2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

Improving Source Code Readability: Theory and Practice

Sarah Fakhoury*, Devjeet Roy*, Sk. Adnan Hassan†, Venera Arnaoudova*
*School of Electrical Engineering and Computer Science, Washington State University, Pullman (WA), USA
†Infosapex Limited, Dhaka, Bangladesh
{sfakhour, droy, varnaoud}@eecs.wsu.edu, adnan@infosapex.com

***Index Terms*—Readability, Code Quality Metrics, Developers' Perception**

***Abstract*—There are several widely accepted metrics to measure code quality that are currently being used in both research and practice to detect code smells and to find opportunities for code improvement. Although these metrics have been proposed as a proxy of code quality, recent research suggests that more often than not, state-of-the-art code quality metrics do not successfully capture quality improvements in the source code as perceived by developers. More specifically, results show that there may be inconsistencies between, on the one hand, the results from metrics for cohesion, coupling, complexity, and readability, and, on the other hand, the interpretation of these metrics in practice. As code improvement tools rely on these metrics, there is a clear need to identify and resolve the aforementioned inconsistencies. This will allow for the creation of tools that are more aligned with developers' perception of quality, and can more effectively help source code improvement efforts.**

In this study, we investigate 548 instances of source code readability improvements, as explicitly stated by internal developers in practice, from 63 engineered software projects. We show that current readability models fail to capture readability improvements. We also show that tools to calculate additional metrics, to detect refactorings, and to detect style problems are able to capture characteristics that are specific to readability changes and thus should be considered by future readability models.

I. Introduction

The code quality of every single software project is assessed at some point within its development life cycle. Researchers have developed several metrics to measure code quality which are widely accepted in practice. However, recent research by Pantiuchina et al. [22] has shown that more often than not, state-of-the-art code quality metrics are unable to capture quality improvements in the source code, as perceived by developers in practice. In fact, metrics are unable to capture the majority of instances where improvements in the source code's cohesion, complexity, coupling, and readability are made. However, when evaluating code quality metric models in research, numbers as high as 85% [28] for accuracy are reported. Therefore, there seems to be a discrepancy between on the one hand, the results from models in research, and on the other hand, the interpretation of these metrics in practice. In order to build tools that can help developers identify a range of opportunities for code quality improvement in practice, we need models that can measure and identify code quality metrics in a way that aligns with developer's perception and can be used in practice.

In this paper we investigate three state of the art readability models, which have been shown to successfully capture source code readability in theory [28], and discuss the results when those models are applied on source code, i.e., in practice. This is in contrast to previous work, which evaluates readability based on surveying techniques. Instead, we aim to understand whether these readability models are able to capture readability improvements as explicitly tagged by developers in their day to day commits. We confirm recent findings by Pantiuchina et al. [22] that current readability models are unable to capture readability improvements during software maintenance activities in practice. We expand existing work by providing the following contributions:

1) We identify metrics captured by current state-of-the-art readability models that do not significantly change during readability improvements in practice.
2) We show that certain structural metrics, refactorings, style problems, and words contained in commit messages can be good indicators of changes that pertain to readability improvement.
3) We provide a replication package that includes an oracle of manually curated commits related to readability improvements [2].

We analyze 548 readability commits from 63 engineered Java projects. We identify a commit as a readability commit if its message explicitly states that the change made by the commit aims to improve readability. To ensure that our dataset contains only files affected by readability commits, we manually untangle each commit to include only files where the stated improvement is applied. Results show that readability commits tend to fix problems introduced by non-readability commits, such as imports, spaces, and braces. We also observe that certain refactorings such as extract method, inline method, and rename class are very frequent in readability commits and almost non-existant in other commits. These changes are confirmed by the commit messages that developers write to summarize the changes of the commits. Thus, readability models should include such information to more accurately detect incremental readability changes.

The rest of this paper is organized as follows: Section II outlines the research questions of the study, defines the

978-1-7281-1519-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPC.2019.00014

readability models and metrics considered, introduces the static analysis tools used, and explains the data collection and processing techniques. Section III contains the results for each of the research questions defined. Section IV outlines the threats to the validity of the study, Section V discusses related work, and Section VI contains concluding remarks.

II. Methodology

A. Research Questions

The *goal* of this study is to investigate the degree to which different models and measures capture readability improvements at commit level. The *quality focus* is the performance of current state-of-the-art readability models, as well as tools for metrics and refactoring. The *perspective* of the study is that of researchers and developers, who are interested in measuring readability in their day to day software maintenance tasks. The evaluation is carried out on 548 commits from 63 engineered Java projects collected from GitHub[1].

This study addresses the following research questions:

1) **RQ1:** ***Are state-of-the-art readability models able to capture readability improvements in practice?***
   To answer this research question we consider three state of the art readability models, Scalabrino's model [29], Dorn's model [8], and the Combined model, which combines the first two models as well as Buse & Weimer's model [6] and Posnett's model [24]. We run each of the models on a set of files before and after readability improvements were made, and investigate whether they are able to capture the improvement.
2) **RQ2:** ***Which source code metrics capture improvement in the readability of source code, as perceived by developers?***
   To answer this research question we run SourceMeter [16] on files before and after readability improvements were made and look for significant differences between the metric values in the two populations.
3) **RQ3:** ***What types of changes do developers perform during readability improvements?***
   To answer this research question we run ChangeDistiller [9], CheckStyle [1], and RefactoringMiner [30] to understand the types of changes and refactorings that are made on source code during readability improvements.
4) **RQ4:** ***What types of changes do developers describe in commit comments related to readability improvements?***
   To answer this research question, we analyze the commits comments from our dataset to understand i) what are the most frequently used words? ii) what types of actions are described, and on what aspects of source code?

B. Readability

1) Identifying Readability and Non-Readability Commits: To answer the research questions defined in Section II-A, we collect commits from engineered projects where developers specifically state a readability improvement was made to the source code. We use Reaper [21], a tool that calculates a score for GitHub repositories to determine whether they are engineered projects or not. Next, we identify candidate commits to include in our oracle by performing a simple keyword match. The keywords we use are related to readability: 'readable', 'readability', 'easier to read', 'comprehension', 'comprehensible', 'understand', 'understanding', and 'clean up'. Next, two authors of this paper manually excluded commits that do not explicitly reflect readability improvements of the source code. For example, commits that improve readability of UI elements of applications such as *"Added user-readable version of the todo.txt"*.

Commits often times target several different changes at once. Therefore, to ensure that we only include files that contain readability changes we manually untangle commits by looking at the commit messages and source code of all commits that alter more than one file. This process resulted in 548 readability commits and 2,323 datapoints from 63 Java projects. A datapoint is a pair of before-after versions of a file. In this paper we often refer to a datapoint simply as *file* but note that this is not the unique number of files affected by the commits. When answering **RQ2**, **RQ3**, and **RQ4**, we compare readability commits to non-readability commits. To identify non-readability commits, we randomly sample 1,231 commits (which corresponds to roughly the same number of datapoints as the one involved in the readability commits) from the 63 engineered projects after removing the readability commits that we identified previously.

2) Readability Models: We consider three state of the art readability models which assess different aspects of the source code as our reference point for the metrics that have been adopted in research for improved readability. The first model we used is based on the original model proposed by Scalabrino et al. [28]; it is dependent on metrics that measure the quality of the source code lexicon as a proxy of readability. The features considered by the model are: comments and identifier consistency, identifier terms in dictionary, narrow meaning identifiers, comments readability, number of meanings, textual coherence, and number of concepts. The model was evaluated on a dataset composed by 200 Java snippets. We use the implementation provided by the authors [28].

The second model we consider is proposed by Dorn et al. as a generalizable model for source code readability [8]. This model relies on various features like visual, spatial, alignment, and linguistic aspects of the source code. The model was evaluated by conducting a survey with around 5,000 participants rating the readability of 360 code snippets written in 3 different programming languages. We use the implementation provided by Scalabrino et al. in their paper comparing state of the art readability models [28].

The last model we consider is implemented by Scalabrino et al. [28] as a combination of multiple state of the art readability models, which considers both structural and linguistic aspects of the source code. This model is referred to as the *Combined*

[1] https://github.com

model, which combines the first two models as well as Buse & Weimer's model [6] and Posnett's model [24]. This model is shown to have the highest accuracy scores when evaluated against all the individual models on the same dataset. The model combines the model proposed by Dorn et al. [8], Buse and Weimer [6], Posnett et al. [24], and Scalabrino et al. [28].

We run all three models on our dataset of files before and after readability commits.

C. Metrics Collection

1) SourceMeter: SourceMeter [16] performs deep static analysis on source code to compute code quality metrics. These metrics are grouped into 6 categories: cohesion, complexity, coupling, documentation, inheritance, and size. Each of these categories includes several metrics. For example, complexity metrics include Halstead Effort (HE), McCabe's Cyclomatic Complexity (McCC) and Weighted Methods per Class (WMC). We use SourceMeter to answer **RQ2**.

2) CheckStyle: Checkstyle [1] is a static analysis tool which checks source code adherence to configurable rules. We use the standalone version of CheckStyle, along with two of the configuration files provided by checkstyle, sunchecks.xml and googlechecks.xml, modified to add a warning for magic numbers. For each file in a readability commit, we run the tool on the before and after commit snapshots of the file. We then extract the number of files affected by each warning between the two sets of before and after files to answer **RQ3**.

3) ChangeDistiller: ChangeDistiller was created by Fluri et al. [9] as a tool to extract and categorize statement level changes in Java source code. ChangeDistiller uses the abstract syntax tree (AST) of the source code to extract fine grained change extraction using the change distiller algorithm [10]. Statement level source code changes are classified according to a taxonomy of 41 change types. We run ChangeDistiller on files before and after readability improvements are made and use the results to answer **RQ3**.

4) RefactoringMiner: RefactoringMiner detects refactorings across the history of Java projects, using the RMiner technique as proposed by Tsantalis et al. [30]. It supports 21 refactoring types, such as Extract Method, Move Method, Replace Variable with Method, and Parameterize Variable. The authors show that RefactoringMiner has 98% precision and 87% recall. To answer **RQ3** we run RefactoringMiner on the files before and after readability improvements were made by developers to identify the types of changes that are being made in practice to improve readability.

D. Analysis Methods

1) Preprocessing: Commit comments often contain the name and email address of the committer as well as URLs. We remove those from the messages using spaCy [14] Name Entity Recognition and regular expression matchers. Next, we used a tokenizer to parse the commit messages into a list of words and remove stop words using the list provided by NLTK [5]. Due to the fact that commits in our oracle are identified using keywords, we added these keywords to the list of stop-words to be removed. Next, we applied Porter Stemmer [23] to identify the stem of each word. We use the preprocessed commits when investigating the top words in commit messages for **RQ4**.

2) Part-Of-Speech (POS) Tagging: To understand the actions that developers document in commit messages we analyze verb-object phrase pairs. To this end, we must tag commit comments with their part of speech. We used the NLTK POS tagger to parse and tag each word in the commit comments. We use POS tagging to answer **RQ4**.

3) Grammatical Dependencies: Grammatical dependencies are direct grammatical relations between terms in a sentence. To analyze which aspects of source code developers are working on while improving readability, we look into the most common objects in commit comments. We use the spaCy dependency parser [14], which generates dependency trees for each sentence in a commit comment. Each dependency tree provides the following features: dependency labels and entity head dependency. To extract object-phrases, rather than singular words, we only use the top-most node in the dependency tree. We use grammatical dependencies to answer **RQ4**.

4) Wilcoxon Signed Rank Test: In **RQ2**, we aim to find metrics which capture improvements in readability of source code as perceived by developers. We group our source code samples into pairs, each pair is comprised of a snapshot of the source file before a commit, and a snapshot of the file after the same commit. We then perform a Wilcoxon signed-rank test, a non-parametric statistical test used to compare two related samples, to assess whether the population mean ranks differ. We choose Wilcoxon because our data was shown to not follow a normal distribution when tested using the Shapiro-Wilk test. In addition, the two groups of commits (before and after snapshots of the same file) are dependent on one another. Our null hypothesis is that there is no difference between our selected source code metrics before and after the readability commits. Our alternative hypothesis is that our selected metrics have significantly different values after readability commits. A significant p-value (<= 0.05) indicates that the metric values for the two populations are significantly different. We also report the percentage of datapoints where values of the metrics increases, decreases, or remain unchanged.

5) Cliff's Delta (d) Effect Size: Cliffs delta (d) effect size [11] is a non-parametric statistic estimating whether the probability that a randomly chosen value from a group is higher than a randomly chosen value from another group, minus the reverse probability. Values range between 1 and 1. When d = 1 there is no overlap between the two groups and all values from group 1 are greater than the values from group 2. When d = 1 there is again no overlap but all values from group 1 are lower than the values from group 2. When d = 0 there is a complete overlap between the two groups and thus there is no effect size. Between 0 and 1 the magnitude of the effect size is interpreted as follows: $0 \leq |d| < 0.147$: negligible, $0.147 \leq |d| < 0.33$: small, $0.33 \leq |d| < 0.474$: medium, $0.474 \leq |d| \leq 1$: large. We use Cliff's Delta to answer **RQ2**.

III. Results

In this section, we report the results of our study, with the aim of answering the research questions formulated in Section II.

A. *RQ1: Are state-of-the-art readability models able to capture readability improvements in practice?*

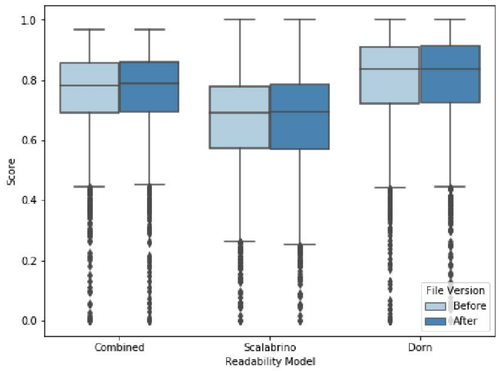

Fig. 1. Readability scores generated by Dorn's, Scalabrino's, and the Combined models for files before and after readability improvements.

Figure 1 contains the readability scores obtained by each of the three readability models. For Dorn's model, the mean readability score for files before and after changes is 0.793 and 0.798, respectively. This is a mean increase of only 0.005 between the two populations. The model indicates that there was a readability improvement in only 40.7% of the files. Considering Scalabrino's model, the mean readability score generated by this model for files before changes is 0.664, and 0.669 for files after changes. The mean readability scores generated by the model increases also by 0.005. The model indicates that there was a readability improvement in only 40.1% of the files. The combined model calculates a mean readability score for files before changes as 0.754 and 0.760 after changes are made by developers. This is a 0.006 point increase. The model indicates that there was a readability improvement in only 35.7% of the files.

Overall, the readability models seem to agree on whether a readability improvement exists for the majority of the files in our dataset. Scalabrino's and Dorn's models indicate an increase in readability for 667 of the same files, which is a 76% intersection. Similarly, for Scalabrino's and the Combined models, the models agree on 559 files which is a 64% intersection. For Dorn's and the Combined models we see 606 files and a 78% intersection.

We observe that all three readability models are unable to capture changes in the readability of source code for the majority of the changed files. This is similar to the results obtained by Pantiuchina et al. [22]; they find that the Scalabrino's model only reports 38%, compared to our 40.1%, of modified classes as improving readability. Similarly to what we can see in Figure 1, the difference in the distribution of scores for the Buse & Weimer's model for before and after files is negligible. One reason for this is that the source code snippets before any changes are made may not be of particularly poor readability and that the improvements made by the developers have a minimal affect on overall readability of a snippet. Note that all three readability models were evaluated by the original authors on datasets composed of source code snippets that have been flagged by external developers as being readable or not readable. Thus, the models may be better able to distinguish between extreme differences in the readability of code snippets.

For example, one change made by a developer to improve readability consisted of renaming a single identifier[2]. In this case, the score for all readability models stayed the same, before and after the change. However, there are also several cases where readability decreases after a change is made. For example, one commit consisted of formatting and regrouping the source code, but no functional change[3]. Scalabrino's model initially rated the code as 0.56 but then decreased to 0.46 after the change was made. Similarly, Dorn's model decreases by 0.04 points and the combined model by 0.01 points. Another change consisted of replacing conditions inside `if` statements with boolean variables[4]. The variable names are descriptive of the conditional statement, which makes it easier to understand the code. The models all register a very slight decrease in readability.

However, if we hope to create tools that help developers to improve the readability of the source code in practice, we must build models that are able to detect incremental improvement. Such models will be more useful to identify improvement opportunities when projects are maintained in day to day tasks. To this end we need to 1) identify which metrics change during readability improvements in practice and 2) identify the types of changes that developers make to improve the readability of their source code.

RQ$_1$ Summary: Our results show that readability models are unable to capture readability improvements during software maintenance for the majority of the changed files, which aligns with recent findings by Pantiuchina et al. [22]. Thus, there is a need for models that are able to capture incremental readability improvements which will be more applicable for measuring readability changes in day to day software maintenance tasks.

B. *RQ2: Which source code metrics capture improvement in the readability of source code, as perceived by developers?*

Using SourceMeter we explore changes in values between files before and after readability improvements for various metrics. Table I shows a list of metrics considered by the three readability models we use for this paper. We also indicate here whether these metrics are also supported using the static

[2] https://tinyurl.com/y3mrduqo

[3] https://tinyurl.com/yxbg9dxx

[4] https://tinyurl.com/yy5hcbel

TABLE I
METRICS CONSIDERED BY EACH OF THE INDIVIDUAL READABILITY MODELS USED. (CS) INDICATES THIS METRIC IS SUPPORTED BY CHECKSTYLE, (RM) REFACTORINGMINER, (CD) CHANGEDISTILLER, AND (SM) SOURCEMETER.

Buse & Weimer		Dorn (Features)		Scalabrino	Posnett
Line Length **(cs)**	# Spaces	Line Length **(cs)**	Spaces	Comment Identifier Consistency	Halstead's vocabulary **(sm)**
# Identifiers	# Assignments	Literals	Assignments	Identifier Terms in Dictionary	Token level entropy
Indentation	# Loops	Indentation **(cs)**	Loops	Narrow Meaning Identifiers	# Lines **(sm)**
# Keywords	# Arithmetic Operators **(sm)**	Keywords	Operators **(sm)**	Flesch-Kinacaid	
Identifier length	# Comparisons	Identifiers **(rm, cd)**	Comparison	Number of Meanings	
# Numbers	# Any char	Numbers	Expressions	Textual Coherence	
# Parentheses **(sm)**	# Any identifier	Parentheses	Periods	Number of Concepts	
# Comments **(sm)**	# Branches	Comments **(sm, rm)**			
# Commas	#Blank Lines **(cs)**	Commas			

analysis tools we use. Table II contains results, as calculated by SourceMeter, for selected metrics that capture similar aspects of the source code as the ones described in Table I. Table II also contains the p-values from the Wilcoxon's signed rank test, and the percentage of files for which the metric increased, decreased, and remained the same for readability and non-readability commits. We computed the Cliff's d to measure the effect size of those differences and for all metrics shown in Table II, and regarding readability commits, the effect size is negligible, except for Number of Incoming Invocations for which we observe a small effect size. First, we observe that the overall, the trends between readability and non-readability commits are very similar. One exception is Number of Incoming Invocations where the percentage of files where the values of the metric increase is 46.88% for readability commits and 26.60% for non-readability commits. Despite the fact that most metrics remained the same after a readability commit, interesting trends can still be seen in the remaining file pairs for which these metrics do change. For example, for Public Undocumented API, we can see that for 56% of the files, the metric didn't change, but we can see that it also decreased in 37% of the files, while only increasing in 5% of the files. In this case, we see that although Public Undocumented API doesn't change in the majority of the files, when it does change, it tends to decrease rather than increase.

As outlined in Table I, Posnett's model uses Halstead's vocabulary as part of the algorithm. On our dataset, Halstead Program Vocabulary is not significantly different before and after readability changes. Posnett et al. explored Halstead Difficulty and Effort during the creation of their model and found that neither metric contributed significantly to the model. However, we observe that Halstead Effort is significantly different between the two populations but with negligible effect size.

Lines of code for file level metrics is significant (with negligible effect size), with 31% of files having an increase in lines of code. However, method level lines of code is not significantly different. Both Dorn' and the Buse & Weimer model's use comments as metrics. SourceMeter calculates several documentation related metrics. Results show that documentation lines of code and comment density are not significantly different for files before and after readability changes. However, API documentation, public undocumented API and public documented API are all statistically significant (with negligible effect size). Public documented API and public undocumented API appear to have several false positives, as manual inspection of some samples revealed that these metrics change across a before-after pair, despite the absence of documentation insertions or removals.

Number of parentheses is also measured by SourceMeter, however there is no significant difference between the two populations.

Results also show that several complexity metrics decrease significantly between files before and after readability improvements are made. McCabes cyclomatic complexity (MCC), and nesting level changes indicate less complex source code after readability changes (negligible effect size). This means that certain complexity features might be important to incorporate into readability models. However, the Maintainability Index, which is calculated based on cyclomatic complexity, Halstead Volume, and lines of code mainly decreases. Maintainablity Index has been criticized for not being well defined as it is built on correlated metrics among others [31].

We see that coupling metrics, such as the Number of Incoming Invocations, are significantly different (with small effect size), and increase for a large part of the files (46.88%) after readability changes are made.

RQ$_2$ Summary: Results show that additional metrics that are currently not used by readability models such as the Number of Incoming Invocations increase significantly when readability is improved and for a larger number of files compared to non-readability commits. Such metrics could be used as complementary measures to measure readability.

C. RQ$_3$: What types of changes do developers perform during readability improvements?

In order to answer this research question we explore the types of changes developers perform using ChangeDistiller, CheckStyle, and RefactoringMiner.

1) ChangeDistiller: ChangeDistiller detected a total of 14,130 and 9,206 changes in the readability and non-readability commits, respectively. In Table III we report selected changes (grouped in four categories: delete, insert,

TABLE II
SOURCE METER RESULTS INDICATING THE PERCENTAGE OF FILES WITH AN INCREASE, DECREASE, AND NO CHANGE FOR EACH METRIC VALUE IN READABILITY AND NON-READABILITY COMMITS. SIGNIFICANT P-VALUES ARE MARKED WITH (*). THE HIGHEST PERCENTAGES ARE IN BOLD.

Category	**Metric**	**Readability Commits**				**Non-Readability Commits**			
		P-Value	**% Increase**	**% Decrease**	**% Equal**	**P-Value**	**% Increase**	**% Decrease**	**% Equal**
Complexity metrics	Halstead Difficulty	0.24	28.12	26.72	**45.14**	0.00 (*)	34.77	26.40	**38.74**
	Halstead Effort	0.00 (*)	28.40	30.08	**38.34**	0.00 (*)	**37.58**	26.32	31.54
	Halstead Program Vocabulary	0.85	28.13	25.22	**46.65**	0.00 (*)	36.09	22.52	**41.31**
	Maintainability Index	0.03 (*)	29.52	**37.17**	33.31	0.00 (*)	25.33	**39.32**	35.26
	MCC	0.00 (*)	8.76	14.00	**77.23**	0.00 (*)	19.04	12.42	**68.46**
	Nesting Level	0.00 (*)	7.59	11.44	**80.97**	0.02 (*)	13.82	10.35	**75.75**
	WMC	0.57	13.17	11.68	**75.14**	0.00 (*)	28.58	6.74	**64.68**
Documentation metrics	Documentation Lines of Code	0.51	8.15	8.09	**83.76**	0.00 (*)	10.71	4.32	**84.97**
	Comment Density	0.88	33.54	**36.07**	30.38	0.00 (*)	20.64	34.02	**45.34**
	API Documentation	0.00 (*)	11.17	42.61	**46.22**	0.00 (*)	10.62	38.77	**50.60**
	Public Undocumented API	0.00 (*)	5.33	37.86	**56.82**	0.00 (*)	9.33	34.02	**56.65**
	Public Documented API	0.00 (*)	2.23	**49.43**	48.34	0.00 (*)	3.11	43.26	**53.63**
Size metrics	# Parantheses	0.94	10.27	10.77	**78.96**	0.83	10.93	12.33	**76.66**
	File Lines of Code	0.00 (*)	31.04	23.71	**45.25**	0.00 (*)	**44.47**	13.82	41.71
	Method Lines of Code	0.53	24.94	24.72	**50.33**	0.00 (*)	30.55	20.53	**48.84**
Coupling metrics	Number of Incoming Invocations	0.00 (*)	46.88	3.07	**50.06**	0.00 (*)	26.60	0.69	**72.71**
	Response set For Class	0.00 (*)	23.20	8.99	**67.81**	0.00 (*)	22.97	5.87	**71.16**
Cohesion metrics	Lack of Cohesion in Methods 5	0.42	4.12	4.87	**91.01**	0.31	4.84	4.58	**90.59**

TABLE III
CHANGES DETECTED BY CHANGEDISTILLER. OVERALL PERCENT OF CHANGES IS CALCULATED ACROSS ALL DETECTED CHANGES; WITHIN GROUP CHANGES ARE CALCULATED PER CHANGE TYPE GROUP.

	Change	**Non-Readability**		**Readability**	
		Overall	**Group**	**Overall**	**Group**
Delete	REMOVED_OBJECT_STATE	1.19%	4.97%	1.45%	4.90%
	COMMENT_DELETE	1.24%	5.15%	1.27%	4.31%
	ALTERNATIVE_PART_DELETE	1.43%	5.96%	0.73%	2.46%
	REMOVED_FUNCTIONALITY	2.42%	10.07%	2.04%	6.89%
	STATEMENT_DELETE	16.93%	**70.42%**	22.05%	**74.50%**
Insert	COMMENT_INSERT	1.93%	4.65%	0.73%	3.84%
	ADDITIONAL_OBJECT_STATE	3.14%	7.55%	1.72%	9.07%
	ADDITIONAL_FUNCTIONALITY	4.68%	11.26%	2.87%	15.12%
	STATEMENT_INSERT	**27.07%**	**65.12%**	10.76%	**56.74%**
	PARAMETER_INSERT	0.95%	2.27%	1.06%	5.60%
Move	STATEMENT_ORDERING_CHANGE	1.68%	19.45%	2.87%	37.42%
	STATEMENT_PARENT_CHANGE	6.53%	**75.41%**	4.46%	**58.06%**
Update	METHOD_RENAMING	0.70%	2.70%	0.53%	1.21%
	DOC_UPDATE	0.96%	3.72%	1.27%	2.91%
	ATTRIBUTE_RENAMING	0.54%	2.11%	2.74%	6.26%
	CONDITION_EXPRESSION_CHANGE	3.49%	13.56%	5.87%	13.42%
	STATEMENT_UPDATE	18.67%	**72.59%**	**30.91%**	**70.61%**

move, and update) that are present in the largest number of files of readability commits.

Update changes make up around 40% of the total changes detected in files by ChangeDistiller. Statement updates are the most frequent type of the changes; they affect every single commit that we analyzed. Moreover, we observe that renaming (attribute, method, and parameter renamings) and documentation updates are among the most common types of update changes. On inspection, we found that a large percentage of the renamings were concerned with changing attribute names to comply with preexisting convention. For example, adding an ‘m’ prefix to signify private class attributes was a frequent renaming change (e.g., `hasValidTlvObject` to `mHasValidTlvObject`). Convention compliance changes also included changing attribute names from camel case to snake case (e.g., `shouldHelpOutWithUnnecessaryCastingOfLists()` to `should_help_out_with_unnecessary_casting_of_lists()`) and changing constant attributes from lower to upper case (e.g., `sMimeTypePriorityList` to `MIME_TYPE_PRIORITY_LIST`). Renaming is also used to give identifiers more descriptive names, for example, `col` is changed to `column` and `color1` to `startColor`. Documentation updates, which indicate a change in leading method or class comments are also popular changes. We sampled 50 of these documentation updates for manual investigation, which revealed that they were mostly formatting changes and minor edits to the language, but did not add any new information about the code block being documented. ChangeDistiller found also 18 documentation updates that concern deprecated methods or classes. However, ChangeDistiller classified 45 instances where a blank line was added to the code as a statement update.

Insert changes are the next most popular type of change, making up 30% of the changes. Statement insert (i.e., new lines added to a method) and additional functionality changes (i.e., new methods added to a class) are the most popular types of changes. Comment and documentation inserts are also popular changes, affecting 51 and 44 files respectively. For example, we found 4 instances where ChangeDistiller detected insertion of TODO comments.

Delete changes make up 25% of the detected changes. As in other groups, statement deletes affect the largest amount of files. Note that comment deletes also affect a similar number of files as comment inserts. We examined a sample of these files that had both comment inserts and deletes, and we found that ChangeDistiller detects a comment update as a comment delete and a comment insertion. Also, reorganizing code along with the comments, for example extracting code into a new method, also counts as a comment insert and a comment delete. ChangeDistiller found 140 instances where a blank line has been removed from the code. Moreover, deletion of TODO comments affected 54 different files. 62 out of 87 instances of deleted documentation has to do with deleting inherited documentation.

We compare the results of ChangeDistiller for the readability and non-readability commits and report the results in Table III. We observe several differences between the types of changes in the two datasets. For instance, the most frequent change detected for readability commits is statement updates followed by statement deletes. For the non-readability commits, the most frequent change detected is statement insert, followed by statement update. Results also show that non-readability commits have more added and removed functionality than readability commits. This could indicate that readability commits tend to preserve functionality and modify existing code, rather than introduce new statements. Readability and non-readability changes follow a similar trend when we consider the the most frequent types of changes within the 4 different change categories. However, considering move changes, the non-readability commits have substantially more statement parent changes than readability commits. In addition, statement ordering changes are seen often in readability commits. Overall, when performing readability improvements developers perform statement updates and reordering more often than in non-readability commits.

2) CheckStyle: Table IV contains the warnings that have the largest difference in percentage of files affected by a warning after readability improvements took place. Overall, we observe that for readability commits warnings get fixed, whereas for non-readability commits they tend to increase.

The AvoidStarImport warning is generated when import statements contain the '*' notation. The rationale behind this check is that using the '*' notation imports all classes from a package or static members from a class. This leads to high coupling between packages or classes, and is considered a poor practice. Results show that this warning affected 15.78% of files before readability changes, and only 7.8% after. This is a 7.82% decrease in the number of files affected by the warning, meaning developers update import statements during readability changes. In contrast, for non-readability commits we observe an increase of 1.24%.

WhitespaceAfter, WhitespaceAround, CommentsIndentation, RightCurly, NeedBraces, and ParenPad are all warnings that are generated by CheckStyle due to formatting and style issues in the code. Files that have been altered to improve readability are affected by less warnings related to formatting and style issues overall. NonEmptyAtClauseDescription checks to make sure that at-clause tags, like @param and @return, are followed by a description. We notice a 2.06% decrease in the number of files affected by this warning, this means that more documentation is written after these tags during readability improvements.

In recent work by Pantiuchina et al. [22], some instances of readability improvements not captured by the state of the art readability models had to do with the removal of magic numbers from the source code. We observe a 1.59% decrease in the number of files containing magic number warnings after readability improvements have been made, which supports their observation and indicates that static analysis tools appear to be complementary to readability models. Overall, readability improvements have the greatest impact on the following changes as detected by CheckStyle: import statements, formatting and style of the source code, and the occurrence of magic numbers.

TABLE IV
CHECKSTYLE WARNINGS IN READABILITY AND NON-READABILITY COMMITS.

Warning	Readability			Non-Readability		
	Before	After	Delta	Before	After	Delta
AvoidStarImport	15.78%	7.96%	↓ 7.82%	10.06%	11.3%	↑ 1.24%
WhitespaceAfter	34.17%	26.82%	↓ 7.35%	16.35%	18.62%	↑ 2.27%
WhitespaceAround	32.13%	25.87%	↓ 6.25%	19.56%	21.23%	↑ 1.67%
CommentsIndentation	17.36%	12.57%	↓ 4.79%	12.03%	12.71%	↑ 0.68%
UnusedImports	17.27%	14.42%	↓ 2.85%	13.14%	14.08%	↑ 0.94%
RightCurly	14.30%	12.05%	↓ 2.25%	10.32%	11.39%	↑ 1.07%
MagicNumber	36.58%	35.00%	↓ 1.59%	22.3%	25.39%	↑ 3.08%
NonEmptyAtclauseDescription	36.98%	34.92%	↓ 2.06%	6.85%	7.28%	↑ 0.43%
ParenPad	8.94%	7.16%	↓ 1.78%	12.2%	14.55%	↑ 2.35%
NeedBraces	23.26%	21.78%	↓ 1.48%	14.21%	15.24%	↑ 1.03%

3) RefactoringMiner: Table V contains the number of files affected by different refactorings as detected by RefactoringMiner for readability and non-readability commits. For readability commits, attribute and variable renamings are the most common type of refactorings; they are less frequent in non-readability commits. Renaming methods and parameters are also popular refactorings. Although the metrics defined in Scalabrino's model were designed to capture readability of source code lexicon, given the model's performance, changes related to identifier naming could be useful in improving the model's performance. We observe that refactorings such as extract method, inline method, parameterize variable, and rename class are also very frequent for readability commits and are almost absent in non-readability commits. The nature of the most frequent refactorings indicates that readability changes may have more to do with the quality of the source code lexicon, through the renaming of identifiers, than the refactoring of design and functional elements of the code. Overall, both lexical changes like renames, and structural changes like extract method and variable are important refactorings that take place during readability improvements.

RQ$_3$ Summary: When comparing readability and non-readability commits we observe that readability commits tend to fix problems that pertain to imports, white spaces, and braces whereas non-readability imports tend to introduce more of such problems. Moreover, refactorings such as extract method, inline method, parameterize variable, and rename class are very specific to readability commits and almost non-existent for non-readability commits. Thus, tools such as CheckStyle and RefactoringMiner can be used in addition to current readability models to detect fine grane readability changes.

TABLE V
REFACTORINGS DETECTED BY REFACTORINGMINER FOR READABILITY AND NON-READABILITY COMMITS.

	Readability		Non-Readability	
Refactoring	**#**	**%**	**#**	**%**
Extract And Move Method	6	0.72%	0	0.00%
Extract Class	1	0.12%	0	0.00%
Extract Method	124	**14.94%**	0	0.00%
Extract Operation	0	0.00%	30	**15.00%**
Extract Variable	37	4.46%	19	9.50%
Inline Method	68	**8.19%**	0	0.00%
Inline Operation	0	0.00%	1	0.50%
Inline Variable	15	1.81%	2	1.00%
Move Attribute	1	0.12%	0	0.00%
Move Class	1	0.12%	1	0.50%
Move Method	1	0.12%	0	0.00%
Parameterize Variable	18	2.17%	2	1.00%
Rename Attribute	183	**22.05%**	19	9.50%
Rename Class	31	3.73%	1	0.50%
Rename Method	96	**11.57%**	57	**28.50%**
Rename Parameter	73	8.80%	16	8.00%
Rename Variable	170	**20.48%**	50	**25.00%**
Replace Variable With Attribute	5	0.60%	2	1.00%

TABLE VI
MOST COMMON WORDS AND THEIR PERCENTAGE IN COMMIT COMMENTS.

Readability		Non-Readability	
Words	**% commits**	**Words**	**% commits**
code	24.32	fix	18.44
improve	21.96	change	15.84
test	17.06	bug	12.19
make	14.36	test	11.29
commit	13.35	use	10.97
method	12.16	commit	9.26
refactor	11.99	add	8.69
use	11.82	log	7.55
add	10.47	remove	7.47
fix	9.79	code	6.25
class	9.62	class	6.17
better	8.95	create	6.09
create	8.44	set	5.76
remove	7.43	update	5.44
move	7.40	support	5.44

D. *RQ4: What types of changes do developers describe in commit comments related to readability improvements?*

To answer this research question we explore the types of changes developers describe in commit comments while improving source code readability and we compare them to changes described in non-readability commits. In particular, we look into frequently used words, part of speech tagging, and grammatical dependencies on commit comments.

1) ***What are the most frequently used words?:*** Table VI contains the 15 most frequent words developers use and the percentage of commit comments in our dataset in which that particular word appeared.

Words such as 'improve', 'refactor', and 'better' indeed appear frequently in commits that improve readability (21.96%, 11.99%, and 8.95%, respectively), as one might expect, and have lower frequencies in commits that do not target readability improvements (1.46%, 4.63%, and 0.81%, respectively). Other words such as 'add', 'remove', 'create', 'class', and 'method' confirm results found in **RQ3**. As discussed in the results for RefactoringMiner in **RQ3** for readability commits, some of the most popular refactorings pertain to extracting, removing, and renaming methods or classes and using and adding/creating inline variables or methods. Examples of such commit messages are: *"removed deprecated methods of RenderEnvironment"*, *"Renamed a method for better readability"*, *"Add boolean variables for improved readability"*, *"Splits the main method in* `ClusterService` *into smaller chunks so that it's easier to understand and simpler to modify in subsequent PRs"*, and *"Minor internal refactor of* `InferJSDocInfo` *to make it more readable"*.

Attribute and variables may not appear in the top most frequent words in the readability commits because developers often refer to them by their explicit names in commit messages. For example the commit comment: *"Renaming* `CheckRequiresForConstructors` *to* `CheckMissingAndExtraRequires` *for readability's sake"*.

2) ***What types of actions are described, and on what aspects of source code?:*** For each verb tagged in the readability commit messages, we retrieve its most common verb-object pairs. In Table VII we show the 5 most frequent verbs found in our dataset for readability commits. For each verb, we show the frequency of the verb in our oracle, the 5 most frequent objects it was used with, and the frequency of the verb-object pair. As we can see from Table VII, developers clean the code, API, test and documentation/usage (e.g., *"cleaned up getDimensionValues code"*) while improving code, class, comment and stack readability (e.g., *"Improve readability of code"*) by adding new fields, documentation, tests etc. (e.g., *"Add a couple new tests for ES6 classes and modules, and reformat existing tests for readability"*). They also make new files, class, and examples (e.g., *"Make uninitialization code in DefaultChannel easier to understand"*) and use new methods, spaces, and lines in the code (e.g., *"Use fireChangeEvent() method to improve code readability"*) for improving source code readability.

To verify which types of changes, as discovered in **RQ3** for readability commits, developers document, we specifically look for the actions found by RefactoringMiner in commit messages. We analyze 5 such verbs and look for some insightful verb-object pairs for each of these verbs in our dataset. In the following we provide examples of pairs and the number of commits in which they occur in parentheses: remove (44 occurrences): 'method', 'test', 'class' (e.g., *"remove duplicate/unneeded methods and fields"*), fix (36 occurrences): 'error', 'bug', 'text' (e.g., *"Fix copy/-paste error Add whitespace to aid readability"*), refactor (28 occurrences): 'method', 'code', 'class' (e.g., *"refactor code to make it more readable, less invasive"*), move (25 occurrences): 'constructor', 'class', 'comment' (e.g., *"move test security classes from templates.security package to com.gemstone.gemfire.security.templates"*), and rename (16

TABLE VII
MOST FREQUENT VERBS WITH TOP 5 OF THEIR OBJECTS.

Verbs	Freq.	Objects (number of co-occurrences with verb)
clean	163	code (21), api (17), test (11), javadoc (6), usage (6)
make	107	code (2), commit (2), class (2), file (2), example (2)
improve	98	readability (76), code (4), class (4), comment (2), stack (2)
add	98	readability (9), test (7), javadoc (6), dialog (5), field (4)
use	54	readability (4), lambda (2), method (2), space (2), line (2)

occurrences): ‘method’, ’variable’, ‘example’ (e.g., *“Rename variables for readability”*). In the results for **RQ$_3$**, we found that the most popular code changes have to do with renaming, extracting, removing, and moving methods, variables and classes for readability commits. These findings further validates our results in **RQ$_3$** by showing that developers explicitly state in the commit messages that the purpose of those refactorings is to improve readability.

To gain more depth on the objects (e.g., when developers are commenting on classes what type of classes they are talking about or when developers are doing tests what type of tests they are performing), we retrieved the most common noun phrases each object was used with. In Table VIII we show the 5 most frequent objects found in our dataset. For each object, we show the frequency of the object in our oracle, the 5 most frequent noun phrases it was used with, and the number of times the noun phrases were used with the respective object. As we can see from Table VIII, developers are talking about changes in the main method (e.g., *“Splits the main method in ClusterService into smaller chunks so that it’s easier to understand and simpler to modify in subsequent PRs”*), removing duplicated code and reusing code (e.g., *“To make it easier to understand PoolChunk and PoolArena we should cleanup duplicated code”*) while improving readability. Also, they look into nested classes, inner classes, and utility classes (e.g., *“Clean-up after extracting nested classes from CallLogFragment”*) and do tests like distributed tests, connectivity tests, builder test, and compare test (e.g., *“Increase timeout for a flaky distributed test”*) while improving the source code readability.

RQ$_4$ Summary: By analyzing the commit messages related to readability commits we find that developers document the types of changes and refactorings that they perform to improve readability. We also found that keywords such as ‘improve’ and ‘refactor’ are more frequent in readability commits in comparison to non-readability commits. Thus, metrics based on commit messages can bring additional information not captured by existing readability models.

IV. THREATS TO VALIDITY

This section discusses the threats to validity that can affect our study. A common classification [32][33] involves five categories, namely threats to conclusion, internal, construct, external, and reliability validity.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. We used multiple statistical procedures, including measures for effect size which indicate the magnitude and significance of the results. We use the appropriate statistical tests for our data, without the assumption of normality or independence between groups.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. The biggest threat to internal validity is the experience of the developers who wrote the code and commit messages used in our dataset. Although we evaluated the commit messages manually, we can not be certain that the developers have sufficient understanding about what makes readable or unreadable source code, or that their perceptions of readability are generalizable to the developer community. To mitigate this threat we control the quality of the repositories used in our dataset by only using engineered projects.

Threats to *construct validity* concern the relation between theory and observation. One of the major threats to construct validity in this work pertains to the creation of the oracle. Misclassification of commits in which developers state readability improvements is possible. To mitigate this threat, two of the authors of this paper went through the set of commits to ensure that developers’ changes are indeed readability changes. In case of a doubt, the commit was excluded from the dataset. Misclassification of tangled commits is another threat. Both validators went through all 548 commits where the change involved more than one file and untangled commits manually. In case of a doubt (e.g., commits that were too tangled, or files that contained too many changes unrelated to readability improvements) commits were discarded. Another threat to construct validity are the metrics considered in the paper. We select a variety of static analysis tools to generate metrics on commit data before and after readability changes are implemented. However, the conclusion drawn depend on the accuracy of these tools, and the metrics they are able to collect. Different tools could lead to different results.

Threats to *external validity* concern the generalizability of the findings outside the experimental settings. Potential threats to external validity in this study include the selection of sampled open source applications, which may not be representative of the studied population. We extracted 548 instances of source code readability from 63 engineered Java software projects. A different dataset could lead to different conclusions.

Threats to *reliability validity* concern the ability to replicate a study with the same data and to obtain the same results. We use open-source software projects whose source code is available. Moreover, we provide all the necessary details to replicate the analysis in our online replication package [2].

V. RELATED WORK

A. Code Readability Models

Buse and Weimer [6] conduct a study investigating code readability metrics and find that structural metrics such as the number of branching and control statements, line length, the number of assignments, and the number of spaces negatively affect readability. They also show that metrics such as the num-

TABLE VIII
MOST FREQUENT OBJECTS WITH TOP 5 OF THEIR NOUN PHRASES.

Objects	Frequency	Phrases (number of co-occurrences with objects)
readability	173	readability (67), better readability (21), code readability (10), improved readability (4), readability of code (3)
code	57	code (19), the code (14), some code (2), duplicated code (1), reused code (1)
method	47	a method (4), methods (3), the method (3), the main method (2), some methods (2)
test	41	test (9), a flaky distributed test (2), connectivity tests (2), the builder test (2), canvas compare tests (2)
class	40	class (4), the class (4), nested classes (3), inner classes (2), stream items utility class (2)

ber of blank lines, the number of comments, and adherence to proper indentation practices positively impact readability.

Posnett et al. [24] show that metrics such as McCabe's Cyclomatic Complexity [18], nesting depth, the number of arguments, Halstead's complexity measures [13], and the overall number of lines of code impact code readability. An empirical evaluation conducted on the same dataset used by Buse and Weimer [6] indicates that the model by Posnett et al. is more accurate than the one by Buse and Weimer.

Scalabrino et al. [28] propose and evaluate 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). The model was evaluated on the two datasets previously introduced by Buse and Weimer [6] and Dorn [8] and on a new dataset, composed by 200 Java snippets, manually evaluated by nine developers. The results indicate that combining the features (i.e., structural and textual) improves the accuracy of code readability models.

In this paper, we do not intend to devise a new readability model rather we investigate three existing state of the art readability models i.e., the original model proposed by Scalabrino et al. [28], the model proposed by Dorn et al. [8], and the combined model proposed by Scalabrino et al. [28].

B. Code Quality Metrics in Practice

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 [15], [20], to recommend refactorings [19], [25], and to predict the code fault- and change-proneness [12], [17], [34]. Some of these applications assume that a strong link between code quality as assessed by metrics and as perceived by developers exists.

Scalabrino et al. [27] perform an extensive evaluation of 121 existing as well as new code-related ([6], [8], [24], [29]), documentation-related ([29] and 2 newly introduced), and developer-related (3 newly introduced) metrics. They try to (i) correlate each metric with understandability and (ii) build models combining metrics to assess understandability. To do this, they use 444 human evaluations from 63 developers and obtain a bold negative result: none of the 121 experimented metrics is able to capture code understandability, not even the ones assumed to assess quality attributes apparently related, such as code readability and complexity.

Indeed, code smell detectors and refactoring recommenders should be able to identify design flaws/recommend refactorings that align with developer's perception in practice. While such an assumption seems reasonable, there is limited empirical evidence supporting it. Pantiuchina et al. [22] 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 [4], [7], [26] surveyed developers to investigate whether metrics align with their perception of code quality, they 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. They use state-of-the-art metrics to measure the changes relative to the specific quality attribute it targets. To measure code readability, the authors exploit two state-of-the-art metrics. The first one was presented by Buse and Weimer et al. [6] and the second metric is the one proposed by Scalabrino et al. [29]. Code readability is the quality attribute for which the authors observed the less perceivable changes in the metrics' values. This holds for both metrics they employed, despite the metrics use totally different features when assessing code readability. The two metrics report only 28% [6] and 38% [29] of the modified classes as improving their readability after the changes implemented in the commits.

In contrast to previous work, which evaluates readability based on surveying techniques, we aim to understand whether state-of-the-art readability models are able to capture readability improvements as explicitly tagged by developers in commits messages.

VI. CONCLUSION

This paper presents a study on the theory and practice of measuring source code readability changes. In particular, we investigate three state of the art models and the degree to which they are able to detect source code readability improvements as tagged by open source developers of 63 engineered Java projects. Our results confirm recent findings that those models fail to capture readability improvements and thus do not appear to be suitable for day to day maintenance tasks. We also study additional metrics that are not considered by state of the art models and we detect changes and refactorings that developers use to implement readability improvements. We show examples of metrics and changes/refactorings that are successful in detecting readability improvements and thus must be considered when building readability models.

As part of the future work we plan to analyze in more details renamings and documentation changes and use tools such as REPENT [3] to classify the renamings.

References

[1] Checkstyle. http://checkstyle.sourceforge.net/, last accessed on March 15, 2019.

[2] Replication package. https://github.com/Smfakhoury/Improving-Source-Code-Readability-Theory-and-Practice.

[3] Venera Arnaoudova, Laleh Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering (TSE)*, 40(5):502–532, 2014.

[4] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers' perception of software coupling. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 692–701, 2013.

[5] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[6] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering (TSE)*, 36(4):546–558, July 2010.

[7] Steve Counsell, Stephen Swift, Allan Tucker, and Emilia Mendes. Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *ACM SIGSOFT Software Engineering Notes*, 31(5):1–10, September 2006.

[8] Jonathan Dorn. A general software readability model. Master's thesis, University of Virginia, 2012.

[9] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 35–45, 2006.

[10] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11), 2007.

[11] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach.* Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[12] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE)*, 31(10):897–910, October 2005.

[13] Maurice H Halstead. Elements of software science. 1977.

[14] Matthew Honnibal and Mark Johnson. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[15] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[16] FrontEndART Software Ltd. Sourcemeter. https://www.sourcemeter.com/, last accessed on March 15, 2019.

[17] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering (TSE)*, 34(2):287–30, 2008.

[18] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, SE-2(4):308–320, 1976.

[19] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[20] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36, January-February 2010.

[21] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

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

[23] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[24] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 73–82, 2011.

[25] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[26] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering (EMSE)*, 16(6):773–811, December 2011.

[27] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[28] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[29] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[30] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. pages 483–494, 2018.

[31] Arie van Deursen. Think twice before using the 'maintainability index'. https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/.

[32] Claes Wohlin, Per Runeson, Höst Martin, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering - An Introduction.* Kluwer Academic Publishers, 2000.

[33] Robert K. Yin. *Case Study Research: Design and Methods.* Sage Publications, 2nd edition, 1994.

[34] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 531–540. ACM, 2008.