Toward Understanding the Impact of Refactoring on Program Comprehension

Giulia Sellitto,[1] Emanuele Iannone,[1] Zadia Codabux,[2] Valentina Lenarduzzi,[1]
Andrea De Lucia,[1] Fabio Palomba,[1] Filomena Ferrucci[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]University of Saskatchewan, Canada — [3]LUT University, Finland
gisellitto@unisa.it, eiannone@unisa.it, zadiacodabux@ieee.org, valentina.lenarduzzi@lut.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 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 large and poorly cohesive classes that centralize the behavior of a software system and that heavily affect 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 now 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], [23]. 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 of 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 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 enhancing 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 readability.

A. 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.

The impact of refactoring on code quality metrics was investigated by Chaparro et al. [44]. They used a tool, RIPE, to study the effect of 12 refactoring operations on 11 quality metrics in 15 OSS. RIPE 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) [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]. They concluded that some refactoring types positively impacted security while others were negatively correlated with security. The effect of refactoring on security has also been explored by other studies [52]–[55].

B. Code Readability

Code readability refers to how easy a text is to understand and is an essential attribute for maintainability and, subse-

quently, code quality [39], [56]. Over the years, studies have investigated various ways to improve readability.

The impact of Java coding practices on readability was investigated using an online survey with students and practitioners [57]. 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. [35], [42] 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 [39], [58]. The impact of looping and nesting on readability was assessed by conducting an experiment with 275 participants on 32 Java methods [40]. 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. Piantadosi et al. [59] 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. [60]. 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. [56] found that using different color codes for various statements helped with readability, while Feigenspan et al. [61] reported that background color to highlight source code improve the comprehension.

C. 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 but the intersection of both, namely, the impact of refactoring on code comprehension and readability 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. Our study was designed based on the guidelines proposed by Runeson and Host [62].

Based on the aforementioned goal, we defined our first Research Question (RQ).

> **RQ$_1$.** *To what extent do refactoring operations affect program readability?*

Our study does not limit establishing 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 identifying a subset of "dangerous" refactorings that undermine the comprehensibility by reducing the readability. Thus, we ask:

> **RQ$_2$.** *Which types of refactoring operations show clear positive or negative impacts on program readability?*

We provide answers to these two RQs in Sections IV and V.

A. Context

The *context* of the study is composed of 365 OSS, and in particular, their change history information. The projects are contained in an online platform called PANDORA[1] 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 our online appendix [63].

B. Data Collection

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 run version 2.1 of RefactoringMiner [41] on each source code change. RefactoringMiner[2] 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 [64]), we opted for RefactoringMiner since it is publicly available and has a detection accuracy close to 100% compared to the performance of other detectors [41].

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 [1], but also detects additional types that cannot be mapped

[1] http://sqa.rd.tuni.fi/superset/dashboard/1/

[2] https://github.com/tsantalis/RefactoringMiner

TABLE I
TYPES OF REFACTORINGS CONSIDERED IN THIS STUDY, MINED USING REFACTORINGMINER [41].

Group	Description	Refactorings
Composing Methods	Re-organize the way methods are composed, such as streamlining their logic or removing unneeded parts.	Extract/Inline Method, Extract/Inline/Split/Merge/Rename Variable, Change Variable Type.
Moving Features between Objects	Re-organize the distribution of functionalities and data among classes.	Extract/Move/Rename Class, Move Method, Move Attribute.
Organizing Data	Re-organize the way data is managed inside a class.	Extract/Split/Merge/Replace/Rename Attribute, Change Attribute Type, Replace Variable With Attribute.
Simplifying Method Calls	Simplify the interactions between classes by making the methods easier to call and understand.	Split/Merge/Add/Remove/Reorder/Rename Parameter, Parameterize Variable, Change Parameter Type, Change Method Access Modifier, Change Return Type, Rename Method.
Dealing with Generalization	Moving functionalities along class inheritance hierarchy.	Extract Superclass, Extract Subclass, Extract Interface, Pull Up/Push Down Attribute, Pull Up/Push Down Method.
Composite	Composite refactorings detected by REFACTORINGMINER.	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.

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 catalog, so that we could group them by the categories defined by Fowler [1], as described in Table I. We preferred keeping the "uncanonical" types—collected in the "Others"group—as they could provide valuable findings to our investigation. For each refactoring commit detected by the tool, we counted the number of occurrences of each refactoring type.

Mining Readability Metrics. The goal of this study is to assess the impact of refactoring on program 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 II. 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 [27], [65]–[67]. Second, Scalabrino et al. demonstrated that their newly-defined metrics are 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 CIC_{syn}) 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 [68], 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 [69].

To extract the values of the readability metrics, we followed the following methdology. First, for each refactoring commit detected by REFACTORINGMINER, we mined the source code of the modified files using PYDRILLER [70], 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 refactorings. Then, we used the tool[3] 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

[3]https://dibt.unimol.it/report/readability

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 is 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) [71], and Token Count (TOK) [72]. 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.

C. 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 Δ 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 has 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 have 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 models were built using the R statistical toolkit exploiting the `multinom` package available in the package `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 the Spearman's rank correlation [73] 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 LOC, CC and 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 model's 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) [74], given by the exponentiation of the logit coefficients. The ORs facilitates the interpretation of the results: the OR 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 $OR > 1$ for the class *"Decreased"* means that a commits having several instances of that refactoring is more likely to cause a decrease to the dependent variable. A refactoring type with both $ORs > 1$ implies that the refactoring type varies the value of the dependent variable, but without any clear distinction between an increase or a decrease.

[4]https://cran.r-project.org/web/packages/nnet/nnet.pdf

TABLE II
LIST OF READABILITY METRICS CONSIDERED IN THIS STUDY TO ANSWER OUR **RQ**S. THE COLUMN 'DESIRABLE' VALUE INDICATES THE VALUE THAT WOULD BENEFIT THE READABILITY.

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

D. 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 our online appendix [63].

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 I.

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 ORs obtained (available in our online appendix [63]), 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 ORs to derive any statistically relevant conclusion. Half of these cases occurred in the model concerning the CR metric (*Comments Readability*), with 17 out of 67 predictors being above the given significance level in that model. This finding was somehow expected, as the CR 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 is 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 ORs below 1, indicating a higher likelihood to keep the CR 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 ORs > 2) on the CR 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., LOC having ORs 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 (AL) being related to an increase in the *Number of Concepts* (NOC) metric, while the Deleted Lines number (DL) related to a decrease of the same metric. This is not surprising as the NOC metric is strongly influenced by the size of files, i.e., the more the LOC, 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 Refactorings. From this group, eight refactoring types were involved in the building the models. The majority of these are characterized by both ORs 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 CIC_{syn} metric, there is a moderate increase, i.e., $OR > 2$ for the

"Increased" category, while $OR < 1$ for the *"Decreased"* one. In other words, increasing the number of *Extract Method*s 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. This happened in the case of APACHE COMMONS COMPRESS at commit `794c20fe`[5], in which from `closeEntry()`, two methods were extracted: `drainCurrentEntryData()` and `getBytesInflated()`, both having dedicated documentation comments (i.e., JAVADOC) reflecting their content, explaining the increase in the CIC_{syn} metric. Curiously, the commit message states that *'a few 'extract method' refactorings to make code more readable [...]'*, indicating that the developer planned to apply this refactoring to improve the readability.

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 NOC variable, which represents the number of concepts detected by clustering the LOC 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 CIC metric, which has a negative effect on readability. This is in line with the definition of CIC, 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.

[5]https://github.com/apache/commons-compress/commit/794c20fe

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 ORs $<$ 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 the *Extract Class*. They impact the readability without clearly indicating whether it was an improvement or not.

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.

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 CIC_{syn} and 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_{syn} and the decrease in the ITID. Likewise, the *Replace Attribute* refactoring—applied to remove unneeded class fields as well—increases CIC_{syn} and decreases ITID. It also decreases the ambiguity (NM metric), while increasing the particularity of the terms (NMI 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.

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 depends on how the refactorings are applied.

Simplifying Method Calls Refactorings. The *Split Parameter* refactoring decreases the value of NM, 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 CIC and ITID while decreasing the CIC_{syn}, 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.

> **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 NM and NOC_{norm} have higher likelihood to increase when there are instances of *Extract Method*s. 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_{norm}. However, the decrease induced by *Extract Subclass* to NOC_{norm} may be caused by the creation of the new subclass containing part of the topics of the refactored class, resulting in a lower mean value.

The other types of refactorings that move class members along the hierarchy (i.e., *Pull Up Attribute*, *Push Down Attribute*, *Pull Up Method*, *Push Down Method*) generally cause variations in all the metrics. In particular, *Pull Up Attribute* has a contrasting effect. On one hand, it improved the readability by increasing CR; on the other hand, it hinders it by increasing the NOC. 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 show 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* refactorings has a moderate-strong impact on many readability metrics. On the one hand, the readability is improved because of the increases in CIC, ITID, and NMI, as well as the decreases in NM and NOC. 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 NOC_{norm}, but this could be caused by the specific destination class, which has fewer lines of code—indeed, NOC_{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 NMI 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 renamings 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 [75], 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 RefactoringMiner, we computed the *purity* metric as the sum of Refactoring Added Lines (RAL) and Refactoring Deleted Lines (RDL) over the sum of Added Lines (AL) and Deleted Lines (DL). 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 [63].

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 represents 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 [76], [77]. 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.

Cohesion Is the Key. Another important aspect emerging from the findings is the importance of creating cohesive components [12], [33], [78], [79]. Indeed, classes and methods with fewer responsibilities have lower number of topics and use terms with lesser ambiguity, translating into better readability. Moreover, a higher cohesion at the finest granularity level, i.e., variable, has positive effects too. While this finding was kind of expected, it still corroborates the need of considering code cohesion during refactoring tasks. As such, the results of the study promote and further encourage the definition of refactoring recommenders that explicitly consider cohesion as one of the targets to optimize.

> 👉 **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, we believe that further analyses would be beneficial to better understand the phenomenon and the implications it has for software maintainability [80], [81].

> 👉 **Lesson 4.** Refactorings applied in sequence show clearer results with respect to their individual application.

Beware of the Noise. As for the detection of refactoring operations among commits, during our work, we noticed that RefactoringMiner presents some limitations. In particular, it only recognizes a limited portion of the refactoring types defined in the literature [1], [82]. 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 really refactorings. These aspects can easily introduce noise in a dataset of *refactoring commits*, since not all the reported modifications are actually refactoring actions. Therefore, we encourage other researchers who might work with refactoring data to perform some preliminary steps of data cleaning, to trim out this noise. In addition, we highlight that there is need for an automated tool that can properly point out refactoring operations and distinguish them from simple code changes.

> 👉 **Lesson 5.** There is a need for automated tools to detect refactoring operations and discern them from simple 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 LOC 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 V, 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 8 metrics. For the sake of verifiability, we made all the data publicly available in our online appendix [63].

Internal Validity. When building statistical models, we selected confounding factors, i.e., the control metrics described in section III, 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 V) 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 [63] 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. In any case, in our online appendix [63], we made available 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, most of the refactoring operations considered do not significantly affect the readability of source code. Noticeable exceptions are the refactoring actions aiming at increasing cohesion and the composite refactoring operations, which generally increase the overall readability of the code. Based on these findings, we identified several lessons learned for researchers and practitioners, which will also drive our future research agenda: we plan to investigate further 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).

References

[1] F. Martin and B. Kent, "Refactoring: Improving the design of existing code," *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[2] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[3] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, Jun 2018.

[4] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and software technology*, vol. 51, no. 9, pp. 1319–1326, 2009.

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

[6] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[7] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, 2007, pp. 252–266.

[8] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *International Conference on Software Engineering*, 2004, pp. 273–281.

[9] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.

[10] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[11] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells—5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[12] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

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

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[15] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[16] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 176–185.

[17] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[18] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *International conference on Software engineering*, 2008, pp. 421–430.

[19] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[20] T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.

[21] C. Vassallo, F. Palomba, and H. C. Gall, "Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 564–568.

[22] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[23] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[24] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 104–113.

[25] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.

[26] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–37, 2014.

[27] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, 1987.

[28] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[29] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *International Conference on Program Comprehension (ICPC)*, 2018, pp. 286–28 610.

[30] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 665–670, 2003.

[31] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *International conference on software engineering*, vol. 2. IEEE, 2010, pp. 223–226.

[32] M. L. Nelson, "A survey of reverse engineering and program comprehension," *arXiv preprint cs/0503068*, 2005.

[33] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *European conference on software maintenance and reengineering*, 2011, pp. 181–190.

[34] S. Blinman and A. Cockburn, "Program comprehension: investigating the effects of naming style and documentation," in *International Conference Proceeding Series*, vol. 104. Citeseer, 2005, pp. 73–78.

[35] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, 2018.

[36] V. Singh, L. L. Pollock, W. Snipes, and N. A. Kraft, "A case study of program comprehension effort and technical debt estimations," in *International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–9.

[37] B. E. Teasley, "The effects of naming style and expertise on program comprehension," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 757–770, 1994.

[38] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 504–507.

[39] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[40] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, "An empirical study assessing source code readability in comprehension," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 513–523.

[41] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 483–494.

[42] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[43] H. Theil, "A multinomial extension of the linear logit model," *International economic review*, vol. 10, no. 3, pp. 251–259, 1969.

[44] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 456–460.

[45] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 465–475.

[46] C. Tavares, M. Bigonha, and E. Figueiredo, "Analyzing the impact of refactoring on bad smells," in *Brazilian Symposium on Software Engineering*, 2020, pp. 97–101.

[47] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 336–345.

[48] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[49] F. A. Fontana, R. Roveda, S. Vittori, A. Metelli, S. Saldarini, and F. Mazzei, "On evaluating the impact of the refactoring of architectural problems on software quality," in *Scientific Workshop XP2016*, 2016, pp. 1–8.

[50] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of design problems through refactorings: are we looking at the right symptoms?" in *International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 148–153.

[51] C. Abid, M. Kessentini, V. Alizadeh, M. Dhouadi, and R. Kazman, "How does refactoring impact security when improving quality? a security-aware refactoring approach," *IEEE Transactions on Software Engineering*, 2020.

[52] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "An empirical study to improve software security through the application of code refactoring," *Information and Software Technology*, vol. 96, pp. 112–125, 2018.

[53] S. Ghaith and M. Ó. Cinnéide, "Improving software security using search-based refactoring," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 121–135.

[54] B. Alshammari, C. Fidge, and D. Corney, "Assessing the impact of refactoring on security-critical object-oriented designs," in *Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 186–195.

[55] B. Alshammari, C. Fidge, and D. Corney, "Security assessment of code refactoring rules," in *National Workshop on Information Assurance Research*. VDE, 2012, pp. 1–10.

[56] G. K. Rambally, "The influence of color on program readability and comprehensibility," in *SIGCSE technical symposium on Computer science education*, 1986, pp. 173–181.

[57] R. M. dos Santos and M. A. Gerosa, "Impacts of coding practices on readability," in *Conference on Program Comprehension*, 2018, pp. 277–285.

[58] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Working conference on mining software repositories*, 2011, pp. 73–82.

[59] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Empirical Software Engineering*, vol. 25, no. 6, pp. 5374–5412, 2020.

[60] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–3483.

[61] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the# ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[62] R. Per and H. Martin, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, 2009.

[63] "Toward understanding the impact of refactoring on program comprehension: Appendix." [Online]. Available: https://figshare.com/s/9550c0f7fd458eb3699b

[64] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.

[65] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Koli Calling International Conference on Computing Education Research*, 2011, p. 1–9.

[66] L. Guerrouj, "Normalizing source code vocabulary to support program comprehension and software quality," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1385–1388.

[67] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: An fmri study," in *International Conference on Software Engineering (ICSE)*, 2021, pp. 524–536.

[68] R. F. Flesch, "A new readability yardstick." *The Journal of applied psychology*, vol. 32 3, pp. 221–33, 1948.

[69] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *ISSE*, vol. 3, pp. 303–318, 11 2007.

[70] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[71] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[72] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977.

[73] C. Spearman, "The proof and measurement of association between two things." 1961.

[74] J. M. Bland and D. G. Altman, "The odds ratio," *Bmj*, vol. 320, no. 7247, p. 1468, 2000.

[75] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[76] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 187–196.

[77] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: what they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, 01 2015.

[78] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012, automated Software Evolution. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121212001057

[79] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121210003195

[80] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Oliveira, M. Kim, and A. Oliveira, "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 186–197. [Online]. Available: https://doi.org/10.1145/3379597.3387477

[81] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. a. L. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca, M. Ribeiro, C. Barbosa, and D. Oliveira, "How does incomplete composite refactoring affect internal quality attributes?" in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 149–159. [Online]. Available: https://doi.org/10.1145/3387904.3389264

[82] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. USA: John Wiley and Sons, Inc., 1998.