# Improving Code: The (Mis)perception of Quality Metrics

Jevgenija Pantiuchina, Michele Lanza, Gabriele Bavota
*REVEAL @ Software Institute, Università della Svizzera italiana (USI), Lugano, Switzerland*
jevgenija.pantiuchina | michele.lanza | gabriele.bavota@usi.ch

***Abstract*—Code quality metrics are widely used to identify design flaws (*e.g.,* code smells) as well as to act as fitness functions for refactoring recommenders. Both these applications imply a strong assumption: quality metrics are able to assess code quality as *perceived* by developers. Indeed, code smell detectors and refactoring recommenders should be able to identify design flaws/recommend refactorings that are meaningful from the developer's point-of-view. While such an assumption might look reasonable, there is limited empirical evidence supporting it.**

**We aim at bridging this gap by empirically investigating whether quality metrics are able to capture code quality improvement as perceived by developers. While previous studies surveyed developers to investigate whether metrics align with their perception of code quality, we mine commits in which developers clearly state in the commit message their aim of improving one of four quality attributes: *cohesion*, *coupling*, *code readability*, and *code complexity*. Then, we use state-of-the-art metrics to assess the change brought by each of those commits to the specific quality attribute it targets. We found that, more often than not the considered quality metrics were not able to capture the quality improvement as perceived by developers (*e.g.,* the developer states "*improved the cohesion of class C*", but no quality metric captures such an improvement).**

***Index Terms*—code quality; metrics; empirical study**

## I. Introduction

Code quality metrics are at the core of many approaches supporting software development and maintenance tasks. They have been used to automatically detect code smells [1], [2], to recommend refactorings [3], [4], and to predict the code fault- and change-proneness [5], [6], [7]. Some of these applications assume that a strong link between code quality as assessed by metrics and as *perceived* by developers exists. For instance, many refactoring recommenders use metrics as fitness functions to identify sequences of refactorings able to maximize cohesion and minimize coupling (*e.g.,* [3], [8], [4]). The final users of these tools are software developers. This means that the recommended refactorings should be meaningful from a developer's point-of-view, and this is only possible if the metrics are actually able to capture cohesion and coupling as perceived by developers.

Previous studies only partially investigated this phenomenon. Revelle *et al.* [9] and Bavota *et al.* [10] surveyed developers to investigate whether their perception of code coupling aligns with the strength of coupling as assessed by quality metrics. Counsell *et al.* [11] performed a similar study, but focusing on cohesion. While the problem investigated in these studies is similar to the one we tackle, we adopt a different methodology and we investigate four code quality attributes (*i.e.,* cohesion, coupling, readability, and complexity).

Instead of surveying software developers, we analyzed real changes they implemented with the stated purpose of improving one of the four considered quality attributes.

We mined over 300M commits performed on GitHub and used a simple "keyword matching mechanism" to identify commit notes reporting one of the following four words: *cohesion*, *coupling*, *readability*, or *complexity*. Then, we excluded commits performed on non-Java systems, and manually analyzed the remaining ones with the goal of identifying those in which developers state in the commit note the intention to improve the corresponding quality attribute. Examples of those commits are: "*Removed* `write()` *method – higher cohesion*" and "*Refactoring* `TexasHoldEmHandFactory` *to reduce cyclomatic complexity.*".A final set of 1,282 commits was considered in our study.

For each of the selected commits $c$, we extracted the list of files $F$ impacted by $c$ before ($F_{before}$) and after ($F_{after}$) $c$'s changes. Then, we use quality metrics designed to assess the quality attribute $c$ aims at improving (*e.g.,* for class cohesion we exploit the Lack of Cohesion of Methods [12] and the Conceptual Cohesion of Classes [6]) to measure such attributes on $F_{before}$ and $F_{after}$. This allows us to investigate whether the quality improvement expected by the developer is also reflected in the metrics' values. Our results show that, more often than not, the studied quality metrics are not able to capture the quality improvements as perceived by the developers.

The contribution of our paper is threefold. First, we provide insights into better understanding the completeness of the considered quality metrics in capturing the quality improvements as perceived by the developers. This is an important step in understanding the practical implications of software measurement theory and identifying future directions. Second, we do not limit our results discussion to numbers and statistics, but report many qualitative examples allowing to better understand the causes behind cases of disagreement between quality metrics and developers' perception of code quality. Finally, the data used in our study is made publicly available [13] for replication purposes.

**Structure of the paper.** Section II presents the study design, while its results are reported in Section III. Section IV discusses the threats that could affect the validity of our results. Finally, Section VI concludes the paper after a discussion of the related literature (Section V).

## II. Study Design

The *goal* of the study is to investigate whether code quality improvements, as subjectively perceived by developers, can be objectively captured with code quality metrics. The *context* consists of 1,282 commits performed in Java systems, in which developers state their intention of improving a specific quality attribute. The study addresses the following research question:

**$RQ_1$:** *To what extent do quality metrics capture code quality improvement as perceived by developers?* Given a commit in which the developer explicitly states her goal to improve a specific quality attribute $Q$ (*e.g.,* cohesion) we use metrics aimed at measuring $Q$ (*e.g.,* the LCOM) to verify whether the improvement perceived by the developer is also reflected in the metrics' value. Note that by answering $RQ_1$ we will indirectly provide insights on the impact of refactoring operations on code quality as assessed by quality metrics. Indeed, in our experimental design, the commits in which developers claim to improve code quality can be seen as refactoring operations[1]. As we discuss in Section V, previous works already investigated the impact of refactoring on code quality [15], [16], [17], [18], but with a different focus and study design.

### A. Code Quality Attributes and Metrics

Given the general formulation of $RQ_1$, we need to clarify what "code quality" means in this context. We assess code quality at class level, and consider four quality attributes of classes with corresponding metrics to automatically measure them. In the following we describe each quality attribute as well as the metrics we selected for it. All metrics' implementation only work for the Java language. This impacted our selection of the commits subject of this study. For the computation of all metrics but the ones related to code readability, we used our own implementation. For readability, we relied on the original tools provided by the authors (details follow).

**Cohesion.** The first quality attribute we consider is class cohesion, assessing the degree to which the responsibilities implemented in a class belong together [19]. High cohesion is desirable, since it promotes the single responsibility principle, fostering code maintainability. We use two metrics to assess the cohesion of classes. The first is the Henderson-Sellers revised Lack of COhesion of Methods (LCOM) [12], considering the shared variables among the methods of a class as a proxy for its cohesion. The LCOM for a class $C$ is computed as:

$$LCOM(C) = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1 - m}$$

where $a$ is the number of $C$'s variables, $\mu(A_j)$ is the number of $C$'s methods accessing the variable $A_j$, and $m$ is the number of methods in $C$. The Henderson-Sellers revised LCOM addresses many of the limitations of the original LCOM [12] and it has the advantage of being normalized between 0 (highest cohesion) and 1 (lowest cohesion), thus easing its interpretation.

The second cohesion metric we use is the Conceptual Cohesion of Classes (C3) [6]. The C3 exploits Latent Semantic Indexing (LSI) [20] to compute the overlap of textual information in a class expressed in terms of textual similarity among methods. The C3 is computed as the average textual similarity between all pairs of methods in a class. The metric is defined in $[0 \dots 1]$, and higher values of C3 indicate higher class cohesion. In our implementation we apply preprocessing to the text representing the class methods in order to (i) remove English stop words and reserved Java keywords, (ii) stem words to their root form, and (iii) split identifiers in the source code based on CamelCase and the underscore separator.

**Coupling.** It is defined as the strength of the dependencies existing between classes [19]. Low coupling is desirable, since it helps in isolating changes. Also in this case we use two metrics to compute it. The first is the Coupling Between Object (CBO) [21], counting the number of dependencies a class has (*i.e.,* the number of other classes it depends on). The higher the CBO the higher the class coupling.

The second metric is the Response for a Class (RFC) [21], calculated as the number of distinct methods and constructors invoked by a class. The RFC can act as a proxy for complexity as well as for coupling. Since the definition of RFC includes methods called from outside of the class, we consider it as a coupling metric: the higher the RFC the higher the coupling.

**Code complexity.** Keeping the complexity of code low is one of the main goals of refactoring. We use the Weighted Methods per Class (WMC) [21] to assess class complexity. WMC for a given class is computed as the sum of the McCabe's cyclomatic complexity [22] of its methods. Being a direct metric, the higher WMC the higher the class complexity.

**Code readability.** Finally, we focus on code readability, a quality attribute that has recently attracted interest in the software engineering community, mostly due to the challenges in automatically measuring it. Readable code is clearly preferable, since it is supposed to foster code comprehensibility. To measure this quality attribute, we exploit two state-of-the-art metrics. The first one was presented by Buse and Weimer [23] (from now on we refer to this metric as B&W), and exploits source code structural aspects (*e.g.,* number of branches, loops etc.) to model the readability of code. We use the original implementation made available by the authors[2].

The second metric is the one proposed by Scalabrino *et al.* [24], and exploits a set of features based entirely on source code lexicon analysis (*e.g.,* consistency between source code and comments, specificity of the identifiers, textual coherence, comments readability, etc.). Their model was evaluated on a dataset composed by 200 Java snippets and the results indicated its accuracy in assessing code readability as perceived by developers. For computing this metric (from now on, $SRead$) we used the original implementation provided by the authors.

[1]Note that these refactorings may or may not fall in the catalogue of refactoring operations generally used in the literature [14].

[2]http://www.arrestedcomputing.com/readability

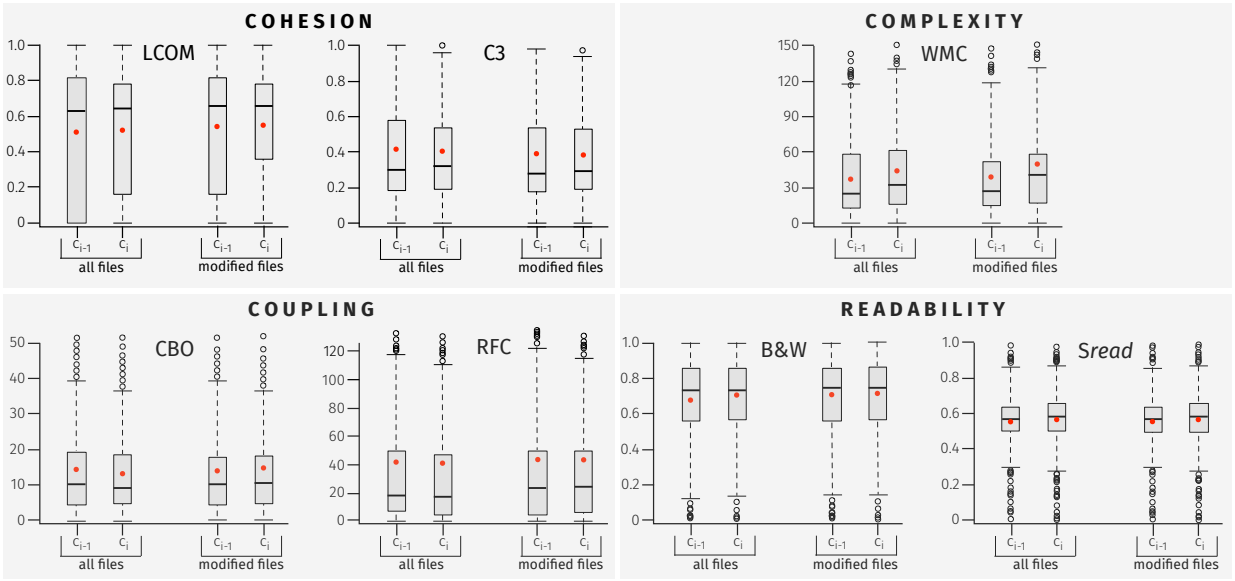


Fig. 1. Metric values before ($c_{i-1}$) and after ($c_i$) commits aimed at improving a specific quality attribute. Results are reported when considering "all files" (*i.e.,* added, deleted, and modified) and only "modified files".

### *B. Data Extraction*

As a first step to answer $RQ_1$ we had to identify code commits in which developers clearly state their aim of improving one of the four quality attributes we considered. To this aim, we mined over 300M commits performed on GitHub between March 2011 and March 2018. We used GitHub Archive, a project recording public GitHub events and making them accessible in JSON format. We used a simple keyword matching mechanism to identify commit notes reporting one of the following four words: *cohesion*, *coupling*, *readability*, or *complexity*. This resulted in 35,239 candidate commits.

Then, we excluded all commits:

1) *Related to non-Java repositories.* This was needed since the tools we use to measure the selected quality metrics are designed to work on Java code. After this filtering 4,221 commits were left.
2) *Modifying more than five Java files.* Such a filtering was applied to reduce the chances of including in our dataset tangled changes [25], meaning commits in which developers implemented changes related to different tasks, one of which being quality improvement. While this is clearly an arbitrary threshold, we only use it as a preliminary automatic filtering step preceding a manual analysis aimed at excluding problematic commits (see step 3). This filtering step resulted in the exclusion of 1,668 commits, leaving 2,459 for manual analysis.
3) *Not having as their main focus code quality improvement.* One of the authors manually inspected the 2,459 remaining commits with the goal of only selecting for the study those in which developers clearly stated their goal of improving one of the four considered quality aspects. The inspector read the commit note and, if needed, inspected the code change by using the Unix `diff`. In case of doubt, the commit was excluded. Then, a second author double-checked only the 1,661 commits classified by the first evaluator as relevant for our study, further excluding 8 of them. This filtering step left 1,653 commits.
4) *Only modifying files implementing test classes.* Since most of the metrics we use are designed to assess code quality on production code, we excluded commits only impacting files implementing test classes. We identified these files by using naming conventions, marking as "test files" those having their name beginning or ending with "`Test`".

We ended up with 1,282 valid commits exctracted from 986 Java systems: 81 commits related to class *cohesion*, 493 to *coupling*, 268 to code *complexity*, and 440 to code *readability*.

For each of the selected commits $c_i$ related to a specific quality attribute (*e.g., cohesion*), we checkout the files impacted by $c_i$ at their snapshots $c_{i-1}$ (*i.e.,* before the changes implemented by $c_i$) and $c_i$ (*i.e.,* after the changes implemented by $c_i$). We consider as impacted files added, deleted, and modified in $c_i$. Note that added files are only available in snapshot $c_i$, deleted files are only available in snapshot $c_{i-1}$, while modified files are available in both snapshots. We refer to the files in $c_{i-1}$ as $F_{c_{i-1}}$ and to the ones in $c_i$ as $F_{c_i}$. Files implementing tests are ignored in both sets for the same reason explained in the commits selection process.

For each production file in $F_{c_{i-1}}$ ($F_{c_i}$), we measure the quality metric(s) selected for the specific quality attribute $c_i$ aims at improving (see Section II-A). For example, if the $c_i$'s commit note states "*improving the class cohesion*", we measure the LCOM and the C3 of all classes in $F_{c_{i-1}}$ and $F_{c_i}$. The goal is to compare the distribution of values for classes in $F_{c_{i-1}}$ and $F_{c_i}$ to check whether there is alignment between the quality improvement as perceived by the developer and the assessment provided by quality metrics. One may argue that comparing metric values in two potentially different sets of files (*i.e.,* $F_{c_{i-1}}$ and $F_{c_i}$) might introduce noise in the data.

Indeed, the files in the two sets are the same only if $c_i$ does not add/delete any file, but limits its action to modify existing files. While we analyze the results by taking this potential source of noise into consideration (details in Section II-C) it is important to clarify why only focusing on commits do not adding/deleting any file is not an option. Several refactoring operations may require the deletion/addition of files. For example, extract class refactoring splits a God class into many classes, possibly distributed into several different files. Thus, this refactoring could result in the deletion of the file implementing the god class and in the addition of many new classes (*i.e.,* disjointed sets of files in $F_{c_{i-1}}$ and $F_{c_i}$). Still, comparing the cohesion of the deleted class in $F_{c_{i-1}}$ to the one of the added classes in $F_{c_i}$ provides an indication of whether the commit $c_i$ reached its goal of increasing class cohesion as assessed by quality metrics.

### C. Data Analysis

We compare the boxplots of the distribution of the selected quality metrics before and after each of the 1,282 commits. For example, we compare the distribution of the LCOM and of the C3 metric for classes impacted in commits stating the intention of improving class cohesion. To account for the possible noise introduced by files added and deleted in commits, we present this data both when considering all files impacted in the commits (*i.e.,* added, deleted, modified) as well as when only focusing on modified files. This latter scenario guarantees that the set of files before and after each commit is exactly the same. We also statistically analyze differences in the metrics' distributions in both scenarios. We use the Mann-Whitney test [26] with results intended as statistically significant at $\alpha = 0.05$. When considering all files impacted in the commits we use the unpaired version of the test. Instead, when focusing only on modified files, we employ the paired version to increase the statistical power. Moreover, since the same files are being repeatedly compared using different metrics, we adjusted our $p$-values using the Holm's correction procedure [27].

We also estimate the magnitude of the differences by using the Cliff's Delta ($d$), a non-parametric effect size measure [28]. We follow well-established guidelines to interpret it: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [28].

Finally, we qualitatively analyze interesting cases of agreement and disagreement between the improvement in quality as perceived by developers and as assessed by quality metrics. To this aim, one of the authors (from now on "inspector") manually inspected all commits, looking at the code diff and at the metrics profile of the impacted code components before/after the commit. The goal was to identify commits suitable and interesting for qualitative analysis. With "suitable", we refer to commits for which the inspector was able to fully understand the context of the commit, meaning the impacted code components. This is not trivial considering that the inspector was not familiar with the analyzed code. With "interesting", we indicate cases that could lead to lessons learned according to the authors.

### D. Replication Package

The data used in our study are publicly available in our replication package [13]. We provide: (i) the raw-data reporting the distribution of metrics pre-/post-commits for each quality attribute, and (ii) the `R` script used to produce the tables and figures reported in this paper.

## III. Results Discussion

Fig. 1 reports the box plots depicting the distribution of metric values before ($c_{i-1}$) and after ($c_i$) commits aimed at improving the specific quality attribute a metric assesses. The box plots are shown both when considering all files impacted in a commit (*i.e.,* the files added, deleted, and modified) as well as when only focusing on modified files. Table I shows the results of the Mann-Whitney test ($p$-value) and of the Cliff's Delta ($d$) when comparing the same distributions depicted in Fig. 1. Significant results are reported in **bold**.

TABLE I
METRICS BEFORE/AFTER COMMITS AIMED AT IMPROVING A QUALITY ATTRIBUTE: MANN-WHITNEY TEST ($p$-VALUE) AND CLIFF'S DELTA ($d$).

**All files**

| Attribute | Metric | $p$-value | $d$ |
|---|---|---|---|
| Cohesion | LCOM | 1.00 | -0.02 (Negligible) |
| | C3 | 1.00 | 0.01 (Negligible) |
| Coupling | CBO | 0.14 | -0.05 (Negligible) |
| | RFC | 0.07 | -0.06 (Negligible) |
| Complexity | WMC | 0.41 | 0.07 (Negligible) |
| Readability | B&W | 1.00 | 0.01 (Negligible) |
| | Sread | 1.00 | 0.02 (Negligible) |

**Modified files**

| Attribute | Metric | $p$-value | $d$ |
|---|---|---|---|
| Cohesion | LCOM | 1.00 | -0.01 (Negligible) |
| | C3 | 1.00 | 0.01 (Negligible) |
| Coupling | CBO | 0.10 | 0.01 (Negligible) |
| | RFC | 0.11 | -0.01 (Negligible) |
| Complexity | WMC | **<0.01** | 0.07 (Negligible) |
| Readability | B&W | 1.00 | -0.01 (Negligible) |
| | Sread | 1.00 | -0.01 (Negligible) |

### A. Cohesion

Fig. 1 shows an interesting trend for the LCOM metric when changes from commits aimed at improving cohesion are implemented. Remember that the LCOM is an inverse cohesion metric: the lower its value the higher the class cohesion. The third quartile is lower for the $c_i$ commits, indicating that developers at least tried to fix the classes having very high values of LCOM (*i.e.,* low cohesion). However, this also resulted in an increase of the first quartile, thus showing some side effect decreasing the cohesion of some of the impacted classes. When comparing the distributions of values before and after the changes, no statistically significant difference is observed (see Table I) both when considering all files as well as when only focusing on modified files.

The second cohesion metric, the C3 (the higher the better), shows a similar trend: the first quartile slightly increases, showing an improvement for classes having a low cohesion.

However, also in this case there is a small drop of cohesion for the highly-cohesive classes, manifested through a lower third quartile. The differences are not statistically significant.

When only considering the modified classes, for which we can compute the metrics both before and after the commit, we found that only 26% and 40% of them improve their LCOM and C3 values, respectively, thanks to the implemented changes.

To better understand cases of agreement and disagreement between what claimed by the developers in the commit message (*i.e.,* their intention of increasing class cohesion) and what reflected in the quality metric values, we qualitatively analyzed some interesting cases. Each case is introduced by a figure summarizing the involved quality attribute (*e.g.,* cohesion), whether the discussed case represents a case of agreement or disagreement, the name of the system and the commit id, the commit note, and the percentage increase/decrease of the quality metrics used to assess the quality attribute. The percentage variation in metrics values is computed by considering the average of all impacted files before and after the commit.

**Cohesion** | **Agreement** | fim@ccf57e72

| Commit note | LCOM | C3 |
|---|---|---|
| *Extract HashProgress to increase cohesion of StateGenerator* | -17% | +27% |

Fig. 2. Class cohesion: agreement case

Fig. 2 summarizes a case of agreement between what claimed by the developer (*i.e.,* the intention of increasing class cohesion) and what assessed by quality metrics. This commit implements an extract class refactoring operation, to extract from the `StateGenerator` class a new class named `HashProgress`, isolating a well-defined responsibility.

This resulted in a strong increase in cohesion for `StateGenerator`, with its LCOM dropping from 0.83 to 0.69 and the C3 metric increasing from 0.26 up to 0.33. The extracted class (*i.e.,* `HashProgress`) also shows a better metrics' profile as compared to the original version of `StateGenerator`, with LCOM=0.71 and C3=0.41. In this case, the structural and semantic cohesion assessed by the LCOM and by the C3 metric, respectively, capture the improvement in class cohesion as perceived by the developer.

**Cohesion** | **Disagreement** | mcts@9598830

| Commit note | LCOM | C3 |
|---|---|---|
| *higher cohesion - move utc calculation from MctsTreeNode to Mcts* | +9% | +6% |

Fig. 3. Class cohesion: partial disagreement case

Fig. 3 reports instead a case of (partial) disagreement between the goal claimed for commit `9598830` and what reflected in quality metrics, with an average increase of the LCOM (*i.e.,* decrease of class cohesion) of 9%. The average value for the C3 metric among the files impacted by the commit increases instead by 6%, supporting the increase of cohesion targeted by the developer. While we also observed cases of complete disagreement (*i.e.,* both metrics confirm a worsening of class cohesion), this commit is interesting when looking at the effect of its changes on each impacted file. The commit modifies the classes `Mcts` and `MctsTreeNode`, with the goal of moving one responsibility, *i.e.,* the computation of the Upper Confidence bound applied to Trees (UCT) for a Monte Carlo search algorithm, from `MctsTreeNode` to `Mcts`. This operation increases the structural cohesion of `MctsTreeNode`, with its LCOM going from 0.82 to 0.79 (-4%) while its semantic cohesion (C3) remains stable. The increase of cohesion obtained for the `MctsTreeNode` class is payed back with a decrease of structural cohesion for `Mcts`, having its LCOM increasing by 22% (from 0.68 to 0.83). The C3 metric provides instead a different assessment, reporting an increase of cohesion for `Mcts` as well (+10%). This case provides us with a number of lessons learned. First, different cohesion metrics provide different hints into code quality variations. This confirms that structural and semantic (*i.e.,* textual) metrics are orthogonal [6], [29] and capture different information in the code. Thus, considering both might provide a more comprehensive view into code quality. Second, the specific formulation of the LCOM metric makes it missing important relationships between methods that should be considered to properly assess class cohesion. Indeed, methods in the same class are considered as related (and thus, as positively contributing to class cohesion) only if they use the same local data. However, two methods could implement strongly related responsibilities without the need for accessing/modifying the same local instance variables. For example, in the discussed case a method moved from `MctsTreeNode` to `Mcts` is `calculateUctValue`, do not accessing any instance variable of its new class (*i.e.,* `Mcts`).

However, this method is clearly related to other methods in `Mcts`, such as `uctSearchWithExploration` or `getNodesBestChildConfidentlyWithExploration`. The latter invokes `calculateUctValue` twice, but these types of structural relationships are simply ignored by the LCOM metric. On the other side, the C3 metric, being computed accordingly to the textual similarity between the pairs of methods in a class, rewards the cohesion of `Mcts`, thanks to the terms shared between the methods (*i.e.,* `uct`, `with`, and `exploration`).

The two above observations also raise a warning for the design of refactoring recommender systems. These tools generally identify refactoring opportunities (*e.g.,* extract class, move method, etc.) by looking for code transformations able to improve some quality indicators, such as the quality metrics we adopted. However, these metrics might not always capture code quality improvement as perceived by developers. This is why semi-automatic techniques putting the developer in the loop while generating refactoring recommendations (see *e.g.,* [30], [31], [32]) could be more effective and identify more meaningful refactorings.

Finally, we discuss an example of complete disagreement between developer's perception and quality metrics (see Fig. 4).

We focus our attention only on a single class modified in the commit (*i.e.,* `TaskLoader`) that was clearly the target of the refactoring (see commit note in Fig. 4). In this case, both metrics report a decrease of class cohesion for `TaskLoader`.

**Cohesion** | **Disagreement** | dooyit@fc054e6

| Commit note | LCOM | C3 |
|---|---|---|
| *Improved cohesion for TaskLoader [...]* | +89% | -22% |

Fig. 4. Class cohesion: disagreement case

Similarly to what observed for the previous case (Fig. 3), this discrepancy is due to the different interpretation that the developer and the quality metrics give to the concept of class cohesion. The developer targets the increase of cohesion by moving into `TaskLoader` methods contained in other classes but closely related to the responsibilities grouped by `TaskLoader` (*e.g.,* the method `loadTask`). Again, the LCOM fails in capturing the relationship existing between the added methods and the ones already present in `TaskLoader` due to the missing sharing of local variables. The C3 is instead penalized by the total lack of comments in the class, representing a precious source of information for textual metrics such as the C3. In cases like this, both metrics fail in capturing the increase of cohesion as perceived by the developer.

By inspecting the code, we noticed that something that could have helped in capturing the relationship between the methods moved into `TaskLoader` and the ones already present in it are the used types. For example, `loadTask` uses objects of type `TaskManager` and `JsonObject`, also used by five of the other eight methods in the class[3]. However, this type of relationship (*i.e.,* shared usage of types) is not exploited by the metrics we used to capture class cohesion. Again, this highlights the many forms that class cohesion can have and the difficulty in capturing all of them with code quality metrics.

### B. Coupling

There is a positive trend for what concerns both the CBO and the RFC metrics, with their value decreasing (*i.e.,* lower coupling) after the changes implemented by developers. When considering all files, the CBO median drops from 10 to 9, while the RFC goes from 19 to 17. When considering only the modified files, the difference can not be observed. Although, we can see that there is a decrease in the classes with very high coupling (see top whisker). Only 16% and 22% of modified classes improve their CBO and RFC values, respectively, thanks to the changes implemented in the commits.

**Coupling** | **Agreement** | Mezzo@11e404d

| Commit note | CBO | RFC |
|---|---|---|
| *Refactored [...] to remove coupling between DropboxSource/Fragment from the MusicSourceFragment* | -15% | -27% |

Fig. 5. Class coupling: agreement case

Fig. 5 shows a case of agreement on the class coupling improvement between developer's perception and what assessed by the CBO and RFC quality metrics. The commit implements refactoring operations involving the `MusicSourceFragment` class and its subclass `DropboxFragment` to reduce the coupling between them. This was realized through the removal of three methods from `MusicSourceFragment` and a *push down method* refactoring (*i.e.,* method `downloadAll`) from `MusicSourceFragment` to its subclass. These changes resulted in a coupling reduction for the `DropboxFragment` class, with a drop of its CBO from 13 to 11 and its RFC from 33 to 24, supporting the developer's aim. Moreover, the `downloadAll` method was only used by the `DropboxFragment` subclass, thus justifying the performed refactoring.

**Coupling** | **Disagreement** | CrowingMiment@0baed82

| Commit note | CBO | RFC |
|---|---|---|
| *Decoupling fragment and activity* | +16% | +54% |

Fig. 6. Class coupling: disagreement case

Fig. 6 reports data about a case of disagreement between developer's perception and quality metrics. The developer summarizes the commit as aimed at *decoupling fragment and activity*. The commit modifies three classes: `MainActivity`, `MomentFragment`, and `MomentListFragment`. All three classes exhibit worsening of their CBO (on average, +16%) and RFC (on average, +54%). While this may look like a strong case of disagreement, the problem here is in the type of coupling meant by the developer in the commit message as opposed to the one assessed by the metrics.

Indeed, the developer aims at reducing the coupling between `MainActivity` and `MomentFragment` by replacing in the former the creation of an object of type `MomentFragment` though its constructor with an invocation to a *factory method* returning an instance of `MomentFragment`. Here, the coupling reduction as intended by the developer seems to be mostly related to isolating future changes, but does not reduce the number of dependencies (CBO) nor the message flow (RFC) of the involved classes. Such a form of coupling reduction could be captured, after the change has been implemented, by analyzing the logical coupling between classes [33] and, in particular, by measuring how frequently the involved pair of classes (*i.e.,* `MainActivity` and `MomentFragment`) co-change before and after the changes introduced by this commit.

Most of the cases of disagreement we identified for the coupling metrics can be represented by the commit we just discussed: The developers targets the isolation of future changes by performing refactoring operations aimed at decoupling a pair of classes and/or two application layers.

However, these refactorings do not always result in an improvement in terms of CBO and/or RFC, since the overall number of dependencies does not decrease and the message flow of the application logic is not simplified.

### C. Complexity

Surprising are the results we achieved for what concerns code complexity. Indeed, the distributions depicted in Fig. 1 indicate an overall increase of the complexity after the commits in which developers claimed to target a code complexity reduction.

[3]Four methods use `TaskManager`, one `JsonObject`.

This is also confirmed by the fact that only 13% of classes modified in these commits exhibit a WMC reduction. Moreover, this is the only case in which the differences are assessed as statistically significant even thought with a negligible effect size (see Table I). Before moving to the qualitative analysis, it is worth remembering that the WMC metric we used to assess complexity is computed as the sum of the McCabe's cyclomatic complexity of its methods, meaning the number of linearly independent paths in all its methods.

Given the achieved results, we focus our attention only on a case of disagreement. Indeed, we only find a few cases of agreement between the reduction of code complexity as perceived by the developers and what indicated by the WMC. These cases are mostly the result of the removal of unneeded selection statements (*e.g.,* `if` branches) in the code. Fig. 7 summarizes an extreme case of disagreement we found.

**Complexity** | **Disagreement** | intellij-haxe@8aab776

| Commit note | WMC |
|---|---|
| *[..] Moved all else if code in expression evaluator to methods to reduce cyclomatic complexity [...]* | +492% |

Fig. 7. Class complexity: disagreement case

The developer explicitly refers to the reduction of cyclomatic complexity as the goal of the commit and also reports the target of this operation being the class `HaxeExpressionEvaluator` (referred to as "*expression evaluator*"). Before explaining the performed refactoring and its consequences on the WMC quality metric, it is important to better clarify the context, and in particular the system in which this commit has been performed. Intellij-haxe is a plugin to develop Haxe programs with Intellij IDEA. Thus, some of its classes implement parsing functionalities and are expected to exhibit high complexity.

The focus of the refactoring performed in commit `8aab776` is the `_handle` method having, in the pre-refactoring version, 834 lines of code, making it a good candidate for extract method refactoring. The complexity of this method is due to the fact that it is used to "*handle*" different types of `Haxe` code statement (*e.g.,* `HaxeForStatement`, `HaxeSwitchCaseBlock`, etc.). Its cyclomatic complexity is negatively affected by the need for implementing two "responsibilities". First, the `_handle` method has to identify the type of statement to handle (*e.g.,* `for`, `throw`, etc.), thus requiring a long list of `if` statements *e.g.,* `if` (`element instanceof HaxeThrowStatement`) *do something*. Second, the logic required to handle each type of statement is implemented in the same method.

In this commit the developer extracted from the `_handle` method 42 new methods, each one focused on the handling on a specific `Haxe` statement type. For example, `handleReturnStatement` is in charge of handling statements of type `HaxeReturnStatement`, and it is invoked by the `_handle` method when needed:

```
if (element instanceof HaxeReturnStatement) {
   return handleReturnStatement(...)
}
```

This series of extract method refactorings is likely to improve the readability and reusability of the code, removing a very long method and extracting from it small methods focused on a precise responsibility. However, based on the WMC assessment, this did not help in terms of complexity, with the WMC for `HaxeExpressionEvaluator` increasing from 25 to 173. This was due to the creation of the 42 new methods and to the modest reduction in cyclomatic complexity of the refactored method (*i.e.,* `_handle`), still in charge of delegating the handling of the different types of statements to the newly created methods (thus, still requiring a high number of `if` statements).

This disagreement case highlights how broad the concept of code complexity is and how difficult it is to distill such a concept inside a metric value. From the developer's point-of-view, the goal was to reorganize the code in order to make it simpler working with it. Looking at the code this goal seems to be fully achieved. However, the overall number of independent paths in the code is increased, leading the WMC metric to assess an increase in code complexity. This further stresses that the blind use of metrics as quality indicators (*e.g.,* to detect code smells, to assess the impact of refactorings, etc.) might lead to questionable conclusions.

### D. Readabilty

Code readability is the quality attribute for which we observed the less perceivable changes in the metrics' values (see bottom part of Fig. 1). This holds for both metrics we employed, despite they use totally different features when assessing code readability. The two metrics report only 28% (B&W) and 38% ($Sread$) of the modified classes as improving their readability after the changes implemented in the commits.

**Readability** | **Agreement** | json-flattener@dc23d2c

| Commit note | B&W | *Sread* |
|---|---|---|
| *Increase code readability* | +9% | +24% |

Fig. 8. Code readability: agreement case

Fig. 8 shows a case of agreement between the developer's perception of code readability improvement and the indications provided by both quality metrics. The developer states the intention of increasing the code readability and pursues this goal by working on a single class named `JsonUnflattener`.

In particular, she tries to simplify the `unflatten` method by extracting from it six new methods, thus reducing its size from 82 to 45 ELOC. This results in an overall increase of readability for the class as assessed by the two employed quality metrics.

The B&W metric increases by 9% (from 0.77 to 0.86) thanks to improvements in several of the code features exploited by it. For example, the average indentation level of the code statements is decreased (*i.e.,* less statements exhibit high indentation) and the number of comments in the class is increased (thanks to the new methods). The metric by Scalabrino *et al.* reports an increase of readability of 24% (from 0.32 to 0.56), mostly due to the high Textual Coherence (TC) [24] of the newly introduced methods.

The textual coherence estimates the number of "concepts" implemented by a method by computing the vocabulary overlap between the syntactic blocks of the method. Given the narrow responsibility implemented by the extracted methods, their TC is very high, thus increasing the overall class readability.

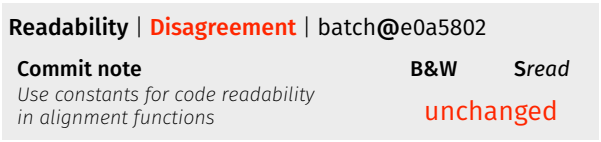


Fig. 9. Code readability: disagreement case

Fig. 9 summarizes what happened in commit `e0a5802` with the developer targeting an increase of readability through the introduction of constants instead of "magic numbers". This refactoring is performed when a literal number in the code has a specific meaning and, thus, can be replaced with a constant having a name making an explicit reference to its meaning. In this case, many magic numbers valued "0" in the code have been replaced with the `StringUtil.LEFT_ALIGNMENT` constant. Such a change, while representing an improvement in code readability for the developer, did not affect the value of the two employed metrics. The reason in this case is fairly simple: none of the two metrics penalizes the readability of a snippet using magic numbers.

### E. *Quality-related commits* vs *general-commits*

While we compared the metric values before/after commits in which developers state their intention of improving specific quality attributes, we did not show what happens in commits where developers do not state the goal of improving code quality (from now one "general-commits").

For this reason, we performed the same analyses adopted in our experimental design for 300 randomly selected commits that were excluded in the manual validation process since not explicitly reporting in the commit note references to code quality improvement. Our results show that for all quality metrics there was no statistically significant difference in the distribution of metrics before/after the commits (adjusted $p$-value always equal 1.0 and $d$ effect size always lower than 0.02). We observed the same trend both when considering all files as well as when only focusing on the modified ones.

Then, we compared the changes in quality metrics (deltas) obtained in quality-related commits to that achieved in general-commits. This means, for example, computing the LCOM values for all classes modified in each commit $c_i$ before and after $c_i$ and, then compute the delta (*e.g.,* LCOM before=0.8, LCOM after=0.6, delta=0.2). Clearly, this analysis is possible when focusing on modified files only, since we need the metric values of the classes before/after the commits. We found significant difference (adjusted $p$-value lower than 0.05) for the LCOM, CBO, WMC, and $Sread$, in all cases accompanied by a negligible or small effect size. No statistically significant differences were observed for C3, RFC, and B&W. Interestingly, while the LCOM, CBO and $Sread$ achieves better deltas in commits targeting their improvement, the WMC deltas are better in general-commits.

These results strengthen our findings highlighting that the considered quality metrics are not "complete", meaning that they are not always able to capture code quality improvement as perceived by developers.

## IV. Threats to Validity

Threats to **construct validity** concern the relation between the theory and the observation, and in this work are mainly due to the collected data and to the measurements we performed. This is the most important kind of threat for our study, and is related to:

*Misclassification of commits in which developers claimed the intention to improve a specific code quality attribute*. One of the authors manually analyzed 2,459 pre-filtered commits to identify the ones in which developers clearly stated the intention to improve one of the four quality attributes considered in our study (*i.e.,* cohesion, coupling, complexity, and readability). In case of doubt, the commit was excluded to reduce the chances of including false positives in our study. On top of that, a second author double-checked the commits selected by the first evaluator as relevant for our study. However, as in any manual process errors are possible, and we cannot completely exclude the presence of false positives in our dataset.

*Imprecision due to tangled code changes [25]*. We cannot exclude that some of the commits we considered grouped together tangled code changes, of which only a subset aimed at improving the quality attribute claimed in the commit. At least, we mitigated such a threat by excluding commits impacting more than five Java files, more likely to represent tangled code changes. Still, this does not exclude the presence in our dataset of commits including multiple edits, only one of which targeting code quality improvement.

*Quality metrics considered in the study*. We selected metrics widely adopted in the literature to assess the quality attributes considered in our study. However, it is possible that other metrics could lead to different results, showing a higher/lower alignment with the code quality improvement as perceived by developers. For example, "comprehension metrics" built on top of findings reported in previous studies [34], [35], [36] could be used to complement the employed readability metrics.
Some studies indicated the need for considering both the WMC and the SLOC when assessing the cyclomatic complexity of a class [37], [38], [39]. However, due to contradicting evidences in the literature [40], [41], we decided to only use the WMC in the assessment of code complexity.

*The considered quality metrics might not have been designed to capture incremental differences of the same code artefact.* Our study assesses whether quality metrics are able to capture code quality improvement as perceived by developers. However, this assumes that quality metrics are able to capture also "incremental differences" in code quality (*i.e.,* small changes implemented by the developers) occurring to the same code component. Such an assumption might not hold for the considered metrics, since some of them were designed to compare the quality of different code artifacts.

*Identification of test classes*. We used a heuristic based on naming conventions to identify test classes and exclude them from our study. Such a heuristic works only if naming conventions are followed by developers.

Thus, it is possible that some test classes have been included in our dataset. However, we did not find any of them during our qualitative analysis.

Threats to **internal validity** concern external factors we did not consider that could affect the variables and the relations being investigated. As opposed to similar studies in the literature investigating the developers' perception of code quality (see Section V), we did not survey developers but preferred to mine real refactorings they performed over the change history of software systems. A drawback of our experimental design is that we do not have demographic information about the developers that performed the considered refactorings. In other words, we cannot assess their programming experience nor whether they are really aware of the four code quality attributes considered in our study. For this reason, our findings cannot be considered representative of a specific population of developers. Also, we did not control for the quality of projects in our dataset. As described in Section II-B, our data extraction process did not include filtering criteria aimed at assessing the "quality level" of the projects from which we mined the commits subject of our study. This means that our dataset could include both commits from students' projects as well as from well-known open source communities. Future work will be devoted to control for these confounding factors.

Threats to **conclusion validity** concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences. When comparing the quality metrics of the files impacted in the commits we use the Mann-Whitney test, which assumes independence of the two groups of compared samples. Such an assumption is not supported in our data, since we considered the quality of code components before/after refactoring.

Threats to **external validity** concern the generalization of results. Our study is performed on a set of 1,282 commits impacting a total of 2,767 classes. Yet, a different dataset and different quality metrics could lead to different conclusions.

## V. Related Work

We discuss the literature studying the (i) developers' perception of code quality, and (ii) impact of refactoring on quality. While our work is mostly related to (i), our experimental design is close to the one adopted in previous study investigating (ii).

**On the developers' perception of code quality**. Counsell *et al.* [11] approached the issue of investigating the developers' perception of class cohesion asking 24 experienced and novice IT professionals to evaluate the cohesion of 10 Java classes. They found that the perceived cohesion (i) is not correlated with the class size, (ii) is correlated to comments density, and (iii) does not depend on the developers' experience.

Revelle *et al.* [9] presented a new coupling metric named Hybrid Feature Coupling (HFC), combining structural and textual features. As part of the HFC evaluation, they surveyed 31 developers asking their assessment of the coupling strength between 16 pairs of classes. Then, they measured the correlation between the developer's responses and the HFC metric, showing that the HFC is able to capture coupling similarly to what assessed by developers.

Bavota *et al.* [10] studied the developers' perception of software coupling. They focused the attention on four different coupling metrics, namely structural [42], dynamic [43], semantic [44], and logical coupling [33]. They asked 76 developers to look into pairs of classes reported as strongly/weakly coupled by each of these metrics and determine the extent to which they were coupled. Then, the authors investigated the level of agreement between the four metrics and the developers' perception of coupling. Their findings highlight that the semantic coupling is the one that better aligns with developers' perception of coupling.

While the goal of our study is similar to the three above described works, we: (i) focus on four different quality attributes (*i.e.,* cohesion, coupling, readability, and complexity), and (ii) adopt a totally different experimental design do not surveying software developers.

Palomba *et al.* [45] focused the attention on the developers' perception of code smells. They surveyed 34 developers showing them code snippets affected and not affected by code smells, and asked whether, in the respondents' opinion, the code component exhibited quality issues. They found that smells related to complex/long source code are generally perceived as an important threat by developers as opposed to those related to good object-oriented programming practices. Our work complements [45] focusing on the developers' perception of software quality from a different perspective.

**On the impact of refactoring on code quality**. Stroggylos and Spinellis [15] studied the impact of refactoring on the values of eight object-oriented code quality metrics. The refactoring operations were identified by looking in the commit messages for mentions of words stemming from the verb "refactor". Their results show the possible negative effects that refactoring can have on some quality metrics.

Moser *et al.* [16] investigated the impact of refactoring on the productivity of an agile team and on the quality of the code they produce. They achieved results show that refactoring increases both software quality but also developers' productivity.

Alshayeb [17] investigated the impact of refactoring on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their study is performed on three systems they refactored, measuring the five attributes on each system pre- and post-refactoring. Their findings highlight that benefits brought by refactoring on some code classes are often counterbalanced by a decrease of quality in some other classes.

Szoke *et al.* [18] investigated the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality (as assessed by a quality model considering different attributes). This is the study mostly related to the one presented in this paper. Indeed, Szoke *et al.* also analyzed the impact on quality of 32 commits explicitly targeting the improvement of a specific quality metric (*e.g.,* lines of code). Our study is performed on a different and larger dataset (1,282 commits). Also, while Szoke *et al.* exploit a single score provided by a quality model to asses the changes in quality, we use specific metrics related to the quality attributes developers aim at improving in the analyzed commits. Despite these differences, the findings reported in the two studies (*i.e.,* [18] and ours) point in the same direction: minor refactoring changes (such as the ones implemented in the commits considered in our study), rarely impact software quality as assessed by metrics.

Finally, Bavota *et al.* [46] mined refactoring operations using the Ref-Finder [47] tool, and showed that 42% of refactoring operations are performed on code entities affected by code smells and only in 7% of cases the refactoring was able to remove the code smell.

In this paper, we looked for commits explicitly reporting the developers' intention of improving a specific quality attribute (*e.g.,* coupling), ignoring whether refactoring operations were actually performed. Then, we studied the impact of that change on quality metrics measuring that specific attribute. The goal was to assess whether the considered quality metrics are able to capture code quality improvement as perceived by developers.

## VI. Conclusion

We presented a study aimed at investigating whether the increase of code quality as perceived by software developers can be captured by code quality metrics. To do that, we identified a set of 1,282 commits in which the developer explicitly states her intention to improve one of four quality attributes (*i.e.,* cohesion, coupling, code complexity, or code readability). Then, for each commit $c$, we used quality metrics designed to capture the quality attribute improved by $c$ (*e.g.,* cohesion) to verify whether the improvement was also reflected in the metrics' values. We found that, more often than not, the considered quality metrics are not able to capture the quality improvement as perceived by developers.

Our analysis provided us with a number of lessons learned:

*It is far from trivial to distill a code quality attribute inside a single metric value*. Several of the cases we discussed highlighted limitations of metrics when assessing the quality attribute they are designed for. For example, when discussing the case in Fig. 4, we noticed that something that could have helped in capturing the improvement of cohesion as perceived by the developer was the sharing of object types used by the methods in the class. However, this information was not considered by both metrics used in our study. A similar observation holds for the code readability case presented in Fig. 9, in which the developer tried to increase readability by replacing magic numbers with constants, an aspect ignored by both state-of-the-art metrics. In general, our analysis suggests that a single metric can only provide a very partial view on a code quality attribute, and that the combination of many quality metrics should be preferred, when possible.

*There might be inconsistency between a code quality attribute as interpreted by developers and as assessed by quality metrics*. The interpretation of a code quality attribute might be very subjective, as demonstrated by the cases discussed for coupling and complexity. In the latter case, while the developer clearly made the code easier to work with, the WMC metric did not capture such an improvement, due to the very specific aspect of code complexity it focuses on. This leads to the next lesson learned.

*Applications of software engineering built on top of quality metrics should be aware of their limitations*. This is especially true for recommender systems aimed at identifying design flaws (*e.g.,* code smells) and suggesting refactorings. Indeed, while a refactoring might make total sense from the quality metrics point-of-view, it might be meaningless for developers (or *vice versa*). This is why semi-automatic techniques putting the developer in the loop while generating recommendations (see *e.g.,* [30], [32]) could be more effective.

*Experimentations based on code quality metrics can provide hints about code quality, but should be complemented by qualitative feedback provided by developers.* Such experimentations are typical of refactoring recommenders: the quality of the refactored code pre- and post-refactoring is assessed by using quality metrics (*e.g.,* cohesion and coupling metrics for an extract class recommender). While quality metrics can provide some hints about code quality, our findings stress the importance of also involving developers in such evaluations, since their view on code quality might be different and, at the end, they are the users of such recommenders.

Future work will be devoted to further investigate the relationship between code quality as perceived by developers and as assessed by metrics. This will be mostly done by combining the results of our study with qualitative insights collected from the developers who authored the commits. In addition to that, we also plan to include new metrics in our study and to enrich our analysis with the goal of defining a taxonomy of "disagreement reasons" for each of the considered metrics (*i.e.,* explain why a disagreement is observed between the developer's perspective of quality improvement and what assessed by a quality metric).

## Acknowledgments

Pantiuchina and Bavota thanks the Swiss National Science foundation for the financial support through SNF Project JITRA, No. 172479.

## REFERENCES


[1] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[2] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[3] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Software Eng.*, pp. 264–282, 2011.

[4] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, pp. 2503–2545, 2016.

[5] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Software Eng.*, pp. 897–910, 2005.

[6] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Software Eng.*, pp. 287–300, 2008.

[7] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *30th International Conference on Software Engineering (ICSE 2008)*, 2008, pp. 531–540.

[8] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, pp. 1617–1664, 2014.

[9] M. Revelle, M. Gethers, and D. Poshyvanyk, "Using structural and textual information to capture feature coupling in object-oriented software," *Empirical Software Engineering*, pp. 773–811, 2011.

[10] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, pp. 692–701.

[11] S. Counsell, S. Swift, A. Tucker, and E. Mendes, "Object-oriented cohesion subjectivity amongst experienced and novice developers: An empirical study," *SIGSOFT Softw. Eng. Notes*, pp. 1–10, 2006.

[12] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Systems*, vol. 3, pp. 143–158, 1996.

[13] *Replication Package*, 2018. [Online]. Available: https://github.com/code-quality-analysis-repo/code-quality-analysis-repo

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07, 2007.

[16] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing agility and formalism in software engineering," 2008, ch. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266.

[17] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, pp. 1319 – 1326, 2009.

[18] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, 2014, pp. 95–104.

[19] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, pp. 115–139, 1974.

[20] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[21] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, June 1994.

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

[23] R. P. L. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[24] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *24th IEEE International Conference on Program Comprehension*, 2016.

[25] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[26] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[27] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[28] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[29] G. Bavota, "Using structural and semantic information to support software refactoring," in *34th International Conference on Software Engineering, ICSE*, 2012, pp. 1479–1482.

[30] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *28th IEEE International Conference on Software Maintenance, ICSM*, 2012, pp. 472–481.

[31] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the developer in-the-loop: An interactive GA for software re-modularization," in *Search Based Software Engineering - 4th International Symposium, SSBSE Proceedings*, 2012, pp. 75–89.

[32] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó. Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *ACM/IEEE International Conference on Automated Software Engineering,ASE*, 2014, pp. 331–336.

[33] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of 14th IEEE International Conference on Software Maintenance*, 1998, pp. 190–198.

[34] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 95–99.

[35] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, June 2006, pp. 3–12.

[36] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *2010 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp. 156–165.

[37] D. Landman, A. Serebrenik, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods," in *30th IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 221–230.

[38] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods and C functions," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 589–618, 2016. [Online]. Available: https://doi.org/10.1002/smr.1760

[39] ——, "Corrigendum: Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods and C functions published on 9 december 2015," *Journal of Software: Evolution and Process*, vol. 29, no. 10, 2017. [Online]. Available: https://doi.org/10.1002/smr.1914

[40] A. Jbara, A. Matan, and D. G. Feitelson, "High-mcc functions in the linux kernel," *Empirical Softw. Engg.*, vol. 19, no. 5, pp. 1261–1298, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9275-7

[41] A. E. H. Israel Herraiz, "Beyond lines of code: Do we need more complexity metrics?" O'Reilly Media, 2010, pp. 126–141.

[42] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring the coupling and cohesion of an object-oriented program based on information flow," in *Proc. of International Conference on Software Quality*, 1995, pp. 81–90.

[43] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, pp. 491–506, 2004.

[44] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, 2006, pp. 469 – 478.

[45] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 101–110.

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

[47] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10.