Toward Understanding the Impact of Refactoring on Program Comprehension

Giulia Sellitto,1 Emanuele Iannone,1 Zadia Codabux,2 Valentina Lenarduzzi,3 Andrea De Lucia,1 Fabio Palomba,1 Filomena Ferrucci1
1Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
2University of Saskatchewan, Canada — 3University of Oulu, Finland
gisellitto@unisa.it, eiannone@unisa.it, zadiacodabux@iee.org, valentina.lenarduzzi@oulu.fi
adelucia@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

Abstract—Software refactoring is the activity associated with developers changing the internal structure of source code without modifying its external behavior. The literature argues that refactoring might have beneficial and harmful implications for software maintainability, primarily when performed without the support of automated tools. This paper continues the narrative on the effects of refactoring by exploring the dimension of program comprehension, namely the property that describes how easy it is for developers to understand source code. We start our investigation by assessing the basic unit of program comprehension, namely program readability. Next, we set up a large-scale empirical investigation – conducted on 156 open-source projects – to quantify the impact of refactoring on program readability. First, we mine refactoring data and, for each commit involving a refactoring, we compute (i) the amount and type(s) of refactoring actions performed and (ii) eight state-of-the-art program comprehension metrics. Afterwards, we build statistical models relating the various refactoring operations to each of the readability metrics considered to quantify the extent to which each refactoring impacts the metrics in either a positive or negative manner. The key results are that refactoring has a notable impact on most of the readability metrics considered.

Index Terms—Software Refactoring; Program Comprehension; Program Readability; Empirical Software Engineering.

I. INTRODUCTION

In the late 1990s, Martin Fowler introduced the term “software refactoring”, defining it as developers’ activity to improve the quality attributes of source code without changing its external behavior [1]. By definition, refactoring is supposed to benefit various attributes concerned with source code quality and maintainability. For instance, let us consider the case of the Extract Class refactoring [1]. This is the operation recommended for the removal of the so-called God Class code smell [1], which refers to a large and poorly cohesive class that centralizes the behavior of a software system, thus heavily affecting change- and fault-proneness [2], [3]. The refactoring aims at (i) grouping together methods of the God Class that contribute to the implementation of similar responsibilities and (ii) extracting them by creating new, more cohesive classes [1]. An Extract Class is, therefore, able to improve the overall structure of source code, helping developers in maintaining smaller classes over the software evolution. Along the same line, other refactoring operations provide specific benefits for quality attributes like cohesion, coupling, complexity, and inheritance [4]–[7]. Nonetheless, theory is not always aligned to practice [8]. The software engineering research community has been investigating refactoring over the last decades under different perspectives [6], [9]–[11], providing (i) recommendation systems to help developers applying refactoring in practice [12]–[14], (ii) insights into the reasons pushing developers to refactor source code [5], [15]–[17], and (iii) evidence of the current barriers preventing refactoring in practice [18]–[21].

Lack of usable automated tools [20], poor confidence in the outcome of a refactoring [19], and the fear of compromising the socio-technical dynamics of the development team [22] represent vital challenges for practitioners. The most relevant consequence of these issues is that developers often apply refactoring in a step-wise manner and without the adoption of any tool [19], [24]. This is why previous research has shown that refactoring might be risky for maintainability: Bavota et al. [24] and Di Penta et al. [25] identified the refactoring operations that negatively affect fault-proneness, while Kim et al. [19] found that developers perceive refactoring as a non-preserving operation, i.e., it may lead to undesired consequences for both source code quality and effectiveness.

Recognizing the effort spent by the research community so far, we point out the lack of empirical investigations into the effect of refactoring on a crucial property for software development and evolution: program comprehension [26]–[28]. This relates to the ability of developers to understand the source code and its functionalities before any changes are undertaken [29]. On the one hand, previous research has shown that code comprehension represents a large portion of the cost of any software [30]–[32]. On the other hand, researchers have been investigating the negative consequences of poorly documented code or poorly designed code on change- and fault-proneness [33]–[37]. More recently, Ammerlaan et al. [38] conducted a controlled experiment with an industrial partner to assess how refactoring improved program comprehension when applied to small coding tasks, finding that it does not always increase the overall ability of developers to deal with newly refactored code. They concluded that refactoring may lower developer’s productivity in the short term in cases where the code reads differently from what developers have grown attached to.

This observation makes it clear that one of the critical elements that might affect a developer’s productivity after the
application of refactoring operations is program/code readability, that is, the property describing how well developers can read the source code [39]. Readability can be considered as the basic unit of program comprehension [40] and, if not preserved, may induce developers to waste more time/resources while evolving source code, other than increasing the risks connected to the introduction of defects [35].

Our Goal. We aim to look closer into the relation between refactoring and code readability, intending to discover the extent to which the application of different refactoring operations may induce variations in terms of code readability.

Our empirical study is conducted on 156 Open-Source Software (OSS). In the first place, we make use of RefactoringMiner [41] to mine the evolution history of those projects and identify the refactoring commits, namely the commits where developers applied one or more refactoring operations. We then compute a set of state-of-the-art program readability metrics defined by Scalabrino et al. [42] on the refactoring commits and on their previous commits: in this way, we could precisely quantify how much the refactoring operations modified the program readability metrics. Finally, after the mining software repository exercise, we fit several mixed-effect Multinomial Log-Linear models [43] to investigate how the amount and type of refactoring operations, along with other confounding factors, affect each of the program readability metrics considered. The key results of the study indicate that refactoring actions have a significant impact on program readability, highlighting that they are crucial for code comprehension and maintainability.

Structure of the paper. The related literature is discussed in Section II. Section III describes the empirical setting of the experiment by elaborating on the research questions, the data collection and analysis procedures. Sections IV and V report and discuss the results of the study, while Section VI provides a more comprehensive analysis of the implications that the study has for researchers and practitioners. In Section VII we discuss the possible threats to the validity of the study and how we mitigated them. Finally, Section VIII concludes the paper and outlines our future research directions.

II. RELATED WORK

This section gives an overview of studies which investigate the impact of refactoring and propose practices to increase code readability.

The Impact of Refactoring. The impact of refactoring on software systems has been thoroughly studied from different perspectives of software quality, including technical debt, code smells, software changes, and metrics, and the results have been conflicting. We report some studies herein.

Chaparro et al. [44] used a tool, RIPE, to study the effect of 12 refactoring operations on 11 quality metrics in 15 OSS, and successfully predicted 38% of the metrics with a low deviation from the actual metric value. Kim et al. [19] conducted a large empirical study at Microsoft to investigate the benefits of refactoring. They reported that developers observed improved readability and maintainability, defect reduction, and features being easier to add, among others, following refactoring. However, they also pointed out that refactoring introduced regression bugs, caused build breaks, and increased testing costs, among others. The relationship between code changes and (28 different types of) refactoring was also explored in three OSS (APACHE ANT, ARGOUML, and APACHE XERCES) by Palomba et al. [16] Duplicated code was the motivation for refactoring. The study reported that refactoring targeting program comprehension and maintainability was mostly applied during bug fixing activities. However, when implementing new features, the refactoring activities pertaining to improving code cohesion were applied. Bavota et al. [24] investigated the impact of refactoring on defects. While generally, refactoring does not induce defects, specific refactoring types such as Pull Up Method and Extract Subclass tend to introduce defects. Cedrim et al. [45] conducted a study on the version histories of 23 projects to explore how ten different refactoring types and a total of 16,500+ refactoring instances affect 13 types of code smells. They reported that only 9.7% refactoring reduced smells while 33% introduced new ones. Tavares et al. [46] also analyzed the impact of ten code smells on refactoring, more specifically, Replace Type refactoring and Move Method. They conducted the study in seven open-source Java systems from the QUALITAS CORPUS [47]. They had mixed results. They observed that refactoring decrease, increase, or have a neutral impact on the smells. They also found that some smells are introduced or removed by these two refactoring types. Tufano et al. [48] noted that refactoring removes a low number (9%) of code smells during their empirical study of 200 OSS from the ANDROID, APACHE and ECLIPSE ecosystems. Fontana et al. [49] and Eposhi et al. [50] focused on the effect of refactoring on the software architecture and design respectively; the former study [49] explored how refactoring impacted architectural smells and anti-patterns. The refactoring activities removed some smells and anti-patterns but also introduced new ones; the latter study [50] observed that refactored classes had more design issues, more specifically, the classes had code smells and were more coupled and complex compared to other classes in the two OSS. Abid et al. conducted an empirical study, followed by a survey, to investigate the impact of refactoring on security in 30 OSS [51] and they concluded that some refactoring types positively impacted security while others negatively did. The effect of refactoring on security has also been explored by other studies [52–55].

Program Comprehension and Code Readability. Code readability refers to how easy a text is to understand and can be considered as the basic unit of program comprehension, a paramount attribute for maintainability and, subsequently, code quality [39], [56]. Over the years, studies have investigated various ways to improve code readability and program comprehension. The impact of Java coding practices on readability
was investigated using an online survey with students and practitioners. Participants were shown code snippets and had to comment on them. The study showed that while some practices promoted readability, others seemed to have no effect and one even had a negative impact. Scalabrino et al. proposed the use of textual features based on source code lexicon analysis to increase code readability. More than 5,000 people manually evaluated the features of code snippets and concluded that their approach improved readability. Other models have been proposed in the literature to increase code readability. The impact of looping and nesting on readability was assessed by conducting an experiment with 275 participants on 32 Java methods. They found that reduced nesting increased readability and understanding but avoiding looping (do-while loops) increased readability, but for simple and short loops only. However, do-while loops are easier to understand compared to while statements. Plantadosi et al. explored the effect of code evolution on code readability at the file level in 25 OSS. They concluded that only a very small number of commits impact the readability, and big commits that introduce new code can decrease readability. The readability of test cases was explored by Grano et al. They reported that test cases are generally less readable than code. They also found that the readability of manually generated test cases (compared to the readability of code, for instance) is not a priority for developers, and the automated test cases are even less readable than the manual ones. The effect of colors on program comprehension and readability was also studied. Rambally et al. found that using different color codes for various statements helped with readability, while Feigenspan et al. reported that background color to highlight source code improve the comprehension.

Reflecting on Previous Work and Our Contribution. Program comprehension and, more specifically, code readability and the impact of refactoring have been studied extensively on their own under different perspectives. However, the intersection of both, namely, the impact of refactoring on code readability and program comprehension is novel. To study the phenomenon, we conduct a large-scale study, analyzing 96,268 commits from 156 OSS and measure code readability using eight state-of-the-art metrics.

III. RESEARCH METHODOLOGY

The goal of this study is to assess the relation between refactoring and program readability, with the purpose of understanding how refactoring actions applied by developers impact the code quality and maintainability in terms of readability. The perspective is of both researchers and practitioners: the former are interested in understanding which additional support developers would require when performing refactoring; the latter are interested in evaluating the potential consequences of refactoring on source code dependability. The context is open source software (OSS). Our study was designed based on the guidelines proposed by Runeson and Host. Based on the aforementioned goal, we defined our first Research Question (RQ).

Q RQ1. To what extent do refactoring operations affect program readability?

Our study is not limited to establish links between refactorings and the overall code readability, but also wants to identify a subset of refactorings that improve the readability of the refactored code, while also recognizing a subset of “dangerous” refactorings that undermine the comprehensibility by reducing the readability. Thus, we ask:

Q RQ2. Which types of refactoring operations show clear positive or negative impacts on program readability?

We provide answers to these two RQs in Sections and Context. The context of the study is composed of 365 Open Source Software (OSS), and in particular, their change history information. The projects are contained in an online platform called PANDORA that continuously collects data from December 2020 investigating the code quality in terms of code violations, technical debt, and code build-stability. We selected this platform to enable further extension of this work, considering different factors that can affect program readability, such as code quality. Projects details are included in the online appendix of our study.

Mining Refactoring Data. We mined the entire change history of the considered projects to identify commits where developers applied refactoring – we will call these refactoring commits. To this aim, we ran version 2.1 of REFACTORINGMINER on each source code change. REFACTORINGMINER is a publicly available tool that can detect 81 distinct types of refactoring operations through the analysis of how the Abstract Syntax Trees of classes and methods have changed concerning the previous commits. The output of REFACTORINGMINER is formatted as a JSON file, reporting, for each commit, the set of refactoring operations applied, as well as the specific code snippets that were involved. Despite the existence of alternative refactoring detectors (e.g., REFDIFF), we opted for REFACTORINGMINER since it is publicly available and has a detection accuracy close to 100% compared to the performance of other detectors.

Among the 81 refactoring types supported, we discarded the Change Package and Move Source Folder refactorings as they only provide marginal (or even no) impact on the syntactic structure of the source code. It is worth noting that REFACTORINGMINER not only detects only a portion of the traditional refactoring classified in the original Fowler’s catalog, but also detects additional types that cannot be mapped to the original catalog, such as the composite refactoring Move and Inline Method. In this regard, we performed a mapping between the detected refactoring types with the ones in the

1 http://sqa.rd.tuni.fi/superset/dashboard/1
2 https://github.com/tsantalis/RefactoringMiner
The goal of this study was to assess the impact of refactoring on program comprehension, expressed in terms of code readability. To do this, we needed to measure the variation caused by refactoring operations in the readability level of the considered programs. We decided to measure the readability by using the eight metrics defined by Scalabrino et al. [35], which are based on textual properties of the source code, described in Table I. Two reasons mainly drove the choice of these metrics. First, several studies highlighted that textual features are significant descriptors in the evaluation of code comprehension and, therefore, are meaningful indicators of the overall readability level of source code. Second, Scalabrino et al. demonstrated that their newly-defined metrics are indeed a proxy of the actual readability perceived by developers [35]. In other words, the considered metrics are suitable to quantitatively assess the readability of source code and qualitatively perceived as relevant by practitioners. For two out of the eight metrics, i.e., Number of Concepts (NOC) and Number of Meanings (NM), having low values would be desirable in order to have higher readability. These two metrics represent the heterogeneity of the topics—detected through clustering—and the polysemy level of the terms used in the program’s source code, respectively. Therefore, code snippets with many topics and polysemous terms should be avoided to increase the overall readability. On the contrary, high Textual Coherence (TC) and Narrow Meaning Identifiers (NMI) values would represent positive traits of the program because they would denote cohesion among syntactic blocks of a method and the ease of interpretation of the terms used, respectively. Similarly, the Comments and Identifiers Consistency (CIC and CICsyn) metrics measure the coherence between the terms used in methods and the ones appearing in the comments that accompanies them (e.g., Javadoc comments). The Comments Readability (CR) metric is based on the Flesch-Kincaid index [69], which is commonly used to assess the readability of natural language texts. Thus, a higher index, which represent easy-to-read text, should always be preferred when writing code comments. Also, the Identifier Terms in Dictionary (ITID) metric relies on natural language: it expresses the percentage of identifiers used in the code that are also part of the English dictionary and this has been shown to be perceived as beneficial [70].

To extract the values of the readability metrics, we applied the following methodology. First, for each refactoring commit detected by RefactoringMiner, we mined the source code of the modified files using PyDriller [71], a Python framework for mining Git repositories that allowed us to obtain the two versions of each file touched by the commit, i.e., the version before and after the commit containing the refactoring. Then, we used the tool [7] by Scalabrino et al. [35] to obtain the values of the readability metrics of these two versions of each file. Afterwards, we computed the delta (Δ) of each readability metric to establish whether there was an increase, a decrease, or no variation in the metric values caused by the actions carried out in the refactoring commit. Since our study sets the granularity at the commit level, we collected all the deltas obtained and aggregated together using the mean operator—following the same aggregation used by the tool to compute the metrics for files. It is worth pointing out that

### Table I

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
<th>Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composing Methods</td>
<td>Re-organize the way methods are composed, such as streamlining their logic or removing unneeded parts.</td>
<td>Extract/Inline Method, Extract/Inline/Split/Merge/Rename Variable, Change Variable Type.</td>
</tr>
<tr>
<td>Moving Features between Objects Organizing Data</td>
<td>Re-organize the distribution of functionalities and data among classes.</td>
<td>Extract/Move/Rename Class, Move Method, Move Attribute.</td>
</tr>
<tr>
<td>Generalizing Data</td>
<td>Re-organize the way data is managed inside a class.</td>
<td>Split/Merge/Replace/Rename Attribute, Change Attribute Type, Replace Variable With Attribute.</td>
</tr>
<tr>
<td>Methods Calls</td>
<td>Simplify the interactions between classes by making the methods easier to call and understand.</td>
<td>Extract Superclass, Extract Subclass, Extract Interface, Pull Up/Push Down Attribute, Pull Up/Push Down Method.</td>
</tr>
<tr>
<td>Dealing with Generalization Others</td>
<td>Moving functionalities along class inheritance hierarchy.</td>
<td>Move And Rename Attribute, Move And Inline Method, Move And Rename Class, Move And Rename Method, Extract And Move Method, Add/Modify/Remove (Class/Attribute/Method/Parameter/Variable) Annotation, Add/Change/Remove Thrown Exception Type, Change Package, Move Source Folder.</td>
</tr>
</tbody>
</table>

To extract the values of the readability metrics, we applied the following methodology. First, for each refactoring commit detected by RefactoringMiner, we mined the source code of the modified files using PyDriller [71], a Python framework for mining Git repositories that allowed us to obtain the two versions of each file touched by the commit, i.e., the version before and after the commit containing the refactoring. Then, we used the tool [7] by Scalabrino et al. [35] to obtain the values of the readability metrics of these two versions of each file. Afterwards, we computed the delta (Δ) of each readability metric to establish whether there was an increase, a decrease, or no variation in the metric values caused by the actions carried out in the refactoring commit. Since our study sets the granularity at the commit level, we collected all the deltas obtained and aggregated together using the mean operator—following the same aggregation used by the tool to compute the metrics for files. It is worth pointing out that
the Textual Coherence (TC) metric fails if the given code snippet cannot be parsed or if it is made only of a single syntactic block (e.g., a method without any branches). As a consequence, the majority of the commits in our context did not receive a valid TC value, leading us not to consider it.

**Mining Control Metrics.** Our aim was to determine if and to what extent refactoring operations impact the readability of source code, so we collected data describing the refactoring modifications performed in the commits and the resulting variations in the readability metrics. However, we recognize that the fluctuations in the readability indicators are not necessarily caused by refactoring in the narrowest sense but can be due to the other changes applied in the commit. Therefore, we exploited PYDRILLER to mine eight control metrics. The first set of control metrics is intended to describe the overall magnitude of the commit. We collected the number of Added Lines (AL), Deleted Lines (DL), and the number of Changed Methods (CM) for each JAVA file involved in a commit. These three metrics alone can describe the commit in general, but they do not provide adequate indications into the actual size of the refactoring. Hence, we defined a heuristic to count the number of lines of code that were added or deleted because they were involved in a refactoring operation, which we named Refactoring Added Lines (RAL) and Refactoring Deleted Lines (RDL), respectively. Specifically, we used PYDRILLER to extract the DIFF, which we related with the code snippets that REFACTORINGMINER pointed out to be part of a refactoring operation. Thus, we computed the intersection between these two sets, so that we could obtain the lines of code added or removed as part of a refactoring operation. In this way, we gained information on the size of the refactoring in the strict sense. The third and last set of control metrics is motivated by the fact that readability may depend on other aspects related to the files that were subject to the change. Therefore, for each JAVA file modified in the commit, we mined three descriptors that could summarize the overall structure of the source code, i.e., Lines of Code (LOC), Cyclomatic Complexity (CC) [72], and Token Count (TOK) [73]. All these metrics were computed at the file level, hence we aggregated their values using the mean operator to bring them at the commit level.

**Data Analysis.** After collecting the data required to address our research questions, we proceeded with the analysis that relates the refactorings with variations in the readability profile of the modified code. In this respect, we were only interested in establishing whether commits containing refactoring operations have either positive, negative or no effect in terms of the considered readability metrics. For this reason, we remapped the \( \Delta \) values to categories that could suit our needs. If a readability metric resulted in \( \Delta > 0 \) in one of the refactoring commits we analyzed, this variation was mapped to the category “Increased”—i.e., the refactoring commit caused an increase in that metric. Similarly, if a readability metric had a \( \Delta < 0 \), it was converted to the category “Decreased”. Otherwise, it was converted to “Stable.” Afterwards, to address our RQs, we built eight different statistical models, one per readability metric, acting as our dependent variables. Then, we used the number of refactoring operators and the control metrics as the independent variables. Since our dependent variables had been made categorical, we fit a mixed-effect Multinomial Log-Linear model [43], a classification method that can generalize logistic regression to multiclass problems, thus fitting our case. We set the “Stable” category for being the model’s baseline, as this type of model requires a reference level to provide the prediction probabilities for the classes “Increased” and “Decreased”. The eight models were built using the R statistical toolkit exploiting the `multinom` package available in nnet\(^4\). Before fitting the models, we took the issue of multicollinearity into account—arising when two or more independent variables are highly correlated—causing biases in the predictive capabilities of the model, and subsequently, in the results interpretation. In this respect, since the data are not normally distributed, we computed Spearman’s rank correlation between all possible pairs of independent variables, with the aim of determining the existence of strongly correlated pairs (i.e., variables for which the Spearman’s \( \rho^2 > 0.6 \)). This analysis discovered strong correlation among Lines of Code (LOC), Cyclomatic Complexity (CC) and Token Count (TOK) control metrics, requiring us to discard two of them. We removed CC and TOK as we preferred keeping the most simple metric (LOC) in terms of explainability.

As for the interpretation of the results, it is worth noting that the models’ logit coefficients are relative to the baseline category “Stable”, indicating how the independent variables affect the probability of the dependent variable being classified as “Increased”, “Decreased” or “Stable”. In this sense, each model provides two distinct logit coefficients per independent variable. For instance, a refactoring type \( r_i \) with a logit coefficient of -1.50 when comparing the “Increased” category with the baseline (“Stable”) implies that a one-unit increase of \( r_i \) leads to a decrease of the chances of the dependent variable being classified as “Increased”, hence having higher probability of remaining unchanged. In addition, we also computed the Odds Ratios (OR) [75], given by the exponentiation of the logit coefficients. The Odds Ratio facilitates the interpretation of the results: it represents the probability of being classified to the given class (“Increased” or “Decreased”) over the probability of being classified as the reference class (“Stable”). As an example, a refactoring type having Odds Ratio > 1 for the class “Decreased” means that a commit having several instances of that refactoring is more likely to cause a decrease to the dependent variable. A refactoring type with both Odds Ratios > 1 implies that the refactoring type varies the value of the dependent variable, but without any clear distinction between an increase or a decrease.

**Verifiability and Replicability.** In order to allow our study to be verified and replicated, we have published the complete raw data, along with the data collection and analysis scripts in the online appendix [64].

\[^4\]https://cran.r-project.org/web/packages/nnet/nnet.pdf
### Table II

<table>
<thead>
<tr>
<th>Readability Metric</th>
<th>Abbrev.</th>
<th>Description</th>
<th>Desirable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments and Identifiers Consistency</td>
<td>CIC</td>
<td>Overlap between the terms used in method comments and the ones in the method bodies.</td>
<td>High</td>
</tr>
<tr>
<td>Comments and Identifiers Consistency - Synonym</td>
<td>CIC$_{syn}$</td>
<td>CIC metric extended considering synonym terms.</td>
<td>High</td>
</tr>
<tr>
<td>Identifier Terms in Dictionary</td>
<td>ITID</td>
<td>Percentage of identifiers used in the code that are also part of the English dictionary.</td>
<td>High</td>
</tr>
<tr>
<td>Narrow Meaning Identifiers</td>
<td>NMI</td>
<td>Sum of the particularity of the identifiers.</td>
<td>High</td>
</tr>
<tr>
<td>Comments Readability</td>
<td>CR</td>
<td>Flesch-Kincaid reading-ease score (FRES) of the comments linked to methods.</td>
<td>High</td>
</tr>
<tr>
<td>Number of Meanings</td>
<td>NM</td>
<td>Polysemy level of the terms appearing in the methods bodies.</td>
<td>Low</td>
</tr>
<tr>
<td>Textual Coherence</td>
<td>TC</td>
<td>Overlap between the terms used in the pairs of syntactic blocks.</td>
<td>High</td>
</tr>
<tr>
<td>Number of Concepts</td>
<td>NOC</td>
<td>Number of topics detected among the statements.</td>
<td>Low</td>
</tr>
<tr>
<td>Number of Concepts - Normalized</td>
<td>NOC$_{norm}$</td>
<td>Number of topics detected among the statements normalized on the number of statements.</td>
<td>Low</td>
</tr>
</tbody>
</table>

### IV. Analysis of the Results

In this section, we report our results. First, we provide a high-level overview of the results, and then we delve into each refactoring group described in Table II.

**Overall Results.** We analyzed 365 OSS. However, the readability metrics collection has only been successful for 156 projects, with a change history composed of 96,268 commits.

By looking at the Odds Ratios obtained (available in our online appendix [64]), we can see that only in very few cases (< 6%) the predictors were not statistically significant ($p < \alpha = 0.05$). Thus, in these cases we were not able to use their Odds Ratios to derive any statistically relevant conclusion. Half of these cases occurred in the model concerning the Comments Readability metric, with 17 out of 67 predictors being above the given significance level in that model. This finding was somehow expected, as the Comments Readability is given by the Flesch-Kincaid index computed on the comments linked to class methods, meaning that it is not related to any structural aspects of the source code. Since the considered refactoring types are only concerned with the syntactic part of the modified code, there are no changes affecting the comments alone. Moreover, the predictors exhibiting a statistical significance, both have their Odds Ratios below 1, indicating a higher likelihood to keep the Comments Readability unchanged (i.e., “Stable” category) as the number of those refactorings increase. From another point of view, Reorder Parameter refactoring turned out not to be significant in the majority of the cases (namely, 5 out of 8 models). This was also not unexpected: Reorder Parameter, despite affecting structural aspects of the source code, is limited to swapping the method’s parameters in the signature, without changing the terms found. Indeed, the readability metrics considered in this study are not influenced by the way the terms are arranged within a sentence, thus explaining the low significance. Curiously, Reorder Parameter refactoring has a moderate impact (i.e., both Odds Ratios > 2) on the Comments Readability metric. This can be explained by the fact that reordering the methods’ parameters has an impact on their documentation comment, causing variations to the Flesch-Kincaid index. Generally speaking, the considered control metrics either have no impact at all, e.g., Lines of Code having Odds Ratios equals to 1, or high likelihood to keep the dependent variable unchanged. There were only some cases in which these metrics had an impact, such as the Added lines number being related to an increase in the Number of Concepts metric, while the Deleted Lines number related to a decrease of the same metric. This is not surprising as the Number of Concepts metric is strongly influenced by the size of files, i.e., the more the Lines of Code, the higher the detected topics number among the statements.

Finding 1. The isolated application of certain refactorings may not affect the program readability, but other collateral code-changing activities may do.

Composing Methods Refactoring. From this group, eight refactoring types were involved in the building the models. The majority of these are characterized by both Odds Ratios being greater than 1, indicating their capability to cause variations in the readability of a file, e.g., the case of the Inline Method refactoring. This, however, does not allow us distinguish clearly whether these variations are actually increasing or decreasing. To some extent, the Extract Method refactoring behaves in a similar manner to its inverse Inline Method—i.e., causes variations to many metrics—but for the Comments and Identifiers Consistency metric, there is a moderate increase, i.e., Odds Ratio > 2 for the “Increased” category, while Odds Ratio < 1 for the “Decreased” one. In other words, increasing the number of Extract Methods applied in a commit...
by one unit causes the probability of having a readability improvement, in terms of the consistency between identifiers and comments, to double with respect to the probability of remaining unmodified. This could be explained by the nature of the Extract Method refactoring itself. An Extract Method is applied on a method \( M \), from which we want to extract and move a set of statements that do not fit the responsibilities of \( M \). For instance, a method `sendToServer()` that both sends some data to a server and prints some information to the standard output, could be refactored by removing the logging logic into a dedicated method, e.g., `log()` which is then referenced by the method `sendToServer()`. Subsequently, such a refactoring moves a set of identifiers and their synonyms from a method in which they are considered “inconsistent” to a method where they are more consistent.

The Split Variable refactoring (a.k.a. Split Temporary Variable \([1]\)) exhibits strong positive effects on multiple readability aspects. This refactoring consists of the addition of new local variables to host the output of expressions instead of using a single variable to host the results of different expressions. In this way, each variable contains the value of a single expression, increasing the cohesion at the granularity level of a variable. The only case in which the readability worsens is with the Number of Concepts dependent variable, which is computed by clustering the Lines of Code according to the terms they share. In this respect, the removal of shared variables reduces the overlap among the lines, increasing the number of concepts, which translates into an increased reading difficulty. However, the inverse refactoring Merge Variable (a.k.a. Inline Temp \([1]\)) was expected to have an opposite effect like the Split Variable; however, it only causes generic variations, with the exception for the Comments and Identifiers Consistency metric, which has a negative effect on readability. This is in line with the definition of Comments and Identifiers Consistency, as the Merge Variable refactoring is meant to reduce the number of variables in a method, so likely increasing the discrepancies between the code and the documentation comment.

### Finding 2. Re-organizing the source code with the goal of creating more cohesive components, e.g., methods, has a positive impact on the readability, especially in terms of coherence between identifiers and documentation comments.

### Finding 3. The refactorings belonging to the category ‘Moving Features between Objects’ determine variations in terms of readability but do not show any clear indication of their positive or negative impact, most likely because the considered metrics cannot properly measure the overall readability of multiple classes together.

### Finding 4. Refactorings aiming at re-organizing data within classes do not seem to have a clear impact on the readability. Removing unnecessary class fields has positive effects, but that depends on how the refactorings are applied.

### Simplifying Method Calls Refactorings. The Split Parameter refactoring decreases the value of Number of Meanings, indicating a readability improvement as it reduces the ambiguity of a parameter by splitting it into two less ambiguous ones. The opposite refactoring Merge Parameter, instead, provides an increase in Comments and Identifiers Consistency (CIC) and Identifier Terms in Dictionary (ITID) while decreasing the \( \text{DIC}_{\text{syn}} \), i.e., the Comments and Identifiers Consistency computed also considering synonyms, so showing mixed effects. Adding or removing parameters to a method signature causes variations without any specific trend. The Rename Parameter refactoring is related to an increase in the ITID, suggesting that developers apply this operation to clarify the names of the parameters, opting for English terms. This effect should have been identical for Rename Variable refactorings, but in those cases, it seems that there is no sufficient empirical evidence as there is only a generic variation.

### Organizing Data Refactorings. All refactorings belonging to this group have diversified impact on the readability. The Extract/Split/Merge Attribute refactorings cause medium-large variations of all the readability metrics, without any fixed trends, with the exceptions of Comments and Identifiers Consistency (CIC\(_{\text{syn}}\)) and Identifier Terms in Dictionary (ITID), for which instances of Merge Attribute refactorings increase and decrease their values, respectively. Such refactoring is applied to remove unneeded class fields, also causing a removal of inconsistent identifiers, and so explaining the increase in the CIC\(_{\text{syn}}\) and the decrease in the ITID. Likewise, the Replace Attribute refactoring—applied to remove unneeded class fields as well—increases CIC\(_{\text{syn}}\) and decreases ITID. It also decreases the ambiguity (Number of Meanings metric), while increasing the particularity of the terms (Narrow Meaning Identifiers metric). All things considered, this refactoring type has predominantly positive effects on the readability, but still has some drawbacks, likely depending on how it is applied. Indeed, if the replaced attribute has a name that is difficult to comprehend, then this would be an appropriate option to improve the readability. Conversely, if the replaced attribute already provided a good contribution to the overall readability, then the refactoring may damage the readability.

### Moving Features between Objects Refactorings. Despite being used for creating more cohesive classes, the Extract Class refactoring does not show any clear changes in the readability profile, but it only makes generic variations (both Odds Ratios < 2). This may be explained by the selected set of readability metrics. In fact, an Extract Class refactoring involves, by its nature, more than one class, while the readability metrics are meant to describe the readability of a single class, explaining the absence of clear variations.

The other types of refactorings in this group, i.e., Move Class/Attribute/Method and Rename Class, follow a similar trend to Extract Class. They impact the readability without clearly indicating whether it was an improvement or not.
Finding 5. The renaming of code elements is expected to improve readability, as it is done for that specific goal in mind. Yet, this does not always happen in practice, probably because the choice of the new name may affect metrics influenced by the vocabulary of the identifiers.

Dealing with Generalization Refactorings. The Extract Superclass refactoring seems to have negative effects on the readability. Indeed, both Number of Meanings (NM) and Normalized Number of Concepts (NOC\textsubscript{norm}) have higher likelihood to increase when there are instances of Extract Methods. This can be explained by the nature of this type of refactoring. The increase of NM may be caused by the addition of a new class (i.e., the extracted superclass) which is characterized, due to its nature, by the use of more generic and ambiguous terms. For the same reason, a superclass may contain a larger number of topics with respect to its subclasses, possibly explaining the increase in NOC\textsubscript{norm}. However, the decrease induced by Extract Subclass to NOC\textsubscript{norm} may be caused by the creation of the new subclass containing part of the topics of the refactored class, thus resulting in a lower mean value. The other types of refactorings that move class members along the hierarchy (i.e., Pull Up Attribute, Pull Up Method, Push Down Attribute, Push Down Method) generally cause variations in all the metrics. In particular, Pull Up Attribute has a contrasting effect. On the one hand, it improves the readability by increasing Comments Readability; on the other hand, it hinders it by increasing the Number of Concepts. Such differences may be explained by a lack of sufficient empirical evidence, hence requiring additional data to derive any relevant conclusions.

Finding 6. Moving the logic along the hierarchy structure creates more cohesive classes and shows better readability to some extent (e.g., fewer number of topics). However, the creation of superclasses inevitably has a negative impact as it makes use of more generic and ambiguous terms.

Other Refactorings. Refactorings in this group represent sequences of two distinct refactoring operations applied one after another. Applying Move and Rename Attribute refactoring has a moderate-strong impact on many readability metrics. On the one hand, the readability is improved because of the increases in Comments and Identifiers Consistency, Identifier Terms in Dictionary, and Narrow Meaning Identifiers, as well as the decreases in Number of Meanings and Number of Concepts. This implies that placing an attribute into a different class and changing its name have two key benefits. First, it improves the coherence of the attribute name within the methods it appears, likely because the moving of an attribute is done to place it in a more appropriate class; second, it happens to be clearer (i.e., using more English terms) and less ambiguous because of the renaming effect. However, there is still a worsening caused by an increase in the Normalized Number of Concepts (NOC\textsubscript{norm}), but this could be caused by the specific destination class, which has fewer lines of code—indeed, NOC\textsubscript{norm} is higher in code snippets exhibiting a smaller number of concepts.

The similar composite refactoring, Move and Rename method, provides a readability improvement by the means of an increase in the Narrow Meaning Identifiers metric, i.e., the new method name is less polysemous. All the other composite refactoring provide generic variations to the readability profile.

Finding 7. Noticeable changes in readability happen when multiple refactoring operations involving renaming are applied in sequence. For example, applying Rename Attribute refactoring alone may not be enough to improve the readability, so Move Attribute refactoring should be contextually done to provide a more suitable class. This is in line with previous findings that showed how the creation of more cohesive components is desirable for readability.

V. Further Analyses

The main findings from our research assert that refactoring operations have an impact on programs’ readability. In our study, we considered the variations in readability caused by refactoring commits, i.e., commits where at least one refactoring modification was performed. However, as previous work has highlighted [76], refactoring commits do not only consist of refactoring operations series, but also include other modifications in the code. In fact, refactoring is mostly performed along with functional modifications, and it is very uncommon to see a commit dedicated explicitly to refactoring. As a matter of fact, in the 156 projects considered, we found that only 3% of the commits containing refactoring operations were explicitly marked as refactoring commits by the developers. Therefore, we questioned our research findings, acknowledging that the variations in readability could be highly influenced by functional modifications in the code rather than being exclusively caused by refactoring. Hence, we decided to perform further analysis, only focusing on commits where most of the modifications were indeed refactoring operations. To do this, we defined a heuristic approach to determine the purity of a refactoring commit, i.e., the extent to which a commit can be considered a refactoring commit, given by the percentage of pure refactoring operations applied among all the modifications performed in the commit. For each refactoring commit reported by REFACTORMINER, we computed the purity metric as the sum of Refactoring Added Lines and Refactoring Deleted Lines over the sum of Added Lines and Deleted Lines. We re-conducted our analysis on three commit subsets, with purity greater than 25%, 50%, and 75%, to assess the validity of our findings. Clearly, the lower the purity threshold was set, the more similarity persisted between the original dataset of commits and the reduced dataset used for the additional analysis. Therefore, as expected, the results obtained in the supplementary analysis considering a 25% purity threshold were consistent with the main ones. Nevertheless, in the other two further investigations, i.e., considering 50% and 75% purity, the findings were coherent with the initial results.
obtained using the whole commit history without considering any purity threshold. Thus, we can conjecture that the presence of non-refactoring modifications in commits does not nullify the effect of refactoring actions on readability. Due to limited space, we do not report the complete results in the paper, but make all the data available in the online appendix [64].

VI. DISCUSSION AND LESSONS LEARNED

The analysis of the results revealed interesting insights that let us distill a number of lessons and/or implications on the impact of refactoring on code readability.

**Refactoring: Pleasure and Pain.** The first key aspect that we would like to point out is related to the overall effect that refactoring has on readability. Our study highlighted that refactoring operations may have a negative impact on readability, confirming that refactoring is a challenging activity that may not necessarily have a positive effect on code quality. Therefore, there is need for automated refactoring recommenders to consider the worthiness of refactoring actions, and help developers perform only quality-enhancing modifications.

**Lesson 1.** Refactoring may not always have a positive impact on code readability. Refactoring recommenders should consider the worthiness of the suggested modifications.

**It Only Tells Part of the Story.** The study that we performed consisted of a quantitative analysis on the impact of refactoring operations on programs’ readability, which can be considered as the basic unit of program comprehension. Although the readability metrics that we used have been validated as good predictors of the developers’ perception of code readability [35], we recognize that they only capture aspects related to vocabulary and lexicon of the code. As we discussed in Section IV, there are some examples (see Finding 6) in which the readability metrics suggest low code quality, while, actually, the maintainability and comprehensibility are high. In fact, code comprehension is concerned with how a developer reads code rather than what s/he reads, and involves more complex cognitive processes [27]. Researchers have been working on the definition of additional, complementary metrics to capture those cognitive processes. For instance, Arnaoudova et al. proposed the so-called linguistic antipatterns, i.e., inconsistencies among the naming, documentation, and implementation of entities [77], [78]. These antipatterns may make code and its purpose not very understandable, thereby confusing developers. It is important to notice that these inconsistencies may still be present in code with good readability values. Therefore, readability metrics alone only tell part of the story of developers’ perception on code comprehensibility. As part of our future research agenda, we plan to extend our empirical study and consider a larger amount of code comprehension metrics. Also, a qualitative analysis will provide a significant insight on the actual developers’ points of view. In this way, we will be able to understand the real impact of refactoring operations on program comprehension, by identifying in details the positive aspects and the problems practitioners face.

| Lesson 2. Readability metrics are only a part of the story. Further analyses on the impact of refactoring on other cognitive processes would provide more complete and concrete insights. |
| Lesson 3. Creating more cohesive components should be encouraged and suggested by refactoring recommenders. |
| Better Together. Our analysis revealed cases in which some refactoring types, e.g., *Rename Attribute*, have different effects depending on whether they are applied individually or in combination with other refactoring operations, e.g., *Move Attribute*. These findings encourage us to investigate the impact of composite refactoring on quality attributes. While some preliminary research efforts have been done [81], [82], we believe that further analyses would be beneficial to better understand the phenomenon and the implications it has for software maintainability. |
| Beware of What You Mine. As for the detection of refactoring operations among commits, during our work, we noticed that *RefactoringMiner* presents some noteworthy characteristics. In particular, it only recognizes a limited portion of the refactoring types defined in the literature [1], [33]. At the same time, it captures plenty of other kinds of elementary code changes, e.g., *Change Attribute Type*, and many composite operations, e.g., *Move and Inline Method*, that are not listed in the original catalog of refactorings [1]. These aspects have to be considered when building a dataset of refactoring commits, since not all the mined modifications are canonical refactoring actions. We decided to analyze anyways the "uncanonical" refactorings, grouping them in the *Others* category, and to then perform additional steps to assess the purity of the refactoring commits (see Sections III and V). Similarly, we encourage other researchers who might work with refactoring data to accurately ponder how to manage the supplementary data provided by *RefactoringMiner*. At the same time, we highlight that there is need for an automated tool that can properly point out canonical refactorings and distinguish them from other code changes. |
Lesson 5. There is a need for automated tools to detect canonical refactoring operations and discern them from other code changes.

VII. Threats to Validity

This section discusses the factors that might have influenced our results and how we mitigated them.

Construct Validity. In the context of our study, we employed automated tools to mine refactoring commits data and compute readability metrics. As for the refactoring commits data, we used REFACTORINGMINER to identify and locate refactoring operations applied in commits. Among the refactoring detectors available in the literature, the selection of REFACTORINGMINER was driven by the results reported by Tsantalis et al. [41], who showed a detection accuracy close to 100%. When it comes to the mining of commit descriptive data, i.e., metrics such as Lines of Code of the source code files touched by the commit, we used PyDRILLER, a Python framework for mining Git repositories. During the data mining phase, we excluded from our study the data related to non-source code files, such as readme and configuration files, because the primary goal of our study was to assess the impact of refactoring operations on code readability. Since we operated at the commit level, the major threat to the validity of our work is related to the difficulty of isolating refactoring operations from mere modifications in the source code. As discussed in Section VII we recognized that we could not exclusively impute to refactoring the variations observed in readability metrics values. Therefore, we defined the purity metric to mitigate this threat and re-conduct our analyses on those commits in which at least a certain percentage of modifications were due to refactoring. Although this custom-defined metric is based on a heuristic approach, we are convinced that it can acceptably summarize to what extent a commit can be considered a refactoring commit, i.e., if its purity is greater than a certain threshold, e.g., 75%. Still, we admit that there is a need for a more precise metric that could express the proportion of refactoring operations applied in the commit, and we encourage the community to propose a more robust approach. As for the readability metrics, we used Scalabrino et al.’s tool [35], which computes the readability level of a Java code snippet or class in terms of eight metrics. For verifiability, all the data are publicly available in the online appendix [64].

Internal Validity. When building statistical models, we selected confounding factors, i.e., the control metrics described in section VII to control our findings for aspects that might have explained the variations in the readability metrics better than refactoring operations. We recognize that there might be additional factors that were not considered in our study, and, as such, replications of our work would be desirable. Nonetheless, our manual follow-up analysis (see Section VII had the goal of further investigating the underlying reasons behind the relation between refactoring and readability, possibly mitigating threats to internal validity.

Conclusion Validity. Concerning the relation between treatment and outcome, a key threat is the statistical methods adopted to address our RQs. We opted for a Multinomial Logistic Linear statistical approach [43] as our problem was a multiclass problem involving both categorical and continuous independent variables. Before fitting the models, we also verified the presence of correlated variables that might lead to multicollinearity. An interesting finding that we observed is that the quantity of pure refactoring modified lines is not correlated with the total number of modified lines, highlighting that refactoring operations’ size is not strictly proportional to the overall size of the commit. We aggregated the descriptive metrics related to all the files in a commit by using the mean operator. While this choice could have been influenced by skewed distributions of metric values, we still opted for this aggregator instead of the median since it could provide an overall description of the commit. In the online appendix [64] we provide all the raw data, along with other aggregations, i.e., median and sum.

External Validity. Our study targeted 156 projects with a change history composed of 96,268 commits. We cannot exclude that different results could be obtained when considering systems of other ecosystems developed using different programming languages and with different maturity levels. Replications targeting a more extensive set of projects would be, therefore, desirable. Therefore, the online appendix [64] provides all the data and scripts to favor researchers interested in replicating our study in other contexts.

VIII. Conclusion

We analyzed the impact of refactoring on code readability and provided initial insights into the relation between refactoring and program comprehension. By mining refactoring data and readability metrics of 156 OSS, we found that refactoring does not always have a positive impact and, indeed, some of the considered refactorings demonstrated to negatively influence the readability of source code. Worthy of note are the refactoring actions aiming at increasing cohesion and the composite refactoring operations, which generally improved the overall readability of the code in a significant manner. Based on these findings, we identified several lessons learned for researchers and practitioners, which will also drive our future research agenda: we plan to further investigate the relation between refactoring and program comprehension, other than devising (semi-)automated recommenders that may find a compromise between multiple quality attributes, including readability and comprehensibility.

Acknowledgement

Zadia is partly supported by the Natural Sciences and Engineering Research Council of Canada, RGPIN-2021-04232 and DGECR-2021-00283. Fabio acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2 186090 (TED).