How does refactoring affect internal quality attributes?

A multi-project study

Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Alessandro Garcia
Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
{alopez, iferreira, emfernandes, dcgrego, afgarcia}@inf.puc-rio.br

ABSTRACT

Refactoring is a technique commonly applied by developers along software maintenance and evolution. Software refactoring is expected to improve the internal quality attributes of a program, such as coupling and cohesion. However, there is limited understanding of whether and to what extent developers achieve this expectation when they refactor their source code. In this study, we investigate how refactoring operations affect five well-known internal quality attributes: cohesion, complexity, coupling, inheritance, and size. For this purpose, we analyze the version history of 23 open source projects with 29,303 refactoring operations. Our analysis revealed interesting observations. First, we noticed that developers apply more than 94% of the refactoring operations to program elements with at least one critical internal quality attribute, as opposed to previous work. Second, in 65% of the cases, the related internal quality attributes are improved, and the remaining 35% operations keep the quality attributes unaffected. Third, whenever pure refactoring operations are applied (i.e., the so-called root-canal refactoring), we confirm that internal quality attributes are either frequently improved or at least not worsened. Finally, while refactoring operations often reach other specific aims, such as adding a new feature or fixing a bug, 55% of these operations surprisingly improve internal quality attributes, with only 10% of the quality decline.

CCS CONCEPTS

•Software and its engineering →Maintaining software; Software evolution; •General and reference →*Measurement; Empirical studies;*

KEYWORDS

Refactoring, code structural quality, software metrics

ACM Reference format:
Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?. In *Proceedings of 31th Brazilian Symposium on Software Engineering, Fortaleza, Ceará, Brazil, Sept 2017 (SBES'17),* 10 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Refactoring is a technique commonly applied by developers to improve code structural quality. Fowler [9] defines refactoring as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*". One way to improve code structural quality concerns the improvement of measurable internal quality attributes, such as *coupling*, *cohesion*, *inheritance*, *complexity*, and *size* [19]. Each refactoring type is expected to relate mostly to a specific set of internal quality attributes. For instance, *Extract Superclass* tends to: (i) reduce the *size* of its subclasses, and (ii) increase their *inheritance* depth to improve reuse.

However, there is limited knowledge on the impact of refactoring on internal quality attributes. Some previous work [6, 21] simply hypothesize that each single refactoring operation improves multiple internal quality attributes. Other studies try to investigate the influence of refactoring on different aspects of structural quality. For instance, Bavota et al. [1] and Cedrim et al. [4] attempt to investigate the relationship of refactoring operations and the presence of poor code structures. In general, their findings suggest that poor code structures are often not removed through refactoring. However, they do not explicitly analyze the impact of refactoring on internal quality attributes. It may be the case that each refactoring operation does not suffice to remove poor structures, but it does improve certain quality attributes.

In this paper, we address the limitations of previous work by investigating *how refactoring operations affect internal quality attributes.* That is, we aim at studying fine-grained relationships between refactoring operations and code structural quality. For this purpose, we analyze the version history of 23 Java open source projects collected from GitHub, with 113,306 commits and 29,303 refactoring operations. We also rely our analysis on 11 widely studied refactoring types [23], and five internal quality attributes often mentioned in the well-known Fowler's catalog [9]: *cohesion*, *complexity*, *coupling*, *inheritance*, and *size.*

We split our analysis into two complementary parts. First, we analyze the frequency of refactoring operations applied to code elements with critical internal quality attributes, i.e., attributes that have a critical value for at least one of its related metrics. This analysis aims at revealing whether refactoring operations often target structurally critical elements in the source code. If so, this means that refactoring indeed has the potential to improve internal structural attributes. Second, we analyze the consequential impact of refactoring operations on internal quality attributes. In other words, we investigate if these attributes tend to improve, decrease or remain unaffected as a consequence of refactoring operations.

The emphasis on structural quality may vary depending on the tactic used by developers to apply refactoring. Thus, in order to better understand the impact of refactoring operations, we distinguish them according to two refactoring tactics: *root-canal refactoring*, in which developers are explicitly concerned with improving the structural code quality, and *floss refactoring*, in which developers use refactoring operations as means to reach other goals rather than only improving the structural code quality, such as adding new features or fixing bugs. Overall, there is limited knowledge regarding the differences of both tactics. Thus, we aim to understand if there are differences between the refactoring tactics applied by developers, regarding their impact on internal quality attributes.

As a result, we find out that developers apply more than 94% of the refactoring operations to code elements with at least one critical internal quality attribute. This observation is valid for all refactoring operations, regardless the refactoring tactics. We then conclude that refactoring operations mostly target on code elements that indeed require structural quality improvement. As far as the consequential impact of refactoring is concerned, we observe that refactoring operations tend to either improve or unaffect the internal quality attributes, regardless the refactoring tactic (*root-canal refactoring* or *floss refactoring*). We also discuss how our results reinforce or contradict previous findings of the literature.

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 discusses related work. Section 4 describes the study design. Section 5 presents results regarding the frequency of refactoring operations applied to code elements with critical internal quality attributes. Section 6 presents results regarding the impact of refactoring operations on different internal quality attributes. Section 7 describes threats to validity. Section 8 concludes the paper and outlines future directions.

2 BACKGROUND

This section provides background information. Section 2.1 discusses refactoring, and Section 2.2 discusses internal quality attributes.

2.1 Refactoring

Refactoring means changing the source code without changing the external behavior of a software project but improving its internal code structure [9]. Each refactoring operation is a micro-transformation at the source code level that affects multiple code elements. An example of refactoring operation is when the developer moves a method from one class to another, to remove excessive dependencies between classes. This refactoring type is called *Move Method*. There are several other refactoring types with different purposes, such as extracting new code elements from excessively complex elements and managing class inheritances.

Table 1 presents the refactoring types analyzed in our study. The first column presents the refactoring type. The second column informs the problem that each refactoring type addresses. The third column describes the solution aimed by applying each refactoring type. We selected 11 refactoring types from the Fowler's catalog [10]. These refactoring types have been largely investigated in the literature [23]. For instance, when a class provides a set of resources to other classes, one may apply a *Extract Interface* to provide only the resources of interest to the client classes.

As aforementioned, a single refactoring operation may affect multiple code elements. Thus, we consider as refactored elements the set of elements directly involved in the refactoring operation. For instance, let us consider the *Move Method* refactoring. In this refactoring type, a method m is moved from class A to class B. Hence, the set of refactored elements is $\{m, A, B\}$.

In addition, this study investigates two refactoring tactics, namely *root-canal refactoring* and *floss refactoring*. Developers apply *root-canal refactoring* when they aim to exclusively improve the code structural quality. In contrast, developers apply *floss refactoring* as means to reach a particular goal rather than improving the code structural quality, such as adding a new feature or fixing a bug. The analysis of both refactoring tactics may help understanding if there is a relationship between the refactoring tactics and the behavior of internal quality attributes.

2.2 Internal Quality Attributes

Internal quality attributes are key indicators of code structural quality [19]. Previous work apply diffirent software quality metrics for measuring internal quality attributes [21]. In this paper, we analyze five internal quality attributes, namely *cohesion*, *complexity*, *coupling*, *inheritance*, and *size*. We selected these internal quality attributes because they are closely related to the 11 refactoring types that we aim to analyze.

We describe each internal quality attributes as follows. *Coupling* is the degree of interdependence between modules or classes [5]. High *coupling* affects maintainability and reusability, for instance. *Cohesion* is the degree to which the internal elements of a module are related to each other. Low *cohesion* may lead to high *complexity* and bug proneness. *Complexity* is the measure of the overload of responsibilities and decision of a module. It affects the code readability, for instance. *Inheritance* represents relationships between superclasses and subclasses. *Inheritance* enables software reusability, but large hierarchies may lead to software maintenance problems. Finally, *size* measures the length of a module.

Existing literature has been using software metrics to capture specific internal quality attributes [6, 21]. For instance, the *Lines of Code (LOC)* metric quantifies the *size* of code elements. Section 4.5 describes the adopted procedure for measuring each of the internal quality attributes analyzed in this study.

3 RELATED WORK

Our study goal is to investigate how refactoring operations affect internal quality attributes. In this section, we discuss previous work that conduct similar investigation. Section 3.1 discusses studies that assess the impact of refactoring operations on software quality from different perspectives. Section 3.2 discusses previous work that are closely related to ours.

3.1 Refactoring and Software Quality

Bavota et al. [1] investigate three software systems aimed at understanding whether refactoring operations are applied on code elements with certain characteristics that suggest an opportunity of refactoring. These characteristics include quality metrics. According to their results, quality metrics do not have a clear relationship with refactoring. In contrast with their study, we investigate a

Table 1: Refactoring types analyzed in this study (extracted from Fowler [9])

Refactoring Type	Problem	Solution
Extract Method	Parts of code should be gathered in a single method	Create a new method with the extracted code
Extract Interface	Class that implement commonly used resources or two classes have part of their interfaces in common	Extract the subset into an interface
Extract Superclass	There are two classes with similar features	Create a superclass and move the common features to the superclass
Inline Method	When a method body is more obvious than the method itself, use this technique	Replace calls to the method with the method's content and delete the method itself
Move Field	A field is, or will be, used by another class more than the class on which it is defined	Create a new field in the target class, and change all its users
Move Method	A method is, or will be, using or used by more features of another class than the class in which it is defined	Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether
Rename Method	The name of a method does not reveal its purpose	Change the name of the method
Pull up Field	Two subclasses have the same field	Move the field to the superclass
Pull up Method	There are methods with identical results on subclasses	Move them to the superclass
Push down Field	A field is used only by some subclasses	Move the field to those subclasses
Push down Method	The behavior on a superclass is relevant only for some of its subclasses	Move it to those subclasses

much larger set of software systems. Also, we have considered 25 quality metrics, 10 metrics more than those analyzed by Bavota et al. [1]. In addition, they did not explicitly analyze internal quality attributes as they focus on individual quality metrics. Our results contradict their findings as we observed that refactoring operations are frequently applied on code elements with at least one critical internal quality attribute.

Other studies [1, 4] classify refactoring operations according to the occurrence, addition or removal of poor code structures. However, the improvement of code structural quality may not be perceived whether only poor code structures are considered. Full removal of these poor code structures may require the simultaneous improvement of more than one internal attribute. Developers may need several refactoring operations to fully remove these structures. To address this literature gap, we investigate the effect of refactoring operations in a finer-grained level, i.e., we analyze the impact of refactoring operations directly on internal quality attributes.

Silva et al. [24] present an empirical study on the motivation of developers when applying refactoring operations. They conclude that refactoring operations are mostly driven by changes in software requirements rather than poor code structures. Even though the authors have not analyzed the relationship between refactoring and internal quality attributes, some of their findings suggest that there might be a relationship between certain motivations and internal quality attributes. For instance, they mention that developers are concerned about removing duplicated code, which is associated with two specific internal quality attributes: *size* and *inheritance*. The same reasoning applies to other types of motivation, such as maintainability and testability. Therefore, the impact of refactoring operations on internal quality attributes may be related to different high-level goals of developers who apply refactoring.

Furthermore, the authors [24] mention that developers are seriously concerned about avoiding code duplication. For that, they apply the *Extract Method* refactoring type to remove it. This finding suggests that, despite of the aim of developers when refactoring code (not explicitly mentioned by the authors), developers are concerned on improving the code structural quality (i.e., they apply *root-canal refactoring*).

3.2 Refactoring and Internal Quality

Previous work [7, 14] investigate whether refactoring operations improve the code structural quality. Du Bois and Mens [7] measure the relationship between only five quality metrics and three refactoring types: *Extract Method*, *Encapsulate Field*, and *Pull up Method*. They proposed a formalism based on abstract syntax tree representation of the source code, extended with cross-references to describe the impact of refactoring on only five metrics. The results of their work showed both positive and negative impacts on the studied metrics.

On the other hand, Kataoka et al. [14] focused only on the *coupling* metrics to evaluate the refactoring effect, comparing the metrics before and after refactoring operations. They analyzed only a single C++ program for *Extract Method* and *Extract Class* refactoring types, which were performed by a single developer. All these limitations pose threats to the validity of their findings.

We have noticed that these studies were limited to only a few internal quality attributes, a few number of projects or/and a few number of refactoring types. In our study, we try to analyze more internal quality attributes to get a more precise indication of whether or not refactoring affects internal quality attributes. Also, we analyze more projects to capture different software development contexts, and we study 11 types of refactoring, which according to Murphy-Hill [23], are the most common refactoring types. Finally, we analyze the impact of refactoring on internal attributes both before and after each refactoring operation.

4 STUDY DESIGN

This section presents the study design. Section 4.1 introduces the research questions. Section 4.2 presents the selection of software projects. Section 4.3 shows how we detected refactoring operations and the procedure to classify refactoring operations in *root-canal refactoring* and *floss refactoring*. Section 4.4 presents the protocol for mapping refactoring operations to internal quality attributes. Finally, Section 4.5 illustrates how we measure internal quality attributes. All study artifacts are available at the study website [8].

4.1 Research Questions

Our study aims at assessing how refactoring affects internal quality attributes. To achieve our goal, we will address the following research questions.

RQ1. Are refactoring operations often applied to code elements with critical internal attributes?

We designed RQ1 to investigate if developers tend to apply refactoring operations on code elements with critical internal quality attributes. Our goal is to understand if refactoring operations are potential indicators of problems in the code structure. In the affirmative case, we can investigate the impacts of these refactoring operations on the internal quality attributes (RQ2).

To answer RQ1, we need to assess the critically of each code element affected by a refactoring operation. For this purpose, we annotated each refactoring operation according to the number of critical internal attributes present in the code elements affected by the refactoring. Thus, refactored code elements can be affected by (i) none critical attribute, elements that do not have any critical attribute, (ii) single critical attribute, elements that have only one bad attribute, and (iii) multiple critical attributes, elements that have more than one critical attribute.

In our study, an attribute is critical whether at least one metric used to measure the attribute is either below a lower threshold or above an upper threshold. Thresholds are computed for each project's commits. These threshold values are calculated based on quartiles [22] for the commit. The *first* and *third* quartiles define the lower and upper thresholds, respectively. For each commit, we collect the metrics related to each code element. Then, we define the lower and upper thresholds of each metric by calculating the *first* and *third* quartile of the metric values obtained from all the code elements. The first quartile (Q1) is the lower threshold, which comprises the 25% of the bottom data (metrics). The third quartile (Q3) is the upper threshold, which comprises the top 25% of the data above it. Thus, if a metric value of an element is above the Q3, then we say that metric got critical in that element. There are few metrics that are critical if their value is below the Q1, according to the nature of the metric. A detailed description of the metrics used are available on the study website [8]. If an element has more that one critical metric, then we say that the internal quality attribute related to those metrics is also critical.

RQ2. What is the impact of refactoring on internal quality attributes?

Each internal quality attribute is measured using a set of quality metrics, where each metric captures a distinct property of each internal attribute. For instance, to measure coupling it is not enough to know the number of elements of a class that calls elements of another class. We also need to capture the number of elements of other classes that call elements of the class being measured. Thus, for each internal attribute, we selected a set of metrics (Section 4.5) that may have improved, worsened or may not be affected after applying a refactoring operation. The behavior of each metric is calculated by comparing the metric value of the code elements in the commits before and after each refactoring operation.

To answer our RQ2, we divide the analysis into two approaches, related to how to classify the behavior of internal quality attributes: (i) *At Least One Metric*, and (ii) *Most Metrics*. In the first approach, we assume that the internal quality attribute improves when any of the metrics that measures such attribute improve. In the second approach, we assume that an internal quality attribute improves when most of the metrics that capture the attribute also improve. Both approaches are explained in more detail below.

Most metrics. An internal quality attribute improves when most of the metrics that capture the attribute also improve. This approach tries to cover the scenarios in which several properties of each attribute are improved by a refactoring operation. This approach is strict because each refactoring operation has to improve several metrics at once. This strict approach does not consider an attribute improvement if only a few metrics that mostly related to the performed operation, in order to consider the improvement of the internal quality attribute.

At least one metric. An internal quality attribute improves when any of the metrics that quantifies such attribute also improve. This approach is interesting because the developer may improve only one of the properties of an internal quality attribute in each refactoring operation. On the other hand, we try to resolve other study problems (Section 3.1) that measure internal quality attributes using only one quality metric. This approach is less strict than the *most metrics* approach because, for each refactoring operation, only one metric has to improve, in order to consider the improvement of the internal quality attribute.

4.2 Selection of Software Projects

To conduct our study, we selected a set of software projects. We focused the data analysis on open source projects to support the study replication and extension. We collected projects from GitHub repositories because we are concerned with the evolution of software projects regarding refactoring operations. In this study, we analyze 23 software projects selected using the following quality criteria. First, Java software projects, a very popular programming language[1]. Second, open source projects, to allow the study replication. Third, highly popular projects, based on the number of stars received by the projects.

Table 2 lists the selected software projects. The first column presents the partial repository path at GitHub. The second column provides the number of commits per project. The third column presents the number of refactoring operations per project. The projects have, in average, 4,926.35 commits and 1,274.04 refactoring operations.

[1]https://www.tiobe.com/tiobe-index/

Table 2: Software Projects Analyzed in this Study

Software Project	# Commits	# Refactorings
alibaba/dubbo	1,836	280
AndroidBootstrap/android-bootstrap	230	8
apache/ant	13,331	2,063
argouml	17,654	2,588
elastic/elasticsearch	23,597	9,507
facebook/facebook-android-sdk	601	341
facebook/fresco	744	161
google/iosched	129	73
google/j2objc	2,823	713
junit-team/junit4	2,113	309
Netflix/Hystrix	1,847	266
Netflix/SimianArmy	710	55
orhanobut/logger	68	20
PhilJay/MPAndroidChart	1,737	398
prestodb/presto	8,056	2,068
realm/realm-java	5,916	1,699
spring-projects/spring-boot	8,529	1,386
spring-projects/spring-framework	12,974	5,320
square/dagger	696	96
square/leakcanary	265	12
square/okhttp	2,645	855
square/retrofit	1,349	232
xerces	5,456	853
Sum	**113,306**	**29,303**
Mean	**4,926.35**	**1,274.04**

4.3 Refactoring Detection and Classification

Since we aim to analyze a large set of commits for different systems, we used the Refactoring Miner tool (version 0.2.0) [25] to detect refactoring operations. Previous work observed a precision of Refactoring Miner equals 96.4% with low rates of false positives [25], which we confirmed in our validation process. Refactoring Miner detects 11 of the most recurring refactoring types investigated in the literature [23].

Validation of Refactoring Types. We conducted a manual validation of the refactoring types identified by the Refactoring Miner tool. Such validation covered a random set of refactoring operations. After applying a statistical test with a confidence level of 95%, we observed a high precision of the tool for each refactoring type, with a median of 88.36% (excluding the *Rename Method* refactoring type). By applying the Grubb outlier test [11] (alpha = 0.05), we could not find any outliers, indicating that no refactoring type strongly influences the median precision found. Thus, the obtained results represent a key factor to provide reliability to the results reported in this work.

Refactoring Tactic Classification. In our study, we manually analyzed a randomly selected sample of refactoring operations to classify them as *root-canal refactoring* or *floss refactoring*. Our goal is to investigate whether the refactoring tactics influences in the frequency and impacts of refactoring operations on the internal quality attributes. For this purpose, we assess whether the changes performed by developers during the refactoring are exclusively refactoring operations. We classify a transformation as floss refactoring when we identify additional changes in the code, such as the addition of methods or changes in a method body unrelated to refactoring operations. Otherwise, we classify the refactoring as *root-canal refactoring*.

4.4 Mapping Refactoring to Internal Quality Attributes

One could expect that each refactoring type is likely to positively affect a subset of quality attributes. Thus, we mapped each refactoring type to one or more internal quality attributes by relying on previous work [9, 13]. Essentially, we infer the internal quality attributes related to each refactoring type addressed in our study from the Fowler's catalog [9]. By doing that, we aim to gain a better understanding of how refactoring operations affect each related internal quality attribute.

Table 3 presents the association of each refactoring type with internal quality attributes, which are expected to be improved by each refactoring type. The first column lists each refactoring type. The second column presents the related internal quality attributes. According to the table, we expect that *Extract Superclass* has a direct impact on *size* and *inheritance*. Thus, if there are two different classes that implement similar functionalities, we apply *Extract Superclass* to create a single superclass that implements such functionalities. Consequently, we remove the duplicate code of both subclasses, thereby reducing their *size*, and increasing the *inheritance* depth.

Table 3: Expected effect on internal quality attributes

Refactoring Type	Related Internal Attributes
Extract Interface	Size, Inheritance
Extract Method	Size, Coupling
Extract Superclass	Size, Inheritance
Inline Method	Size, Coupling
Move Field	Coupling, Cohesion, Inheritance
Move Method	Coupling, Cohesion, Complexity, Inheritance
Pull Up Field	Size
Pull Up Method	Size
Push Down Field	Inheritance
Push Down Method	Inheritance
Rename Method	Size

4.5 Measuring Internal Quality Attributes

In Section 4.4, we mapped each refactoring type to its related internal quality attributes. After, we investigated the quality metrics for quantifying each internal quality attribute. We defined a set composed by 25 quality metrics based on previous work [2, 5, 15–18, 20]. To select appropriate metrics per internal quality attribute, we applied the following criteria. First, we selected well-known metrics from the literature [2, 15–18], including the CK suite [5] and McCabe's cyclomatic complexity [20]. Second, we selected metrics that assess different properties of each internal quality attribute [3, 5, 19, 20]. For example, LOC measures the number of lines of code, CBO measures the number of classes to which a class is coupled, and CC measures the complexity of a module's decision structure. Third, we selected metrics with evidence of accuracy in capturing and predicting problems in structural code quality. For instance, LOC helps in identifying classes having a poor design from the viewpoint of software developers [1].

Table 4 presents the set of 25 metrics obtained through the aforementioned criteria. The first column lists the internal quality attributes. The second column presents the software metrics related to each internal quality attribute. Finally, the third column describes each software metric. To compute each metric, we used the non-commercial license of the Understand tool[2].

5 REFACTORING AND CRITICAL ELEMENTS

In this section, we assess whether refactoring operations are often applied to code elements with critical internal quality attributes. We aim to observe if developers target their refactoring efforts mostly on code elements that require structural quality improvement. By answering the RQ1, we will be able to assess the impact of refactoring operations on internal quality attributes (RQ2). Therefore, we answer RQ1 (*Are refactoring operations often applied to code elements with critical internal quality attributes?*) as follows.

To address this research question, we have analyzed the critical internal quality attributes targeted by refactoring operations. In particular, we computed the number of critical internal quality attributes associated with code elements being refactored. Algorithm 1 illustrates this process. The algorithm iterates over the refactoring operations (line 2). For each refactoring r, the code elements affected by r are iterated (line 4). For each code element e, we compute the number of critical internal quality attributes (line 5). Finally, we sum the total number of critical internal attributes for the current code elements affected by a refactoring operation (line 6). The total number of critical internal attributes is used to classify the refactoring operation as *None* if the number equals zero, *Any* if the number equals one, and *Multiple* if the number is greater than one (line 7).

Algorithm 1 Classifying refactoring operations

```
1: Initilize $R$ = refactoring operations
2: for $r \in \mathcal{R}$ do
3:     Initilize $E$ = code elements affected by $r$
4:     for $e \in \mathcal{E}$ do
5:         Initilize $c$ = number of critical internal attributes of $e$
6:         $t$ += $c$
7: Classify $r$ as $None$, $Single$, or $Multiple$
```

After computing the number of critical internal quality attributes present in code elements affected by refactoring, we are able to analyze the results. Thus, Table 5 presents the number of refactoring operations applied on code elements with critical internal quality attributes. The first column lists each experimental group: *Sample*, composed of the refactoring operations that we manually classified (Section 4.3), and *All*, composed by all the refactoring operations collected from the target projects. The second column indicates the tactics for conducting refactoring operations, namely *Root-canal refactoring* and *Floss refactoring*. Note that the *All* group includes all refactoring operations regardless the refactoring tactic applied by developers. The third column provides the total number of refactoring operations per refactoring tactic and experimental group. The fourth column presents the number and percentage of operations applied to code elements without critical internal attributes. The fifth and sixth columns present the number and percentage of operations applied to code elements with at least one critical internal attribute, i.e., the *Any* columns (see Section 4.1). We divided these operations into two, namely: *Single*, i.e., the ones with exactly one critical internal attribute, and *Multiple*, i.e., the ones with two or more critical internal attributes.

The data in Table 5 suggest that developers tend to apply refactoring operations most frequently to code elements with at least one critical internal quality attribute, as indicated in the *Any* columns. By summing the refactoring operations with a *Single* and *Multiple* critical internal quality attributes, the total frequency represents 94.64% (27,733) of the refactoring operations. This result takes into consideration the *All* line regarding all analyzed refactoring operations. Moreover, most of these refactoring operations are applied to *Multiple* critical internal quality attributes, i.e., two or more attributes. It represents 79.43% (23,275) of the refactoring operations, against only 15.21% (4,458) for code elements with a *Single* critical internal quality attribute.

These observations suggest that critical internal quality attributes may indicate code elements that require refactoring operations for improving their structural quality. Also, they contradict the finding of Bavota et al. [1], which suggest that there is no clear relationship between quality metrics and refactoring operations. One of the reasons for not having a clear relationship is that Bavota et al. [1] considers the improvement of an internal quality attribute if the same metric related to that attribute often improves along all refactoring operations. In contrast, our study analyzes multiple metrics for the same internal quality attribute, which may have increased the chances of capturing the positive impact of refactoring operations on internal quality attributes.

Also from data in Table 5, we can draw conclusions on the frequency of refactoring operations per refactoring tactic. By comparing *Root-canal refactoring* and *Floss refactoring*, we observe no relevant differences between the percentages of operations per number of critical attributes. As in the aforementioned overall analysis, most of the operations are applied to code elements with at least one critical attribute. The percentages are 98.78% and 99.74% for *root-canal refactoring* and *floss refactoring*, respectively. Thus, regardless the aim of developers on improving or not the structural program quality, they tend to apply refactoring operations in elements with many critical attributes.

Finding 1. Most of the refactoring operations (94.64% in average) are very often applied to code elements with at least one critical internal quality attribute. This observation is also valid for both root-canal refactoring (98.78%) and floss refactoring (99.74%).

[2]http://www.scitools.com

Table 4: Quality metrics used in this study

Attribute	Metric	Metric description
Coupling	Coupling Between Objects (CBO) [5]	The number of classes to which a class is coupled
	Fan-in (FANIN)	The number of other classes that reference a class
	Fan-out (FANOUT)	The number of other classes referenced by a class
	Coupling Intensity (CINT) [15]	The number of distinct operations called by the measured operation
	Coupling Dispersion (CDISP) [15]	The number of classes in wich the operations called from the measured operation are defined by
Cohesion	Lack of COhesion of Methods 2 (LCOM2) [5]	Number of pairs of methods that do not share attributes, minus the number of pairs of methods that share attributes
	Lack of COhesion of Methods 3 (LCOM3) [16]	Number of disjoint components in the graph that represents each method as a node and the sharing of at least one attribute as an edge
	Tight Class Cohesion (TCC) [2]	Ratio of number of similar method pairs to total number of method pairs in the class
Complexity	Cyclomatic Complexity (CC) [20]	Measure of the complexity of a module's decision structure.
	Weighted Method Count (WMC) [5]	The sum of Cyclomatic Complexity [20] of all methods declared in the given class
	Essential Complexity (Evg) [20]	Measure of the degree to which a module contains unstructured constructs
	Paths (NPATH)	Number of unique paths though a body of code, not counting abnormal exits or gotos
	Nesting (MaxNest)[18]	Maximum nesting level of control constructs
Inheritance	Depth of Inheritance Tree (DIT) [5]	The depth of a class as the number of its ancestor classes
	Number Of Children (NOC) [5]	The number of direct descendants (subclasses) of a class
	Base Classes (IFANIN)	Number of immediate base classes
	Override Ratio (OR) [15]	The number of methods of the measured class that override methods from the base class, divided by the total number of methods in the class
Size	Lines of Code (LOC) [18]	The number of lines of code excluding white spaces and comments
	Lines with Comments (CLOC) [18]	Number of lines containing comment
	Statements (STMTC) [18]	Number of statements
	Classes (CDL)	Number of classes
	Instance Variables (NIV) [18]	Number of instance variables
	Instance Methods (NIM) [18]	Number of instance methods
	Weight of Class (WOC) [15]	The sum of "funtional" public methods divided by the total number of public members
	Number of Public Attributes (NOPA) [15]	The number of publics attributes of a class

Table 5: Refactoring operations applied on code elements with critical internal quality attributes

Experimental Groups	Refactoring Tactic	Total	None	Any	
				Single	Multiple
Sample	Root-Canal	576	7 (1.22%)	17 (2.95%)	552 (95.83%)
	Floss	1,543	4 (0.26%)	13 (0.84%)	1,526 (98.90%)
All	-	29,303	1,570 (5.36%)	4,458 (15.21%)	23,275 (79.43%)

Table 6: Notation for Metric Behavior

Symbol	Description
↑	The metric improves
↓	The metric worsens
-	The metric remains unaffected

6 IMPACT ON INTERNAL ATTRIBUTES

In this section, we assess the impact of refactoring operations on different internal quality attributes. We aim to understand if developers improve at least one internal quality attribute after refactoring code elements, as previously suggested by Fowler [9]. On the other hand, we are interested in analyzing if developers affect most of the internal attributes when applying a refactoring operation. Thus, we answer RQ2 (*What is the impact of refactoring operations on internal quality attributes?*) as follows.

After applying a refactoring operation to a code element, the metrics that capture each internal quality attribute may present different behaviors. For instance, the value of a given metric may increase, decrease, or remain unaffected. Also, multiple metrics help to capture a single internal quality attribute. Thus, to understand the impact of refactoring operations on internal quality attributes, we defined a notation for the metric behavior. Table 6 presents the notation used to categorize the metric behavior. This notation aims to support the discussion of study results in Sections 6.1 and 6.2.

We address RQ2 by analyzing the behavior of internal quality attributes in two different approaches: *Most Metrics* and *At Least One Metric*. Section 6.2 describe each approach.

6.1 The *Most Metrics* Approach

Table 7 presents the impact of refactoring operations per type using the *Most Metrics* approach. The first column lists the 11 refactoring types under analysis. The second column provides the total number of refactoring operations per type. The remainder columns concern the five internal quality attributes investigated in this study. For each column, the table presents the predominant behavior for the respective refactoring type. For this purpose, we use the notation described in Table 6. These columns also provide the percentage of refactoring operations categorized in the predominant behavior for the respective internal quality attribute. Highlighted cells in the table indicate that a given internal quality attribute (i.e., a column of the table) is expected to improve after applying a given refactoring type (i.e., a line of the table). The gray highlight on each cell was assigned by the mapping of refactoring to internal quality attributes described in Table 3. These gray cells represent the cases of internal quality attributes expected to be improved by the corresponding refactoring types.

Table 7: Refactoring impact using the *Most Metrics* approach

Refactoring Type	Total	Cohesion	Coupling	Complexity	Inheritance	Size
Extract Superclass	341	↑ 51.32%	– 53.67%	– 60.12%	– 54.25%	↑ 73.02%
Extract Interface	133	– 83.46%	– 69.92%	– 94.74%	↑ 64.66%	– 42.11%
Move Field	4,355	↑ 63.31%	– 55.57%	– 88.4%	– 90.13%	– 50.95%
Inline Method	1,525	↑ 58.3%	↓ 39.87%	– 48.92%	– 92.92%	↑ 56.59%
Push down Field	78	↓ 47.44%	↑ 41.03%	– 87.18%	– 97.44%	↑ 46.15%
Extract Method	7,513	↓ 59.03%	↓ 45.77%	↑ 44.95%	– 93.81%	↓ 58.53%
Move Method	1,404	↓ 46.44%	↓ 41.17%	– 68.87%	– 81.05%	↑ 43.73%
Push down Method	114	↓ 42.11%	↑ 44.74%	– 59.65%	– 89.47%	↓ 58.77%
Pull up Method	629	↓ 43.88%	↓ 67.89%	– 69.16%	– 85.06%	↓ 52.94%
Pull up Field	465	↓ 59.35%	↓ 66.45%	– 68.6%	– 81.94%	↓ 63.87%
Rename Method	12,746	– 100%	– 82.93%	– 97.98%	– 100%	– 86.11%

Our results show that for the *Extract Superclass*, *Extract Interface*, and *Move Field* one or more internal quality attributes improved, and the others remained unaffected. Furthermore, one related internal quality attribute to a given refactoring type has improved. Overall, 3 out of 11 (27%) of the refactoring types have improved for one or more internal quality attributes. Although we have found a slight improvement of the internal quality attributes considering most of the metrics, there is some evidence that using a less strict approach we can find better results. Section 6.2 aims at analyzing the internal quality attributes in a less strict approach, i.e, using the *At Least One Metric* approach.

Finding 2. For some refactoring types, the internal quality attributes tend to improve or remain unaffected.

Based on Table 7 we compute the total internal quality attributes that improve, worsen and remain unaffected for each approach. Concerning the *Most Metrics* approach, the amount of related internal quality attributes (gray cells) is equal to 4 (20%); thus, we observed some positive impact of refactoring operations on certain internal quality attributes. However, the amount of worsening internal attributes is equal to 7 (35%), and 9 (45%) remain unaffected. In fact, the number of worsened internal quality attributes surpasses the number of improved attributes. However, it represents only a half of the total number of improved summed with unaffected internal quality attributes. These results suggest we should study if less strict approaches (e.g., the *At Least One Metric* approach) may better capture the positive impact of refactoring operations.

Bavota et al. [1] shows that there is no relationship between refactoring operations and metrics that capture key quality attributes, such as *cohesion*. Table 7 shows that *cohesion* is expected to be improved by *Move Method and Move Field* refactoring types. We observed that *Move Field* improves cohesion while *Move Method* has an opposite effect, using the *Most Metrics* approach. One of the possible reasons for this diverging results (with respect to Bavota et al. [1]) is that our study used a different set of metrics to capture *cohesion*. In their study, they analyze the LCOM[5] metric, that according to Henderson-Sellers [12] is not an appropriate metric to capture and measure *cohesion*.

6.2 The *At Least One Metric* Approach

Table 8 presents the impact of refactoring operations per type using the *At Least One Metric* approach. The structure of this table is similar to Table 7. Using this approach, when any of the metrics used to quantify the internal quality attribute improve, it means that the internal quality attribute also improve. An analysis of Table 8 reveals that, in most cases, refactoring operations tend to improve the related internal quality attributes. For instance, the *Extract Superclass* refactoring type is expected to improve two internal quality attributes, namely *inheritance* and *size* (see Table 3). Table 8 shows that this refactoring type improves *inheritance* in 92.67% of the cases, and *size* in 81.52%. Besides that, it also improves *cohesion* and *coupling*.

Table 8: Refactoring impact using the *At Least One Metric* approach

Refactoring Type	Total	Cohesion	Coupling	Complexity	Inheritance	Size
Extract Superclass	341	↑ 51.32%	↑ 48.68%	– 60.12%	↑ 92.67%	↑ 81.52%
Inline Method	1,525	↑ 58.3%	↑ 77.64%	– 48.07%	– 91.48%	↑ 88.98%
Extract Method	7,513	↓ 59.03%	↑ 71.4%	↑ 47.03%	– 93.09%	↑ 85.81%
Move Method	1,404	↓ 46.44%	↑ 48.01%	– 68.87%	– 74.29%	↑ 82.55%
Push down Field	78	↓ 47.44%	↑ 53.85%	– 87.18%	– 94.87%	↑ 79.49%
Push down Method	114	↓ 42.11%	↑ 77.19%	– 59.65%	– 85.09%	↑ 78.07%
Extract Interface	133	– 83.46%	– 63.16%	– 94.74%	↑ 68.42%	– 39.85%
Move Field	4,355	↑ 63.31%	– 49%	– 88.4%	– 87.83%	– 48.45%
Pull up Field	465	↓ 59.35%	↓ 66.45%	– 68.6%	– 70.97%	↑ 81.29%
Pull up Method	629	↓ 43.88%	↓ 67.89%	– 69.16%	– 83.47%	↑ 82.99%
Rename Method	12,746	– 100%	– 80.63%	– 97.98%	– 100%	– 85.74%

Based on Table 8 we compute the total internal quality attributes that improve, worsen and remain unaffected for each approach. With respect to *At Least One Metric* approach, the amount of related internal quality attributes (grey cells) is equals to 11 (55%). The amount of worsen internal attributes is equals to 1 (5%) and 8 (40%) remain unaffected. As expected, the number of improved internal quality attributes when using a less strict approach has increased in 35% when compared to the *Most Metrics* approach. In turn, the number of worsened attributes has declined in 30% when compared to the more strict approach.

Finding 3. The refactoring operations tend to improve or remain unaffected the internal quality attributes. This observation is valid when considering at least one metrics per internal quality attribute.

In the study of Bavota et al. [1] is evidenced a low relationship between refactoring operations and metrics that capture *coupling* internal quality attributes. In contrast, our study shows that refactoring operations in most of the cases improve the *coupling* internal quality attribute. For instance, in our work, we assume that *coupling* internal quality attribute is related with *Inline Method, Extract Method, Move Method, and Move Field* refactoring types. Using the *At Least One Metric* approach, we observed that *Inline Method, Extract Method and Move Method* refactoring types improve *coupling* and *Move Field* refactoring type does not affect *coupling*. One of the possible reasons for that is the different set of metrics to capture *coupling*. Bavota et al. [1] analyze three metrics that capture *coupling*, while we used five metrics (See Table 4).

6.3 Root-canal *versus* Floss Refactoring

This section discusses the impact of *root-canal refactoring* and *floss refactoring* using both approaches. Table 9 presents the general results for the *Most Metrics* approach. The first column lists each metric behavior. The second and third columns present data regarding *root-canal refactoring*. Each cell presents the number and percentage of internal quality attributes in the case of related attributes and all attributes, respectively. The fourth and fifth columns present data regarding *floss refactoring*. Similarly to the *root-canal refactoring*, each cell presents the number and percentage of internal quality attributes in the case of the related internal quality attributes and all attributes, respectively.

Table 9: General behavior using the *Most Metrics* approach by refactoring tactic

Behavior	Root-canal Refactoring		Floss Refactoring	
	Related Attributes	All Attributes	Related Attributes	All Attributes
↑	8 (40%)	14 (25.45%)	5 (25%)	10 (18.18%)
↓	4 (20%)	11 (20.00%)	7 (35%)	20 (36.36%)
-	8 (40%)	30 (54.55%)	8 (40%)	25 (45.45%)

Data in Table 9 point out interesting observations for the *Most Metrics* approach. For instance, for both *root-canal refactoring* and *floss refactoring*, the number of worsened attributes represents a half of the number of improved attributes summed with the number of unaffected attributes. These results suggest that regardless the refactoring tactic, refactoring operations tend to improve at least one metric of the attribute.

Table 10 presents the general results for the *At Least One Metric* approach. The first column lists each metric behavior. The second and third columns present, for root-canal refactoring, the number and percentage of internal quality attributes per behavior, in the case of related (second column) and all attributes (third column). The fourth and fifth columns present, for *floss refactoring*, the number, and percentage of internal quality attributes per behavior, in the case of related (fourth column) and all attributes (fifth column).

Considering the *At Least One Metric* approach, the number of improved and unaffected attributes are both higher than the number of worsened attributes. This observation applies to both refactoring tactics. As expected by using a less strict approach, these results evidence that refactoring operations have mostly at least a slightly positive impact on internal quality attributes.

Finding 4. *Root-canal refactoring* tends to improve internal quality attributes when considering at least one metric per attribute. Moreover, *floss refactoring* tends to improve at least one metric per attribute, even if developers are may not be explicitly concerned with this improvement.

Table 10: General behavior using *At Least One Metric* approach by refactoring tactic

Behavior	Root-canal Refactoring		Floss Refactoring	
	Related Attributes	All Attributes	Related Attributes	All Attributes
↑	13 (65%)	23 (41.82%)	11 (55%)	23 (41.82%)
↓	0 (0%)	5 (9.09%)	2 (10%)	11 (20%)
-	7 (35%)	27 (49.09%)	7 (35%)	21 (38.18%)

Comparing both approaches, we observed that the incidence of worsened internal attributes is higher in the *Most Metrics* approach than such incidence in the *At Least One Metric* approach. This observation applies to both refactoring tactics. This means that whenever refactoring operations worsen a quality attribute, those operations tend to considerably deteriorate the attribute. In other words, negative refactoring operations more often decrease multiple (rather than a single) metrics of each negatively affected attribute.

7 THREATS TO VALIDITY

We discuss threats to the study validity [26], with respective treatments, as follows.

Construct and Internal Validity. The threats to internal validity of this work concern the data collection procedure. Regarding the refactoring detection, we used the Refactoring Miner tool. To mitigate this threat, we selected random samples by refactoring type and performed a manual validation of it. The idea was to check whether we could have confidence on Refactoring Miner precision. We observed an average precision of Refactoring Miner equals to 96.4% with low rates of false positives. The set of metrics used in this study also represents a threat to validity, because it may not capture all relevant properties of the internal quality attributes. To mitigate this threat, we did not choose a random set of metrics. We chose metrics that assess different properties of each internal quality attribute [3, 5, 19, 20]. Another criteria was used to chose our set of metrics that are detailed in Section 4.5.

We analyzed 11 refactoring types detected by the Refactoring Miner tool. This amount represents a threat to validity because they do not cover all refactoring types reported by Fowler [9]. We mitigated this threat by considering the 11 refactoring types that represent the most common types according to the literature [23].

Conclusion and External Validity. In this paper, we analyzed different tactics to perform refactoring operations, namely *root-canal refactoring* and *floss refactoring*. Due the lack of effective tools for classifying refactoring operations by tactics (*root-canal* *refactoring* and *floss refactoring*), we decided to perform a manual classification of 2,119 refactoring operations (about 7% of all refactoring operations). This manual classification represents a threat to validity. Then, two researchers performed double-checking for a random subset of the classified refactoring operations and obtained a high precision (>95%) in the classification.

8 CONCLUSION

This paper investigates the impact of refactoring operations on internal quality attributes. For this purpose, we analyze 23 Java open source projects with 29,303 refactoring operations. Our study focuses on 11 commonly studied refactoring types and five internal quality attributes. We split our study into two parts. First, we investigate whether refactoring operations are often applied to code elements with critical internal quality attributes. Second, we assess the impact of refactoring operations on internal quality attributes using two complementary approaches: *Most Metrics*, when most of the metrics that capture an internal quality attribute improves, and *At Least One Metric*, when at least one of the metrics improves. We also analyze two different refactoring tactics: *root-canal refactoring*, when developers are explicitly concerned on improving the code structure quality, and *floss refactoring*, when quality improvement is a means to reach other goals.

As a result, we have noticed that developers apply more than 94% of the refactoring operations to program elements with at least one critical internal quality attribute. Furthermore, in 65% of the cases, the related internal quality attributes are improved, and the remaining 35% operations keep the quality attributes unaffected. Also, whenever pure refactoring operations are applied (*root-canal refactoring*), we confirm that internal quality attributes are either frequently improved or at least not worsened. Finally, while refactoring operations often reach other specific aims, such as adding a new feature or fixing a bug, 55% of these operations improve internal quality attributes, with only 10% of the quality decline.

As future work, we intend to reflect upon our findings in order to improve refactoring tools. For instance, one could think of a recommender system that ranks refactoring opportunities in terms of the structural criticality of the code elements in a program. Another direction of future work consists of conducting other studies using software systems written in different programming languages, such as C++ or C#. Finally, it is interesting to understand the effect of refactoring operations in proprietary software systems. Our present study focused only on the analysis of popular open source projects, which may have different structural degradation behaviors as compared to proprietary software systems.

REFERENCES

[1] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.* 107 (2015), 1–14.

[2] James Bieman and Byung-Kyoo Kang. 1995. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Software Engineering Notes*, Vol. 20. 259–262.

[3] James Bieman and Linda Ott. 1994. Measuring functional cohesion. *IEEE Trans. Softw. Eng.* 20, 8 (1994), 644–657.

[4] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. Does refactoring improve software structural quality?. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES)*. 73–82.

[5] Shyam Chidamber and Chris Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (1994), 476–493.

[6] Bart Du Bois, Serge Demeyer, and Jan Verelst. 2004. Refactoring-improving coupling and cohesion of existing code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*. 144–151.

[7] Bart Du Bois and Tom Mens. 2003. Describing the impact of refactoring on internal program quality. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), co-located with 19th ICSM*. 37–48.

[8] Chávez et al. 2017. Research website. (2017). https://alexchavezlop.github.io/refactoring-and-internal-attributes/

[9] Martin Fowler. 1999. *Refactoring*. Addison-Wesley Professional.

[10] Martin Fowler. 2013. Catalog of Refactorings. (2013). http://refactoring.com/catalog/

[11] Frank Grubbs. 1969. Procedures for detecting outlying observations in samples. *Technometrics* 11, 1 (1969), 1–21.

[12] Brian Henderson-Sellers. 1995. *Object-oriented metrics*. Prentice-Hall, Inc.

[13] Kerievsky Joshua. 2005. Refactoring to Patterns. (2005).

[14] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM)*. 576–585.

[15] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice*. Springer Science & Business Media.

[16] Wei Li and Sallie Henry. 1993. Object-oriented metrics that predict maintainability. *J. Syst. Softw.* 23, 2 (1993), 111–122.

[17] Yixun Liu, Denys Poshyvanyk, Rudolf Ferenc, Tibor Gyimóthy, and Nikos Chrisochoides. 2009. Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*. 233–242.

[18] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics*. Prentice Hall.

[19] Ruchika Malhotra. 2016. *Empirical research in software engineering*. CRC Press.

[20] Thomas McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 4 (1976), 308–320.

[21] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139.

[22] David Moore, William Notz, and Michael Fligner. 2015. *The basic practice of statistics*. W. H. Freeman.

[23] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. 2012. How we refactor, and how we know it. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 5–18.

[24] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*. 858–870.

[25] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Proceedings of the 23rd Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. 132–146.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.