On the Impact of Refactoring on the Relationship between Quality Attributes and Design Metrics

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

***Abstract*—Background: Refactoring is a critical task in software maintenance and is generally performed to enforce the best design and implementation practices or to cope with design defects. Several studies attempted to detect refactoring activities through mining software repositories allowing to collect, analyze and get actionable data-driven insights about refactoring practices within software projects.**

Aim: We aim at identifying, among the various quality models presented in the literature, the ones that are more in-line with the developer's vision of quality optimization, when they explicitly mention that they are refactoring to improve them.

Method: We extract a large corpus of design-related refactoring activities that are applied and documented by developers during their daily changes from 3,795 curated open source Java projects. In particular, we extract a large-scale corpus of structural metrics and anti-pattern improvement changes, from which we identify 1,245 quality improvement commits with their corresponding refactoring operations, as perceived by software engineers. Thereafter, we empirically analyze the impact of these refactoring operations on a set of common state-of-the-art design quality metrics.

Results: The statistical analysis of the obtained results shows that (i) a few state-of-the-art metrics are more popular than others; and (ii) some metrics are being more emphasized than others.

Conclusions: We verify that there are a variety of structural metrics that can represent the internal quality attributes with different degrees of improvement and degradation of software quality. Most of the metrics that are mapped to the main quality attributes do capture developer intentions of quality improvement reported in the commit messages, but for some quality attributes, they don't.

***Index Terms*—refactoring, software quality, empirical study**

I. Introduction

Being the *de facto* practice of improving software design without altering its external behavior, refactoring has been the focus on several studies, which aim to support its application by identifying refactoring opportunities, in the source code, through the optimization of structural metrics, and the removal of code smells [8], [16], [21], [36], [46], [47], [49]. Therefore, several studies have been analyzing the impact of refactoring on existing literature quality attributes, structural metrics, and code smells [2], [3], [5], [6], [17], [25], [35], [50]. The spectrum of quality attributes, structural metrics and code smells, represent the main driver for studies aiming to imitate the human decision making, and automate the refactoring process.

Despite the growing effort in recommending refactorings by structural metrics optimization and code smells removal, there is very little evidence on whether developers follow that intention when refactoring their code. A recent study by Pantiuchina et al. [31] has shown that there is misperception between the state-of-the-art structural metrics, widely used as indicators for refactoring, and what developers actually consider to be an improvement their source code. Thus, there is a need to distinguish, among all the structural metrics, typically used in refactoring literature, the particular ones that are of a better representation of the developers' perception of software quality improvement.

This paper aims in identifying, among the various quality models presented in literature, the ones that are more in-line with the developer's vision of quality optimization, when they explicitly mention that they are refactoring to improve them.

We start with reviewing literature studies, which propose software quality attributes and their corresponding measurement in the source code, in terms of metrics. Software quality attributes are typically characterized by a high-level definition whose interpretation open the possibility of multiple ways to calculate them in the source code level. Thus, there is little consensus on what would be the optimal match between high-quality attributes, and code-level structural and design metrics. For instance, as shown later in Section II, the notion of complexity was the subject of many studies that proposed several metrics to calculate it. Therefore, we investigate which code-level metrics are more representative to the high-level quality attributes, when their optimization is explicitly stated by the developer, when applying refactorings.

Practically, we have classified 1,245 commits, as quality improvement commits, by manually analyzing their messages and identifying an explicit statement of improving an internal quality attribute, along with detecting their refactoring activities. We mined these commits from 3,795 well-engineered, open-source projects. We identify their refactoring operations by applying state-of-the-art refactoring mining tools [38], [48]. We refine our dataset by untangling each commit to only select refactored code elements. Then, we cluster commits according to the quality attribute they state they are optimizing (complexity, inheritance, etc.). Afterward, for each quality attribute, we calculate the values of its corresponding structural metrics, in the files, before and after their refactorings. And

finally, we empirically compare the variation of these values, to distinguish the metrics that are significantly impacted by the refactorings, and so they better reflect the developer's intention of optimizing its corresponding quality attribute. To the best of our knowledge, no previous study has investigated the relationship between quality attributes and their corresponding structural metrics, from the developer's perception. Our key findings show that not all state of the art structural metrics equally represent internal quality attributes; some quality attributes are being more emphasized than others by developers. This paper extends the existing knowledge of empirically investigating exploring the relationship between refactoring and quality as follows:

1) We extensively review the literature of quality attributes, used in the literature of software quality, and their corresponding possible measurements, in terms of metrics. Then we mine a large scale dataset from GitHub[1] that consist of 1,245 commits from 3,795 software projects, proven to contain refactoring operations, and illustrating developers self-stated intentions to optimize our studied quality attributes.
2) For each quality attribute, we empirically investigate which metrics are most impacted by refactorings, and so, the closest to capture the developer's intention.
3) For reproducibility and extension, we provide a dataset of commits, their refactoring operations, and their impact on several quality metrics[2].

The remainder of this paper is organized as follows: Section II reviews the existing studies related to measuring software quality and analyzing the relationship between quality attributes and refactoring. Section III outlines our empirical setup in terms of data collection, analysis and research questions. Section IV discusses our findings, while Section V captures any threats to the validity of our work, before concluding with Section VI.

II. 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 implementations bad practices [14]. In recent year, there is much research efforts have focused on studying and exploring the impact of refactoring on software quality [2], [3], [5], [6], [17], [25], [35], [50]. 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 the impact of refactoring on software quality.

In an academic setting, Stroulia and Kapoor [41] investigate the effect of size and coupling measures on software quality after the application of refactoring. The results in Stroulia and Kapoor's work showed that size and coupling metrics decreased after refactorings. Kataoka et al. [21] used only coupling measures to study the impact of *Extract Method* and *Extract Class* refactoring operations on the maintainability of a single C++ software system, and found that refactoring has positive impact on system maintainability. Demeyer [9] performed a comparative study to investigate the impact of refactoring on performance. The results of Demeyer's study show that program performance is enhanced after the application of refactoring. Moreover, Sahraoui et al. [34] used coupling and inheritance measures to automatically detect potential antipatterns and predict situations where refactoring could be applied to improve software maintainability. The authors found that quality metrics can help to bridge the gap between design improvement and its automation, but in some situations the process cannot be fully automated as it requires the programmer's validation through manual inspection.

Tahvildari et al. [44] proposed a software transformation framework that links software quality requirements like performance and maintainability with program transformation to improve the target qualities. The results show that utilizing design patterns increase system's maintainability and performance. In another study, Tahvildari and Kontogiannis [43] used the same framework to evaluate four object-oriented measures (*i.e.,* cohesion, coupling, complexity, and inheritance) in addition to software maintainability. Leitch and Stroulia [22] used dependency graph-based techniques to study the impact of two refactorings, namely, *Extract Method* and *Move Method*, on software maintenance using two small systems. The authors found that refactoring enhanced the quality by (1) reducing the design size, (2) increasing number of procedures, (3) reducing the data dependencies, and (4) reducing regression testing. Bios and Mens [13] proposed a framework to analyze the impact of three refactorings on five internal quality attributes (*i.e.,* cohesion, coupling, complexity, inheritance, and size), and their findings show positive and negative impacts on the selected measures. Bios et al. [11] provided a set of guidelines for optimizing cohesion and coupling measures. This study shows that the impact of refactoring on these measures ranged from negative to positive. In a follow-up work, Bios et al. [12] conducted a study to differentiate between the application of Refactor to Understand and the traditional Read to Understand pattern. Their findings show that refactoring plays a role in improving the understandability of the software.

Geppert et al. [15] investigated the impact of refactoring on changeability focusing on three factors for changeability, namely, customer-reported defect rates, change effort, and scope of changes. Their findings show a significant decrease in the first two factors. Ratzinger et al. [33] analyzed the historical data of a large industrial system and focused on reducing change couplings. Based on the identified change couplings, they also analyzed code smell changes for the purpose of identifying where to apply refactoring efficiently. They concluded that refactoring is able to enhance software evolvability (*i.e.,* reduce the change coupling). In an agile development environment, Moser et al. [26] used internal measures (*i.e.,* CK, MCC, LOC) to explore the effect of refactoring on the reusability of the code using a commer-

[1] https://github.com
[2] (Hidden for Double-Blind Review)

Table (I) A summary of the literature on the impact of refactoring activities on software quality attributes.

Study	Year	Approach	Software Metric	Internal QA	External QA
Sahraoui et al. [34]	2000	Analyzing code histories	CLD / NOC / NMO / NMI NMA / SIX / CBO / DAC IH-ICP / OCAIC / DMMEC / OMMEC	Inheritance / Coupling	Fault-proneness / Maintainability
Stroulia & Kapoor [41]	2001	Performing a case study	LOC / LCOM / CC	Size / Coupling	Design extensibility
Kataoka et al. [21]	2002	Analyzing code histories	Coupling measures	Coupling	Maintainability
Demeyer [9]	2002	Analyzing code histories	N/A	Polymorphism	Performance
Tahvildari et al. [44]	2003	Analyzing code histories	LOC / CC / CMT / Halstead's efforts	Complexity	Performance / Maintainability
Leitch & Stroulia [22]	2003	Analyzing code histories	SLOC / No. of Procedure	Size	Maintainability
Bois & Mens [13]	2003	Analyzing code histories	NOM / CC / NOC / CBO RFC / LCOM	Inheritance / Cohesion / Coupling / Size / Complexity	N/A
Tahvildari & Kontogiannis [43]	2004	Analyzing code histories	LCOM / WMC / RFC / NOM CDE / DAC / TCC	Inheritance / Cohesion / Coupling / Complexity	Maintainability
Bois et al. [11]	2004	Analyzing code histories	N/A	Cohesion / Coupling	Maintainability
Bois et al. [12]	2005	Analyzing code histories	N/A	N/A	Understandability
Geppert et al. [15]	2005	Performing a case study	N/A	N/A	Changeability
Ratzinger et al. [33]	2005	Mining commit log Analyzing code histories	N/A	Coupling	Evolvability
Moser et al. [26]	2006	Analyzing code histories	CK / MCC / LOC	Inheritance / Cohesion / Coupling / Complexity	Reusability
Wilking et al. [50]	2007	Analyzing code histories	CC / LOC	Complexity	Maintainability / Modifiability
Stroggylos & Spinells [40]	2007	Mining commit log	CK / Ca / NPM	Inheritance / Cohesion / Coupling / Complexity	N/A
Moser et al. [25]	2008	Analyzing code histories	CK / LOC / Effort (hour)	Cohesion / Coupling / Complexity	Productivity
Alshayeb [2]	2009	Analyzing code histories	CK / LOC / FANOUT	Inheritance / Cohesion / Coupling / Size	Adaptability / Maintainability / Testability / Reusability Understandability
Hegedus et al. [17]	2010	Analyzing code histories	CK	Coupling / Complexity / Size	Maintainability / Testability / Error Proneness / Changeability Stability / Analizability
Shatnawi & Li [35]	2011	Analyzing code histories	CK / QMOOD	Inheritance / Cohesion / Coupling / Polymorphism / Size Encapsulation / Composition / Abstraction / Messaging	Reusability / Flexibility / Extendibility / Effectiveness
Bavota et al. [4]	2013	Analyzing code histories Surveying developers	ICP / IC-CD / CCBC	Coupling	N/A
Szoke et al. [42]	2014	Mining commit log Surveying developers	CC / U / NOA / NII / NAni LOC / NUMPAR / NMni / NA	Size / Complexity	N/A
Bavota et al. [3]	2015	Mining commit log Analyzing code histories	CK / LOC / NOA / NOO C3 / CCBC	Inheritance / Cohesion / Coupling / Size / Complexity	N/A
Cedrim at al. [5]	2016	Mining commit log Analyzing code histories	LOC / CBO / NOM / CC FANOUT / FANIN	Cohesion / Coupling / Complexity	N/A
Chavez et al. [6]	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	Inheritance / Cohesion / Coupling / Size / Complexity	N/A
Pantiuchina et al. [31]	2018	Mining commit log Analyzing code histories	LCOM / CBO / WMC / RFC C3 / B&W / SRead	Cohesion / Coupling / Complexity	Readability

cial system, and found that refactoring was able to improve the reusability of hard-to-reuse classes. Wilking et al. [50] empirically studied the effect of refactoring on non-functional aspects, *i.e.,* the maintainability and modifiability of system systems. They tested the maintainability by explicitly adding defects to the code, and then measured the time taken to remove them. Modifiability, on the other hand, was examined by adding new functionalities and then measuring the LOC metric and the time taken to implement these features. The authors did not find a clear effect of refactoring on these two external attributes.

Stroggylos and Spinellis [40] opted for searching words stemming from the verb "refactor" such as "refactoring" or "refactored" to identify refactoring-related commits to study the impact of refactoring on quality using eight object-oriented metrics. Their results indicated possible negative effects of refactoring on quality, *e.g.,* increased LCOM metric. Moser et al. [25] studied the impact of refactoring on the productivity in an agile team. The achieved results show that refactoring improved software developers' productivity besides several aspects of quality, *e.g.*, maintainability. Alshayeb [2] conducted a study aiming at assessing the impact of eight refactorings on five external quality attributes (*i.e.,* adaptability, maintainability, understandability, reusability, and testability). The author found that refactoring could improve the quality in some classes, but could also decrease software quality to some extent in other classes. Hegedus et al. [17] examined the effect of singular refactoring techniques on testability, error proneness, and other maintainability attributes. They concluded that refactoring could have undesired side effects that can degrade the quality of the source code.

In an empirical setting, Shatnawi and Li [35] used the hierarchical quality model to assess the impact of refactoring on four software quality factors, namely, reusability, flexibility, extendibility, and effectiveness. The authors found that the majority of refactoring operations exhibit positive impact on quality; however, some operations deteriorated quality. Bavota et al. empirically investigated the developers' perception of coupling, as captured by structural, dynamic, semantic, and logical coupling measures. They found that semantic coupling measure aligns with developers' perceptions better that the other coupling measures. In a more recent study, Bavota et al. [3] used RefFinder [32], a version-based refactoring detection tool, to mine the evolution history of three open-source systems. They mainly investigated the relationship between refactoring and quality. The study findings indicate that 42% of the performed refactorings are affected by code smells, and refactorings were able to eliminate code smells in only 7% of the cases.

Cedrim et al. [5] conducted a longitudinal study of 25 projects to investigate the improvement of software structural quality. They analyzed the relationship of refactorings and code smells by classifying refactorings according to the addition or removal of poor code structures. The study results indicate that only 2.24% of refactorings removed code smells, and 2.66% introduced new ones. Recently, Chavez et al. [6] studied the effect of refactoring on five internal quality attributes, namely, cohesion, coupling, complexity, inheritance, and size, using 25 quality metrics. The study shows that root-canal refactoring-related operations are either improved or at

least not worsened the internal quality attributes. Additionally, when floss refactoring-related operations are applied, 55% of these operations improved these attributes, while only 10% of quality declined.

In particular, two studies [31], [42] are most related to our work have analyzed the comment commits in which developers stated the purpose of improving the quality. Szoke et al. [42] studied 198 refactoring commits of five large-scale industrial systems to investigate the effects of these commits on quality of several revisions for a period of time. To know the purpose of the applied refactorings, they trained developers and asked them to state the reason when committing the changes to the repositories, which could be related to (1) fix coding issues, (2) fix antipatterns, and (3) improve certain metrics. The study results show that performing a single refactoring could negatively impact the quality, but applying refactorings in blocks (*e.g.,* fixing more coding issues or improving more quality metrics) can significantly improve software quality. More recently, Pantiuchina et al. [31] empirically investigated the correlation between seven code metrics and the quality improvement explicitly reported by developers in 1,282 commit messages. The study shows that quality metrics sometimes do not capture the quality improvement reported by developers. A common indicator to assess the quality improvements between these studies resides in the use the quality metrics. Both of these studies found that minor refactoring changes rarely impact the quality of the software.

All of the above-mentioned studies have focused on assessing the impact of refactorings on the quality by either considering the internal or the external quality attributes using a variety of approaches. Among them, few studies [3], [5], [6], [31], [33], [40], [42] took the approach of mining software repositories to explore the impact on quality. In particular, the vast majority of these studies, used a limited set of projects and mined general commits without applying any form of verification regarding whether refactorings have actually been applied.

Our work is different from these studies as our main purpose is to explore if there is an alignment between quality metrics and quality improvements that are documented by developers in the commit messages. As we summarize these state-of-the-art studies in Table I. We identify 8 popular quality attributes, namely *Cohesion*, *Coupling*, *Complexity*, *Inheritance*, *Polymorphism*, *Encapsulation*, *Abstraction* and *Design size*. As different studies advocate for various metrics to calculate these quality attributes, we extract and calculate 27 structural metrics. In particular, on a more qualitative sense, we conduct an empirical study using 1,245 commits that are proven to contain real-world instances of refactoring activities, in the purpose of improving software design. To the best of our knowledge, no previous study has empirically investigated, using a curated set of commits, the representativeness of structural design metrics for internal quality attributes. In the next section, we detail the steps we took to design our empirical setup.

III. Empirical Study Setup

This section presents our empirical study. Our main goal is to investigate whether the developer perception of quality improvement (as expected by developers) aligns with the real quality improvement (as assessed by quality metrics). In particular, we address the following research question:

- *Do the developer perception of quality improvement align with the quantitative assessment of code quality?*

To answer our research question, we conduct a three-phased empirical study. An overview of the experiment methodology is depicted in Figure 1. The initial phase consists of selecting and mining a large number of open-source Java projects and detecting refactoring instances that occur throughout their development history, *i.e.,* commit-level code changes, of each considered project. The second phase consists of analyzing the commit messages as a mean of identifying refactoring commits in which developers document their perception of internal quality attributes. Thereafter, the third phase involves the selection of software quality metrics to compare its values before and after the selected refactoring commits.

A. Selection of Quality Attributes and Structural Metrics

To setup a comprehensive set of quality attributes, to be assessed in our study, we first conduct a literature review on existing and commonly acknowledged software quality attributes [7], [10], [19], [23], [24], [30]. Then, we checked if the metrics assess several object-oriented design aspects in order to map each internal quality attribute to the appropriate structural metric(s). For example, the Response For Class (RFC) metric is typically used to measure Coupling and Complexity quality attributes. More generally, we extract, from literature review, all the associations between metrics (*e.g.,* CK suite [7], McCabe [24] and Lorenz and Kidd's book [23]) with internal quality attributes.

The extraction process results in 19 distinct structural metrics as shown in Table II. 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 existing static analysis tools. For this study, all metrics values are automatically computed using the UNDERSTAND[3], a popular static analysis framework.

B. Refactoring Detection

To collect the necessary commits, we refer to an existing large dataset of links to GitHub repositories. [1]. We perform an initial filtering, using Reaper [27], to only navigate through well-engineered projects. So, we ended up reducing the number of selected projects from 57,447 to 3,795. To extract the entire refactoring history in each project, we use two popular refactoring mining tools, namely Refactoring Miner [37] and ReffDiff [38]. We selected both tools because they are known to be in the top of refactoring detection tools, in terms of accuracy [45], [48] (precision of 98% and 100%, and recall of 87% and 88%, respectively), and because they are both built

[3]https://scitools.com/

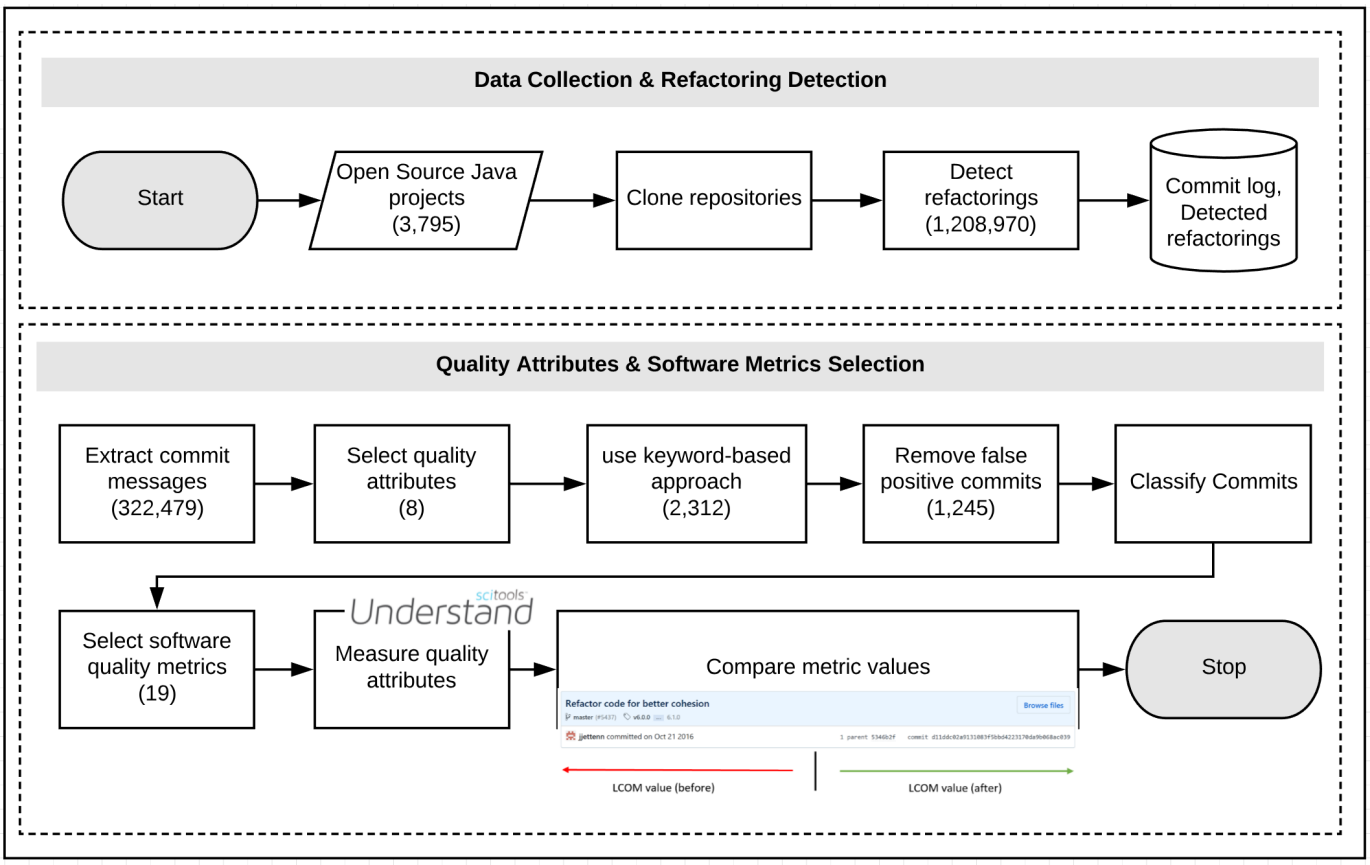


Figure (1) Empirical study design overview.

Table (II) Internal quality attributes and their corresponding structural metrics used in this study.

Quality Attribute	Study	Software Metrics
Cohesion	[6], [31]	Lack of Cohesion of Methods (LCOM) [7]
Coupling	[6], [31]	Coupling Between Objects (CBO) [7]
	[31]	Response For Class (RFC) [7]
	[6]	Fan-in (FANIN) [19]
	[6]	Fan-out (FANOUT) [19]
Complexity	[6]	Cyclomatic Complexity (CC) [24]
	[6], [31], [39]	Weighted Method Count (WMC) [7]
	[29], [39]	Response For Class (RFC) [7]
	[39]	Lack of Cohesion of Methods (LCOM) [7]
	[6]	Essential Complexity (Evg) [24]
	[6]	Paths (NPATH) [30]
	[6]	Nesting (MaxNest) [23]
Inheritance	[6], [39]	Depth of Inheritance Tree (DIT) [7]
	[6], [39]	Number of Children (NOC) [7]
	[6]	Base Classes (IFANIN) [10]
Polymorphism	[39]	Weighted Method Count (WMC) [7]
	[29], [39]	Response For a Class (RFC) [7]
Encapsulation	[39]	Weighted Method Count (WMC) [7]
	[39]	Lack of Cohesion of Methods (LCOM) [7]
Abstraction	[39]	Weighted Method Count (WMC) [7]
	[39]	Lack of Cohesion of Methods (LCOM) [7]
Design Size	[6]	Lines of Code (LOC) [23]
	[6]	Lines with Comments (CLOC) [23]
	[6]	Statements (STMTC) [23]
	[6]	Classes (CDL) [23]
	[6]	Instance Variables (NIV) [23]
	[6]	Instance Methods (NIM) [23]

in analyze code changes in git repositories to detect applied refactorings, which is the case for our intended data, along with being suitable for our study that requires a high degree of automation in data mining. As for the selection of commits with refactorings, we perform a voting process between both tools, *i.e.,* in order for a given commit to be selected, it has to be detected by both tools as a container to at least one refactoring operation. We perform this voting process to raise the likelihood of the existence of refactoring within the commit. Since the accuracy of the tools is out of the scope of this work, and since we do not perform any refactoring-related analysis, we do not care if the detection results overlap or not.

In this phase, We collect a total of 1,208,970 refactoring operations from 322,479 commits, applied during a period of 23 years (1997-2019). An overview of the studied benchmark is provided in Table III.

Table (III) Studied dataset statistics.

Item	Count
Studied projects	3,795
Commits with refactorings	322,479
Refactoring operations	1,208,970
Commits with refactorings & Keywords	2,312
Remove false positive commits	1,067
Final dataset	1,245

C. Data Extraction

After extracting all refactoring commits, we continue to filter them, using the content of their messages at this stage. We start with using a keyword-based search to find commits whose messages contain one of the keywords (*i.e., Cohesion, Coupling, Complexity, Inheritance, Polymorphism, Encapsulation, Abstraction, size*) related to our internal quality attributes.

This keyword-based filtering resulted in only selecting 2,312 commit messages. We notice that the ratio of these commits is very small in comparison with the total number of refactoring commits, *i.e.,* 322,479. However, these observations are aligned with previous studies [28], [42] as developers typically do not provide details when they document their refactorings. To ensure that these commits reported developers' intention to improve quality attributes, we manually inspect and read through these refactoring commits to remove false positives. An example of a discarded commit is: "*Refactored EphemeralFileSystemAbstraction*", we discarded this commit because the quality attribute is actually part of the identifier name of the class. In case of agreement between the authors on the inclusion of a certain commit, the commit was excluded. This step resulted in only considering 1,245 commits. During this process, we manually classified our 1,245 commits with respect to their quality attributes, as one commit could belong to more than one quality attribute. Our goal is to have a *gold set* of commits in which the developers explicitly reported the quality attributes improvement. This *gold set* will serve to check later if there is an alignment between the real quality metrics affected in the source code and the quality improvement documented by developers. Examples of commit messages belonging to the *gold set*, are showcased in Table IV.

Since commits typically contain multiples changed files, which may not all be involved in the refactoring, we make to filter them out, as we checkout, for each commit, its changed Java files, and keep only those involved in the refactoring operation(s), associated with that commit. The resulting commits, correspond to our data points, each data point is represented by a set of *pre-refactoring* and *post-refactoring* Java files. These data points will be used in the experiments, to measure the effect of changes in terms of structural metrics, with respect to the quality attribute, announced in the commit message.

IV. Empirical Study Results & Discussion

For each refactoring commit having a documented internal quality attribute by developers, we compute the corresponding metric values (see Table II) before and after the commit. For instance, for commit messages related to reducing the complexity of the source code, we calculate seven corresponding metric values before and after the selected refactoring commit, *i.e.*, Cyclomatic Complexity (CC), Weighted Method Count (WMC), Response For Class (RFC), Lack of Cohesion of Methods (LCOM), Essential Complexity (Evg), Paths (NPATH), and Nesting (MaxNest) [7], [23], [24], [30], as shown in Table II. As we calculate the metrics values of pre- and post-refactoring, we want to distinguish, for each metric, whether there is a variation on its pair of values, whether this variation indicates an improvement, and whether that variation is statistically significant. Therefore, we use the Wilcoxon test, a non-parametric test, to compare between the group of metric values before and after the commit since these groups are dependent on one another. The Null hypothesis is defined by no variation in the metric values of pre- and post-refactored code elements. Thus, the alternative hypothesis indicates that there is a variation in the metric values. In each case, a decreased metric value is considered desirable (i.e., an improvement). Additionally, the variation between values of both sets of is considered significant its associated p-value is less than 0.005. It is important to note that, in many cases, the same metric is used to evaluate several quality attributes. In the following, we report the results of our research questions.

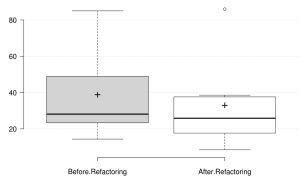
The boxplots in Figure 2 show the distribution of each metric before and after each of the examined commits.

To answer our main research question, we provide a detailed analysis of each of the eight quality attributes as reported in Table II. Table V shows the overall impact of refactorings on quality.

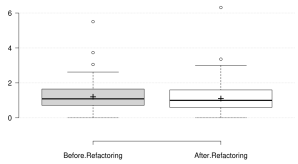
1) Cohesion: For commits whose messages report the optimization of the cohesion quality attribute, the boxplot sketched in Figure 2a shows the pre- and post-refactoring results of the normalized LCOM, used in literature to estimate the cohesion. A poor LCOM metric value implies generally that the classes should be split into 1 or more classes with better cohesion. Thus, if the value of this metric is low, it indicates a strong cohesiveness of the class. We have selected the normalized LCOM metric as it has been widely acknowledged in the literature [6], [18], [31] as being the alternative to the original LCOM, by addressing its main limitations (artificial outliers, misperception of getters and setters, etc.). As can be seen from the boxplot in Figure 2a, the median drops from 28.12 to 25.86 and the third quartile is significantly lower which shows a decrease in variation for commits after refactoring. This result indicates that LCOM is capturing the developer's intention of optimizing the cohesion quality attribute. Furthermore, as shown in Table V, LCOM has a positive impact on cohesion quality, as it decreases in the refactored code. This implies that developers did improve the cohesion of their classes, as outlined in their commit messages.

> *Summary*. The normalized LCOM metric does not only represent a good replacement to the original LCOM, but also represents the cohesion quality attribute. Its positive variation is in line with the developer's intention in improving cohesion.

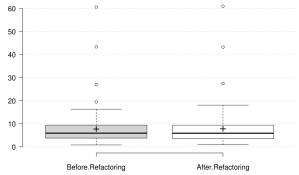
2) Coupling: For commits whose messages report the optimization of the coupling quality attribute, the boxplots sketched in Figures 2b, 2c, 2d, 2e show the pre- and post-refactoring results of four structural metrics, *i.e.*, CBO, RFC, FANIN, and FANOUT, used in literature to estimate the coupling. We observe from the figure that three out of the four coupling metrics experienced a degradation in the median



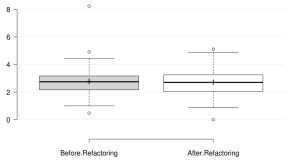
(a) Cohesion - LCOM



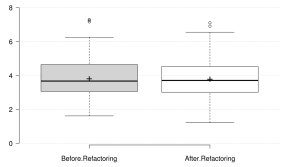
(b) Coupling - CBO



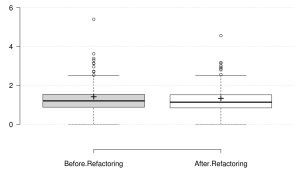
(c) Coupling - FANIN



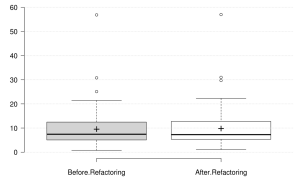
(d) Coupling - FANOUT



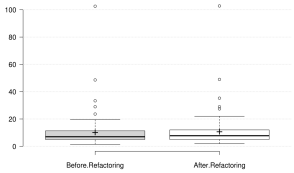
(e) Coupling - RFC



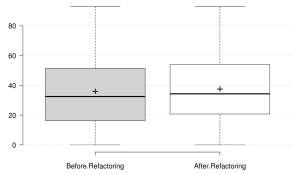
(f) Complexity - CC



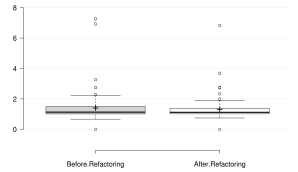
(g) Complexity - WMC



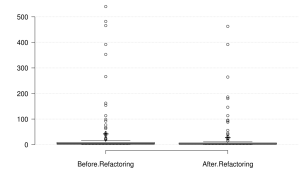
(h) Complexity - RFC



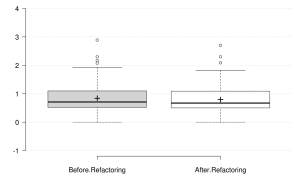
(i) Complexity - LCOM



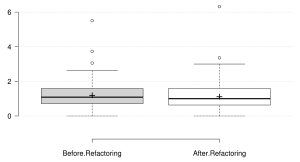
(j) Complexity - Evg



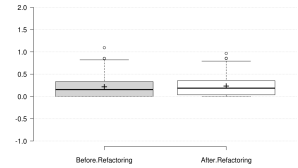
(k) Complexity - NPATH



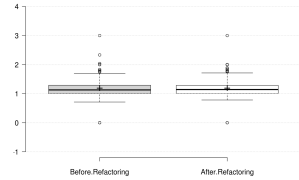
(l) Complexity - MaxNest



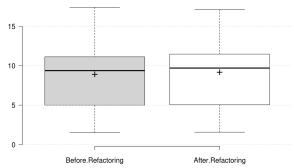
(m) Inheritance - DIT



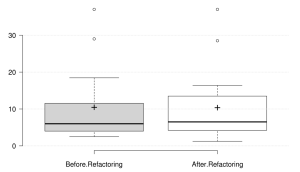
(n) Inheritance - NOC



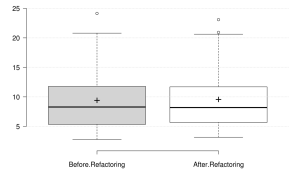
(o) Inheritance - IFANIN



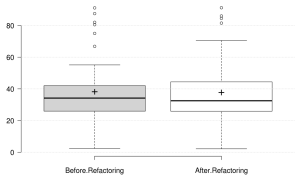
(p) Polymorphism - WMC



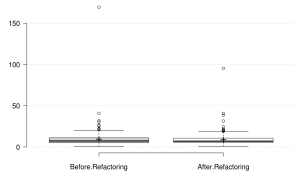
(q) Polymorphism - RFC



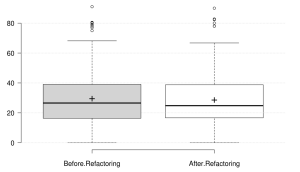
(r) Encapsulation - WMC



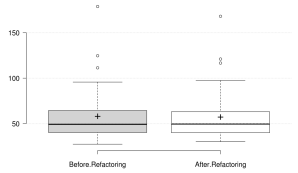
(s) Encapsulation - LCOM



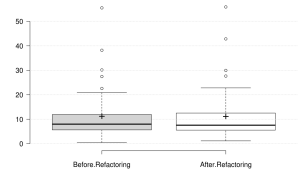
(t) Abstraction - WMC



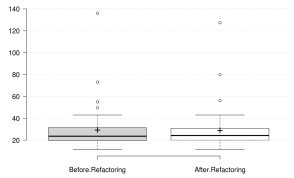
(u) Abstraction - LCOM



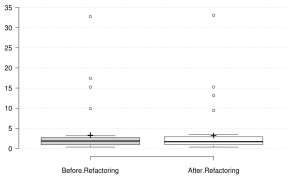
(v) Design Size - LOC



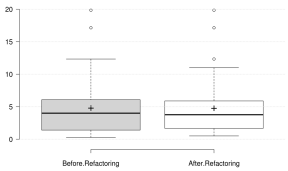
(w) Design Size - CLOC



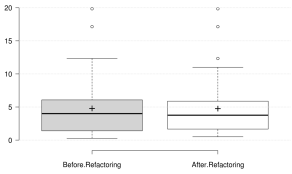
(x) Design Size - STMTC



(y) Design Size - CDL



(z) Design Size - NIV



(aa) Design Size - NIM

Figure (2) Boxplots of metrics values of pre- and post-refactored files.

Table (IV) Examples of selected commit messages.

Quality Attribute	Commit Message
Cohesion	*Refactor code for better cohesion*
Coupling	*Reduce coupling between packages*
Complexity	*reducing complexity by refactoring*
Inheritance	*refactored document requests code to better reflect inheritance ...*
Polymorphism	*Enhance field manager to account for polymorphism when getting a field from a ceiling class*
Encapsulation	*Refactored transactional observer code for better encapsulation and runtime performance*
Abstraction	*code refactored in order to improve the abstraction*
Design Size	*Major refactoring to reduce code size and have at least halfway reasonable structure ...*

values. For instance, CBO, FANIN and FANOUT medians dropped, respectively, from 1.19 to 1.00, from 5.94 to 5.91, and from 2.75 to 2.68. Coupling Between Objects (CBO) counts of the number of classes that are coupled to a particular class either through method or attribute calls. Calls are counted in both directions. CBO values have significantly decreased, which makes it a good representative of coupling. FANIN represents how useful is a code element to other code elements, while FANOUT counts the number of outsider code elements, a particular code element depends on. While both metrics are found to be degrading as developers intend to optimize coupling, only the FANOUT's variation was statistically significant. Interestingly, the Response for a Class (RFC), which counts the visibility of a class to outsider classes, has increased as developers intend to optimize coupling. In theory, increasing the visibility of a class increases the possibility to other classes to reach it, and so, it increases its coupling. However, this does not necessarily hold according to our results, but the variation is not statistically significant.

The manual inspection, of the refactored code, indicates that developers typically decrease coupling by reducing (1) the strength of dependencies that exist between classes, (2) the message flow of the classes, and (3) the number of inputs a method uses plus the number of subprograms that call this method. The code was improved as expected from the developer intentions in their commit message.

> *Summary*. CBO, FANIN and FANOUT generally decrease as developer intends to improve coupling. However, only CBO and FANOUT variation is significant. RFC exhibits an opposite variation to coupling, but it is not statistically significant. Finally, at least one metric has a significant positive variation which matches the developer's perception of improving coupling.

3) Complexity: As for the complexity quality attribute, we consider seven literature metrics, shown in Table II, to investigate the code complexity reduction as perceived by developers. As seen in the boxplots in Figures 2f, 2g, 2j, 2k, 2l, we observe that the majority metrics *i.e.,* CC, WMC, Evg, NPATH, and MaxNest, experienced a degradation in the median values. Furthermore, all the variations are statistically significant. Despite being associated with several, metrics, which are different in their definitions, our results indicate that 5 out the 7 metrics, accurately represent the complexity quality attribute. Moreover, RFC's opposed increase, when optimizing complexity, is found to be statistically significant.

In particular, we observe through a manual inspection of the collected dataset that developers tend to reduce the number of local methods, simplify the structure statements, reduce the number of paths in the body of the code, and lower the nesting level of the control statements (*e.g.*, selection and loop statements) in the method body.

> *Summary*. CC, WMC, Evg, NPATH, and MaxNest generally decrease as developer intends to improve complexity, and all their variation is significant. Furthermore, our empirical investigation discards RFC from being an indicator for complexity. Finally, at least one metric has a significant positive variation which matches the developer's perception of improving complexity.

4) Inheritance: For commits whose messages report the optimization of the inheritance quality attribute, the boxplots sketched in Figures 2m, 2n, 2o show the pre- and post-refactoring results of three structural metrics, *i.e.*, DIT, NOC, and IFANIN, used in literature to estimate the inheritance. We observe that only one metric out of the three experienced a degradation in the median values. For instance, the median decreases from 1.09 to 1.00 for DIT, whereas the medians increase from 0.15 to 0.19 and from 1.13 to 1.14 for NOC and IFANIN respectively. This indicates that developers probably decrease the depth of the hierarchy by adding more methods for a class to inherit, increasing the number of immediate subclasses, and increasing the number of immediate base classes. Although we observed certain cases that show inheritance improvement as perceived by developers, the overall depth of the inheritance tree and the number of immediate subclasses and superclasses did not decrease. The interpretation of the metric improvement highly depends on the quality of the code and the developer's design decisions. The statistical test shows that the differences are statistically significant for DIT and NOC, but they are not for IFANIN.

Summary. DIT generally decreases as developer intends to improve inheritance, and its variation is significant. IFANIN exhibit opposite variations to inheritance, but it is not statistically significant. Furthermore, our empirical investigation discards NOC from being an indicator for inheritance. Finally, at least one metric has a significant positive variation which matches the developer's perception of improving inheritance.

5) Polymorphism: For commits whose messages report the optimization of the polymorphism quality attribute, the boxplots sketched in Figures 2p, 2q show the pre- and post-refactoring results of two structural metrics, *i.e.*, WMC and RFC, used in literature to estimate the polymorphism. We observe that none of these metrics experienced a degradation in the median values.

The concept of polymorphism is closely related to inheritance. When developers inherit instance variables and methods from another class, polymorphism techniques allow the subclasses to use these variables and methods to perform different tasks. For this quality attribute, we observe similar trends to inheritance. There is a rise in the median for both WMC and RFC. When developers explicitly refer to polymorphism aspect improvement as a target in the commit messages, they tend to increase the number of local and inherited methods. The statistical test shows that the differences are not statistically significant.

Summary. WMC and RFC exhibit opposite variations to polymorphism, but they are not statistically significant. Therefore, we could not find any metric that has a significant positive variation which matches the developer's perception of improving polymorphism.

6) Encapsulation: For commits whose messages report the optimization of the encapsulation quality attribute, the boxplots sketched in Figures 2r, 2s show the pre- and post-refactoring results of two structural metrics, *i.e.*, WMC and the normalized LCOM, used in literature to estimate the encapsulation. We observe that both metrics experienced a degradation in the median values. However, the variations are statistically significant.

From a qualitative perspective, we observe that developers prevent access to attributes and methods by defining them to be private and enclosing them within a single construct. Although the results for the encapsulation metrics are not statistically significant, the significant results for cohesion and complexity-related commits discussed previously might indicate that the information hiding mechanism could generally help in reducing the complexity of the software systems when developers are actually limiting the inter-dependencies between components, and thus promote cohesion and modularity.

Summary. WMC and the normalized LCOM generally decrease as developer intends to improve encapsulation, but their variations are not significant. Therefore, we could not find any metric that has a significant positive variation which matches the developer's perception of improving encapsulation.

7) Abstraction: For this quality attribute that is measuring the generalization-specialization aspect of the design, we noticed an improvement of both the WMC and the normalized LCOM metrics, as shown in Figures 2t, 2u. The differences are not statistically significant as shown from the statistical test. Using this attribute, developers seem to practically handle the complexity of the methods when adding one or more descendants by actually hiding the implementation details, and increasing the class cohesion.

Summary. WMC and the normalized LCOM generally decrease as developer intends to improve abstraction, but their variations are not significant. Therefore, we could not find any metric that has a significant positive variation which matches the developer's perception of improving abstraction.

8) Design Size: For commits whose messages report the optimization of the design size quality attribute, the boxplots sketched in Figures 2v, 2w, 2x, 2y, 2z, 2aa show the pre- and post-refactoring results of five structural metrics, *i.e.*, LOC, CLOC, STMTC, CDL, NIV, NIM, used in literature to estimate the design size. We notice the improvement of four metrics, namely CLOC, CDL, NIV, and NIM after the commits in which developers explicitly target the improvement of the size of the classes. As can be seen in the box plots, the medians decreased in general. On the other hand, we notice an increase in LOC and STMTC. Regardless of the increase or decrease of metric values, their variations are not statistically significant. This indicates that developers reduce (1) line containing comments, (2) the number of classes and (3) the number of declared instance variables and methods. As for LOC and STMTC, we observed minor increases in the metric values.

Summary. CLOC, CDL, NIV, and NIM generally decrease as developer intends to improve design size, but their variations are not significant. Therefore, we could not find any metric that has a significant positive variation which matches the developer's perception of improving design size.

V. Threats to Validity

Our study has used a few thousands of refactoring commits in various systems. Since the analysis was not carried out in a controlled environment, few threats are discussed in this section as follows:

Table (V) Effect of refactoring on structural metrics, clustered by their corresponding internal quality attribute. **bold** indicates statistical significance. *italic* indicates improvement.

Quality Attribute	Metric	Impact	*p*-value
Cohesion	LCOM	+ve	***0.0346***
Coupling	CBO	+ve	***0.0400***
	RFC	-ve	0.2729
	FANIN	+ve	*0.2338*
	FANOUT	+ve	***0.0456***
Complexity	CC	+ve	***0.0001***
	WMC	+ve	***0.0062***
	RFC	-ve	**0.0021**
	LCOM	-ve	0.2431
	Evg	+ve	***0.0010***
	NPATH	+ve	***< 0.0001***
	MaxNest	+ve	***0.0026***
Inheritance	DIT	+ve	***0.0439***
	NOC	-ve	**0.0208**
	IFANIN	-ve	0.3987
Polymorphism	WMC	-ve	0.5137
	RFC	-ve	0.7983
Encapsulation	WMC	+ve	*0.1769*
	LCOM	+ve	*0.7737*
Abstraction	WMC	+ve	*0.1924*
	LCOM	+ve	*0.6988*
Design Size	LOC	-ve	0.8245
	CLOC	+ve	*0.7855*
	STMTC	-ve	0.3311
	CDL	+ve	*0.4870*
	NIV	+ve	*0.2757*
	NIM	+ve	*0.6043*

Internal Validity. Our analysis is mainly threatened by the accuracy of the refactoring mining tools because the tool may miss the detection of some refactorings. However, previous studies [37], [48] report that Refactoring Miner has high precision and recall scores compared to other state-of-the-art refactoring detection tools, which gives us confidence in using the tool. Another potential threat to validity relates to commit messages. This study does not exclude commits containing tangle code changes [20], in which developers performed changes related to different tasks and one of these tasks could be related to quality enhancement. If these changes were committed at once, there is a possibility that the individual changes are merged and cannot trace it back to the original task. We did not consider filtering out such changes in this study. Moreover, our manual analysis is a time consuming and error prone task, which we tried to mitigate by focusing mainly on commits known to contain refactorings.

Another potential threat to validity is the sample bias, where the choice of the data may directly impact the results. Therefore, we explored a large sample of projects, we made sure they are well engineered to ensure the quality of the findings along with diversifying the sources to reduce the bias of data belonging to the same entity. During our qualitative analysis, we considered only commits where a consensus between authors was made about whether a commit is clearly stating that it is optimizing a particular quality attributes. Commits which were debatable were discarded. We also provide our dataset online for further refinement and analysis.

External Validity. Our analysis was limited to only open source Java projects. However, we were still able to analyze 3,795 projects that are well-commented, and varied in size, contributors, number of commits and refactorings.

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 [7].

VI. Conclusion

Software developers do explicitly report target quality improvements in the commit messages of versioned repositories. In this work, we performed an exploratory study to investigate the alignment between quality improvement and software design metrics by focusing on 8 internal quality attributes and 19 distinct quality metrics. In summary, the main conclusions are:

- A variety of structural metrics can represent the internal quality attributes considered in this study. Based on our empirical investigation, for metrics that are associated with quality attributes, there are different degrees of improvement and degradation of software quality.
- Most of the metrics that are mapped to the main quality attributes, *i.e.,* cohesion, coupling, and complexity, do capture developer intentions of quality improvement reported in the commit messages. In contrast, there is also a case in which the metrics do not capture quality improvement as perceived by developers. We summarize our findings as follows:
  - **Cohesion.** In contrast with previous studies, cohesion tends to be well represented by LCOM as we found the metrics to be significantly improved in the refactored code (p-value $\leq$ 0.05).
  - **Coupling.** Similarly to cohesion, optimizing the coupling quality attribute was empirically measured using both CBO and FANOUT (p-value $\leq$ 0.05), in comparison with FANIN and RFC.
  - **Complexity.** One of the popular quality attributes that is being approximated by developers using a variety of metrics, namely CC, WMC, RFC, Evg, NPATH, and MaxNest (p-value $\leq$ 0.05).
  - **Inheritance.** While DIT has been found be the metric that matches the developer's perception (p-value $\leq$ 0.05), NOC, known to be a measure for Inheritance in literature, tends to increase instead in practice (p-value $\leq$ 0.05).
  - As for **Encapsulation**, **Abstraction** and **Design Size**. We cannot find any metric that can represent developer's intention of optimizing these quality attributes, and so these findings motivates a deeper investigation on understanding the mismatch between theory and practice.

As future work, we plan to empirically assess the impact of external quality metrics (*e.g.,* testability and readability) as documented by developers in their commit messages on quality and compare and contrast them with the findings for the internal ones. This will give us an indication which quality attributes are optimized the most by developers. Also, we plan

on investigating the impact of composed refactorings on each of the quality attributes, in contrast with existing studies which analyze each refactoring type individually. Another potential research direction is to explore what factors might contribute to the significant improvement of the quality metrics that are aligned with developer perception tagged in the commit messages (*e.g.,* developer experience, proximity to release date, and refactoring community culture).

ACKNOWLEDGMENT

We sincerely thank the authors of the refactoring mining tools that we have used in this study, for providing their tools open source and for allowing the community to benefit from them.

REFERENCES

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

[2] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

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

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

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

[6] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 74–83. ACM, 2017.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[8] C. M. S. Couto, H. Rocha, and R. Terra. A quality-oriented approach to recommend move method refactorings. In *Proceedings of the 17th Brazilian Symposium on Software Quality*, pages 11–20. ACM, 2018.

[9] S. Demeyer. Maintainability versus performance: What's the effect of introducing polymorphism. *Edegem, Belgium: Universiteit Antwerpe*, 2002.

[10] G. Destefanis, S. Counsell, G. Concas, and R. Tonelli. Agile processes in software engineering and extreme programming. chapter Software Metrics in Agile Software: An Empirical Study, pages 157–170. Springer-Verlag, Berlin, Heidelberg, 2014.

[11] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*, pages 144–151. IEEE, 2004.

[12] B. Du Bois, S. Demeyer, and J. Verelst. Does the" refactor to understand" reverse engineering pattern improve program comprehension? In *Ninth European Conference on Software Maintenance and Reengineering*, pages 334–343. IEEE, 2005.

[13] B. Du Bois and T. Mens. Describing the impact of refactoring on internal program quality. In *International Workshop on Evolution of Large-scale Industrial Software Applications*, pages 37–48, 2003.

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

[15] B. Geppert, A. Mockus, and F. Robler. Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–pp. IEEE, 2005.

[16] I. Griffith, S. Wahl, and C. Izurieta. Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In *Proceedings of the ISCA 24rd International Conference on Computer Applications in Industry and Engineering*. Citeseer, 2011.

[17] G. Hegedűs, G. Hrabovszki, D. Hegedűs, and I. Siket. Effect of object oriented refactorings on testability, error proneness and other maintainability attributes. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*, page 8. ACM, 2010.

[18] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[19] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, (5):510–518, 1981.

[20] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016.

[21] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585. IEEE, 2002.

[22] R. Leitch and E. Stroulia. 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)*, pages 309–322. IEEE, 2003.

[23] M. Lorenz and J. Kidd. *Object-oriented software metrics*, volume 131. Prentice Hall Englewood Cliffs, 1994.

[24] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[25] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. 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*, pages 252–266. Springer, 2007.

[26] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *International Conference on Software Reuse*, pages 287–297. Springer, 2006.

[27] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

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

[29] C. Neelamegam and M. Punithavalli. A survey-object oriented quality metrics. *Global Journal of Computer Science and Technology*, 9(4):183–186, 2009.

[30] B. A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[31] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[32] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[33] J. Ratzinger, M. Fischer, and H. Gall. *Improving evolvability through refactoring*, volume 30. ACM, 2005.

[34] H. A. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *icsm*, page 154. IEEE, 2000.

[35] R. Shatnawi and W. Li. 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.

[36] D. Silva, R. Terra, and M. T. Valente. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 146–156. ACM, 2014.

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

[38] D. Silva and M. T. Valente. Refdiff: detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 269–279. IEEE Press, 2017.

[39] V. Singh and V. Bhattacherjee. Evaluation and application of package level metrics in assessing software quality. *International Journal of Computer Applications*, 58(21), 2012.

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

[41] E. Stroulia and R. Kapoor. Metrics of refactoring-based development: An experience report. In *OOIS 2001*, pages 113–122. Springer, 2001.

[42] G. Szóke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy. 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*, pages 95–104. IEEE, 2014.

[43] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 183–192. IEEE, 2003.

[44] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software*, 66(3):225–239, 2003.

[45] L. Tan and C. Bockisch. A survey of refactoring detection tools.

[46] R. Terra, M. T. Valente, S. Miranda, and V. Sales. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19–36, 2018.

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

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

[49] N. Ubayashi, Y. Kamei, and R. Sato. Can abstraction be taught? refactoring-based abstraction learning. In *6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018*, pages 429–437. SciTePress, 2018.

[50] D. Wilking, U. F. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007.