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

Refactoring for Reuse: An Empirical Study

Eman Abdullah AlOmar · Tianjia Wang · Vaibhavi Raut · Mohamed Wiem Mkaouer · Christian Newman · Ali Ouni

Received: date / Accepted: date

Abstract Refactoring is the *de-facto* practice to optimize software health. While several studies propose 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 that were intended to improve the software reusability. We analyze the impact of reusability refactorings on the 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. These findings provide practical insights into the current practice of refactoring in the context of code reuse involving the act of refactoring.

Keywords refactoring, reusability, software metrics, quality

Eman Abdullah AlOmar · Tianjia Wang · Vaibhavi Raut · Mohamed Wiem Mkaouer · Christian Newman
Rochester Institute of Technology
E-mail: {eaa6167,twang,vraut,mwmvse,cdnvse}@rit.edu

Ali Ouni
ETS Montreal, University of Quebec
E-mail: ali.ouni@etsmtl.ca

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 [53, 32,12]. Refactoring is one of the commonly-used techniques to improve software quality [68,32]. 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 [32].

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. [50] 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. This paper extends a quantitatively and qualitatively our previous study [17]. This papers analyzes the impact of reusability refactorings at a wider set of structural metrics, allowing a better profiling of how the intent of improving design, impacts, either positively or negatively, various design level metrics. From Qualitative point of view, we manually investigate to categorize the intents behind refactoring reusable code. We particularly investigate what triggers developer to refactor the code for the purpose of code reuse.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 [73] 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 [50,18]. 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.

RQ3. *What triggers developers to refactor the code for the purpose of code reuse?*

To answer this research question, we perform case studies that demonstrate GitHub developers' intentions when refactoring source code to improve code reusability.

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[1].

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 6 describes the threats the validity to our study and any mitigation we took to minimize those threats, and Section 7 summarizes the contributions and results of our study.

2 Related Work

It is widely acknowledged in the literature of software refactoring that it has the ultimate goal to improve software quality and fix design and implementation bad practices [32]. As shown in Table 1, there is much research effort have focused on studying and exploring the impact of refactoring on software quality [49,75,18, 67,22,25,46,24,36,15,35]. The vast majority of studies have focused on measuring the internal and external quality attributes to determine the overall quality of a software system being refactored. In this section, we review and discuss the relevant literature on software quality in general and reusability in particular.

2.1 Studies on Software Reusability

Software reusability has been a topic of interest since the 1970s, because of that, a large amount of literature that discusses it is available. This section, although not accounting for all available work, does its best in presenting multiple papers from the different ways reusability is discussed. Previous work discussed general aspects of reusability, identifying challenges, topics, issues and principles [10,20,38,43,47,77,78].

Ahmaro et al. [10] conducted a systematic literature review to identify the definition, approaches, benefits, reusability levels, factors, and adoption of software reusability. The authors found that the concept of software reusability consisted of 11 approaches, namely,

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

Table 1: A summary of the literature on the impact of refactoring activities on quality.

Study	Year	Approach	Software Metric
Sahraoui et al. [64]	2000	Analyzing code histories	CLD / NOC / NMO / NMI NMA / SIX / CBO / DAC IH-ICP / OCAIC / DMMEC / OMMEC
Stroulia & Kapoor [69]	2001	Performing a case study	LOC / LCOM / CC
Tahvildari et al. [72]	2003	Analyzing code histories	LOC / CC / CMT / Halstead's efforts
Leitch & Stroulia [40]	2003	Analyzing code histories	SLOC / No. of Procedure
Bois & Mens [28]	2003	Analyzing code histories	NOM / CC / NOC / CBO RFC / LCOM
Tahvildari & Kontogiannis [71]	2004	Analyzing code histories	LCOM / WMC / RFC / NOM CDE / DAC / TCC
Moser et al. [50]	2006	Analyzing code histories	CK / MCC / LOC
Wilking et al. [75]	2007	Analyzing code histories	CC / LOC
Stroggylos & Spinells [68]	2007	Mining commit log	CK / Ca / NPM
Moser et al. [49]	2008	Analyzing code histories	CK / LOC / Effort (hour)
Alshayeb [18]	2009	Analyzing code histories	CK / LOC / FANOUT
Hegedus et al. [36]	2010	Analyzing code histories	CK
Shatnawi & Li [67]	2011	Analyzing code histories	CK / QMOOD
Bavota et al. [23]	2013	Analyzing code histories Surveying developers	ICP / IC-CD / CCBC
Szoke et al. [70]	2014	Mining commit log Surveying developers	CC / U / NOA / NII / NAni LOC / NUMPAR / NMni / NA
Bavota et al. [22]	2015	Mining commit log Analyzing code histories	CK / LOC / NOA / NOO C3 / CCBC
Mkaouer et al. [46]	2016	Many-objective SBSE	QMOOD
Cedrim at al. [24]	2016	Mining commit log Analyzing code histories	LOC / CBO / NOM / CC FANOUT / FANIN
Chavez et al. [25]	2017	Mining commit log Analyzing code histories	CBO / WMC / DIT / NOC LOC / LCOM2 / LCOM3 / WOC TCC / FANIN / FANOUT / CINT CDISP / CC / Evg / NPATH MaxNest / IFANIN / OR / CLOC STMTC / CDL / NIV / NIM / NOPA
Pantiuchina et al. [55]	2018	Mining commit log Analyzing code histories	LCOM / CBO / WMC / RFC C3 / B&W / Sread
AlOmar et al. [15]	2019	Mining commit log Analyzing code histories	CK / FANIN / FANOUT / CC / NIV / NIM Evg / NPath / MaxNest / IFANIN LOC / CLOC / CDL / STMTC
AlOmar et al. [17]	2020	Mining commit log Analyzing code histories	CK / CC / LOC
Hamdi et al. [35]	2021	Mining commit log Analyzing code histories	LCOM / CBO / WMC / RFC NOSI / TCC / LCC / LOC VQYT / DIT

design patterns, component-based development, application frameworks, legacy system wrapping, service-oriented systems, application product lines, COTS integration, program libraries, program generators, aspect-oriented software development and configurable vertical applications. A study on the relationship of complexity and reuse design principles is reported by Anguswamy and Frakes [20]. Their findings show that the higher the complexity the lower the ease of reuse. Lubars et al. [43] contrasted code reusability in the large versus code reusability in small with regards to several aspects, including, size, complexity, application, and problems associated with locating and reusing the code. The author highlighted that code reusability in the small has had limited impact because of its strongly self-centered orientation, whereas code reusability in the large has had limited impact because of its high degree of difficulty in finding the reusable components. In a similar context, Mockus [47] performed large-scale code reuse study in open source software and found that more than 50% of the files were used in more than one project. Yin and Lee [77] conducted a survey to examine the characteristics of software reusability from the points of view of software engineering as well as knowledge engineering. Younoussi and Roudies [78] presented a systematic literature review on software reusability. They pointed out that few studies examined barriers of reusability, and organizations need to adapt software reusability approaches.

Reusability and code reuse are also discussed in relation to specific topics [34,44,57,65,56,42,19,9,11,30]. Lotter et al. [42] explored code reuse between Stack Overflow and Java open-source systems in order to understand how the practice of reusing code could affect future software maintenance and the correct use of license. Their findings show that there is up to 3.3 % code reuse within Stack Overflow, while 1.0 % of Stack Overflow code is reused in Java projects. Patrick [56] investigated reusability metrics with Q&A forum data. The author proposed an approach (LANLAN), using word embeddings and machine learning, to classify Q&A forum posts into support requests and problem reports, as well as reveal information in relation to software reusability and explore potential reusability metrics. In another context, Abdalkareem et al. [9] performed an exploratory study on 22 Android apps to explore how much, why, when, and who reuses code. They found that 1.3 % of the Apps were constructed from Stack Overflow posts, and discovered that mid-aged and older apps reuse Stack Overflow code later in their lifetime. An et al. [19] also explored Android apps and found that 15.5 % of the apps contained exact code clones, and 60 out of 62 apps, had potential license violations. Recently, AlOmar et al. [11] presented insights regarding how developers discuss software reuse by analyzing Stack Overflow. These findings can be used to guide future research and to assess the relevance of software reuse nowadays. Their findings show that software reuse is a decreasing trend in Stack Overflow which might indicate that developers have widely adopted this practice and thus few questions regarding it emerge as it is well grasped by the community. Further, Feitosa et al. [30] studied the relation between software reuse at the class level and technical debt. The authors found that reused classes tend to concentrate more principally, and reused code usually has less technical debt interest.

2.2 Studies on Software Quality

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. [55] 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. [29] 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 [13,58,14] 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.

AlOmar et al. [15] 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. [50] showed that refactoring increases the quality and reusability of classes in an industrial, agile

Table 2: Summary of related studies on developer perception and quality.

Study	Year	Focus	Dataset Size	Quality Attribute	Software Metric
Moser et al. [50]	2006	Reusability measurement over time.	30 Java classes	Reusability	LCOM / RFC / CC CBO / WMC /LOC DIT /NOC
Pantiuchina et al. [55]	2018	Developer's perception & quality	1,282 commits	Cohesion / Coupling Complexity / Readability	LOCM / C3 / CBO RFC / WMC / B&W Sread
Fakhoury et al. [29]	2019	Developer's perception & quality	548 commits	Readability	B&W / Sread / Dorn
AlOmar et al. [15]	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
AlOmar et al. [17]	2020	Developer's perception & quality	1,967 commits	Reusability	LCOM / CBO / RFC CC / WMC / LOC DIT / NOC

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 2 shows the summary of each study related to our work.

Our work highlights on the aspect of reusability refactorings, and is different from the above-mentioned studies as our main purpose is to explore if there is an alignment between quality metrics and reusability quality improvement that are documented by developers in the commit messages. To the best of our knowledge, no previous study has empirically investigated, using a curated set of commits, and the representativeness of structural design metrics for reusability quality attribute.

3 Experimental Design

According to the guidelines reported by Runeson and Höst [63], we design an empirical study that is supported by explanatory case studies [61]. Our research method consists of three steps as depicted in Figure 1. We detail each activity of our methodology in the subsequent subsections. The dataset utilized in this study is available for extension and replication purpose [2].

3.1 Selection of Quality Attributes and Structural Metrics

We started by conducting a literature review on existing and well-known software quality metrics and their corresponding possible measurements [26,41,45]. 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. More generally, we extract, from literature review, all the associations between metrics (e.g., CK suite [26], McCabe [45]) with reusability quality attribute.

The process left us with 8 object-oriented metrics as shown in Table 3. 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 be calculated by the tool we considered. All metrics values are automatically computed using the tool Understand[3], a software quality assurance framework.

3.2 Refactoring Detection

The projects in our study consist of 1,828 open-source Java projects, which were curated projects hosted on

[2] https://smilevo.github.io/self-affirmed-refactoring/
[3] https://scitools.com/

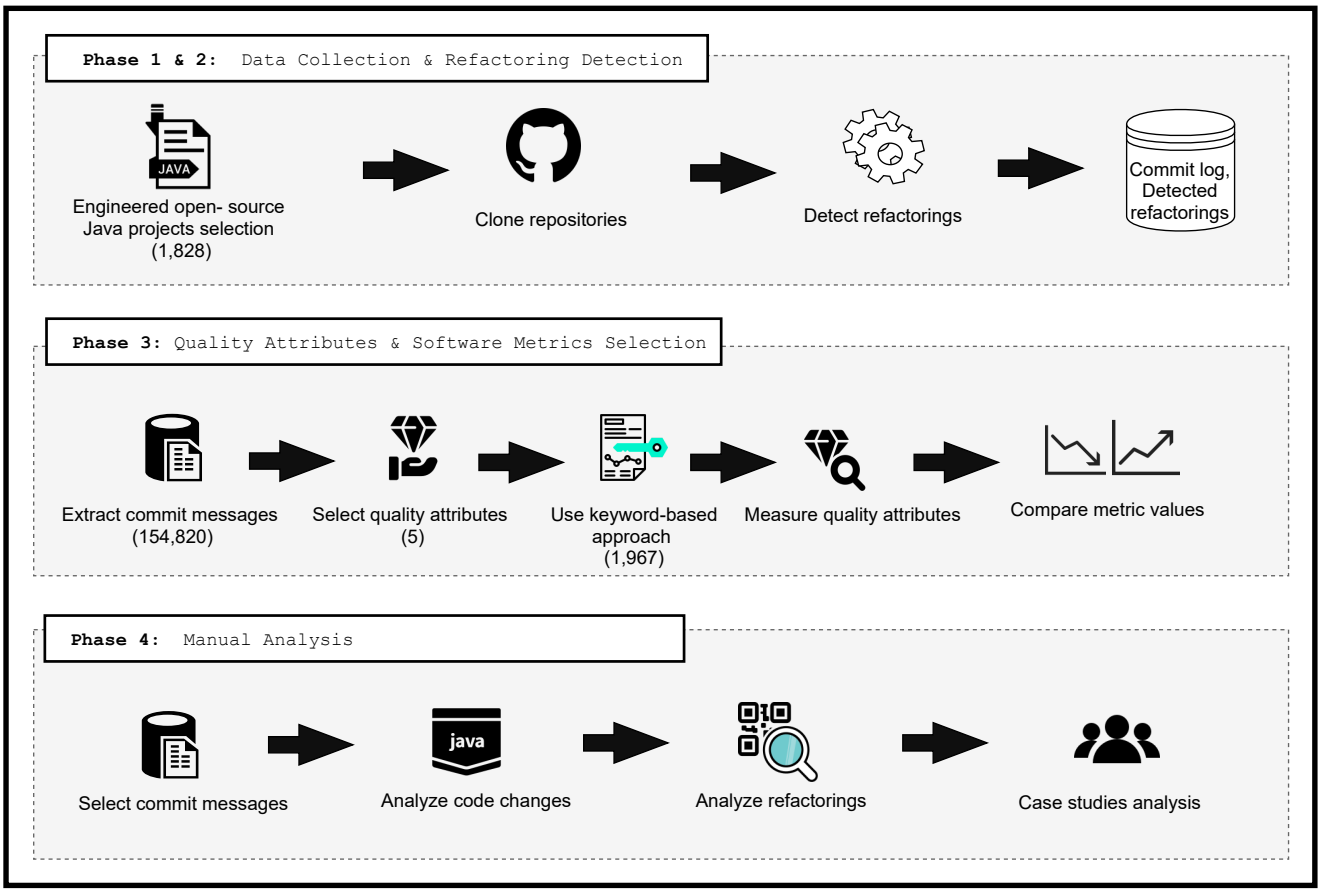


Fig. 1: Empirical study design overview.

GitHub. These projects were selected from a dataset made available by Munaiah et al. [51], while verifying that these are Java-based projects since this is the only language the Refactoring Miner [73] supports. These projects utilize software engineering practices such as documentation and testing.

We utilize Refactoring Miner [73] 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 4.

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) [13,16,14]. As for the commit message selection, we initially use a keyword-based approach to find those commits that contain the keywords *reus**[4] 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 [66]. 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 was done by the first two authors. 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.

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

[4] Regular expression was used to capture all expansions of reus such as reuses, reusing, reuse, etc.

Table 3: Reusability and its corresponding structural metrics used in this study.

Quality Attribute	Study	Software Metric
Cohesion	[18,50]	Lack of Cohesion of Methods (LCOM)
Complexity	[50]	Response for Class (RFC)
	[50]	Cyclomatic Complexity (CC)
	[25,15]	Paths (NPATH)
	[25,15]	Nesting (MaxNest)
Coupling	[18,50]	Coupling Between Objects (CBO)
	[25,15]	Fan-in (FANIN)
	[25,15]	Fan-out (FANOUT)
Design Size	[18,50]	Weighted Method per Class (WMC)
	[18,50]	Line of Code (LOC)
	[25,15]	Lines with Comments (CLOC)
	[25,15]	Statements (STMTC)
	[25,15]	Classes (CDL)
	[25,15]	Instance Variables (NIV)
	[25,15]	Instance Methods (NIM)
Inheritance	[18,50]	Depth of Inheritance Tree (DIT)
	[18,50]	Number of Children (NOC)
	[25,15]	Base Classes (IFANIN)

Table 4: 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

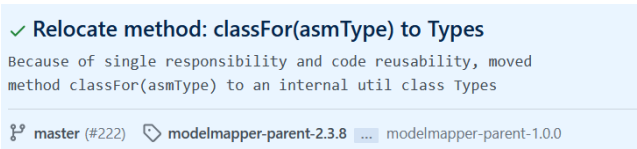


Fig. 2: A sample instance of our dataset.

and code reusability[5]. 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 tool, specifically using Understand[6]. The metrics we used to evaluate the code quality are summarized in Table 3.

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. The goal of using Git diff is to track the changed files in each commit and categorize them into the 'before/after' directory for further analysis. For each file being modified in a commit, the Git diff shows the file paths and a status related to the change as follows:

- 'M' (modification of the contents or mode of a file)
- 'R' (renaming of a file)
- 'D' (deletion of a file)
- 'A' (addition of a file)

Based on the four different status, we performed different actions on the file.

- For 'M', we keep the original file before the change in 'before' directory, and the modified file after the change in 'after' directory.
- For 'R', we keep the original file before the change, and the renamed file after the change in 'after' directory.
- For 'D', we keep the file in the 'before' directory.
- For 'A', we keep the file in the 'after' directory.

In another words, 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

[5] https://github.com/modelmapper/modelmapper/commit/6796071fc6ad98150b6faf654c8200164f977aa4

[6] https://scitools.com/features/

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 [74] 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 the following hypotheses:

H_0: *There was no improvement in the metrics we analyzed between before and after the reusability refactoring*.

H_1: *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.

3.5 Manual Analysis

To get a more qualitative sense of the context of code reuse involving the act of refactoring, we create case studies that demonstrate GitHub developers' intentions when refactoring source code for the purpose of code reuse. Case study is one of the empirical methods used for studying phenomena in a real-life context [76]. In our study, we performed a combination of manual analysis and quantitative analysis. For each case study, we checkout the corresponding commit to obtain the source code, then two authors manually analyze the code changes. We provide the commit message and its corresponding refactoring operations detected by the tool Refactoring Miner. We elaborate in detail these case studies in Section 4.3, where we report on our results.

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 [73,15,58,59]. 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 [74]. 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 TOP 3

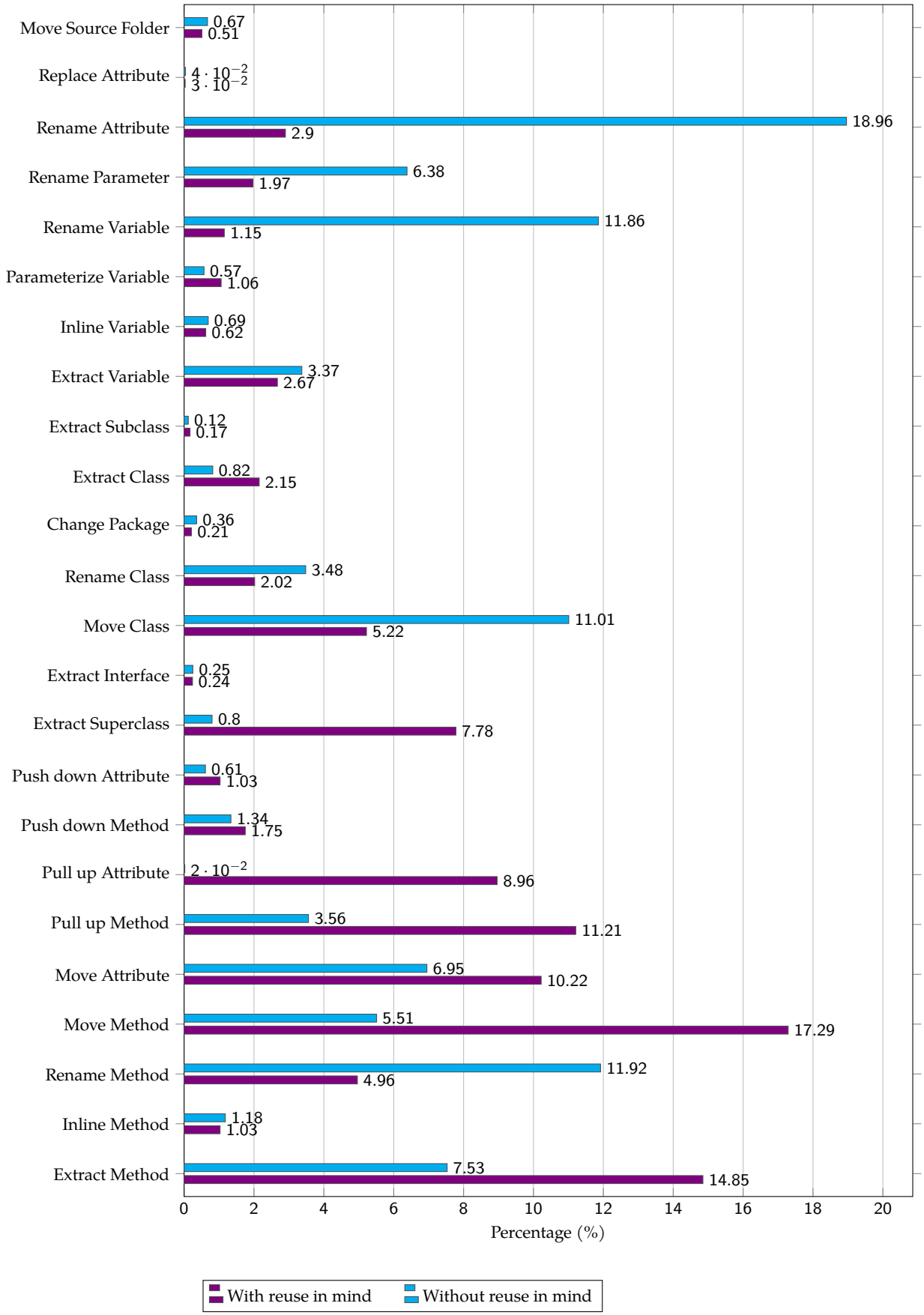


Fig. 3: Percentages of *reusability refactoring* and *non-reusability refactorings*, clustered by type.

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, 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.

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 5.

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 Meth-

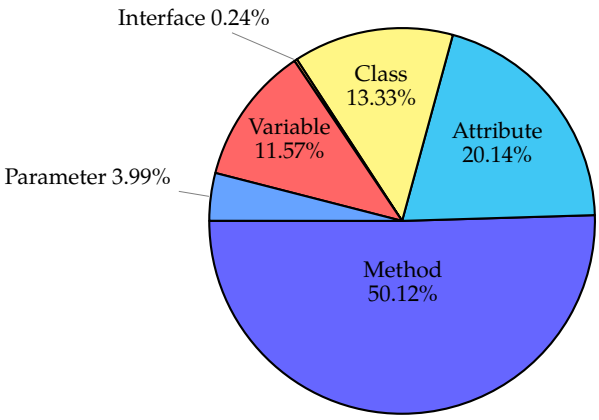


Fig. 4: Distribution of code elements in reusability refactoring commits.

ods (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 shown in Figure 6, we measure an increase in the following metrics: the Number of Paths (NPATH), the Fan-in (FANIN), the Fan-out (FANOUT), the Number of Instance Methods (NIM), and the variation is found to be not statistically significant. This observation indicates that developers increase number of possible paths, the number of calling subprograms plus global variables read, the number of called subprograms plus global variables set, and the number of instance methods. We also observe We notice the improvement of fine metrics, namely in the Nesting (MaxNesting), the Lines with Comments (CLOC), the Number of Statements (STMTC), the Number of Classes (CDL), the Number of Instance Variables (NIV), after the commits in which developers explicitly target the improvement of the reusability refactoring, but with no statistical significance. This indicates that developers reduce the maximum nesting level of control constructs, the line containing comments, the number of statements, the number of classes and the number of declared instance variables. Further, Similar to the findings of the Depth of Inheritance Tree (DIT) and the Number of Children (NOC), reusability refactorings had no impact on the Number of Base Classes (IFANIN), which emphasizes on the fact that most of the reusability refactorings are not targeting immediate base classes.

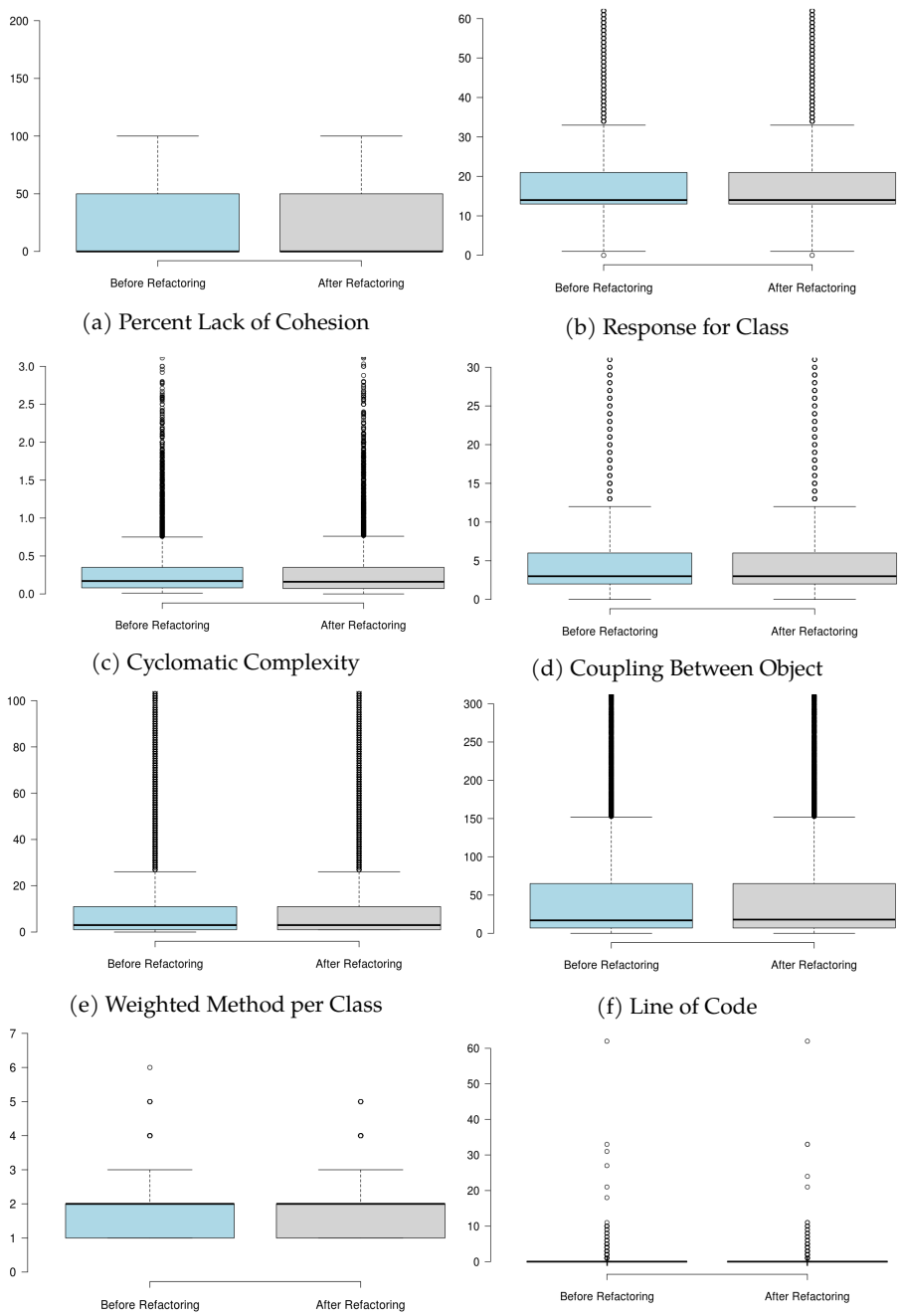
As a meta-review, the majority of state-of-the-art metrics did not capture any improvement, or captured non-significant improvement, when developers refactor their code for the purpose of reusability. This is an interesting finding for our future research directions, as we want to further increase our 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 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.

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.

4.3 **RQ3. What triggers developers to refactor the code for the purpose of code reuse?**

This research question aims at understanding the development contexts that trigger developer to perform refactoring activities for the purpose of code reuse. Upon the manual inspection of the reusability refactoring commits performed by the two authors, we identify and categorize the context of the code reuse used to describe the motivation behind the refactoring operations into nine main categories:

- Design Patterns



(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

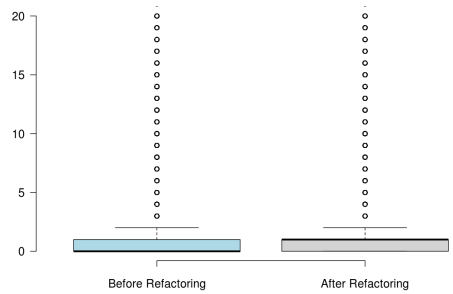
(g) Depth of Inheritance Tree

(h) Number of Children

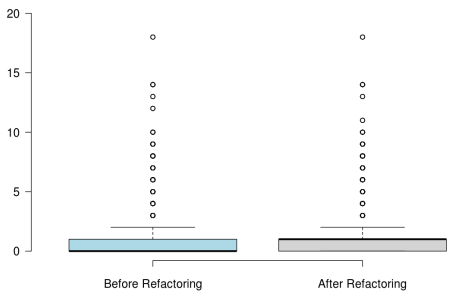
Fig. 5: Boxplots for metric values before and after reusability commits for different sets of code elements.

- Duplicate Code Removal
- API Management
- Features Updates
- Bug Fix
- Extract Reusable Component
- Test Code Management
- Visibility Changes
- Other refactoring operations

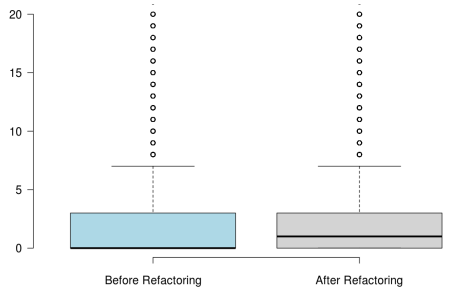
Figure 7 depicts the categorization of reusability refactoring commits. The Extract Reusable Components category had the highest number of commits with 23.96% followed by Others and Bug Fix categories with a slight advantage to the first category, since its ratio was 19.94%, while the Bug Fix category had a ratio of 14.34%. We observe that a few categories had almost a uniform distribution of refactoring classes with low variability. For instance, Test Code Management, Design Patterns, and



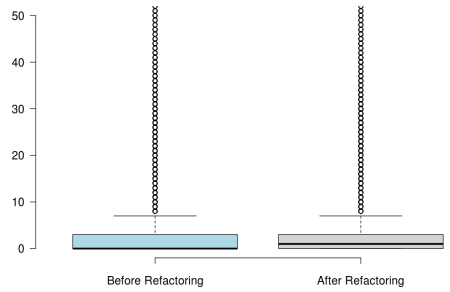
(a) Paths



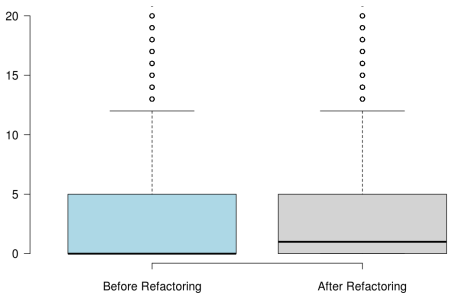
(b) Nesting



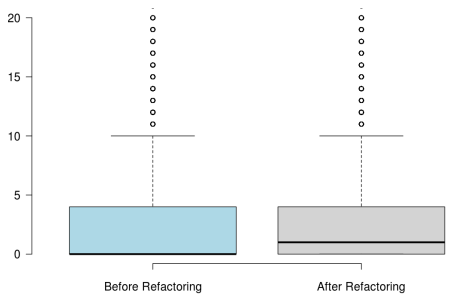
(c) Fan-in



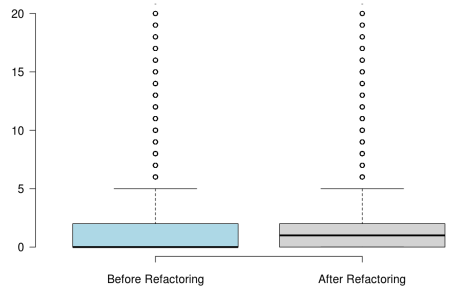
(d) Fan-out



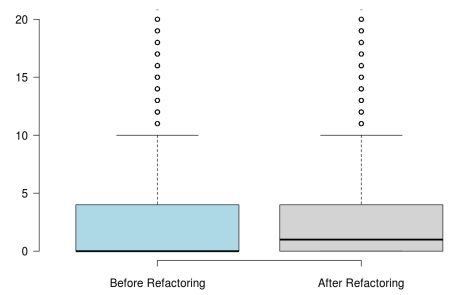
(e) Lines with Comments



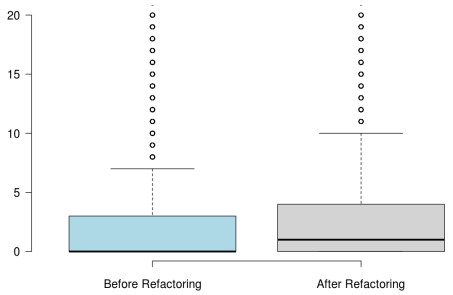
(f) Statements



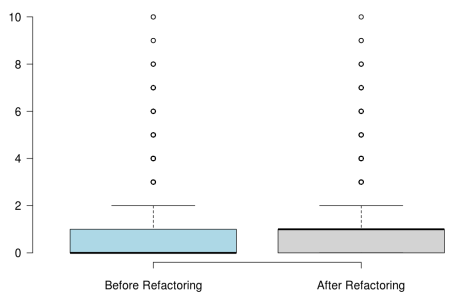
(g) Classes



(h) Instance Variables



(i) Instance Methods



(j) Base Classes

Fig. 6: Boxplots for metric values before and after reusability commits for different sets of code elements (Cont.).

Table 5: 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
Paths (NPATH)	0.5	-ve	False
Nesting (MaxNesting)	0.12	+ve	False
Fan-in (FANIN)	0.97	-ve	False
Fan-out (FANOUT)	0.94	-ve	False
Lines with Comments (CLOC)	0.71	+ve	False
Statements (STMTC)	0.19	+ve	False
Classes (CDL)	0.35	+ve	False
Instance Variables (NIV)	0.30	+ve	False
Instance Methods (NIM)	0.45	-ve	False
Base Classes (IFANIN)	1.0	No	False

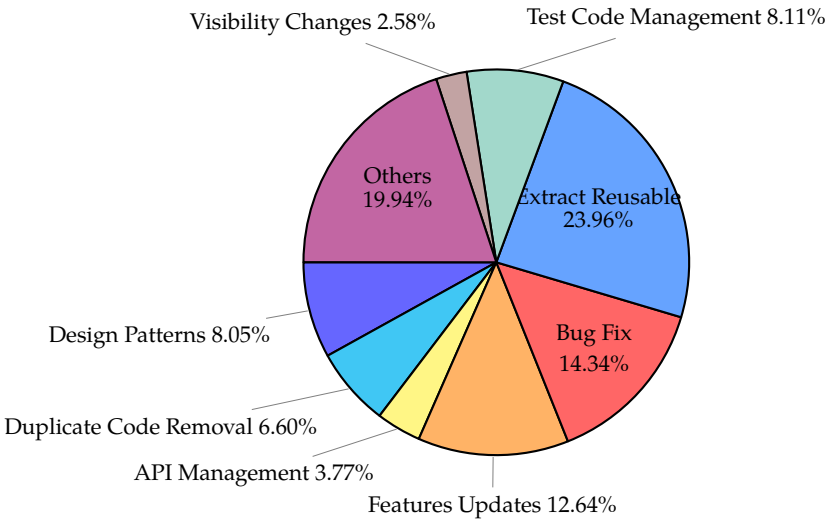


Fig. 7: Distribution of code reuse context in reusability refactoring commits.

Duplicate Code Removal had commit message distribution percentages of 8.11%, 8.05%, and 6.60%, respectively. The API Management and Visibility Changes had the least popular categories. In the following section, we provide a case study for design patterns, duplicate code removal, and API management categories due to their popularity within the context of code reuse.

4.3.1 Design Patterns.

Design patterns are technique for achieving reuse of software architectures [33]. Refactoring is another technique used for producing better maintainable and reusable designs. Design patterns and refactoring are powerful techniques for code reuse and are also related in the sense that design pattern can be used to guide refactoring [60]. Fowler et al. [32] demonstrated such correlation by illustrating how the State Pattern is used to guide transformations of a program step by step using refactoring rules. Further, Kerievsky [39] discussed the pattern-directed refactoring, which can be seen as big-step refactoring rules toward patterns. In the Gang-of-Four book [33], design patterns are classified into three categories: Creational, Structural and Behavioral patterns. In this paper, we will take one representative pattern from the Creational design pattern category to demonstrate the case of code reuse.

Case study. Builder pattern is considered one of the creational design patterns that helps construct complex objects step by step. The pattern allows to produce different types and representations of an object using the same construction code. The commit message (see Fig-

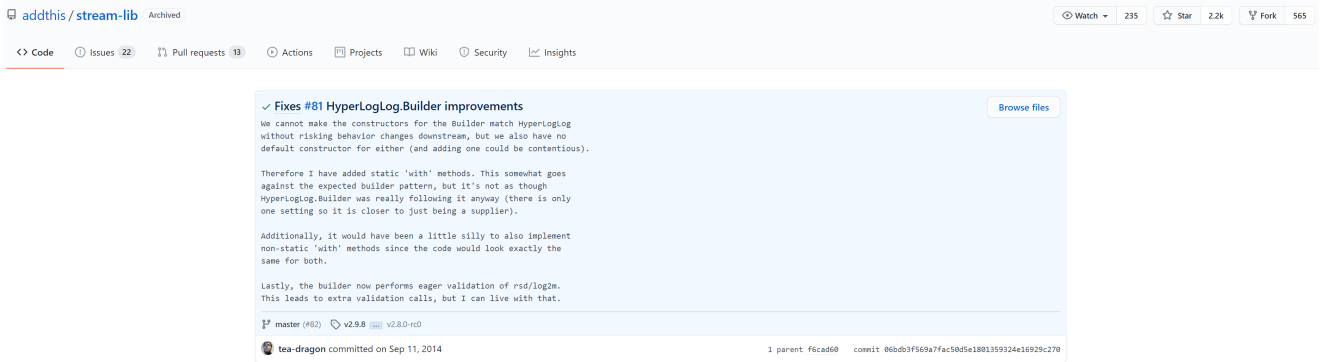


Fig. 8: Commit message stating the implementation of a Builder design pattern [1].

ure 8) indicates improvement in the builder class by making changes in the code by using Builder pattern. The developer is trying to match the constructors of the two classes `HyperLogLog` and `Builder` without changing the behaviour of the flow of code. The developer instead adds a private constructor to try and reduce the risk of behaviour. Also, another interesting comment in this commit is:

> *'This somewhat goes against the expected Builder pattern, but it's not as though HyperLogLog. Builder was really following it anyway'*

This indicates that they have used a Builder design pattern although there is a possibility that it has been tweaked according to the developer's needs. This shows the versatility of the developers and how design patterns can be changed according to one's need.

After further investigation we found that the developer has used the Builder design pattern in the code although it has been changed to accommodate the developer's needs. While we have a reader class `Builder` and the Builder interface `IBuilder` interface, the `Builder` class is embedded in the `HyperLogLog class`. The `IBuilder` interface consists of method calls `build()` and `sizeOf()`, in Listing 1, which are implemented in the `Builder` class. The `build()` method is of type `HyperLogLog` while `sizeOf()` is of Type `int` as shown in the Listing 2.

The refactoring operations performed in this commit are *Extract Method* and *Rename Method*. *Extract Method* is performed on the function public `HyperLogLog()` from which the code snippet, as shown in Listings 3 and 4, respectively, is extracted to a new function `private static void validateLog2m()`. This function is then called in the `public HyperLogLog()` and also in the private `Builder()` constructor as shown in in Listing 5. This refactoring operation can be considered as reusability, instead of writing the code snippet twice, it can be simplified and made more efficient by calling a function. The second refactoring operation that has been performed is renaming an attribute, indicated in Listing 5. The variable `private double rsd` is renamed to `private final int log2m` to keep the naming convention consistent with the rest of the code.

4.3.2 Duplicate Code Removal.

Reuse of code fragments by copying and pasting from one location to another is very common approach during software development. Code duplication is generally discouraged as it might make the software maintenance difficult [37,62,48]. For instance, if the existing code fragment is copied, bugs need to be fixed at multiple places, which makes the task inefficient and error-prone.

Case study. In this case study we are investigating the refactoring operations performed by the developers which claim to remove duplicate codes in their git commits. Extending our research in reusability refactoring commits, we venture into the domain of code smells and try to find instances where refactoring operations could aid in improving not only the quality of code but also remove code smells to some extent. The commit message in Figure 9, indicates refactoring operation being performed to remove duplicates from code. *Extract Superclass* is a common refactoring operation for removing the clones while two classes share some common methods. For example, the detected refactorings are *Extract Superclass* operation followed with several *Pull Up Method* operations.

By looking into the actual code fragments, Listings 6 and 7 show that the `DelegatingList` class and `LazyComponentList` class have duplicated methods. Note that there are totally 31 duplicated methods before the refactoring, we only show parts of them as examples for a better visualization. To fix this, the developer introduces a new class `ListImpl`, pulls up those duplicated methods to `ListImlp`, then makes both `DelegatingList` and `LazyComponentList` extends `ListImpl`. List-

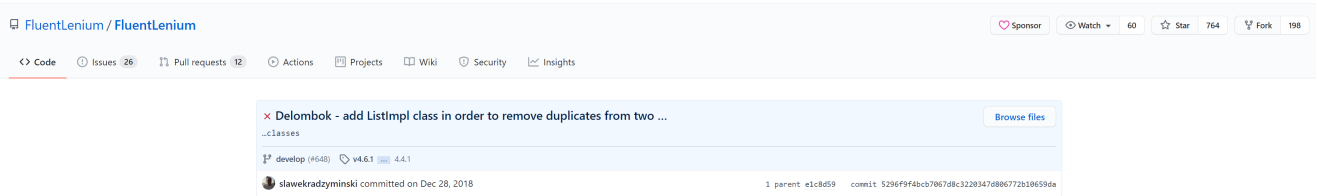
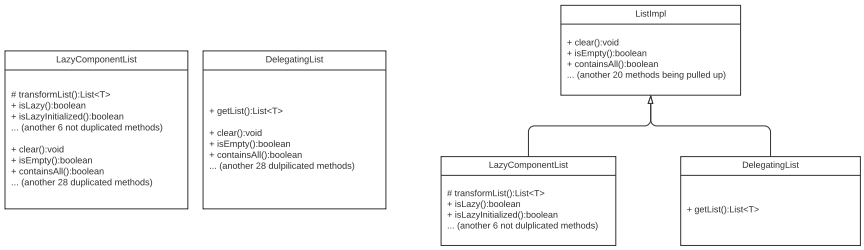


Fig. 9: Commit message stating the removal of a duplicate code [2].



(a) Before Refactoring (b) After Refactoring

Fig. 10: Duplicate code removal after the application of refactoring.

Listing 1: IBuilder Interface

```
package com.clearspring.analytics.util;

public interface IBuilder<T> {
T build();
int sizeof();
}
```

Listing 2: Implementation of build() and sizeof()

```
public static class Builder implements IBuilder
<ICardinality>, Serializable {
-         private double rsd;
+         private final int log2m;
+         /**
+          * Uses the given RSD percentage to determine
+          * how many bytes the constructed HyperLogLog
+          * will use.
+          *
+          * @deprecated Use {@link #withRsd(double)}
+          * instead. This builder's constructors did
+          * not match the (already
+          * themselves ambiguous) constructors of the
+         * HyperLogLog class, but there is no way to
+          * make them match without
+          * risking behavior changes downstream.
+          */
+         @Deprecated
         public Builder(double rsd) {
-        this.rsd = rsd;
+         this(log2m(rsd));
+         }
+         /** This constructor is private to prevent
+         behavior change for ambiguous usages.
+         (Legacy support). */
+         private Builder(int log2m) {
+         validateLog2m(log2m);
+         this.log2m = log2m;
         }
         @Override
         public HyperLogLog build() {
-         return new HyperLogLog(rsd);
+         return new HyperLogLog(log2m);
         }
         @Override
         public int sizeof() {
-         int log2m = log2m(rsd);
         int k = 1 << log2m;
         return RegisterSet.getBits(k) * 4;
         }
```

ings 8, 9 and 10 show part of the refactored codes from Github. We can see that in Listings 8 and 9 the duplicated methods such `clear`, `isEmpty` and `containsAll` are being removed, and the `DelegatingList` class and `LazyComponentList` class now extends the newly introduced `ListImpl` class. Listing 10 shows that the methods being removed in Listings 8 and 9 are now pulled up to the superclass `ListImpl`. Listings 11 and 12 show the UML diagrams before the commit and after the commit. Note that there are 31 duplicated methods before the refactoring, and only 23 of them are being pulled up. That is because there are 8 unnecessary methods overloading being permanently removed after refactoring.

The UML diagrams (see Figures 10a and 10b) clearly show that the clones have been removed by the *Extract Superclass* operation and the *Pull Up Method* operations. This case study gives us an example that the refactorings removing the code duplication such as *Extract Superclass* and *Pull Up Method* could be helpful for improving the reusability. With those duplicated methods being pulled to a superclass, if a new type of List is introduced, it can reuse those methods since they are common list operations.

Listing 3: Before refactoring

```
    @Deprecated
    public HyperLogLog(int log2m, RegisterSet
    registerSet) {
        if (log2m < 0 || log2m > 30) {
           throw new IllegalArgumentException
           ("log2m argument is "
             + log2m + " and is outside the range
         [0, 30]"); }

        this.registerSet = registerSet;
        this.log2m = log2m;
        int m = 1 << this.log2m;
        alphaMM = getAlphaMM(log2m, m);
    }
    }
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

Listing 4: After refactoring

```
+   private static void validateLog2m(int log2m) {
+        if (log2m < 0 || log2m > 30) {
+            throw new IllegalArgumentException
+
     ("log2m argument is " + log2m + " and is
+             outside the range [0, 30]");
+        }
+    }
+
    /**
     * Create a new HyperLogLog instance.  The log2m
         parameter defines the accuracy of
     * the counter.  The larger the log2m the better
      the accuracy.
@@ -117,10 +124,7 @@ public HyperLogLog(int log2m) {
     */
    @Deprecated
    public HyperLogLog(int log2m, RegisterSet
    registerSet) {
-        if (log2m < 0 || log2m > 30) {
-            throw new IllegalArgumentException
-            ("log2m argument is "
-             + log2m + " and is outside the range
-         [0, 30]"); }
+        validateLog2m(log2m);
        this.registerSet = registerSet;
        this.log2m = log2m;
        int m = 1 << this.log2m;
        alphaMM = getAlphaMM(log2m, m);
    }
```

Listing 5: validateLog2m() is called in Builder() constructor

```
public static class Builder implements IBuilder
<ICardinality>, Serializable {
-         private double rsd;
+         private final int log2m;
+         /**
+
     * Uses the given RSD percentage to determine
+
     * how many bytes the constructed HyperLogLog
+          * will use.
+          *
+          * @deprecated Use {@link #withRsd(double)}
+          * instead. This builder's constructors did
+          * not match the (already
+          * themselves ambiguous) constructors of the
+         * HyperLogLog class, but there is no way to
+          * make them match without
+          * risking behavior changes downstream.
+          */
+         @Deprecated
         public Builder(double rsd) {
-        this.rsd = rsd;
+         this(log2m(rsd));
+         }
+         /** This constructor is private to prevent
+         behavior change for ambiguous usages.
+         (Legacy support). */
+         private Builder(int log2m) {
+         validateLog2m(log2m);
+         this.log2m = log2m;
         }
```

4.3.3 API Management.

Dig et al. [27] studied the role of refactoring in API evolution with the goal of reducing the burden of reuse on maintenance. This requires either minimizing the amount of change or reducing the cost of adapting to change. For instance, when a class library that is reused in many systems is refactored, the systems that reuse it must change.

Case study. Figure 11 provides an example of API management code reuse context. Refactoring Miner detects 13 refactoring operations. Although this is a large number of refactoring operations, only 3 operations (i.e., *Move Class*) are directly related to API management. The commit message shows that the intention of the refactoring is to separate APIs into their own classes. Before the refactoring, the APIs are in their corresponding Repository classes which is a bad practice since the Repository classes are handling extra functionalities that should not belong to them. Separating APIs into their own classes will reduce the coupling in the code and allow the developer to efficiently reuse the APIs and their functionalities. By looking at the refactoring in details, we can see that the `GithubApi`, and `TokenApi` are being refactored with the move class operations. These API are being extracted and moved from their original Repository classes to their own classes to improve code reusability. Next, we will explain each refactored APIs to help better understand the developer's intention.

Listings 13, 15, show that before the refactoring `GithubApi`, `TokenApi`, interface are part of `LiveWikiRepository.java`[7]

[7] Full file path: LiveGithubRepository.java/TokenRepository.java/LiveWikiRepository.java

Listing 6: DelegatingList Class

```
public void clear() {
    getList().clear();
    }
    public void forEach(Consumer<? super T> action) {
    getList().forEach(action);
    }
    public <T> T[] toArray(IntFunction<T[]> generator
        )
    {
    return getList().toArray(generator);
    }
    public boolean isEmpty() {
    return getList().isEmpty();
    }
    public boolean removeIf(Predicate<?super T>filter
        )
    {
    return getList().removeIf(filter);
    }
    public Spliterator<T> spliterator() {
    return getList().spliterator();
    }
    public T set(int index, T element) {
    return getList().set(index, element);
    }
    public boolean containsAll(Collection<?> c) {
    return getList().containsAll(c);
    }
```

Listing 7: LazyComponentList Class

```
  public void clear() {
    getList().clear();
    }
    public void forEach(Consumer<?super T> action) {
    getList().forEach(action);
    }
    public <T> T[] toArray(IntFunction<T[]> generator
        )
    {
    return getList().toArray(generator);
    }
    public boolean isEmpty() {
    return getList().isEmpty();
    }
    public boolean removeIf(Predicate<?super T>filter
        )
    {
    return getList().removeIf(filter);
    }
    public Spliterator<T> spliterator() {
    return getList().spliterator();
    }
    public T set(int index, T element) {
    return getList().set(index, element);
    }
    public boolean containsAll(Collection<?> c) {
    return getList().containsAll(c);
    }
```

Listing 8: LazyComponentList Class (before removing duplicate)

```
- public class LazyComponentList<T> implements
- List<T>, WrapsElements, LazyComponents<T> {
    private final ComponentInstantiator instantiator;
    private final Class<T> componentClass;

@@ -120,127 +110,4 @@ public String toString() {
    return isLazyInitialized() ? getList().toString()
    : elements.toString();
    }
-    public void clear() {
-    getList().clear();
-    }
-
     public void forEach(Consumer<? super T> action) {
-    getList().forEach(action);
-    }
-
     public <T> T[] toArray(IntFunction<T[]> generator)
-    {
-    return getList().toArray(generator);
-    }
-    public boolean isEmpty() {
-    return getList().isEmpty();
-    }
-
     public boolean removeIf(Predicate<?super T>filter)
-    {
-    return getList().removeIf(filter);
-    }
-    public Spliterator<T> spliterator() {
-    return getList().spliterator();
-    }
-    public T set(int index, T element) {
-    return getList().set(index, element);
-    }
-    public boolean containsAll(Collection<?> c) {
-   return getList().containsAll(c);
-    }
```

Listing 9: LazyComponentList Class (after removing duplicate)

```
+ public class LazyComponentList<T> extends
+ ListImpl<T> implements List<T>, WrapsElements,
+ LazyComponents<T>
{
    private final ComponentInstantiator instantiator;
    private final Class<T> componentClass;

@@ -120,127 +110,4 @@ public String toString() {
    return isLazyInitialized() ? getList().toString()
    : elements.toString();
    }
    }
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

Listing 10: DelegatingList Class (before removing duplicate)

```
- public class DelegatingList<T> implements List<T> {
    protected final List<T> list;
    /**
@@ -34,127 +24,4 @@ public DelegatingList(List<T> list)
    {
    return list;
    }
-   public void clear() {
-    getList().clear();
-    }
-    public void forEach(Consumer<? super T> action) {
-    getList().forEach(action);
-    }
-    public <T> T[] toArray(IntFunction<T[]> generator)
-    {
-    return getList().toArray(generator);
-    }
-    public boolean isEmpty() {
-    return getList().isEmpty();
-   }
-    public boolean removeIf(Predicate<?super T>filter)
-    {
-    return getList().removeIf(filter);
-    }
-    public Spliterator<T> spliterator() {
-    return getList().spliterator();
-    }
-    public T set(int index, T element) {
-    return getList().set(index, element);
-    }
-    public boolean containsAll(Collection<?> c) {
-    return getList().containsAll(c);
-    }
```

Listing 11: DelegatingList Class (after removing duplicate)

```
+ public class DelegatingList<T> extends ListImpl<T>
+ implements List<T> {
    protected final List<T> list;

    /**
@@ -34,127 +24,4 @@ public DelegatingList(List<T> list)
    {
        return list;
    }
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

Listing 12: ListImpl Class

```
+ public abstract class ListImpl<T> implements List<T>
+   {
+    public ListImpl() {
+    super();
+   }
+    public abstract List<T> getList();
+    public void clear() {
+    getList().clear();
+    }
+    public boolean isEmpty() {
+    return getList().isEmpty();
+    }
+    public T set(int index, T element) {
+    return getList().set(index, element);
+    }
+    public boolean containsAll(Collection<?> c) {
+    return getList().containsAll(c);
+    }
+    public List<T> subList(int fromIndex, int toIndex)
+    {
+    return getList().subList(fromIndex, toIndex);
+    }
+    public boolean add(T e) {
+    return getList().add(e);
+    }
+    public boolean remove(Object o) {
+    return getList().remove(o);
+    }
```

and is highly coupled with it, which is bad for code reuse. Listings 14. 16, show after the refactoring, for better reuse of the `GithubApi`, `TokenApi`, interface, it is being extracted and moved to `WikiApi.java`[8]. In this case, like Listings17 and 18 show, when the other classes need `GithubApi`, `TokenApi`, they can just import and use it instead of using the whole `LiveGithubRepository`, `TokenRepository`, that are loaded with redundant functionalities.

4.3.4 *Feature Updates.*

Code reuse can be used for the purpose of adding a new function or update an existing one.

Case study. Figure 12 shows the commit message and the intentions of the developer which clearly state the purpose of this refactoring operation, that is, code reusability. The developer explains the need to no longer register runners explicitly as agent searches for runner implementations, which can mean that the developer intended to remove these runner implementations from the base classes and store them separately which can be called implicitly as and when needed.

In Listings 19, 20, and 21, we can see that all these classes, namely: `PackagesInstallerRunner`, `PackagesPublishRunner` and `PackRunner` respectively implement interfaces `AgentBuildRunner` and `AgentBuildRunnerInfo` provided by jetbrains, that explicitly running

[8] Ful file path: GithubApi.java/TokenApi.java/WikiApi.java

Fig. 11: Commit message stating the management of the API [3].

function `getRunnerInfo()` provided by `AgentBuildRunnerInfo()` and `canRun()`.

Apart from the two functions the constructors in each class in Listings 19, 20, and 21, repeat variables `myNuGetActionFactory` and `myNuGetActionFactory` which can be considered as repeating code or duplicate code. This can affect the time consumed by the program overall at runtime as the program runs all these lines of code even when not required. Thus, the developer extracts these repeating components from the classes in Listings 19, 20, and 21 and moves them to a superclass `NuGetRunnerBase` in Listing 22. This superclass is then extended by each class and this superclass implements the interfaces `AgentBuildRunner` and `AgentBuildRunnerInfo` provided by jetbrains. This is considered an example of feature update, as the features used from the jetbrains interface classes and the common functionalities among the classes: `PackagesInstallerRunner`, `PackagesPublishRunner` and `PackRunner`, are now being used by each class implicitly via the `NuGetRunnerBase`, thus promoting reusability as well.

4.3.5 Bug Fix.

Due to the habit of copy-and-paste in programming, similar code fragments may contain the same bug which may be neglected to fix during software maintenance. Developers can reuse existing code and interleave it with fixing the bug.

Case study. This case study provides an example of Bug Fix context. The commit (see Figure 13) is in project JLine which is a Java library for handling console input. A user reported a bug in the code that *the arrow keys are all reported as ASCII code 27 with the `readCharacter` method*. The problem behind this is there is no real ASCII code for the arrow keys, and they are usually represented by two characters rather than one character. One of the possible solutions is to allow the user to define their own key bindings in KeyMap, then while reading an input stream, wait until a valid binding is found in KeyMap. Currently, method `readLine` contains the statements related to processing the predefined key bindings, and there is no access point for the user to use their own key bindings. To decouple the functionality of processing key bindings from a large method and improve the code reusability, the developer decides to extract those statements from the `readLine` method.

As Listings 23 and 24 show, the developer extracted the part processing the key bindings in method `readLine` to a new method `readBinding` which can be publicly reused. The `readBinding` method will take a KeyMap as an argument, which could be defined by the user. It will block until there is a matching binding found or reach the end of line, so it can support reading the two characters arrow keys.

4.3.6 Extract Reusable Component.

Extracting code elements can be performed with reusability in mind and there is a high possibility that used components of the code were extracted.

Case study. The developer mentions pushing some common methods down into an abstract class in Figure 14, which can be considered as an extract refactoring operation and thus help improve reusability as the code can now be used by other classes as well and the implementations can be easily changed in a common file unless overridden. This saves the developer time and makes programming very easy and structured.

As shown in Listing 25, the methods in class PubSubElementProcessorAbstract have been extracted from class NodeCreate as shown in Listings 25 and 26. Upon further investigation, we checked if there were other classes that extended the `PubSubElementProcessorAbstract` class and we found multiple other classes that extend this class. Also, we searched for any functions used from `PubSubElementProcessorAbstract` class in any of the classes that extend it and found a few of classes, which show the use of functions `getNodeConfigurationHelper()` and `setOutQueue()`.

4.3.7 Test Code Management.

Code reuse helps in reducing testing efforts. Reusing the code fragments, which have already been unit tested,

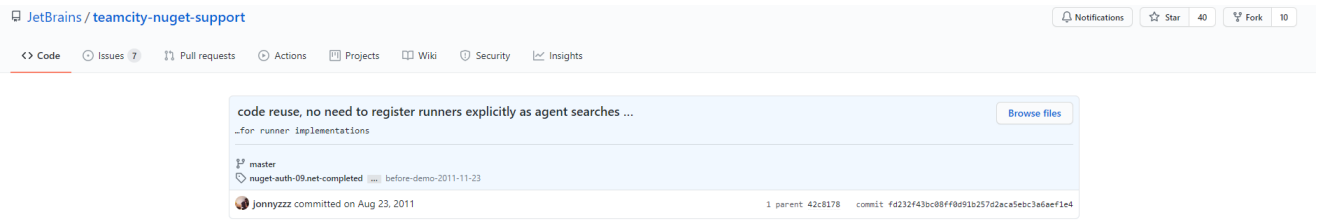


Fig. 12: Commit message stating the update of a feature [4].

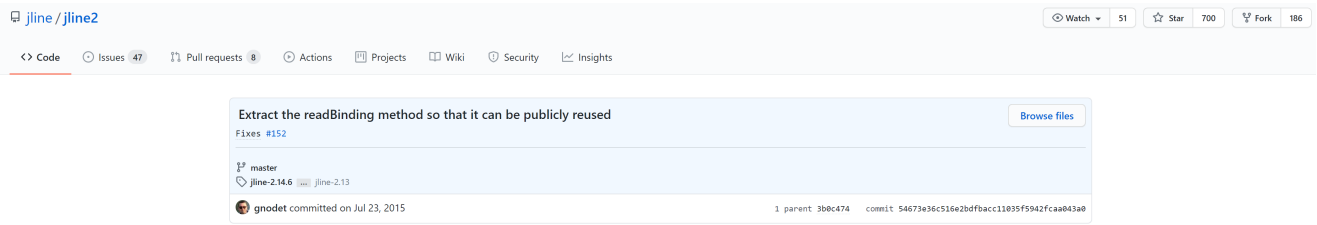


Fig. 13: Commit message stating the fixing of a bug [5].

reduces and saves testing effort, both in terms of avoiding the need to write additional tests and also by not having to run those additional tests each time the full test suite is run.

Case study. Figure 15 provides an example of Test Code Management context in which developer refactored the `transcoder` tests for reuse. There are 39 detected refactoring operations including 1 *Extract Superclass* and 38 *Pull Up Method*, where 3 classes are involved - `WhalinTranscoderTest`, `SerializingTranscoderTest` and `BaseTranscoderCase`. The `WhalinTranscoderTest` and `SerializingTranscoderTest` share 18 duplicated methods. To achieve the goal stated by developer in the commit message, the developer pulled up 18 duplicated methods, and extracted 2 methods(`testShort`, `testCharacter`) that are not duplicated but helpful for future reuse, to a newly introduced class `BaseTranscoderCase`. In the future, the classes that need to be tested could reuse the test cases in `BaseTranscoderCase` to perform some basic tests such as type check and boundary check.

For example, Listings 27 and 28 illustrate the refactoring of 4 duplicated methods - `testLong`, `testInt`, `testChar`, `testBoolean` shared by both `SerializingTranscoderTest` and `WhalinTranscoderTest`, and there are 1 method `testShort` that is only in `WhalinTranscoderTest`. Listing 29 shows after the refactoring, all of these 5 methods are being pulled up and extracted to the new class `BaseTranscoderCase` to improve reusability.

Listing 13: Before refactoring - LiveGithubRepository.java

```
public class LiveGithubRepository implements
GithubRepository {

-    interface GithubApi {
-        @GET("repos/d4rken/reddit-android-appstore
-        /releases/latest")
-        Observable<Release> getLatestRelease();
+    private Observable<GithubApi.Release>
+    latestReleaseCache;
+    private Observable<List<GithubApi.Contributor>>
+    latestContributorsCache;

-        @GET("repos/d4rken/reddit-android
-        appstore/contributors")
-        Observable<List<Contributor>>
-        getContributors();
+
     public LiveGithubRepository(GithubApi githubApi) {
+        this.githubApi = githubApi;
    }

    @Override
-    public Observable<Release> getLatestRelease() {
+    public Observable<GithubApi.Release>
+    getLatestRelease() {
        if (latestReleaseCache == null)
        latestReleaseCache
        = githubApi.getLatestRelease().cache();
        return latestReleaseCache;
    }

    @Override
-    public Observable<List<Contributor>>
-    getContributors() {
+   public Observable<List<GithubApi.Contributor>>
+   getContributors() {
        if (latestContributorsCache == null)
        latestContributorsCache = githubApi
        .getContributors().cache();
        return latestContributorsCache;
    }
```

Listing 14: After refactoring - GithubApi.java

```
+ package subreddit.android.appstore.backend.github;

+ import com.google.gson.annotations.SerializedName;

+ import java.util.Date;
+ import java.util.List;

+ import io.reactivex.Observable;
+ import retrofit2.http.GET;

+ public interface GithubApi {
+     String BASEURL = "https://api.github.com/";

+     @GET("repos/d4rken/reddit-android-appstore/
+     releases/latest")
+     Observable<Release> getLatestRelease();

+     @GET("repos/d4rken/reddit-android-appstore/
+     contributors")
+     Observable<List<Contributor>> getContributors();

+     class Release {
+         @SerializedName("url") public String
+         releaseUrl;
+         @SerializedName("tag_name") public String
+         tagName;
+         @SerializedName("name") public String
+         releaseName;
+         @SerializedName("body") public String
+         releaseDescription;
+         public boolean prerelease;
+         @SerializedName("published_at") public Date
+         publishDate;
+         public List<Assets> assets;

+        public static class Assets {
+             @SerializedName("browser_download_url")
+             public String downloadUrl;
+             public long size;
+         }
+     }

+     class Contributor {
+     @SerializedName("login") public String username;
+     }
+}
```

Listing 15: Before refactoring - TokenRepository.java

```
-  interface TokenApi {
-         @FormUrlEncoded
-         @POST("api/v1/access_token")
-         Observable<Token> getUserlessAuthToken(
-            @Header("Authorization") String
-            authentication,
-            @Field("device_id") String deviceId,
-            @Field("grant_type") String grant_type,
-            @Field("scope") String scope
-         );
-     }
```

4.3.8 Visibility Changes.

Case study. The developers mention splitting up line of code:- util.MojiBakeMapper and Core::MojiBake for reuse in the commit message shown in Figure 16. This refers to `mojibake.rb` file, Listing 30, although the extract refactoring operation is seen to be performed on `MojiBakerFilter` and `MojiMapper` indicated by the

Listing 16: After refactoring - TokenApi.java

```
+ package subreddit.android.appstore.backend.reddit;


+ import io.reactivex.Observable;
+ import retrofit2.http.Field;
+ import retrofit2.http.FormUrlEncoded;
+ import retrofit2.http.Header;
+ import retrofit2.http.POST;

+ public interface TokenApi {
+     String BASEURL = "https://www.reddit.com/";

+     @FormUrlEncoded
+     @POST("api/v1/access_token")
+     Observable<TokenApi.Token> getUserlessAuthToken
+     (@Header("Authorization") String authentication,
+     @Field("device_id") String deviceId,
+     @Field("grant_type") String grant_type,
+     @Field("scope") String scope);

+     class Token {
+         String access_token;
+         String token_type;
+         long expires_in;
+         String scope;
+         final long issuedTime =
+         System.currentTimeMillis();

+         public boolean isExpired() {
+            return System.currentTimeMillis()
+            > issuedTime + expires_in * 1000;
+         }

+         public String getAuthorizationString() {
+             return token_type + " " + access_token;
+         }
+     }
+ }
```

Listing 31, respectively. The extract refactoring operation is performed on `MojiBakeFilter` from which the private method recover of type `CharSequence` was extracted to `MojiMapper` class as a public method recover of type `CharSequence`. Although the commit message suggests reuse, the refactoring operation is performed on another file, which can be considered as a case study for visibility changes as the access modifier of the method recover of type `Charsequence` was changed from private to public. On further investigation, we searched for any traces of reusability of the method recover being extracted and found a few classes that used recover method.

Summary. Code reuse helps wit various software development activities. Our analysis found that the extraction of reusable component leads the rationale behind reuse-related code changes. Refactoring reusable code also helps with API management, duplicate code removal, changes to visibility, test code reorganization, and implementing design patterns.

Listing 17: Github API (before)

```
import android.support.annotation.Nullable;

- import subreddit.android.appstore.backend.github.
- GithubRepository;

import subreddit.android.appstore.util.mvp.
    BasePresenter;
import subreddit.android.appstore.util.mvp.BaseView;

@@ -14,22 +14,22 @@

        void selectFilter(CategoryFilter filter);

-        void showUpdateSnackbar(@Nullable
-        GithubRepository.Release release);


        void showUpdateErrorToast();

-        void enableUpdateAvailableText(@Nullable
-        GithubRepository.Release release);


        void showDownload(String url);

-        void showChangelog(GithubRepository.Release
-        release);

    }

    interface Presenter extends BasePresenter<View> {
        void notifySelectedFilter(CategoryFilter
        categoryFilter);

-   void downloadUpdate(GithubRepository.Release
-   release);


-   void buildChangelog(GithubRepository.Release
-   release);

    }
}
```

Listing 18: Github API (after)

```
import android.support.annotation.Nullable;

+ import subreddit.android.appstore.backend.github.
+ GithubApi;
import subreddit.android.appstore.util.mvp.
    BasePresenter;
import subreddit.android.appstore.util.mvp.BaseView;

@@ -14,22 +14,22 @@

        void selectFilter(CategoryFilter filter);

+        void showUpdateSnackbar(@Nullable GithubApi
+        .Release release);

        void showUpdateErrorToast();


+        void enableUpdateAvailableText(@Nullable
+        GithubApi.Release release);

        void showDownload(String url);

+
    void showChangelog(GithubApi.Release release);
    }

    interface Presenter extends BasePresenter<View> {
        void notifySelectedFilter(CategoryFilter
        categoryFilter);

+   void downloadUpdate(GithubApi.Release release);


+   void buildChangelog(GithubApi.Release release);
    }
}
```

5 Implications

The main implications of this study are as follows:

– **Further exploiting quality metrics and reusability refactoring.** The existing literature discusses different automatic refactoring approaches that help practitioners in detecting anti-patterns or code smells. More recently, Baqais and Alshayeb [21] show that there is an increase in the number of studies on automatic refactoring approaches and researchers have begun exploring how machine learning can be used in identifying refactoring opportunities. Since the features play a vital role in the quality of the obtained machine learning models, this study can help determine which metrics can be used as effective features in machine learning algorithms to accurately predict refactoring opportunities at different levels of granularity (i.e., class, method, field), which can assist developers in automatically making their decisions. For example, using the most impactful metrics as a feature to predict whether a given piece of code should undergo a specific refactoring operation makes developers more confident in accepting the recommended refactoring or picking out the most suitable reusable candidate. Such knowledge is needed as, in practice, the built model should require as little data as possible. Further, since we observe from RQ2 that most of the reusability metrics did not capture any improvement, we plan to conduct more experiments to validate the effectiveness of reusability metrics to explore if the observations are due to the appropriateness of the reusability quality metrics or to the needed validation and clarity of developers perception.
Reducing the amount of efforts to refactor the code to improve its reusability. Generally, reused classes tend to be more maintainable than native classes. One particular aspect of refactoring is to increase the reusability of software components. However, a recent study [30] found that the reused code is in need for various refactorings even though the produced code obeys to good object-oriented practices. Our study sheds light on developers' strategies to refactor the code to improve its reusability that is different from refactoring applied in mainstream development (e.g., reusability refactorings heavily impact methods while typical refactorings, impact all code elements). Understanding such strategies assist in providing developers with a more efficient way to utilize existing code to create new functionality, and facilitate development and maintenance since less work is needed to accomplish additional functionality.

Listing 19: PackagesInstallerRunner Class

```
- public class PackagesInstallerRunner implements
- AgentBuildRunner, AgentBuildRunnerInfo {
-   private static final Logger LOG = Logger.
-   getInstance
-  (PackagesInstallerRunner.class.getName());

-  private final NuGetActionFactory
- myNuGetActionFactory;
-  private final PackagesParametersFactory
-  myParametersFactory;

-  public PackagesInstallerRunner(@NotNull
-  final NuGetActionFactory nuGetActionFactory,
+ public class PackagesInstallerRunner extends
+  NuGetRunnerBase {
+  public PackagesInstallerRunner(@NotNull
+  final NuGetActionFactory actionFactory,
  @NotNull final PackagesParametersFactory
  parametersFactory) {
-    myNuGetActionFactory = nuGetActionFactory;
-    myParametersFactory = parametersFactory;
+    super(actionFactory, parametersFactory);
  }

  @NotNull
@@ -72,7 +66,7 @@ private void createStages(@NotNull
final BuildRunnerContext context,
            parameters,
            context.getBuild().getBuildLogger(),
            new PackagesInstallerBuilder(
-                    myNuGetActionFactory,
+                    myActionFactory,
                    stages,
                    context,
                    installParameters,
@@ -82,27 +76,8 @@ private void createStages(@NotNull
final BuildRunnerContext context,
    stages.getLocateStage().pushBuildProcess(locate);
  }

-  @NotNull
-  public AgentBuildRunnerInfo getRunnerInfo() {
-    return this;
-  }

  @NotNull
  public String getType() {
    return PackagesConstants.INSTALL_RUN_TYPE;
  }

-  public boolean canRun(@NotNull
-  BuildAgentConfiguration agentConfiguration) {
-    if (!agentConfiguration.getSystemInfo().
 -   isWindows()) {
      LOG.warn("NuGet packages installer available
      only under windows");
      return false;
    }
```

Listing 20: PackagesPublishRunner Class

```
- public class PackagesPublishRunner implements
- AgentBuildRunner, AgentBuildRunnerInfo {
-  private static final Logger LOG =

-  Logger.getInstance(PackagesPublishRunner.
-  class.getName());

-  private final NuGetActionFactory myActionFactory;
-  private final PackagesParametersFactory
-  myParametersFactory;

+ public class PackagesPublishRunner extends
+ NuGetRunnerBase {
  public PackagesPublishRunner
  (@NotNull final NuGetActionFactory actionFactory,
  @NotNull final PackagesParametersFactory
  parametersFactory) {
-    myActionFactory = actionFactory;
-    myParametersFactory = parametersFactory;
+    super(actionFactory, parametersFactory);
  }

  @NotNull
@@ -63,27 +58,8 @@ public void
fileFound(@NotNull File file) throws
     RunBuildException {
    return process;
  }

-  @NotNull
-  public AgentBuildRunnerInfo getRunnerInfo() {
-    return this;
-  }

  @NotNull
  public String getType() {
    return PackagesConstants.PUBLISH_RUN_TYPE;
  }

-  public boolean canRun(@NotNull
-  BuildAgentConfiguration agentConfiguration) {
-    if (!agentConfiguration.getSystemInfo().
-     isWindows()) {
-      LOG.warn("NuGet packages installer available
-      only under windows");
-     return false;
-   }
```

- **Examining the code reuse potentials with refactoring**. Our study reveals the context in which developers refactor the code for the purpose of improving code reusability. Our future research direction can focus on providing a comprehensive taxonomy of reusability-aware refactorings. This taxonomy can show various contexts of reusability refactoring and demonstrate different forms of reuse. Thereafter, researchers can build on top of our RQ3 findings to better understand developers practices and investigate to what extent this reusability-aware refactoring taxonomy improves the quality of the system.
- **Understanding the completeness of the quality metrics in capturing the reusability improvements as documented by developers.** We observed that not all of the quality metrics are able to capture the reusability improvement as perceived by developers in their commit messages. While quality metrics can help pinpoint design flaws for refactoring recommendation systems, such recommendation would

Listing 21: PackRunner Class

```
- public class PackRunner implements AgentBuildRunner,
- AgentBuildRunnerInfo {
-  private static final Logger LOG =
-  Logger.getInstance(PackRunner.class.getName());

-  private final NuGetActionFactory myActionFactory;
-  private final PackagesParametersFactory
-  myParametersFactory;

+ public class PackRunner extends NuGetRunnerBase {
  public PackRunner(
    @NotNull final NuGetActionFactory actionFactory,
    @NotNull final PackagesParametersFactory
    parametersFactory) {
-    myActionFactory = actionFactory;
-    myParametersFactory = parametersFactory;
+    super(actionFactory, parametersFactory);
  }

  @NotNull
@@ -54,28 +49,8 @@ public BuildProcess
createBuildProcess(@NotNull final AgentRunningBuild
runningB
    return process;
  }

-  @NotNull
-  public AgentBuildRunnerInfo getRunnerInfo() {
-    return this;
-  }

  @NotNull
  public String getType() {
    return PackagesConstants.PACK_RUN_TYPE;
  }

-  public boolean canRun(@NotNull
-  BuildAgentConfiguration
-    agentConfiguration) {
-    if (!agentConfiguration.getSystemInfo()
-    .isWindows()) {
-      LOG.warn("NuGet packages installer available
-      only under windows");
-      return false;
-    }
-
-    if (!agentConfiguration.
-      getConfigurationParameters().
-    containsKey(DotNetConstants.
-    DOT_NET_FRAMEWORK_4_x86)) {
-      LOG.warn("NuGet requires
-    .NET Framework 4.0 x86 installed");
-      return false;
-    }

-    return true;
-  }
```

Listing 22: NuGetRunnerBase Class

```
/**
 * @author Eugene Petrenko (eugene.petrenko@gmail.com
     )
 *         Date: 23.08.11 18:32
 */
+ public abstract class NuGetRunnerBase implements
+ AgentBuildRunner, AgentBuildRunnerInfo {
+  protected final Logger LOG = Logger.getInstance
+  (getClass().getName());

+  protected final NuGetActionFactory
+ myActionFactory;
+  protected final PackagesParametersFactory
+  myParametersFactory;

+  public NuGetRunnerBase(NuGetActionFactory
+ actionFactory,
+  PackagesParametersFactory parametersFactory) {
+    myActionFactory = actionFactory;
+    myParametersFactory = parametersFactory;
+ }

+  @NotNull
+  public AgentBuildRunnerInfo getRunnerInfo() {
+    return this;
+  }

+  @NotNull
+  public abstract String getType();

+  public boolean canRun(@NotNull
+  BuildAgentConfiguration
+  agentConfiguration) {
+    if (!agentConfiguration.getSystemInfo()
+    .isWindows()) {
+      LOG.warn("NuGet packages installer available
+      only under windows");
+      return false;
+    }

+    if (!agentConfiguration.
+    getConfigurationParameters().
+    containsKey(DotNetConstants.
+    DOT_NET_FRAMEWORK_4_x86)) {
+      LOG.warn("NuGet requires .NET Framework
+      4.0 x86 installed");
+      return false;
+    }

+    return true;
+  }
+ }
```

be meaningful if it is complemented with qualitative insights from developers.

– **Extending/varying the basic structure of design patterns to support real-world applicability.** Our qualitative analysis (see Figure 8) shows that developers in practice are not following the exact basic structure of the design patterns that usually appears in the original documentation of the well-known catalog of patterns [33]. Developers instead alter the implemented design patterns according to the developer's needs. Future researchers are encouraged to perform a deeper investigation on understanding the mismatch between theory and practice, and propose an approach that can detect not only patterns in their basic form but also modified versions of patterns.

6 Threats to Validity

Internal Validity. 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. [32]. However, in a previous study, Murphy-Hill

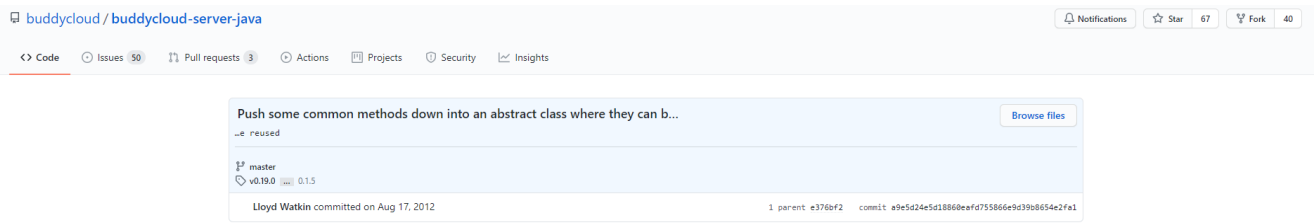


Fig. 14: Commit message stating the extraction of reusable component [6].

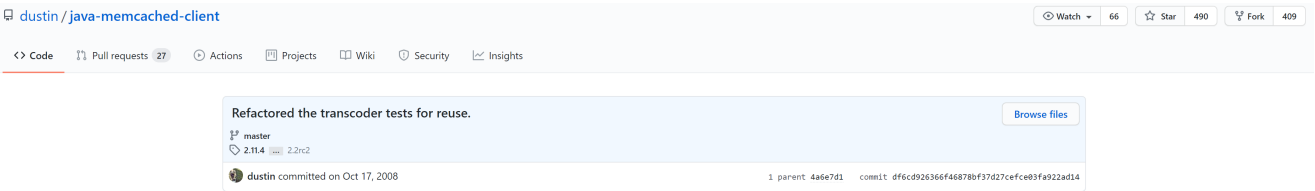


Fig. 15: Commit message stating management of the test code [7].

Listing 23: Before refactoring

```
-   if (c == 27
-     && pushBackChar.isEmpty()
-     && in.isNonBlockingEnabled()
-     && in.peek(escapeTimeout) == -2) {
-         Object otherKey = ((KeyMap) o).
-          getAnotherKey();
-   if (otherKey == null) {
-         otherKey = ((KeyMap) o).
-         getBound(Character.toString((char) c));}
-         o = otherKey;
-   if (o == null || o instanceof KeyMap) {
-                          continue; }
-         sb.setLength(0);  }
-   else {                 continue; }
 *
 *
-    while ( o == null && sb.length() > 0 ) {
-     c = sb.charAt( sb.length() - 1 );
-         sb.setLength( sb.length() - 1 );
-         Object o2 = getKeys().getBound( sb );
-         if ( o2 instanceof KeyMap ) {
-             o = ((KeyMap) o2).getAnotherKey();
-         if ( o == null ) {
-                        continue;
-         } else {
-                        pushBackChar.push( (char) c );
-         }}
*
*
*
*
*
*
```

Listing 24: After refactoring

```
+   if (c == ESCAPE
+     && pushBackChar.isEmpty()
+     && in.isNonBlockingEnabled()
+     && in.peek(escapeTimeout) == READ_EXPIRED) {
+         Object otherKey = ((KeyMap) o).
+         getAnotherKey();
+     if (otherKey == null) {

+
      otherKey = ((KeyMap) o).getBound(Character.
+                    toString((char) c));
+                    }
+                    o = otherKey;
+                    if (o == null ||
+                    o instanceof KeyMap) {
+                        continue;
+                    }
+                    opBuffer.setLength(0);
+                } else {
+                    continue;
+                }
*
*
+     while (o == null && opBuffer.length() > 0) {
+       c = opBuffer.charAt(opBuffer.length() - 1);
+
      opBuffer.setLength(opBuffer.length() - 1);
+                Object o2 = keys.getBound(opBuffer);
+                if (o2 instanceof KeyMap) {
+
      o = ((KeyMap) o2).getAnotherKey();
+                    if (o == null) {
+                        continue;
+                    } else {
+                        pushBackChar.push((char) c);
+                    }}
```

et al. [52] 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. [73] 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

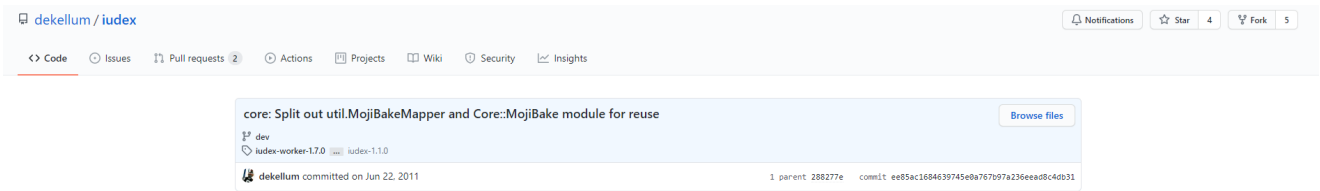


Fig. 16: Commit message stating the case of visibility changes [8].

Listing 25: PubSubElementProcessorAbstract Class

```
+ public abstract class PubSubElementProcessorAbstract
+    implements PubSubElementProcessor
+ {

+    protected BlockingQueue<Packet> outQueue;
+    protected DataStore             dataStore;
+    protected Element               element;
+    protected IQ                    response;
+    protected IQ                    request;
+    protected JID                   actor;
+    protected String                serverDomain;
+    protected String                topicsDomain;
+    protected String                node;
+    protected Helper     configurationHelper;
+   public void setOutQueue(BlockingQueue<Packet>
+   outQueue)
+   {
+      this.outQueue = outQueue;
+   }
+   public void setDataStore(DataStore dataStore)
+   {
+      this.dataStore = dataStore;
+   }
+   public void setServerDomain(String domain)
+   {
+      serverDomain = domain;
+   }
*
*
*
*
*
*
*
*
```

Listing 26: NodeCreate Class

```
- public class NodeCreate implements
- PubSubElementProcessor
+ public class NodeCreate extends
+ PubSubElementProcessorAbstract
{
-    private static final Logger LOGGER =
-    Logger.getLogger(NodeCreate.class);
-    private BlockingQueue<Packet> outQueue;
-    private DataStore             dataStore;
-   private Element               element;
-    private IQ                    response;
-    private IQ                    request;
-    private JID                   actor;
-   private String                serverDomain;
-   private String                topicsDomain;
-   private String                node;
-   private Helper     configurationHelper;
    private static final Pattern nodeExtract =
        Pattern
    .compile("^/user/[^@]+@([^/]+)/[^/]+$");
    private static final String NODE_REG_EX
    = "^/user/[^@]+@[^/]+/[^/]+$";
@@ -46,16 +36,6 @@ public NodeCreate
(BlockingQueue<Packet> outQueue, DataStore dataStore)
        setOutQueue(outQueue);
    }

-   public void setOutQueue(BlockingQueue<Packet>
-   outQueue)
-   {
-      this.outQueue = outQueue;
-   }
-
-   public void setDataStore(DataStore dataStore)
-   {
-      this.dataStore = dataStore;
-   }
```

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.

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 1,828 projects that are highly varied in size, contributors, number of commits and refactorings.

Construct Validity. A potential threat to construct validity relates to the set of metrics, as it may miss some properties of the selected internal quality attributes. To mitigate this threat, we select well-known metrics that cover various properties of each attribute, as reported in the literature [26].

While our experiments rely on mining the intention of developers through their explicit documentation in code, which is in line with what has been done by various recent studies [55,15,11,54,31,35], this may not cover the whole spectrum of all the code changes done with reuse in mind. Thus, we might be missing some code changes that were performed with that aspect but without any explicit documentation about it (i.e., false negatives). Therefore, it would be interesting to further investigate the impact of reusability on code changes by interviewing developers about it.

Listing 27: Before refactoring - In SerializingTranscoderTest.java)

```
-   public void testLong() throws Exception {
-
    assertEquals(9231, tc.decode(tc.encode(9231)));
-   }
-   public void testInt() throws Exception {
-       assertEquals(923, tc.decode(tc.encode(923)));
-   }
-   public void testChar() throws Exception {
-       assertEquals('c', tc.decode(tc.encode('c')));
-   }
-   public void testBoolean() throws Exception {
-       assertSame(Boolean.TRUE,
-       tc.decode(tc.encode(true)));
-       assertSame(Boolean.FALSE,
-       tc.decode(tc.encode(false)));
-   }
```

Listing 28: Before refactoring - In WhalinTranscoderTest.java

```
-   public void testLong() throws Exception {
-
    assertEquals(9231, tc.decode(tc.encode(9231)));
-   }
-   public void testInt() throws Exception {
-       assertEquals(923, tc.decode(tc.encode(923)));
-   }
-   public void testShort() throws Exception {
-       assertEquals((short)923, tc.decode(
-       tc.encode((short)923)));
-   }
-   public void testChar() throws Exception {
-       assertEquals('c', tc.decode(tc.encode('c')));
-   }
-   public void testBoolean() throws Exception {
-       assertSame(Boolean.TRUE,
-       tc.decode(tc.encode(true)));
-       assertSame(Boolean.FALSE,
-       tc.decode(tc.encode(false)));
-   }
```

Listing 29: After refactoring - In BaseTranscoderCase.java

```
+   public void testLong() throws Exception {
+
    assertEquals(9231, tc.decode(tc.encode(9231)));
+   }
+   public void testInt() throws Exception {
+       assertEquals(923, tc.decode(tc.encode(923)));
+   }
+   public void testShort() throws Exception {
+       assertEquals((short)923,
+       tc.decode(tc.encode((short)923)));
+   }
+   public void testChar() throws Exception {
+       assertEquals('c', tc.decode(tc.encode('c')));
+   }
+   public void testBoolean() throws Exception {
+       assertSame(Boolean.TRUE,
+       tc.decode(tc.encode(true)));
+       assertSame(Boolean.FALSE,
+       tc.decode(tc.encode(false)));
+   }
```

Listing 30: mojibake.rb

```
require 'iudex-core'
require 'java'

-module Iudex::Core::Filters
-  import 'iudex.core.filters.MojiBakeFilter'
-
-  # Re-open iudex.core.filters.MojiBakeFilter to add
-  config file
-  # based initialization.
-  class MojiBakeFilter
+ module Iudex::Core

+  module MojiBake
    DEFAULT_CONFIG = File.join( File.dirname
    ( __FILE__ ),
                                  '..', '..', 'config',
                                  'mojibake' )
```

Listing 31: MojiBakeFilter Class

```
-    private CharSequence recover( CharSequence in )
-    {
-        Matcher m = _mojiPattern.matcher( in );
-        StringBuilder out = new StringBuilder(
-        in.length() );
-        int last = 0;
-       while( m.find() ) {
-            out.append( in, last, m.start() );
-            String moji = in.subSequence( m.start(),
-            m.end() ).toString();
-            out.append( _mojis.get( moji ) );
-            last = m.end();
-        }
-        out.append( in, last, in.length() );

-        if( out.length() < in.length() ) {
-           return recover( out );
-        }
-        else {
-            return out;
-        }
-    }

-    private final Key<CharSequence> _field;
-
-    private final Pattern _mojiPattern;
-    private final HashMap<String, String> _mojis;
+    private final MojiBakeMapper _mapper;
}
```

Listing 32: MojiBakeMapper Class

```
+ public class MojiBakeMapper
+ {
+    public MojiBakeMapper( String regex,
+
    Map<String,String> mojis )
+    {
+        _mojiPattern = Pattern.compile( regex );
+
    _mojis = new HashMap<String,String>( mojis );
+    }
+    public CharSequence recover( CharSequence in )
+    {
+        Matcher m = _mojiPattern.matcher( in );
+        StringBuilder out = new StringBuilder(
+        in.length() );
+        int last = 0;
+        while( m.find() ) {
+            out.append( in, last, m.start() );
+            String moji = in.subSequence( m.start(),
+            m.end() ).toString();
+            out.append( _mojis.get( moji ) );
+            last = m.end();
+        }
+        out.append( in, last, in.length() );

+        if( out.length() < in.length() ) {
+            return recover( out );
+        }
+        else {
+            return out;
+        }
+   }
+    private final Pattern _mojiPattern;
+    private final HashMap<String, String> _mojis;
+ }
```

7 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.

Further, we designed our study with the goal of better understanding developer perception of code reusability within the open source community. Further research in this regard is needed (e.g., running contextual interviews with developers to uncover the underpinning reasons for code reuse during refactorings). As with every study, the results may not generalize to other contexts. Extending this work with the industry partners is part of our future investigation to challenge our current findings.

References

1. https://github.com/addthis/stream-lib/commit/06bdb3f569a7fac50d5e1801359324e16929c270
2. https://github.com/FluentLenium/FluentLenium/commit/5296f9f4bcb7067d8c3220347d806772b10659da
3. https://github.com/d4rken/reddit-android-appstore/commit/7f5d41f6d16fb445b139cad034ca4d312c7ab320
4. https://github.com/JetBrains/teamcity-nuget-support/commit/fd232f43bc08ff0d91b257d2aca5ebc3a6aef1e4
5. https://github.com/jline/jline2/commit/54673e36c516e2bdfbacc11035f5942fcaa043a0
6. https://github.com/buddycloud/buddycloud-server-java/commit/a9e5d24e5d18860eafd755866e9d39b8654e2fa1
7. https://github.com/dustin/java-memcached-client/commit/df6cd926366f46878bf37d27cefce03fa922ad14
8. https://github.com/dekellum/iudex/commit/ee85ac1684639745e0a767b97a236eead8c4db31
9. Abdalkareem, R., Shihab, E., Rilling, J.: On code reuse from stackoverflow: An exploratory study on android apps. Information and Software Technology **88**, 148–158 (2017)
10. Ahmaro, I., Abualkishik, A., Yusof, M.: Taxonomy, definition, approaches, benefits, reusability levels, factors and adaption of software reusability: A review of the research literature. Journal of Applied Sciences **14** (2014)
11. AlOmar, E.A., Barinas, D., Liu, J., Mkaouer, M.W., Ouni, A., Newman, C.: An exploratory study on how software reuse is discussed in stack overflow. In: International Conference on Software and Software Reuse, pp. 292–303. Springer (2020)
12. AlOmar, E.A., Mkaouer, M.W., Newman, C., Ouni, A.: On preserving the behavior in software refactoring: A systematic mapping study. Information and Software Technology p. 106675 (2021)
13. AlOmar, E.A., 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)
14. AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Toward the automatic classification of self-affirmed refactoring. Journal of Systems and Software p. 110821 (2020)
15. 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)
16. AlOmar, E.A., Peruma, A., Mkaouer, M.W., Newman, C., Ouni, A., Kessentini, M.: How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications p. 114176 (2020)
17. AlOmar, E.A., Rodriguez, P.T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A., Mkaouer, M.W.: How do developers refactor code to improve code reusability? In: International Conference on Software and Software Reuse, pp. 261–276. Springer (2020)
18. Alshayeb, M.: Empirical investigation of refactoring effect on software quality. Information and software technology **51**(9), 1319–1326 (2009)
19. An, L., Mlouki, O., Khomh, F., Antoniol, G.: Stack overflow: A code laundering platform? In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 283–293. IEEE (2017)
20. Anguswamy, R., Frakes, W.B.: A study of reusability, complexity, and reuse design principles. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, p. 161–164. Association for Computing Machinery, New York, NY, USA (2012). DOI 10.1145/2372251.2372280. URL https://doi.org/10.1145/2372251.2372280
21. Baqais, A.A.B., Alshayeb, M.: Automatic software refactoring: a systematic literature review. Software Quality Journal **28**(2), 459–502 (2020)
22. Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship

between quality and refactoring. Journal of Systems and Software **107**, 1–14 (2015)
23. Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: 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 (2013)
24. Cedrim, D., Sousa, L., Garcia, A., Gheyi, R.: 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 (2016)
25. Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., Garcia, A.: How does refactoring affect internal quality attributes?: A multi-project study. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, pp. 74–83. ACM (2017)
26. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
27. Dig, D., Johnson, R.: The role of refactorings in api evolution. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 389–398. IEEE (2005)
28. Du Bois, B., Mens, T.: Describing the impact of refactoring on internal program quality. In: International Workshop on Evolution of Large-scale Industrial Software Applications, pp. 37–48 (2003)
29. 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)
30. Feitosa, D., Ampatzoglou, A., Gkortzis, A., Bibi, S., Chatzigeorgiou, A.: Code reuse in practice: Benefiting or harming technical debt. Journal of Systems and Software **167**, 110618 (2020)
31. Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., Oizumi, W.: Refactoring effect on internal quality attributes: What haven't they told you yet? Information and Software Technology **126**, 106347 (2020)
32. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
33. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Patterns, D.: Elements of reusable object-oriented software. Design Patterns. massachusetts: Addison-Wesley Publishing Company (1995)
34. Ghofrani, J., Kozegar, E., Bozorgmehr, A., Soorati, M.D.: Reusability in artificial neural networks: An empirical study. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B, SPLC '19, p. 122–129. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3307630.3342419. URL https://doi.org/10.1145/3307630.3342419
35. Hamdi, O., Ouni, A., AlOmar, E.A., Cinnéide, M.Ó., Mkaouer, M.W.: An empirical study on the impact of refactoring on quality metrics in android applications. In: 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), pp. 28–39. IEEE (2021)
36. Hegedűs, G., Hrabovszki, G., Hegedűs, D., Siket, I.: Effect of object oriented refactorings on testability, error proneness and other maintainability attributes. In: Proceedings of the 1st Workshop on Testing Object-Oriented Systems, p. 8. ACM (2010)
37. Hotta, K., Sano, Y., Higo, Y., Kusumoto, S.: Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pp. 73–82 (2010)
38. Jones, B., Litvintchouk, S., Mungle, J., Krasner, H., Mellby, J., Willman, H.: Issues in software reusability. Ada Lett. **IV**(5), 97–99 (1985). DOI 10.1145/1041339.1041345. URL https://doi.org/10.1145/1041339.1041345
39. Kerievsky, J.: Refactoring to patterns. Pearson Deutschland GmbH (2005)
40. Leitch, R., Stroulia, E.: Assessing the maintainability benefits of design restructuring using dependency analysis. In: Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717), pp. 309–322. IEEE (2003)
41. Lorenz, M., Kidd, J.: Object-oriented software metrics, vol. 131. Prentice Hall Englewood Cliffs (1994)
42. Lotter, A., Licorish, S.A., Savarimuthu, B.T.R., Meldrum, S.: Code reuse in stack overflow and popular open source java projects. In: 2018 25th Australasian Software Engineering Conference (ASWEC), pp. 141–150. IEEE (2018)
43. Lubars, M.D.: Code reusability in the large versus code reusability in the small. SIGSOFT Softw. Eng. Notes **11**(1), 21–28 (1986). DOI 10.1145/382300.382307. URL https://doi.org/10.1145/382300.382307
44. Makady, S., Walker, R.J.: Test code reuse from oss: Current and future challenges. In: Proceedings of the 3rd Africa and Middle East Conference on Software Engineering, AMECSE '17, p. 31–36. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3178298.3178305. URL https://doi.org/10.1145/3178298.3178305
45. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
46. Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó., Deb, K.: On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empirical Software Engineering **21**(6), 2503–2545 (2016)
47. Mockus, A.: Large-scale code reuse in open source software. In: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS '07, p. 7. IEEE Computer Society, USA (2007). DOI 10.1109/FLOSS.2007.10. URL https://doi.org/10.1109/FLOSS.2007.10
48. Mondal, M., Rahman, M.S., Saha, R.K., Roy, C.K., Krinke, J., Schneider, K.A.: An empirical study of the impacts of clones in software maintenance. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 242–245. IEEE (2011)
49. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G.: A case study on the impact of refactoring on quality and productivity in an agile team. In: IFIP Central and East European Conference on Software Engineering Techniques, pp. 252–266. Springer (2007)
50. Moser, R., Sillitti, A., Abrahamsson, P., Succi, G.: Does refactoring improve reusability? In: International Conference on Software Reuse, pp. 287–297. Springer (2006)
51. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017)
52. 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)
53. Opdyke, W.F.: Refactoring object-oriented frameworks (1992)
54. Paixão, M., Uchôa, A., Bibiano, A.C., Oliveira, D., Garcia, A., Krinke, J., Arvonio, E.: Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 125–136 (2020)
55. 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)

56. Patrick, M.T.: Exploring software reusability metrics with q&a forum data. Journal of Systems and Software **168**, 110652 (2020)
57. Patwa, S., Malviya, A.K.: Reusability metrics and effect of reusability on testing of object oriented systems. SIGSOFT Softw. Eng. Notes **37**(5), 1–4 (2012). DOI 10.1145/2347696.2347708. URL https://doi.org/10.1145/2347696.2347708
58. 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)
59. 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)
60. Quan, L., Zongyan, Q., Liu, Z.: Formal use of design patterns and refactoring. In: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 323–338. Springer (2008)
61. Robson, C.: Real world research: A resource for social scientists and practitioner-researchers. Wiley-Blackwell (2002)
62. Roy, C.K., Zibran, M.F., Koschke, R.: The vision of software clone management: Past, present, and future (keynote paper). In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 18–33. IEEE (2014)
63. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering **14**(2), 131–164 (2009)
64. Sahraoui, H.A., Godin, R., Miceli, T.: Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In: icsm, p. 154. IEEE (2000)
65. Sharma, A., Grover, P.S., Kumar, R.: Reusability assessment for software components. SIGSOFT Softw. Eng. Notes **34**(2), 1–6 (2009)
66. 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)
67. Shatnawi, R., Li, W.: An empirical assessment of refactoring impact on software quality using a hierarchical quality model. International Journal of Software Engineering and Its Applications **5**(4), 127–149 (2011)
68. 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)
69. Stroulia, E., Kapoor, R.: Metrics of refactoring-based development: An experience report. In: OOIS 2001, pp. 113–122. Springer (2001)
70. Szóke, G., Antal, G., Nagy, C., Ferenc, R., Gyimóthy, T.: Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 95–104. IEEE (2014)
71. Tahvildari, L., Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations. In: Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., pp. 183–192. IEEE (2003)
72. Tahvildari, L., Kontogiannis, K., Mylopoulos, J.: Quality-driven software re-engineering. Journal of Systems and Software **66**(3), 225–239 (2003)
73. 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)
74. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics bulletin **1**(6), 80–83 (1945)
75. Wilking, D., Kahn, U.F., Kowalewski, S.: An empirical evaluation of refactoring. e-Informatica **1**(1), 27–42 (2007)
76. Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A.: Experimentation in software engineering: an introduction. (2000)
77. Yin, W., Tanik, M.M., Yun, D.Y.Y., Lee, T.J., Dale, A.G.: Software reusability: A survey and a reusability experiment. In: Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow, ACM '87, p. 65–72. IEEE Computer Society Press, Washington, DC, USA (1987)
78. Younoussi, S., Roudies, O.: All about software reusability: A systematic literature review. Journal of Theoretical and Applied Information Technology **76**, 64–75 (2015)