796071fc6ad98150b6faf654c8200164f977aa4
[9] https://scitools.com/features/

We then used SQL queries to find reusability commits in the dataset and their associated project links to clone using Git and exported the results from our dataset to a combined Comma-Separated Value (CSV) file. Using a shell script, we cloned the projects, checked out the versions for each commit, and ran the Git diff command to see which files changed in each commit. If files were deleted in a commit, we included the metric values for those files before the commit but not after it. If files were added in a commit, we included the metric values for those files after the commit but not before it. If files were renamed or moved in a commit, then we included the metric values for those files both before and after the commit. Our shell script then ran the UNDERSTAND tool to generate metrics for the changed files for the versions before and after each reusability commit, resulting in two files containing metric values for each commit: (1) one file for the files changed before the commit and (2) another file for the files changed after the commit.

Since each metric value before and after the commit are dependent to each other, we decided to use the Wilcoxon Signed-rank Test [20] to determine whether or not there were statistically significant differences in the metric values for all changed files before and after the reusability commits. We formulated our null hypothesis as follows: *there was no improvement in the metrics we analyzed between before and after the reusability refactoring.* We formulated our alternate hypothesis as follows: *there was an improvement shown as an increase.* To achieve that, we created Python scripts to order and sort all the values from the above results from UNDERSTAND to ensure that the rows in both before and after files are corresponding to each other. Next, we combined the data in the CSV files before and after the commits together into another two CSV files each have a total of 185,244 metric values: one CSV file for all code elements in changed files before the reusability commits, and another CSV file for all code elements in changes files after the commits. The Wilcoxon Signed-rank Test allowed us to determine if any metrics were statistically significantly changed when developers performed self-proclaimed reusability refactorings.

4 Results

This section reports and discusses our experimental results and aims to answer our research questions.

4.1 RQ1. Do developers refactor code differently for the purpose of improving reusability?

This research question aims to compare refactoring activity in reusability commits with the refactoring activity that can be found in mainstream development tasks (feature updates, bug fix, etc.). Since we have a dataset of all refactorings performed in the 1,828 projects that we study, we separate refactorings that belong to the reusability commits (refactorings performed for the purpose of improving reusability), which we refer to as *reusability refactorings*. We refer to the remaining refactorings as *non-reusability refactorings*. Then, for each group, we calculate the percentage of each refactoring type, among the total refactorings of that group.

Figure 3 visualizes, by percentage of the total refactoring operations in each of the respective sets, the distributions of refactoring operations. We observe that the distribution of *reusability refactorings* varies from the *non-reusability refactorings*. In fact, the top frequent types in reusability refactorings are, *Move Method*, *Extract Method*, and *Pull-Up Method*, whose percentages are respectively, 17.29%, 14.85%, and 11.21%. For non-reusability refactorings, the top frequent type were *Rename Attribute*, *Rename Method*, and *Rename Variable*, as their percentages are respectively, 18.96%, 11.92%, and 11.86%. While the *move* related types were highly solicited in reusability refactorings, the *rename* activity was dominant for non-reusability refactorings, which was expected since previous studies who analyzed mainstream refactoring has found that renames are the most popular refactorings [19,3,15,16]. However, reusability refactorings seem to be different. To analyze the extent to which reusability and non-reusability refactorings vary, we compare the distribution of refactoring refactorings identified for each group using the Wilcoxon signed-rank test, a pairwise statistical test verifying whether two sets have a similar distribution [20]. If the p-value is smaller than 0.05, the distribution difference between the two sets is considered statistically significant. The choice of Wilcoxon comes from its non-parametric nature with no assumption of a normal data distribution. Upon running the statistical test, the null hypothesis was rejected and the difference between group distributions was found to be statistically significant.

Another interesting observation that we draw is the popularity of method-level refactoring, being in in TOP 3 most frequent reusability refactorings. Figure 4 shows the distribution of code elements impacted by refactorings, and we notice that more than 50% of refactorings were performed at the method level.

To better understand the observed results, we sampled a subset of reusability refactorings, and we have extracted two main patterns:

Functionality extraction. When developers are interested in a needed functionality, which is found inside a long method, containing various functionalities, they extract the code elements, belonging to the needed functionality, into a newly created separate method, and they update the original method with the appropriate method calls. This decomposition process is known as *Extract Method*. The newly extracted method has its own visibility, which is independent from the original method, and so developers can increase its visibility of the purpose of reuse, and so other objects and methods can now access it.

Functionality movement. To increase the reusability of a given method, we have noticed that developers typically move methods from less visible classes, into more visible classes, in the system. Various methods were moved into utility classes, which are eventually offering their services to the other classes in the system, this explains why *Move Method* was the most popular type in reusability refactorings, according to Figure 3. Our qualitative analysis has also shown scenarios of moving method up, from a child class, into a super class,

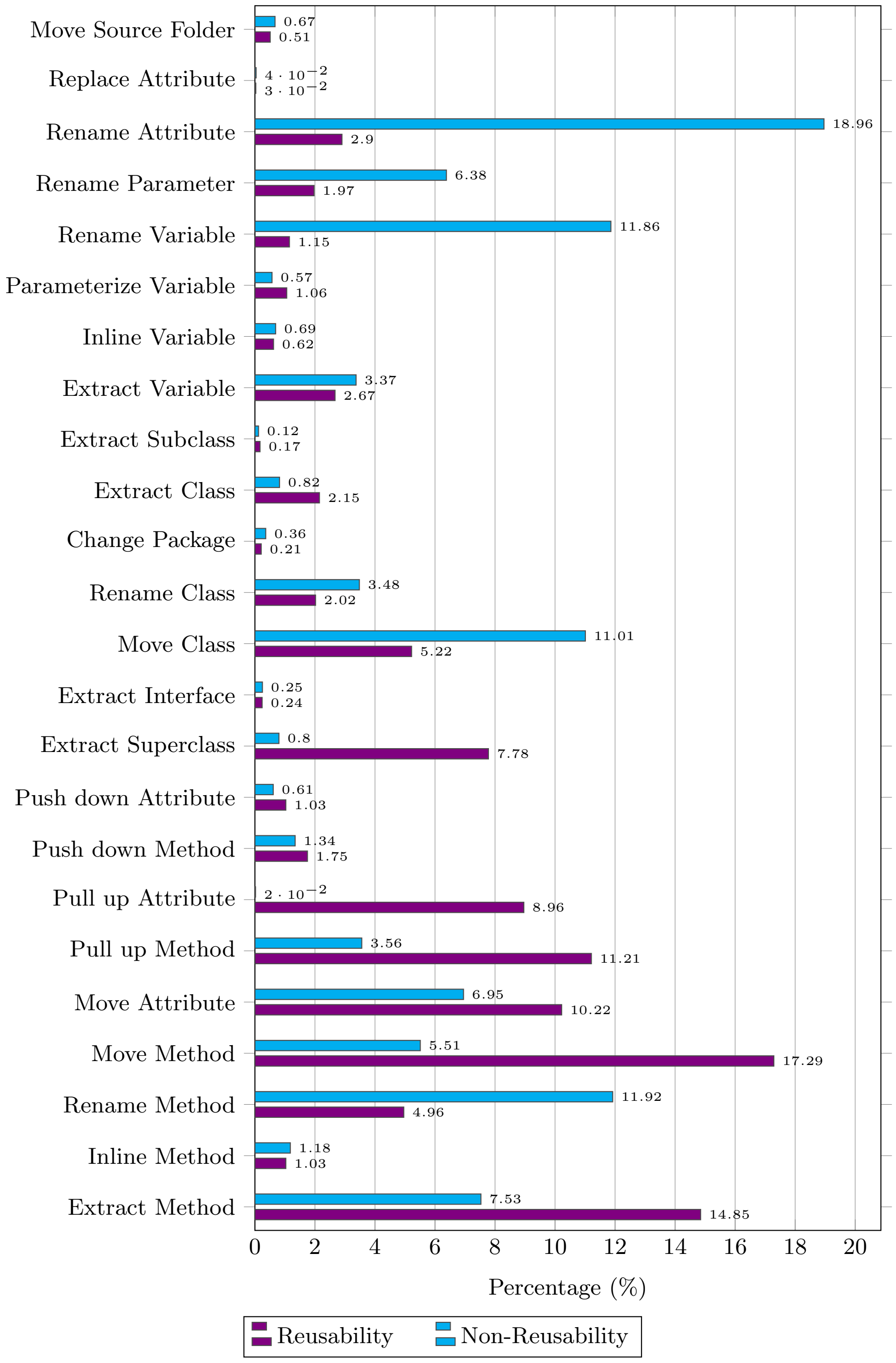


Fig. 3: Percentages of *reusability refactoring* and *non-reusability refactorings*, clustered by type.

for the purpose of sharing its behavior across all subclasses through inheritance. This refactoring is known as *Pull-Up Method*, which was found to be the third popular type in reusability refactorings, while being not popular in non-reusability refactorings.

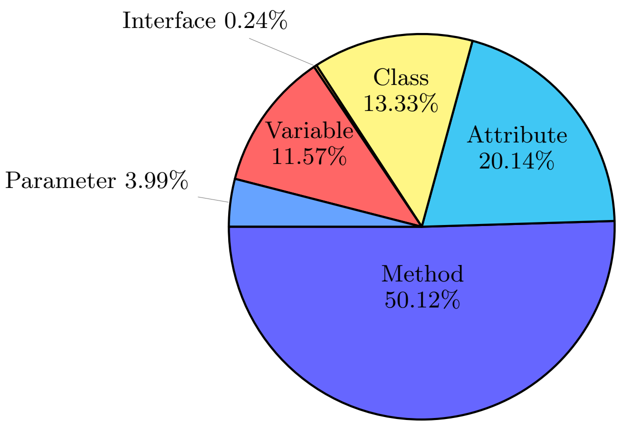


Fig. 4: Distribution of code elements in reusability refactoring commits.

Summary. We have shown that the distribution of refactoring types, applied in the context of reusability, is different from the distribution of refactoring types in mainstream development. In the refactorings performed to improve reusability, files are subject to more design level types of refactorings (e.g., *Move Method*, *Extract Method*) in general, and inheritance-related refactorings (e.g., *Pull-up Method*, *Pull-up Attribute*) in particular, while in other refactorings, files tend to undergo more renames (e.g., *Rename Method*, *Rename Variable*) and data type changes (e.g., *Change Variable Type*) to identifiers. Reusability refactorings heavily impact, high-level code elements, such as packages, classes, and methods, while typical refactorings, impact all code elements, including identifiers, and parameters.

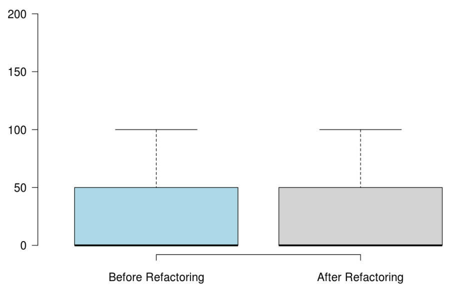
4.2 RQ2. What is the impact of reusability refactorings on structural metrics?

To answer this research question, we investigate the impact of reusability refactorings on the state-of-the-art metrics, which have been used by previous studies, to recommend reusability changes. As a reminder, we aim to look at the variation of each metric value after the execution of the refactoring, therefore, we checkout the project files, right before the reusability commit, and we calculate metrics values, and after the reusability commit, and we recalculate the metrics values. Note that we only consider files that were involved in the commit, as there files are considered part of developer's intention of improving reusability. The results of metrics boxplots are outlined in Figure 5. To further investigate the significance of difference between the boxplots, we also use the Wilcoxon Signed-rank Test. Statistical settings included using a 0.05 alpha value for the significance level. We hypothesize that reusability refactorings will optimize metrics by reducing them (the lower is the value of the metric, the better is the software structural quality). Our alternative hypothesis is accepted if the *before refactoring* boxplot is significantly larger than the *after refactoring* boxplot. The Wilcoxon Signed-rank Test results indicating whether or not there were statistically significant improvements before and after reusability commits is shown in Table 4.

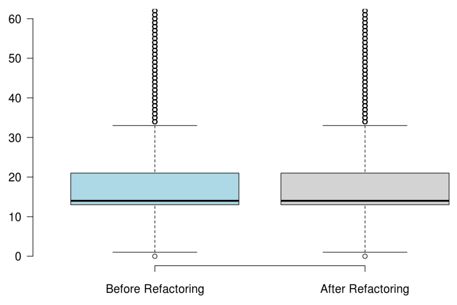
According to Figure 5, reusability refactorings had no impact on the Number of Children (NOC) Depth of Inheritance Tree (DIT), and Response for Class (RFC). These results can be explained by the fact that the majority of reusability refactoring are not targeting classes. In fact, if we refer to Figure 4, only 13.3% of reusability refactoring targeted classes, and exctrating subclasses, which would have impacted these metrics, represent only 0.13%, and so, its impact is negligible.

On the other hand, we measure an increase in the weighted methods per class, and the variation is found to be statistically significant ($p < 0.05$). According to Figure 3, the *Extract Method* refactoring has been found to be very popular in reusability refactoring, and so, developers tend to create new methods while extracting the reusable code from the longer methods. This implies the sudden increase of methods count, per class. While developers are expected to keep the number of methods lower in classes, the impact of reusable functionality from longer classes, creates free methods that can be pulled up to either superclasses, and be shared with all children, or relocated to operate on variables that may not belong to its original class. This explains decrease of the Coupling Between Objects (CBO) and the slight decrease in the Lack of Cohesion of Methods (LCOM), which means that methods have become more cohesive. However, its corresponding statistical test show no significant different, but its value was close to 0.05. Similarly, we notice slight improvement in the Lines of Code (LOC), with no statistical significance but close p-value (i.e., 0.066). The extraction of methods helps in reducing cloning functionalities in multiple locations in the code. Also, pulling methods up the hierarchy, will allow subclasses to inherit it, and so, lines of code will decrease, unless when the method gets overridden. Moreover, the Cyclomatic Complexity (CC) has decreased after reusability code changes with no statistical significance. A proper extraction of sub-methods tends to break down long methods, and slightly decrease their complexity.

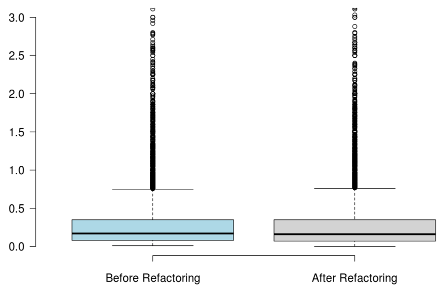
As a meta-review, the majority of state-of-the-art metrics did not capture any improvement, or captured non-significant improvement, when developers refactor their code for the purpose of reusability. This is an interesting finding for our future research directions, as we want to further increase our dataset, in terms of projects, and programming languages, in order to experiment whether there is a shortage of metrics that properly measure what developers consider to be



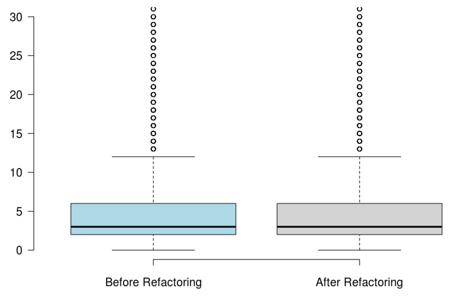
(a) Percent Lack of Cohesion



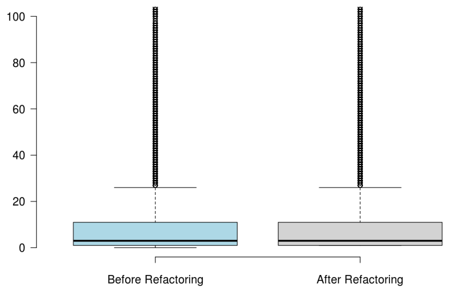
(b) Response for Class



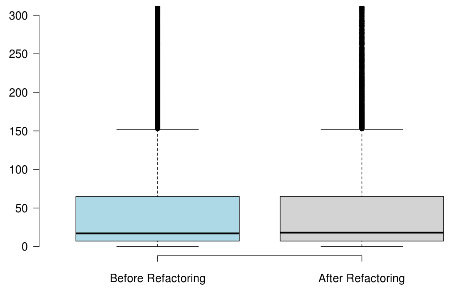
(c) Cyclomatic Complexity



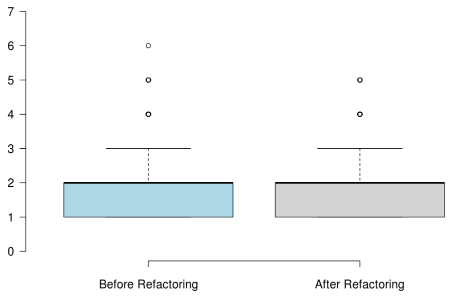
(d) Coupling Between Object



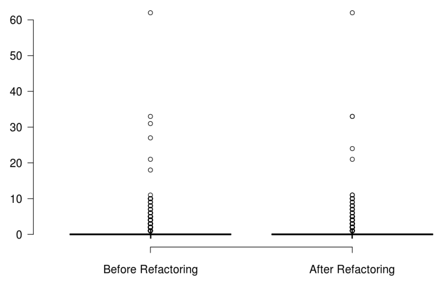
(e) Weighted Method per Class



(f) Line of Code



(g) Depth of Inheritance Tree



(h) Number of Children

Fig. 5: Boxplots for metric values before and after reusability commits for different sets of code elements.

at design level change to improve reusability. Such investigations will bridge the gap between how existing research on software reuse evaluates code changes, and how developers concretely achieve it.

Table 4: Wilcoxon Signed-Rank Test results for all code elements between before-after versions of reusability commits.

Metric	**p-value**	**Impact**	**Reject H_0?**
Percent Lack of Cohesion (LCOM)	0.0707	+ve	False
Response for Class (RFC)	0.2925	No	False
Cyclomatic Complexity (CC)	0.3298	+ve	False
Coupling Between Objects (CBO)	0.2739	+ve	False
Weighted Method per Class (WMC)	0.0372	-ve	True
Line of Code (LOC)	0.06621	+ve	False
Depth of Inheritance Tree (DIT)	0.7446	No	False
Number of Children (NOC)	0.5292	No	False

Summary. When developers refactor their code for the purpose of reusability, we found that the number of methods significantly increased, but the majority of the state-of-the-art metrics did not capture any improvement, or captured non significant improvement.

5 Threats to 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 1,828 projects that are highly varied in size, contributors, number of commits and refactorings. Additionally, 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. [7]. However, in a previous study, Murphy-Hill et al. [12] 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 [19] reported 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.

Another threat to validity is that, as we mentioned above, while we determined whether a commit has a reusability change, we only look for terms like *reus* in the commit message, although not all reusability commit messages may contain those words.

Another critical threat, is the fact that not all refactorings are root-canal. Developers may be interleaving refactorings with other types of changes, and so, this may become a noise in our measurements. To mitigate this issue, we considered commits that both contain an explicit statement about reusability, and contain at least one refactoring operation, in order to correlate between the refactoring and its documentation. Also, the existence of several unrelated files, in the commit, as part of other changes, can also become a noise for our metrics measurements. To mitigate this threat, we measure the metrics for code elements that are being refactored, and not all the changed files in the reusability commit.

6 Conclusion

In this paper, we performed a study on analyzing reusability refactorings based on information in Java projects from our dataset. We found that in reusability refactorings, the changes developers performed would significantly affect metrics pertaining to methods, but not significantly affect metrics regarding comments or cohesion of classes. We also found that less than 0.4% commits are reusability refactorings in 154,820 commits. Another fact we found is that method is modified more frequently in reusability refactoring changes. Our results have shown some existing facts in reusability refactorings, and those findings could help developers to make better decisions while performing reusability refactorings in the future.

Some recommendations that we have for future work involve comparing different subsections of data, and determining what refactorings are related to reusability. Specifically, we think that it would be interesting to compare the results that we got to instances where each individual refactoring detected was analyzed to explore if it was done for reusability or not, to see if us grouping all refactorings in a commit for reusability and non-reusability is similar. We also think that analyzing the code before and after the reusability commits for different metrics that are more usability based, such as adaptability, understandability, or portability, could be an interesting future work, though an issue might arise to finding specific ways to measure those metrics. Moreover, we plan to find a better way to figure out if a commit was a reusability refactoring or not. Since this work relies on the commit message, there could be commits incorrectly labeled, or commits that are reusability but not labeled as such that we are missing.

Acknowledgements This material is based on work supported by the National Science Foundation under Grant No. 1757680.

References

1. AlOmar, E., Mkaouer, M.W., Ouni, A.: Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR). pp. 51–58. IEEE (2019)

2. AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Toward the automatic classification of self-affirmed refactoring. Journal of Systems and Software p. 110821 (2020)
3. AlOmar, E.A., Mkaouer, M.W., Ouni, A., Kessentini, M.: 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 (2019)
4. Alshayeb, M.: Empirical investigation of refactoring effect on software quality. Information and software technology **51**(9), 1319–1326 (2009)
5. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
6. Fakhoury, S., Roy, D., Hassan, A., Arnaoudova, V.: Improving source code readability: theory and practice. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). pp. 2–12. IEEE (2019)
7. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
8. Lorenz, M., Kidd, J.: Object-oriented software metrics, vol. 131. Prentice Hall Englewood Cliffs (1994)
9. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
10. Moser, R., Sillitti, A., Abrahamsson, P., Succi, G.: Does refactoring improve reusability? In: International Conference on Software Reuse. pp. 287–297. Springer (2006)
11. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017)
12. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering **38**(1), 5–18 (2012)
13. Opdyke, W.F.: Refactoring object-oriented frameworks (1992)
14. Pantiuchina, J., Lanza, M., Bavota, G.: Improving code: The (mis) perception of quality metrics. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 80–91. IEEE (2018)
15. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: Contextualizing rename decisions using refactorings, commit messages, and data types. Journal of Systems and Software p. 110704 (2020)
16. Peruma, A., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F.: An exploratory study on the refactoring of unit test files in android applications. In: Conference on Software Engineering Workshops (ICSEW'20) (2020)
17. Sharma, A., Kumar, R., Grover, P.: A critical survey of reusability aspects for component-based systems. World academy of science, Engineering and Technology **19**, 411–415 (2007)
18. Stroggylos, K., Spinellis, D.: Refactoring–does it improve software quality? In: Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007). pp. 10–10. IEEE (2007)
19. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering. pp. 483–494. ACM (2018)
20. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics bulletin **1**(6), 80–83 (1945)