
Università degli Studi del Molise

Dipartimento di Bioscienze e Territorio

Corso di Dottorato in Bioscienze e Territorio
Ciclo XXXI
S.S.D. ING-INF/05

PhD Thesis

# Automatically Assessing and Improving Code Readability and Understandability

Coordinatore

Chiar.ma Prof.ssa
Gabriella Stefania Scippa

Relatore/Tutor

Chiar.mo Prof.
Rocco Oliveto

Candidato
Simone Scalabrino

Anno Accademico 2018/2019

**Abstract**

Reading and understanding code is an inherent requirement for many maintenance and evolution tasks. Developers continuously read the code and they make an effort to understand it before being able to perform any task. Without a thorough understanding of source code, developers would not be able to fix bugs or add new features timely.

Automatically assessing code readability and understandability can help in estimating the effort required to modify code components. Besides, having an objective metric for such aspects could be important to improve the quality of automatically generated test cases.

In this thesis, we improve the accuracy of existing readability models by introducing textual features. Besides, we try to go further, and we use a large number of new and state-of-the-art metrics to automatically assess code understandability. However, our results show that it is still not possible to automatically assess the understandability of source code.

In the context of software testing, we introduce a new metric, namely Coverage Entropy, which is aimed at estimating the understandability of a test case. We use Coverage Entropy in TERMite, a novel test case generation technique aimed at improving the understandability of generated tests. Our results show that TERMite generates a lower number of eager tests compared to the state-of-the-art and it also improves other quality aspects, such as test cohesion.

# Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

Software developers read code all the time. The very first step in each software evolution and maintenance task is carefully reading and understanding code, even when the maintainer is the original author. Developers spend much time reading code, far more than writing it from scratch [46, 102].

Furthermore, incremental change [17, 120, 118], which requires to perform concept location, impact analysis, and the corresponding change implementation/propagation, needs a prior code reading step before it can take place.

Reading, however, is just what developers do aiming to *understand* the code. Despite they are highly entangled, reading and understanding are two different concepts, and so are readability and understandability: on one hand, *readability* regards the form of the code, *i.e.,* how the code is able to convey information about the concepts implemented in it to the reader; *understandability*, instead, regards the substance, *i.e.,* the very nature of those concepts. If the code is readable it is easier to acquire the information necessary to understand it, while if the code is understandable it is easier to process such information. Modifying unreadable code is like assembling a piece of furniture using a manual written in a

foreign language: the task is not impossible, but more difficult, and a few screws still may remain unused. Trying to modify not understandable code, instead, is like being a kid willing to fix a car: even using a good (readable) manual, there are few chances to succeed.

Focusing on code, to better get the difference between the two concepts, consider the snippet below:

```
AsyncHttpClient client = new AsyncHttpClient();
String cookies = CookieManager.getInstance().getCookie(url);
Log.e(TAG, cookies);
client.addHeader(SM.COOKIE, cookies);
```

Any developer would consider such code as *readable*, since it is concise and it uses meaningful identifier names. Nevertheless, this snippet of code is not necessarily easy to understand for any given developer, because the used APIs could be unknown to her and even poorly documented. For example, the developer may not understand the implications of the `getCookie(url)` method call without prior experience using the API or without reading the documentation, *e.g.,* she might not know whether `getCookie(url)` could throw an exception, return a `null` value, or produce some other side effects.

Several facets, like complexity, usage of design concepts, formatting, source code vocabulary, and visual aspects (*e.g.,* syntax highlighting) have been widely recognized as elements that impact program understanding [96, 105, 15]. Only recently, automatic code *readability* estimation techniques started to be developed and used in the research community [21, 119, 43]. Such models provide a binary classification (*readable* or *unreadable*) for a given piece of code. On the other hand, there is no empirical foundation suggesting how to objectively assess the *understandability* of a given piece of code. Indeed, our knowledge of factors affecting (positively or negatively) code understandability is basically tied to *common beliefs* or it is focused on the cognitive process adopted when understanding code [136, 137]. For example, we commonly assume that code complexity can be used to assess the effort required to understand a given piece of code. However, there is no empirical evidence that this is actually the case.

Improving the performance of the existing code readability models would be important to help planning code cleaning activities, focusing only on the code that actually needs it with a higher confidence. More importantly, having a model

that estimates the effort required to understand a given piece of code would have a strong impact on several software engineering tasks. For example, it would be possible to use such a model to (i) improve the estimation of the time needed to fix a bug (the lower the understandability, the higher the time to comprehend the code and thus to fix the bug); (ii) create search-based refactoring recommender systems using the predicted code understandability as a fitness function; or (iii) assess the quality of code changes during code reviews.

As previously mentioned, the importance of both code readability and, above all, code understandability is undisputed for maintenance-related activities [3, 141, 24, 121, 135, 31]. However, developers tend to neglect such aspects for test code [61], even if software testing is one of the most important and expensive software development activities [16]. The poor quality of test cases represents a threat to their evolution. Indeed, test cases are not written in stone; they need to change over time along with the source code they exercise [114, 103]. A change in a feature might break some tests. In some cases, a new feature changes the behavior of the code and some tests are not valid anymore. On the other hand, changes might introduce bugs that are revealed by tests. Therefore, developers need to detect the cause of a failure and either fix the code or update the test [114]. In both the cases, understanding the failing test is the first step to perform any kind of maintenance task.

In the last years, automated generation of test data and test cases have been widely investigated [99, 98]. The proposed approaches aim at reducing the costs of testing. In particular, search-based test case generation techniques proved to be effective to achieve this goal [99]. Such approaches use meta-heuristics to select the most effective tests in a search space composed by all the possible tests. Most of the effort of the research community has been devoted to improve the effectiveness of automatically generated tests (*e.g.,* code coverage and mutation score) [55, 98, 111, 25]. However, the generated tests generally suffer from poor quality: they are even less readable than manually written ones [61, 113] and they contain a larger number of test smells [107]). Improving the understandability and, in general, the quality of generated tests would be of primary importance to promote the usage of such test case generation techniques.

In this thesis, we introduce approaches and tools aimed at measuring and improving readability and understandability of both source code and test code. In Part I we provide a literature review on the topic. Specifically, in Chapter 2 we report the work about code readability and understandability, while in Chapter 3 we first provide background information about search-based test case generation and, then, we report the related work on the quality of generated tests.

In Part II we report our work on automatically assessing and improving source code readability and understandability. Specifically, in Chapter 4 we introduce textual features for improving the accuracy of readability prediction models. Then, in Chapter 5 we describe our study about the correlation between code readability and the presence of FindBugs warnings, showing that an improved readability estimation results in a higher correlation. Given the importance of textual features and, specifically, of identifiers for code readability, in Chapter 6 we introduce an approach that suggests identifier renaming operations aimed at improving the naming consistency in a given project. Finally, in Chapter 7 we make the first step to go beyond code readability assessment and we try, for the first time, to automatically assess code understandability.

In Part III we report our work on automatically assessing and improving the understandability of generated test cases. Specifically, in Chapter 8 we introduce a metric, namely Coverage Entropy, aimed at estimating the focus of a given test case and, thus, its understandability. Then, in Chapter 9 we introduce TERMite, a novel approach to automatically improve the focus of tests generated using search-based approaches.

Finally, in Chapter 10 we conclude this thesis, providing a summary of the contributions and the achieved results.

# Part I

# Background

*If I have seen further it is by standing on the shoulders of Giants.*

Isaac Newton, letter to Robert Hooke (1675)

# CHAPTER 2

# Readability and Understandability of Source Code

**Contents**



## 2.1 Source Code Readability

In the next sub-sections we highlight the importance of source code vocabulary for software quality; in addition, we describe state-of-the-art code readability models. To the best of our knowledge, three different models have beed defined in the literature for measuring the readability of source code [21, 119, 43]. Besides estimating the readability of source code, readability models have been also used for defect prediction [21, 43].

### 2.1.1 Software Quality and Source Code Vocabulary

Given a source code snippet $s$, the *vocabulary* of $s$ is the set of terms used in it; such a set contains words from both identifiers and comments. The source code vocabulary play a crucial role in program comprehension and software quality since developers express domain knowledge through the names they assign to the identifiers of a program (*e.g.,* variables and methods) and through the comments they add to make it more clear to future readers [86, 85, 29, 83, 45]. For example, Lawrie *et al.* [83] showed that identifiers containing full words are more understandable than identifiers composed of abbreviations. From the analysis of source code identifiers and comments it is also possible to glean the "semantics" of the source code. Consequently, identifiers and comments can be used to measure the conceptual cohesion and coupling of classes [95, 117], and to recover traceability links between documentation artifacts (*e.g.,* requirements) and source code [6]. Butler *et al.* [23] showed that there is a relationship between the quality of names chosen for identifiers and the source code quality. Moreover, given the importance of linguistic elements in source code, Arnaoudova *et al.* [8, 7] defined the concept of linguistic antipatterns. Linguistic antipatterns are poor solutions to recurring problems that developers adopt when defining identifiers, such as method names, or when commenting the code. For example, the antipattern "Get method does not return" occurs when the name of a method without return type starts with "get", conventionally used to acquire pieces of information about an instance of a class.

Although the importance of meaningful identifiers for program comprehension is widely accepted, there is no agreement on the importance of the presence of comments for increasing code readability and understandability. Also, while previous studies have pointed out that comments make source code more readable [44, 139, 134], the more recent study by Buse and Weimer [21] showed that the number of commented lines is not necessarily an important factor in their readability model. However, the consistency between comments and source code has been shown to be more important than the presence of comments, for code quality. Binkley *et al.* [19] proposed the QALP tool for computing the textual similarity between code and its related comments. The QALP score has been shown to correlate with human judgements of software quality and is useful

| Feature | Avg | Max |
|---|---|---|
| Line length (characters) | ▼ | ▼ |
| N. of identifiers | ▼ | ▼ |
| Indentation (preceding whitespace) | ▼ | ▼ |
| N. of keywords | ▼ | ▼ |
| Identifiers length (characters) | ▼ | ▼ |
| N. of numbers | ▼ | ▼ |
| N. of parentheses | ▼ | |
| N. of periods | ▼ | |
| N. of blank lines | ▲ | |
| N. of comments | ▲ | |
| N. of commas | ▼ | |
| N. of spaces | ▼ | |
| N. of assignments | ▼ | |
| N. of branches (if) | ▼ | |
| N. of loops (for, while) | ▼ | |
| N. of arithmetic operators | ▲ | |
| N. of comparison operators | ▼ | |
| N. of occurrences of any character | | ▼ |
| N. of occurrences of any identifier | | ▼ |

Table 2.1: Features used by Buse and Weimer's readability model.

for predicting faults in modules. Specifically, the lower the consistency between identifiers and comments in a software component (*e.g.,* a class), the higher its fault-proneness [19]. Such a result has been confirmed by Ibrahim *et al.* [71]; the authors mined the history of three large open source systems observing that when a function and its comment are updated inconsistently (*e.g.,* the code is modified, whereas the related comment is not updated), the defect proneness of the function increases. Unfortunately, such a practice is quite common since developers often do not update comments when they maintain code [51, 93, 88, 94, 92, 37].

### 2.1.2 Source Code Readability Models

Buse and Weimer [21] proposed the first model of code readability and provided evidence that a subjective aspect like readability can be actually captured

and predicted automatically. The model operates as a binary classifier, which was trained and tested on code snippets manually annotated (based on their readability). Specifically, the authors asked 120 human annotators to evaluate the readability of 100 small snippets. In total, their dataset is composed by 12,000 human judgements. The features used by Buse and Weimer to predict the readability of a snippet are reported in Table 2.1: the triangles indicate if the feature is positively (up) or negatively (down) correlated with high readability, while the color indicates the predictive power (green = "high", yellow = "medium", red = "low"). It is worth noting that the features consider only structural aspects of source code. The model succeeded in classifying snippets as "readable" or "not readable" in more than 80% of the cases. From the 25 features, *average number of identifiers*, *average line length*, and *average number of parentheses* were reported to be the most useful features for differentiating between readable and non-readable code. Table 2.1 also indicates, for each feature, the predictive power and the direction of correlation (positive or negative).

Posnett *et al.* [119] defined a simpler model of code readability as compared to the one proposed by Buse and Weimer [21]. The approach by Posnett *et al.* uses only three features: *lines of code*, *entropy*, and *Halstead's Volume metric*. Using the same dataset from Buse and Weimer [21], and considering the Area Under the Curve (AUC) as the effectiveness metric, Posnett *et al.*'s model was shown to be more accurate than the one by Buse and Weimer.

Dorn introduced a "general" model, which relies on a larger set of features for code readability. Such features are organized into four categories: *visual*, *spatial*, *alignment*, and *linguistic* [43]. The rationale behind the four categories is that a better readability model should focus on how the code is read by humans on screens. Therefore, aspects such as syntax highlighting, variable naming standards, and operators alignment are considered by Dorn [43] as important for code readability, in addition to structural features that have been previously shown to be useful for measuring code readability. Table 2.2 reports all the features introduced by Dorn [43] and it maps categories to individual features. The four categories of features used in Dorn's model are described as follows:

- **Visual features**: In order to capture the visual perception of the source code, two types of features are extracted from the source code (including

| Feature | Visual | Spatial | Alignment | Textual |
|---|---|---|---|---|
| Line length | • | | | |
| Indentation length | • | | | |
| Assignments | • | | | |
| Commas | • | | | |
| Comparisons | • | | | |
| Loops | • | | | |
| Parentheses | • | | | |
| Periods | • | | | |
| Spaces | • | | | |
| Comments | • | • | | |
| Keywords | • | • | | |
| Identifiers | • | • | | • |
| Numbers | • | • | | |
| Operators | • | • | • | |
| Strings | | • | | |
| Literals | | • | | |
| Expressions | | | • | |

Table 2.2: Features defined by Dorn.

syntax highlighting and formatting provided by an IDE) when represented as an image: (i) a ratio of characters by color and colored region (e.g., comments), and (ii) an average bandwidth of a single feature (e.g., indentation) in the frequency domain for the vertical and horizontal dimensions. For the latter, the Discrete Fourier Transform (DFT) is computed on a line-indexed series (one for each feature). For instance, the DFT is applied to the function of indentation space per line number.

- **Spatial features**: Given a snippet $S$, for each feature $A$ marked in Table 2.2 as "Spatial", it is defined as a matrix $M^A \in \{0,1\}^{L \times W}$, where $W$ is the length of the longest line in $S$ and $L$ is the number of lines in $S$. Each cell $M^A_{i,j}$ of the matrix assumes the value 1 if the character in line $i$ and column $j$ of $S$ plays the role relative to the feature $A$. For example, if we consider the feature "comments", the cell $M^C_{i,j}$ will have the value "1" if the character in line $i$ and column $j$ belongs to a comment; otherwise, $M^C_{i,j}$ will be "0". The matrices are used to build three kind of features:
  - Absolute area ($AA$): it represents the percentage of characters with the role $A$. It is computed as: $AA = \frac{\sum_{i,j} M^A_{i,j}}{L \times W}$;
  - Relative area ($RA$): for each couple of features $A_1, A_2$, it represents the quantity of characters with role $A_1$ with respect to characters with role $A_2$. It is computed as: $RA = \frac{\sum_{i,j} M^{A_1}_{i,j}}{\sum_{i,j} M^{A_2}_{i,j}}$;
  - Regularity: it simulates "zooming-out" the code "until the individual letters are not visible but the blocks of colors are, and then measuring the relative noise or regularity of the resulting view"[43]. Such a measure is computed using the two-dimensional Discrete Fourier Transform on each matrix $M^A$.

- **Alignment features**: Aligning syntactic elements (such as "=" symbol) is very common, and it is considered a good practice in order to improve the readability of source code. Two features, namely operator alignment and expression alignment, are introduced in order to measure, respectively, how many times the operators and entire expressions are repeated on the same column/columns.

- **Natural-language features**: For the first time, Dorn introduces a textual-based factor, which simply counts the relative number of identifiers composed by words present in an English dictionary.

The model was evaluated by conducting a survey with 5K+ human annotators judging the readability of 360 code snippets written in three different programming languages (*i.e.,* Java, Python and CUDA). The results achieved on this dataset showed that the model proposed by Dorn achieves a higher accuracy as compared to the Buse and Weimer's model re-trained on the new dataset [43].

Summarizing, existing models for code readability mostly rely on structural properties of source code. Source code vocabulary, while representing a valuable source of information for program comprehension, has been generally ignored for estimating source code readability. Some structural features, such as the ones that measure the number of identifiers, indirectly measure lexical properties of code, such as the vocabulary size. However, only Dorn provides an initial attempt to explicitly use such valuable source of information [43] by considering the number of identifiers composed of words present in a dictionary.

## 2.2 Source Code Understandability

Previous studies focused on understanding the factors affecting program comprehension and the process used by developers. Lawrance *et al.* [79, 81, 80] showed that information foraging models are able to explain the paths walked by developers during code comprehension for maintenance and debugging tasks. Avidan *et al.* [10] showed that the identifiers used for the parameter names influence comprehension more than the ones used for local variables.

Other previous studies introduced metrics for measuring understandability (at system level) as a single quality attribute and as part of a quality model.

While readable code might directly impact program comprehension, code readability metrics are not sufficient to measure to what extent the code allows developers to understand its purpose, relationships between code entities, and the latent semantics at the low-level (*e.g.,* statements, beacons, motifs) and high-level structures (*e.g.,* packages, classes). Program understanding is a non-trivial mental process that requires building high-level abstractions from code

statements or visualizations/models [137, 24]. There have been several metrics designed to evaluate software understandability by focusing on complexity as well as source-level metrics.

Lin *et al.* [24] proposed a model for assessing understandability by building an understandability matrix from fuzzy maximum membership estimation for population of fog index, comment ratio, the number of components, CFS, Halstead Complexity, and DMSP. The authors then used PCA and factor analysis to get the weights for the column vectors, which can be multiplied by the matrix to get the Synthesis Vector of Understandability. Finally, the understandability is calculated by using the fuzzy integral. The authors did not empirically evaluate the proposed metric.

Misra and Akman [104] performed a comparative study between existing cognitive complexity measures and their proposed measure: cognitive weight complexity measure (CWCM), which assigns weights to software components by analyzing their control structures. The authors performed a theoretical validation of these metrics based on the properties proposed by Weyuker [148]. They found that only one metric, Cognitive Information Complexity Measure (CICM), satisfied all nine properties, while the others satisfied seven of the nine.

Thongmak *et al.* [141] considered aspect-oriented software dependence graphs to assess understandability of aspect-oriented software, while Srinivasulu *et al.* [135] used rough sets and rough entropy (to filter outliers) when considering the following metrics: fog index, comment ration, the number of components, CFS, Halstead Complexity, and DMSC. These metrics are computed at system level for nine projects, and subsequently the rough entropy outlier factor was calculated for the metrics to identify the outliers, which correspond to either highly understandable or not understandable software based on the metric values.

Capiluppi *et al.* [28] proposed a measure of understandability that can be evaluated in an automated manner. The proposed measure considers: (i) the percentage of micro-modules (*i.e.,* the numbers of files) that are within the macro-modules (*i.e.,* the directories), and (ii) the relative size of the micro-modules. The authors calculated the proposed measure on the history of 19 open source projects, finding that understandability typically increased during the life-cycle of the systems. Yet, no evaluation is provided for such a measure.

Understandability has also been a factor in quality models to assess software maintainability. Aggarwal *et al.* [3] investigated the maintainability of software and proposed a fuzzy model, which is composed of three parts: (i) readability of code, (ii) documentation quality, and (iii) understandability of the software. To quantify understandability, the authors utilize a prior work that defines language of software as the symbols used, excluding reserved words. The authors constructed rules based on the ranges of the three factors to determine maintainability.

Similarly, Chen *et al.* [31] investigated the COCOMO II Software Understandability factors by conducting a study with six graduate students asked to accomplish 44 maintenance tasks, and found that higher quality structure, higher quality organization, and more self-descriptive code were all correlated with less effort spent on the tasks, which leads to high maintainability.

Bansiya and Davis [12] proposed a model where metrics are related to several quality attributes, including understandability. In terms of understandability, the model considers encapsulation and cohesion to have positive influences, while abstraction, coupling, polymorphism, complexity, and design size have a negative influence. The authors validated the model by analyzing several versions of two applications and found that understandability decreases as a system evolves with many new features. Additionally, 13 evaluators analyzed 14 versions of a project and the authors found a correlation between the evaluators' overall assessment of quality and the models assessment for 11 out of 13 evaluators.

Kasto and Whalley [74] analyzed the performance of 93 students in their final examination for the Java programming course and they correlated their results with five metrics. They show that such metrics can be used to automatically assess the difficulty of examination questions.

Several studies have explored software understandability and program comprehension with either students or practitioners. Shima *et al.* considered the understandability of a software system by assessing the probability that a system can be correctly reconstructed from its components [132]. The authors asked eight students to reconstruct a system and the results suggest that faults tend to occur in hard to understand files or very simple files. Roehm *et al.* performed an observational study with 28 developers to identify the steps developers perform

when understanding software and the artifacts they investigate [121]. The authors found that developers are more inclined towards relying upon source code as well as discussing with colleagues over utilizing the documentation. The authors also identified some behaviors that improve comprehension, such as consistent naming conventions or meaningful names.

Understandability has been mostly analyzed from the perspective of (i) the quality attribute at the software level, *i.e.,* understandability as the "*The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use*" [72]; and (ii) the theories, challenges, and models for program understanding at cognitive levels [137, 136]. However, as of today, we still lack models for assessing code understandability at snippet-level, similarly to code readability. The only work we found that relates to a code understandability model is based on complexity and size source code level metrics [24, 91].

# CHAPTER 3

# Readability and Understandability of Generated Test Code

## Contents



Source code and test code are different. The former is more abstract, it aims at solving a problem in general, not just one of its instances. The latter, instead, should be very concrete: a test case is aimed at providing evidence that the source code does not work, and this can be done only specifying precise input data. Previous work, indeed, tend to distinguish quality problems affecting source code and test code. For example, test smells [146] were defined as code smells [52] that can affect test code. In general, test code can be affected by different problems regarding program comprehension with respect to source code. For this reason, previous work introduced techniques specifically aimed at improving the understandability of test code [39].

Quality problems are particularly present in automatically generated test cases [107, 61]. For this reason, in this chapter we first introduce search-based

test case generation and then we provide details about the the related work on test case quality.

## 3.1 Search-Based Test Case Generation

Many tools have been proposed to automate the generation of test cases in last decade [106, 53, 125]. Among them, EvoSuite [53], a tool that automatically generates test suites for Java classes, has been exploited in a plethora of studies [56, 57, 54]. EvoSuite implements several search-based test generation strategies, such as whole suite generation [55], MOSA [111], and LIPS [125] Moreover, it also implements other techniques, such as dynamic symbolic execution [127, 59].

EvoSuite uses genetic algorithms (GA) [60] to evolve populations of test cases. During the evolutionary process, the population of candidate solutions (*i.e.,* test cases) is evolved using operators imitating natural evolution: first, a *selection* operator is used to select the best test cases as for their fitness value; then, it is used a *crossover* operator to mix couples of test cases from the resulting population; finally, random *mutations* are performed to the test cases to add diversity to the population.

The fitness value of a test case depends on the test generation strategy used and on the chosen coverage goals. For branch coverage, it is usually defined in terms of a combination of *branch distance* and/or *approach level* [98] computed on one or multiple branches of the class under test. Besides, EvoSuite can be used to generate tests that satisfy other coverage criteria, such as exception and method coverage [53].

EvoSuite is able to evolve either entire test suites or single test cases. A test case is represented as sequences of statements containing (i) variable declarations/definitions and (ii) method calls, using the chromosome representation defined by Tonella [142]. Therefore, each statement of a test case can define new variables and use variables defined in previous statements. For example, the statement `a.push(x)` uses the variable `a` and `x`, and defines no variables (*i.e.,* the method `push` does not have a return value). Test cases can have arbitrary length.

Some statements can be inserted to exercise the class under test, while others are needed to set up complex objects.

Test cases evolved by EvoSuite do not contain oracles. However, the tool allows one to automatically add them at the end of the generation process: it captures the status of the objects in the tests and it adds assertions based on such values. Such assertions are particularly useful for regression testing, *i.e.,* to find variations in the behavior of the class under test (CUT) from a version to another [149]. Given a generated unit test, a large number of potential assertions can be generated. The choice of such assertions is very important: presenting all the assertions to the developer is problematic, as there might be too many of them, and many of them would be irrelevant [146]. In order to determine the important and effective assertions, EvoSuite applies mutation testing [53]: After the test case generation process, EvoSuite runs each test case on the unmodified software as well as all mutants that are covered by the test case, while recording information on which mutants are detected by which assertions. Then, it calculates a reduced set of assertions that is sufficient to detect all the mutants of the unit under test that the given test case may reveal.

*Whole Suite Generation* (WS) is a test case generation strategy defined by Fraser and Arcuri [55]. Such a strategy consists in evolving entire test suites to optimize the total coverage of the test suite. More specifically, the fitness function is defined as:

$$\mathit{fitness}(T) = |M| - |M_T| + \sum_{b \in B} d(b, T)$$

where $M$ is the set of all the methods of the class under test, $M_T$ is the set of executed methods, $B$ is the set of branches in the class under test and $d(b, T)$ is defined as follows:

$$d(b,T) = \begin{cases} 0, & \text{if } b \text{ was covered} \\ v(d_{min}(b,T)), & \text{if the predicate has been executed} \\ 1, & \text{otherwise} \end{cases}$$

In such a formula, $d_{min}$ computes the minimum branch distance, which is useful when the predicate is executed more than once, while $v$ is a normalization function ($\frac{x}{x+1}$). The genetic algorithm aims to find the test suite $T$ with minimum

*fitness*$(T)$ in order to cover the highest number of branches in the class under test.

*Many Objective Sorting Algorithm* (MOSA) is a test generation strategy defined by Panichella *et al.* [111]. Test cases evolved by MOSA are evaluated using the branch distance and approach level for each branch in the CUT. The fitness function used is represented as a vector $f$ of $n$ objectives, where $n$ is the number of branches of the CUT. Each element $f_i$ of the fitness function is the sum of branch distance and approach level of branch $b_i$. MOSA tries to find solutions (*i.e.,* test cases) that separately cover the target branches, *i.e.,* tests $T$ having a fitness score $f_i(T) = 0$ for at least one uncovered branch $b_i$.

MOSA works as follows. It randomly generates a starting population of test cases and evolves them to find the set of test cases that are able to cover the largest number of branches. At the beginning of each generation, MOSA generates new offsprings using crossover and mutation operators on the current population. Then, it creates the population for the next generation by selecting tests among the union of parents and offsprings as follows: it builds a first front $F_0$ of test cases using the *preference criterion*. Formally, a test case $T_a$ is preferred over another test $T_b$ for a given branch $b_i$ if $f_i(T_a) < f_i(T_b)$. This is called *main criterion*. In addition, if the two test cases are equally good in terms of branch distance and approach level for the branch $b_i$ (*i.e.,* $f_i(T_a) = f_i(T_b)$), the shorter test is preferred. This is called *secondary criterion*. Then, the remaining tests are grouped in subsequent fronts $F_1, \ldots, F_k$ using the non-dominated sorting algorithm [111]. MOSA composes the population used in the next generation picking tests in order of fronts (*i.e.,* first from $F_0$, than from $F_1$, and so on) until it reaches a fixed population size $M$. MOSA relies on the crowding distance as a secondary selection criterion to increase the diversity of the population. MOSA uses an archive to store test cases that cover previously uncovered branches. When a new test that covers an already covered branch is found, MOSA uses the secondary criterion to chose which one to keep between the one in the archive and the new one.

## 3.2 Test Code Quality

Few studies directly focus on test case readability and understandability. While it could be argued that code readability is not related to the type of code, since it regards the form of the code (*e.g.,* long lines of code negatively affect readability in both source and test code), the same is not true for code understandability. Indeed, previous studies specifically addressed the problem of the quality of test cases such as *test smells*. In general, code smells are indicators for design flaws with respect to the maintainability of a software system [52], while test smells are code smells that regard test cases [146]. Test smells are not desirable because they reduce the maintainability of the test cases and, thus, their understandability. Test code has a distinct set of smells that relate to the ways in which test cases are organized, how they are implemented, and how they interact with each other. Van Deursen *et al.* [146] identified eleven static test code smells. Such smells refer to tests making inappropriate assumptions on the availability of external resources (*Mystery Guest* and *Resource Optimism*), tests that are long and complex (*General Fixture*, *Eager Test*, *Lazy Test*, *Indirect Testing*), tests containing bad programming decisions (*Assertion Roulette* and *Sensitive Equality*), and tests exposing signs of redundancy (*Test Code Duplication*).

Palomba *et al.* [107] conducted an empirical investigation on the diffuseness of tests smells in test cases automatically generated by EvoSuite [53]. They found the 83% of the tests to be affected by at least one smell: among the others, *Eager Test* was present in about a third of the generated classes. This smell occurs when a test case exercises more than a single functionality [146]. Best practices suggest that maintainable tests should not verify many functionalities at once, in order to avoid test obfuscation [100]. Instead, test suites should be structured following the *Single-Condition Tests* principle, as this provides better defect localization [100] and, thus, understandability.

Besides being affected by a higher number of test smells, generated test cases are also affected by other quality issues. Grano *et al.* [61] reported that automatically generated test cases are significantly less readable than manually written ones. Also, Shamshiri *et al.* [129] conducted a large empirical study in which they

showed that developer take longer to understand automatically generated tests compared to manually written ones.

Some steps were taken to tackle the problem of the low quality of automatically generated tests. Daka *et al.* [38] proposed a domain-specific model of test readability and an algorithm for producing more readable tests. They found that: (i) their model outperforms previous work in terms of agreement with humans on test case readability; (ii) their approach generates tests that were 2% more readable, on average; (iii) humans prefer their optimized tests 69% of the times and they can answer questions about tests 14% faster. However, they also report that improved tests did not result in higher understandability. Specifically, the correctness of the answers given by the developers about test cases with improved readability was not significantly higher. This suggests that other factor should also be considered to assess and improve the understandability of test cases.

Palomba *et al.* [108] defined two textual-based test code metrics, *i.e.,* the Coupling Between Test Methods (CBTM) and the Lack of Cohesion of a Test Method (LCTM), and they use them as a proxy for test case quality. Their findings show that using their metrics as a secondary criterion for MOSA allows them to improve test quality.

Finally, Daka *et al.* [39] introduced an automated approach to generate descriptive names for automatically generated unit tests by summarizing API-level coverage goals to provide the benefits of well-named tests also to automated unit test generation. They show that developers find generated names meaningful.

# Part II

# Source Code Readability and Understandability

*"ἔοικα γοῦν τούτου γε σμικρῶ τινι αυτῶ τούτω σοφώτερος ειναι, ότι ἀ μὴ οἶδα ουδὲ οίομαι ειδέναι."*

*"I seem, then, in just this little thing, to be wiser than this man at any rate, that what I do not know I do not think I know either."*

Plato (quoting Socrates), Apology of Socrates.

# CHAPTER 4

# Using Textual Information to Measure Code Readability

## Contents

## 4.1 Introduction

Three models for source code readability prediction have been proposed in the literature [21, 119, 43], as previously discussed in Chapter 2.1. Such models aim at capturing how the source code has been constructed and how developers perceive it. The process consists of (i) measuring specific aspects of source code, *e.g.,* line length and number of white lines, and (ii) using these metrics to train a binary classifier that is able to tell if a code snippet is "readable" or "unreadable". State-of-the-art readability models define more than 80 features which can be mostly divided in two categories: structural and visual features. The metrics belonging to the former category aim at capturing bad practices such as *lines too long* and good practices such as the *presence of white lines*; the ones belonging to the latter category are designed to capture bad practices such as *code indentation issues* and good practices such as *alignment of characters*. However, despite a plethora of research that has demonstrated the impact of the source code vocabulary on program understanding [86, 85, 29, 41, 83, 45, 138], the code readability models from the literature are still syntactic in nature. The textual features that reflect the quality of the source code vocabulary are very limited: only two features from the previous models take are based on text, but they only take into account the size of the vocabulary.

In this chapter we introduce a set of textual features that can be extracted from source code to improve the accuracy of state-of-the-art code readability models. Indeed, we hypothesize that source code readability should be captured using both syntactic and textual aspects of source code. Unstructured information embedded in the source code reflects, to a reasonable degree, the concepts of the problem and solution domains, as well as the computational logic of the source code. Therefore, textual features capture the domain semantics and add a new layer of semantic information to the source code, in addition to the programming language semantics. To validate the hypothesis and measure the effectiveness of

the proposed features, we performed a two-fold empirical study: (i) we measured to what extent the proposed textual features complement the ones proposed in the literature for predicting code readability; and (ii) we computed the accuracy of a readability model based on structural, visual, and textual features as compared to the state-of-the-art readability models. Both parts of the study were performed on a set of more than 600 code snippets that were previously evaluated, in terms of readability, by more than 5,000 participants.

This chapter is organized as follows. Section 4.2 presents in detail the textual features defined for the estimation of the source code readability. Section 4.3 describes the empirical study we conducted to evaluate the accuracy of a readability model based on structural, visual and textual features. Finally, Section 4.5 concludes the chapter after a discussion of the threats that could affect the validity of the results achieved in our empirical study (Section 4.3.4).

## 4.2 Text-based Code Readability Features

Well-commented source code and high-quality identifiers, carefully chosen and consistently used in their contexts, are likely to improve program comprehension and support developers in building consistent and coherent conceptual models of the code [86, 85, 138, 41, 65, 18]. Our claim is that the analysis of the source code vocabulary cannot be ignored when assessing code readability. Therefore, we propose seven textual properties of source code that can help in characterizing its readability. In the next subsections we describe the textual features introduced to measure code readability.

The proposed textual properties are based on the syntactical analysis of the source code by looking mainly for terms in source code and comments (*i.e.,* the source code vocabulary). Note that we use the word *term* to refer to any word extracted from source code. To this, before computing the textual properties, the terms were extracted form source code by following a standard pre-processing procedure:

1. Remove non-textual tokens from the corpora, *e.g.,* operators, special symbols, and programming language keywords;

2. Split the remaining tokens into separate words by using the under score or camel case separators; *e.g., getText* is split into *get* and *text*;

3. Remove words belonging to a stop-word list (*e.g.,* articles, adverbs) [11].

4. Extract stems from words by using the Porter algorithm [116].

### 4.2.1 Comments and Identifiers Consistency (CIC)

This feature is inspired by the QALP model proposed by Binkley *et al.* [19] and aims at analyzing the consistency between identifiers and comments. Specifically, we compute the *Comments and Identifiers Consistency* (*CIC*) by measuring the overlap between the terms used in a method comment and the terms used in the method body:

$$CIC(m) = \frac{|Comments(m) \cap Ids(m)|}{|Comments(m) \cup Ids(m)|}$$

where *Comments* and *Ids* are the sets of terms extracted from the comments and identifiers in a method $m$, respectively. The measure has a value between $[0, 1]$, and we expect that a higher value of *CIC* is correlated with a higher readability level of the code.

Note that we chose to compute the simple overlap between terms instead of using more sophisticated approaches such as Information Retrieval (IR) techniques (as done in the QALP model), since the two pieces of text compared here (*i.e.,* the method body and its comment) are expected to have a very limited verbosity, thus making the application of IR techniques challenging [40]. Indeed, the QALP model measures the consistency at file level, thus focusing on code components having a much higher verbosity.

One limitation of *CIC* (but also of the QALP model) is that it does not take into account the use of synonyms in source code comments and identifiers. In other words, if the method comment and its code contain two words that are synonyms (*e.g.,* car and automobile), they should be considered consistent. Thus, we introduce a variant of *CIC* aimed at considering such cases:

$$CIC(m)_{syn} = \frac{|Comments(m) \cap (Ids(m) \cup Syn(Ids(m)))|}{|Comments(m) \cup Ids(m) \cup Syn(Ids(m))|}$$

where $Syn(Ids(m))$ is the set of all the synonyms of the terms in $Ids(m)$. With such a variant the use of synonyms between comments and identifiers contributes to improving the value of *CIC*. We use WordNet [101] to determine the set of synonyms of a given word.

### 4.2.2 Identifier Terms in Dictionary (ITID)

Empirical studies have indicated that full-word identifiers ease source code comprehension [86]. Thus, we conjecture that the higher the number of terms in source code identifiers that are also present in a dictionary, the higher the readability of the code. Thus, given a line of code $l$, we measure the feature *Identifier terms in dictionary* (*ITID*) as follows:

$$ITID(l) = \frac{|Terms(l) \cap Dictionary|}{|Terms(l)|}$$

where $Terms(l)$ is the set of terms extracted from a line $l$ of a method and *Dictionary* is the set of words in a dictionary (*e.g.,* English dictionary). As for the *CIC*, the higher the value of *ITID*, the higher the readability of the line of code $l$. In order to compute the feature *Identifier terms in dictionary* for an entire snippet $S$, it is possible to aggregate the $ITID(l)$ for all the $l \in S$ —computed for each line of code of the snippet— by considering the min, the max or the average of such values. Note that the defined $ITID$ is inspired by the *Natural Language Features* introduced by Dorn [43].

### 4.2.3 Narrow Meaning Identifiers (NMI)

Terms referring to different concepts may increase the program comprehension burden by creating a mismatch between the developers' cognitive model and the intended meaning of the term [41, 9]. Thus, we conjecture that a readable code should contain more *hyponyms*, *i.e.,* terms with a specific meaning, than *hypernyms*, *i.e.,* generic terms that might be misleading. Thus, given a line of code $l$, we measure the feature *Narrow meaning identifiers* (*NMI*) as follows:

$$NMI(l) = \sum_{t \in l} particularity(t)$$

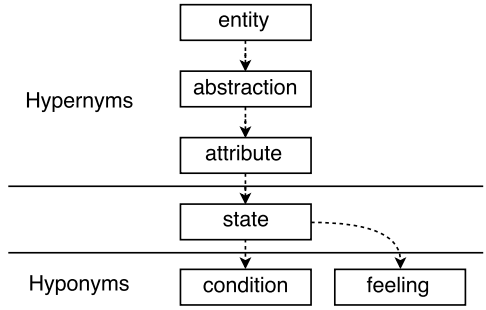



Figure 4.1: Example of hypernyms and hyponyms of the word "state".

where $t$ is a term extracted from the line of code $l$ and $particularity(t)$ is computed as the number of hops from the node containing $t$ to the root node in the hypernym tree of $t$. Specifically, we use hypernym/hyponym trees for English language defined in WordNet [101]. Thus, the higher the *NMI*, the higher the particularity of the terms in $l$, *i.e.,* the terms in the line of code $l$ have a specific meaning allowing a better readability. Figure 4.1 shows an example of hypernyms/hyponyms tree: considering the word "state", the distance between the node that contains such a term from the root node, which contains the term "entity", is 3, so the particularity of "state" is 3. In order to compute the *NMI* for an entire snippet $S$, it is possible to aggregate the $NMI(l), \forall_l \in S$, by considering the min, the max or the average of such values.

### 4.2.4 Comments Readability (CR)

While many comments could surely help to understand the code, they could have the opposite effect if their quality is low. Indeed, a maintainer could start reading the comments, which should ease the understanding phase. If such comments are poorly written, the maintainer will waste time before starting to read the code. Thus, we introduced a feature that calculates the readability of comments (*CR*) using the Flesch-Kincaid [49] index, commonly used to assess readability of natural language texts. Such an index considers three types of elements: words, syllables, and phrases. A *word* is a series of alphabetical characters sepa-

rated by a space or a punctuation symbol; a *syllable* is "a word or part of a word pronounced with a single, uninterrupted sounding of the voice [...] consisting of a single sound of great sonority (usually a vowel) and generally one or more sounds of lesser sonority (usually consonants)" [1]; a *phrase* is a series of words that ends with a new-line symbol, or a strong punctuation point (*e.g.,* a full-stop). The Flesch-Kincaid (FK) index of a snippet $S$ is empirically defined as:

$$FK(S) = 206.835 - 1.015\frac{words(S)}{phrases(S)} - 84.600\frac{syllables(S)}{words(S)}$$

While word segmentation and phrase segmentation are easy tasks, it is a bit harder to correctly segment the syllables of a word. Since such features do not need the exact syllables, but just the number of syllables, relying on the definition, we assume that there is a syllable where we can find a group of consecutive vowels. For example, the number of syllables of the word "definition" is 4 (de<u>fi</u>ni<u>ti</u><u>on</u>). Such an estimation may not be completely valid for all the languages.

We calculate the *CR* by (i) putting together all commented lines from the snippet $S$; (ii) joining the comments with a "." character, in order to be sure that different comments are not joined creating a single phrase; (iii) calculating the Flesch-Kincaid index on such a text.

### 4.2.5 Number of Meanings (NM)

All the natural languages contain polysemous words, *i.e.,* terms which could have many meanings. In many cases the context helps to understand the specific meaning of a polysemous word, but, if many terms have many meanings it is more likely that the whole text (or code, in this case) is ambiguous. For this reason, we introduce a feature which measures the number of meanings (*NM*), or the level or polysemy, of a snippet. For each term in the source code, we measure its number of meanings derived from WordNet [101]. In order to compute the feature *Number of Meanings* for an entire snippet $S$, it is possible to aggregate the *NI(l)* values—computed for each line of code $l$ of the snippet—considering the max or the average of such values. We do not consider the minimum but still consider the maximum, because while it is very likely that a term with few meanings is present, and such a fact does not help in distinguishing readable

```
1  public void buildModel() {
2      if (getTarget() != null) {
3          Object target = getTarget();
4          Object kind = Model.getFacade().getAggregation(target);
5          if (kind == null
6                  || kind.equals(Model.getAggregationKind().getNone())) {
7              setSelected(ActionSetAssociationEndAggregation.NONE_COMMAND);
8          } else {
9              setSelected(ActionSetAssociationEndAggregation.AGGREGATE_COMMAND);
10         }
11     }
12 }
13
```

Figure 4.2: Example of computing textual coherence for a code snippet.

snippets from not-readable ones, the presence of a term with too many meanings could be crucial in identifying unreadable snippets.

### 4.2.6 Textual Coherence (TC)

The lack of cohesion of classes negatively impacts the source code quality and correlates with the number of defects [95, 145]. Based on this observation, our conjecture is that when a snippet has a low cohesion (*i.e.,* it implements several concepts), it is harder to comprehend than a snippet implementing just one "concept". The textual coherence of the snippet can be used to estimate the number of "concepts" implemented by a source code snippet. Specifically, we considered the syntactic blocks of a specific snippet as documents. We parse the source code and we build the Abstract Syntax Tree (AST) in order to detect syntactic blocks, which are the bodies of every control statement (e.g., `if` statements). We compute (as done for *Comments and Identifiers Consistency*) the vocabulary overlap between all the possible pairs of distinct syntactic blocks. The *Textual coherence (TC)* of a snippet can be computed as the max, the min or the average overlap between each pairs of syntactic blocks. For instance, the method in Figure 4.2 has three blocks: $B_1$ (lines 2-11), $B_2$ (lines 5-8), and $B_3$ (lines 8-10); for computing $TC$, first, the vocabulary overlap is computed for each pair of blocks, ($B_1$ and $B_2$, $B_1$ and $B_3$, $B_2$ and $B_3$); then the three values can be aggregated by using the average, the min, or the max.

### 4.2.7 Number of Concepts (NOC)

Textual Coherence tries to capture the number of implemented topics in a snippet at block level. However, its applicability may be limited when there are few syntactic blocks. Indeed, if a snippet contains just a single block, such a feature is not computable at all. Besides, Textual Coherence is a coarse-grain feature, and it works under the assumption that syntactic blocks are self-consistent. Therefore, we introduced a measurement which is able to directly capture the Number of Concepts implemented in a snippet at line-level. It is worth noting that such features can be computed also on snippets that may not be syntactically correct. In order to measure the Number of Concepts, as a first step, we create a document for each line of a given snippet. All the empty documents, resulting from empty lines or lines containing only non-alphabetical characters, are deleted. Then, we use a density-based clustering technique, DBSCAN [47, 123], in order to create clusters of similar documents (i.e., lines). We measure the distance between two documents (represented as sets of terms) as using the Jaccard index:

$$NOC_{dist}(d_1,\, d_2) = \frac{|d_1 \cap d_2|}{|d_1 \cup d_2|}$$

Finally, we compute the "Number of Concepts" ($NOC$) of a snippet $m$ as the number of clusters ($Clusters(m)$) resulting from the previous step:

$$NOC(m) = |Clusters(m)|$$

We also compute an additional feature $NOC_{norm}$ which results from normalizing $NOC$ with the number of documents extracted from a snippet $m$:

$$NOC_{norm}(m) = \frac{|Clusters(m)|}{|Documents(m)|}$$

It is worth noting that $NOC$ and $NOC_{norm}$ measure something that has an opposite meaning with respect to Textual Coherence. While Textual Coherence is *higher* if different blocks contain many similar words, Number of Concepts is *lower* if different lines contain many similar words. This happens because when

```
public boolean isPlaying(TGMeasure measure) {
  // thread safe
  TGMeasure playMeasure = this.playMeasure;

  return (isPlaying() && playMeasure != null && measure.equals(playMeasure));
}
```

Figure 4.3: Example of a small method.

several lines contain similar words, they are put in the same cluster and, thus, the number of clusters is lower, as well as the whole $NOC$ and $NOC_{norm}$ features.

Figure 4.3 shows an example of a method with just a block. In this case, $TC$ can not be computed. On the other hand, $NOC$ and $NOC_{norm}$ are computed as follows. As a first step, 4 documents are extracted from the snippet in Figure 4.3, namely: "public boolean is playing TG measure measure", "thread safe", "TG measure play measure this play measure", "return is playing play measure null measure equals play measure". Assuming that such documents are clustered all together, except for "thread safe", which constitutes a cluster on its own, we have that $NOC(\texttt{isPlaying}) = 2$ and $NOC_{norm}(\texttt{isPlaying}) = \frac{2}{4} = 0.5$.

DBSCAN does not need to know the number of clusters, which is, actually, the result of the computation that we use to define $NOC$ and $NOC_{norm}$. Instead, this algorithm needs the parameter $\epsilon$, which represents the maximum distance at which two documents need to be in order to be grouped in the same cluster. We did not choose $\epsilon$ arbitrarily; instead we tuned such a parameter, by choosing the value that allows the features $NOC$ and $NOC_{norm}$ to achieve, alone, the highest readability prediction accuracy. In order to achieve this goal, we considered all the snippets and the oracles from the three data sets described in Section 4.3 and we trained and tested nine classifiers, each of which contained just a feature, $NOC^{\epsilon}$, where $NOC^{\epsilon}$ is $NOC$ computed using a specific $\epsilon$ parameter for DBSCAN. Since the distance measure we use ranges between 0 and 1, also $\epsilon$ can range between such values and, thus, the values we used as candidate $\epsilon$ for the nine $NOC^{\epsilon}$ features are $\{0.1, 0.2, ..., 0.9\}$; we discarded the extreme values, 0 and 1, because in these cases each document would have been in a separate cluster or all documents would have been in the same cluster, respectively. We use each classifier containing a single $NOC^{\epsilon}$ feature to predict readability, and we

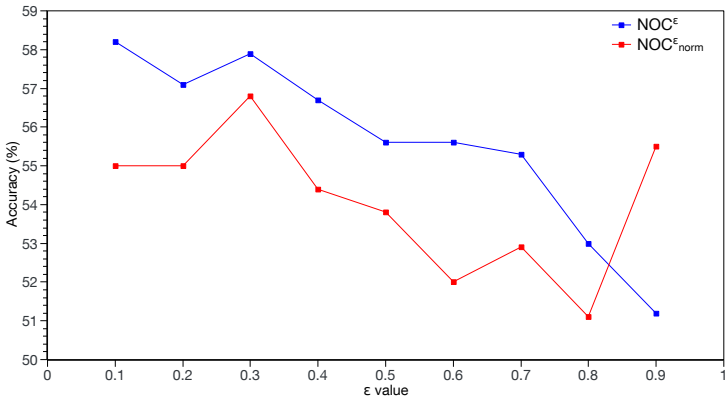



Figure 4.4: Accuracy of different classifiers based only on $NOC^{\epsilon}$ (blue) and $NOC^{\epsilon}_{norm}$ (red).

pick the value of $\epsilon$ that leads to the best classification accuracy with 10-fold cross-validation. The classification technique used for tuning $\epsilon$ was Logistic Regression, also used in Section 4.3. We repeated the same procedure for $NOC_{norm}$. Figure 4.4 shows the accuracy achieved by each classifier containing different $NOC^{\epsilon}$ (in blue) or $NOC^{\epsilon}_{norm}$ (in red). The best $\epsilon$ value for $NOC$ is 0.1, while for $NOC_{norm}$ it is 0.3, as the chart shows.

### 4.2.8 Readability vs Understandability

Posnett *et al.* [119] compared the difference between readability and understandability to the difference between syntactic and semantic analysis. Readability measures the effort of the developer to access the information contained in the code, while understandability measures the complexity of such information. We defined a set of textual features that still capture aspects of code related to the difficulty of accessing the information contained in a snippet. For example, $NOC$ estimates the number of concepts implemented in a snippet. A snippet with a few concepts, potentially more readable, can still be hard to understand if a few concepts are not easy to understand. In our opinion, textual features,

which do not take into account semantics, like the ones we defined, can be used to measure readability.

## 4.3 Evaluation

The *goal* of this study is to analyze the role played by textual features in assessing code readability, with the *purpose* of improving the accuracy of state-of-the-art readability models. The *quality focus* is the prediction of source code readability, while the *perspective* of the study is of a researcher, who is interested in analyzing to what extent structural, visual, and textual information can be used to characterize code readability.

We formulated the following research questions (RQs):

- **RQ**$_1$: *To what extent the proposed textual features complement the ones proposed in the literature for predicting code readability?* With this preliminary question we are interested in verifying whether the proposed textual features complement the ones from the state-of-the-art when used to measure code readability. This represents a crucial prerequisite for building an effective comprehensive model considering both families of features.

- **RQ**$_2$: *What is the accuracy of a readability model based on structural, visual, and textual features as compared to the state-of-the-art readability models?* This research question aims at verifying to what extent a readability model based on structural, visual, and textual features overcomes readability models mainly based on structural features, such as the model proposed by Buse and Weimer [21], the one presented by Posnett *et al.* [119], and the most recent one introduced by Dorn [43].

### 4.3.1 Data Collection

An important prerequisite for evaluating a code readability model is represented by the availability of a reliable oracle, *i.e.,* a set of code snippets for which the readability has been manually assessed by humans. This allows measuring to what extent a readability model is able to approximate human judgment of source code readability. All the datasets used in the study are composed of code snippets

for which the readability has been assessed via human judgement. In particular, each snippet in the data sets is accompanied by a flag indicating whether it was considered readable by humans (*i.e.*,, binary classification). The first dataset (in the following $D_{b\&w}$) was provided by Buse and Weimer [21] and it is composed of 100 Java code snippets having a mean size of seven lines of code. The readability of these snippets was evaluated by 120 student annotators. The second dataset (in the following $D_{dorn}$) was provided by Dorn [43] and represents the largest dataset available for evaluating readability models. It is composed of 360 code snippets, including 120 snippets written in CUDA, 120 in Java, and 120 in Python. The code snippets are also diverse in terms of size including for each programming language the same number of small- (∼10 LOC), medium- (∼30 LOC) and large- (∼50 LOC) sized snippets. In $D_{dorn}$, the snippets' readability was assessed by 5,468 humans, including 1,800 industrial developers.

The main drawback of the aforementioned datasets ($D_{b\&w}$ and $D_{dorn}$) is that some of the snippets are not complete code entities (*e.g.,* methods); therefore, some of the data instances in $D_{b\&w}$ and $D_{dorn}$ datasets are code fragments that only represent a partial implementation (and thus they may not be syntactically correct) of a code entity. This is an impediment for computing one of the new textual features introduced in this chapter: *textual coherence (TC)*; it is impossible to extract code blocks from a snippet if an opening or closing bracket is missing. For this reason, we built an additional dataset ($D_{new}$), by following an approach similar to the one used in the previous work to collect $D_{b\&w}$ and $D_{dorn}$ [21, 43]. Firstly, we extracted all the methods from four open source Java projects, namely *jUnit*, *Hibernate*, *jFreeChart* and *ArgoUML*, having a size between 10 and 50 lines of code (including comments). We focused on methods because they represent syntactically correct and complete code entities of code.

Initially, we identified 13,044 methods for $D_{new}$ that satisfied our constraint on the size. However, the human assessment of all the 13K+ methods is practically impossible, since it would require a significant human effort. For this reason, we evaluated the readability of only 200 sampled methods from $D_{new}$. The sampling was not random, but rather aimed at identifying the most representative methods for the features used by all the readability models defined and studied in this chapter. Specifically, for each of the 13,044 methods we calculated all

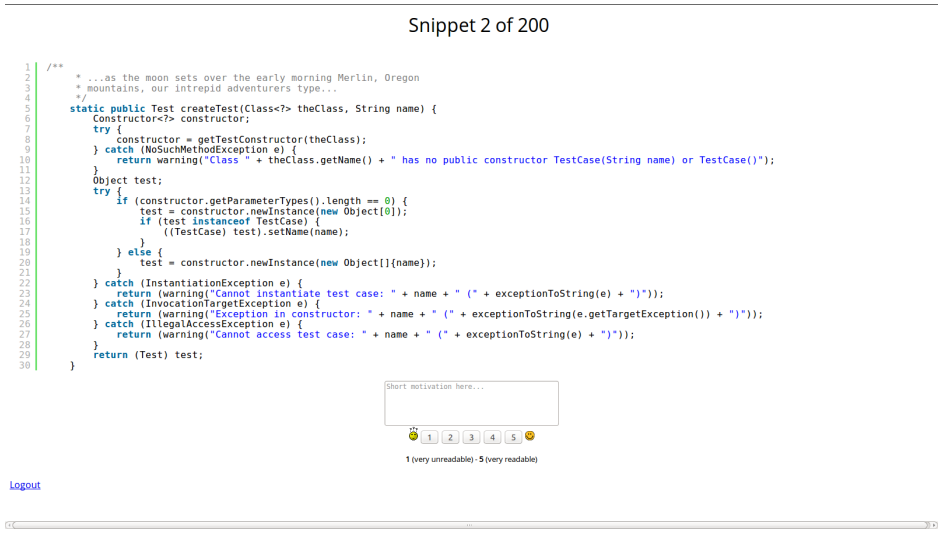



Figure 4.5: Web application used to collect the code readability evaluation for our new dataset $D_{new}$.

the features (*i.e.,* all the features proposed in the literature and textual ones proposed in this chapter) aiming at associating each method with a feature vector containing the values for each feature. Then, we used a greedy algorithm for center selection [77] to find the 200 most representative methods in $D_{new}$. The distance function used in the implementation of such algorithm is represented by the Euclidean distance between the feature vector of two snippets. The adopted selection strategy allowed us (i) to enrich the diversity of the selected methods avoiding the presence of similar methods in terms of the features considered by the different experimented readability models, and (ii) to increase the generalizability of our findings.

After selecting the 200 methods in $D_{new}$, we asked 30 Computer Science students from the College of William and Mary to evaluate the readability $r$ of each of them. The participants were asked to evaluate each method using a five-point Likert scale ranging between 1 (*very unreadable*) and 5 (*very readable*). We collected the rankings through a web application (Figure 4.5) where participants were able to (i) read the method (with syntax highlighting); (ii) evaluate its

readability; and (iii) write comments about the method. The participants were also allowed to complete the evaluation in multiple rounds (*e.g.,* evaluate the first 100 methods in one day and the remaining after one week). Among the 30 invited participants, only nine completed the assessment of all the 200 methods. This was mostly due to the large number of methods to be evaluated; the minimum time spent to complete this task was about two hours. In summary, given the 200 methods in $m_i \in D_{new}$ and nine human taggers $t_j \in T$, we collected readability rankings $r(m_i, t_j), \forall_{i,j}, i \in [1, 200], j \in [1, 9]$.

After having collected all the evaluations, we computed, for each method $m \in D_{new}$, the mean score that represents the final readability value of the snippet, i.e., $\bar{r}(m) = \frac{\Sigma_1^9 r(m,j)}{9}$. We obtained a high agreement among the participants with Cronbach-$\alpha$=0.98, which is comparable to the one achieved in $D_{b\&w}$=0.96. This confirms the results reported by Buse and Weimer in terms of humans agreement when evaluating/ranking code readability: "*humans agree significantly on what readable code looks like, but not to an overwhelming extent*" [21]. Note that code readability evaluation by using crisp categories (*e.g.,*, *readable*, *non-readable*) is required to build a readability model over the collected snippets; therefore, for the methods in $D_{new}$, we used the mean of the readability score among all the snippets as a cut-off value. Specifically, methods having a score below 3.6 were classified as *non-readable*, while the remaining methods (*i.e.,* $\bar{r}(m) \geq 3.6$ ) as *readable*. A similar approach was also used by Buse and Weimer [21]. Figure 4.6 shows the distribution of the average readability scores for the snippets in the new dataset.

### 4.3.2 Analysis Method

In order to answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we built readability models (*i.e.,* binary classifiers) for each dataset (*i.e.,* $D_{b\&w}$, $D_{dorn}$, and $D_{new}$) by using different sets of state-of-the-art and our textual features: Buse and Weimer's ($BWF$) [21], Posnett's ($PF$) [119], Dorn's ($DF$) [43], our textual features ($TF$), and all the features (*All-Features*= $BWF \cup PF \cup DF \cup TF$). With notational purposes, we will use $R\langle Features \rangle$ to denote a specific readability model $R$ we built using a set of *Features*. For instance, $R\langle TF \rangle$ denotes the textual features-based readability model. It is worth noting that with our experiments we are not running the same

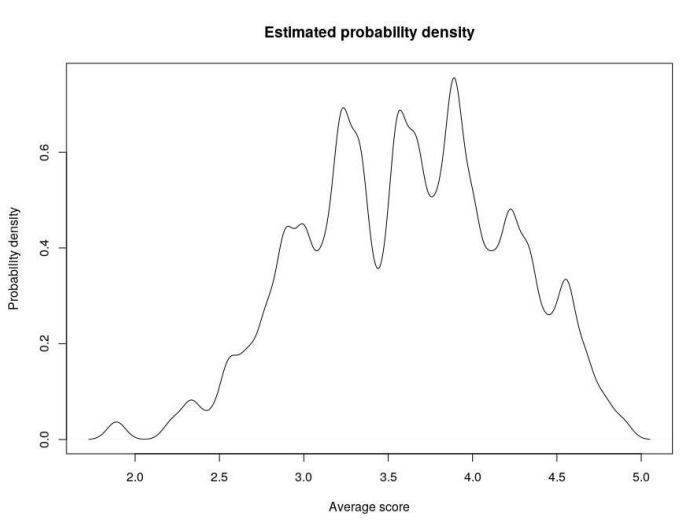



Figure 4.6: Estimated probability density of the average readability scores (bandwidth: 0.05).

models proposed in the prior works, but, we are using the same features proposed by previous works.

As for the classifier used with the models, we relied on logistic regression because it has been shown to be very effective in binary-classification and it was used by Buse and Weimer for their readability model [21]. Besides, such a technique does not require parameter tuning. To avoid over-fitting, we performed feature selection by using linear forward selection with a wrapper strategy [64] available in the Weka machine learning toolbox. In the wrapper selection strategy each candidate subset of features is evaluated through the accuracy of the classifier trained and tested using only such features. The final result is the subset of features which obtained the maximum accuracy. We used 10 as "search termination" parameter: such a parameter indicates the amount of backtracking of the algorithm.

In the case of $\mathbf{RQ}_1$ we analyzed the complementarity of the textual features-based model with the models trained with state-of-the-art features, by computing overlap metrics between $R\langle TF \rangle$ and each of the three competitive models (*i.e.,*

$R\langle BWF\rangle$, $R\langle PF\rangle$, $R\langle DF\rangle$). Specifically, given two readability models under analysis, $R\langle TF\rangle$ a model based on textual features, and $R\langle SF\rangle$ a model based on state-of-the-art features (*i.e.,* $SF \in \{BWF, PF, DF\}$) [1], the metrics are defined as in the following:

$$\xi\big(R\langle TF\rangle \cap R\langle SF\rangle\big) = \frac{|\xi\big(R\langle TF\rangle\big) \cap \xi\big(R\langle SF\rangle\big)|}{|\xi\big(R\langle TF\rangle\big) \cup \xi\big(R\langle SF\rangle\big)|}\%$$

$$\xi\big(R\langle TF\rangle \setminus R\langle SF\rangle\big) = \frac{|\xi\big(R\langle TF\rangle\big) \setminus \xi\big(R\langle SF\rangle\big)|}{|\xi\big(R\langle TF\rangle\big) \cup \xi\big(R\langle SF\rangle\big)|}\%$$

$$\xi\big(R\langle SF\rangle \setminus R\langle TF\rangle\big) = \frac{|\xi\big(R\langle SF\rangle\big) \setminus \xi\big(R\langle TF\rangle\big)|}{|\xi\big(R\langle TF\rangle\big) \cup \xi\big(R\langle SF\rangle\big)|}\%$$

where $\xi\big(R\langle TF\rangle\big)$ and $\xi\big(R\langle SF\rangle\big)$ represent the sets of code snippets correctly classified as readable/non-readable by $R\langle TF\rangle$ and the competitive model $R\langle SF\rangle$ ($SF \in \{BWF, PF, DF\}$), respectively. $\xi\big(R\langle TF\rangle \cap R\langle SF\rangle\big)$ measures the overlap between code snippets correctly classified by both techniques, $\xi\big(R\langle SF\rangle \setminus R\langle TF\rangle\big)$ measures the snippets correctly classified by $R\langle TF\rangle$ only and wrongly classified by $R\langle SF\rangle$, and $\xi\big(R\langle TF\rangle \setminus R\langle SF\rangle\big)$ measures the snippets correctly classified by $R\langle SF\rangle$ only and wrongly classified by $R\langle TF\rangle$.

Turning to the second research question ($\mathbf{RQ_2}$), we compared the accuracy of a readability model based on both all the structural, visual, and textual features ($R\langle$ *All-Features* $\rangle$) with the accuracy of the three baselines, *i.e.,* $R\langle BWF\rangle$, $R\langle PF\rangle$ and $R\langle DF\rangle$. To further show the importance of the new textual features, we also compared $R\langle$ *All-Features* $\rangle$ to an additional baseline, namely a model based on all the state-of-the-art features ($R\langle SVF\rangle = R\langle BWF + PF + DF\rangle$). In order to compute the accuracy, we fist compute:

- True Positives ($TP$): number of snippets correctly classified as *readable*;
- True Negatives ($TN$): number of snippets correctly classified as *non-readable*;
- False Positives ($FP$): number of snippets incorrectly classified as *readable*;

[1] Note that later in this chapter we will replace $R\langle SF\rangle$ with $R\langle BWF\rangle$, $R\langle PF\rangle$, or $R\langle DF\rangle$.

- False Negatives ($FN$): number of snippets incorrectly classified as *non-readable*;

then, we compute accuracy as $\frac{TP+TN}{TP+TN+FP+FN}$, *i.e.,* the rate of snippets correctly classified.

In addition, we report the accuracy achieved by the readability model only exploiting textual features (*i.e.,* $R\langle TF\rangle$). In particular, we measured the percentage of code snippets correctly classified as readable/non-readable by each technique on each of the three datasets. We also report the AUC achieved by all the models, in order to compare them taking into account an additional metric, widely used for evaluating the performance of a classifier.

Each readability model was trained on each dataset individually and a 10-fold cross-validation was performed. The process for the 10-fold cross-validation is composed of five steps: (i) randomly divide the set of snippets for a dataset into 10 approximately equal subsets, regardless of the projects they come from; (ii) set aside one snippet subset as a test set, and build the readability model with the snippets in the remaining subsets (*i.e.,* the training set); (iii) classify each snippet in the test set using the readability model built on the snippet training set and store the accuracy of the classification; (iv) repeat this process, setting aside each snippet subset in turn; (v) compute the overall average accuracy of the model.

Finally, we used statistical tests to assess the significance of the achieved results. In particular, since we used 10-fold cross validation, we considered the accuracy achieved on each fold by all the models. We used the Wilcoxon test [42] (with $\alpha = 0.05$) in order to estimate whether there are statistically significant differences between the classification accuracy obtained by $R\langle TF\rangle$ and the other models. Our decision for using the Wilcoxon test, is a consequence of the usage of the 10-fold cross validation to gather the accuracy measurements. During the cross-validation, each fold is selected randomly, but we used the same seed to have the same folds for all the experiments. For example, the $5^{\text{th}}$ testing fold used for $R\langle BWF\rangle$ is equal to the $5^{\text{th}}$ testing fold used with $R\langle$*All-Features*$\rangle$. Consequently, pairwise comparisons are performed between related samples.

Because we performed multiple pairwise comparisons (*i.e., All-features* vs. the rest), we adjusted our $p$-values using the Holm's correction procedure [70].

| **Dataset** | $R\langle TF\rangle \cap R\langle BWF\rangle$ | $R\langle TF\rangle \setminus R\langle BWF\rangle$ | $R\langle BWF\rangle \setminus R\langle TF\rangle$ |
|---|---|---|---|
| $D_{b\&w}$ | 72% | 10% | 18% |
| $D_{dorn}$ | 69% | 15% | 16% |
| $D_{new}$ | 64% | 20% | 16% |
| Overall | **68%** | **15%** | **17%** |
| | $R\langle TF\rangle \cap R\langle PF\rangle$ | $R\langle TF\rangle \setminus R\langle PF\rangle$ | $R\langle PF\rangle \setminus R\langle TF\rangle$ |
| $D_{b\&w}$ | 71% | 12% | 17% |
| $D_{dorn}$ | 66% | 20% | 14% |
| $D_{new}$ | 72% | 20% | 8% |
| Overall | **70%** | **17%** | **13%** |
| | $R\langle TF\rangle \cap R\langle DF\rangle$ | $R\langle TF\rangle \setminus R\langle DF\rangle$ | $R\langle DF\rangle \setminus R\langle TF\rangle$ |
| $D_{b\&w}$ | 70% | 11% | 19% |
| $D_{dorn}$ | 78% | 10% | 12% |
| $D_{new}$ | 76% | 12% | 12% |
| Overall | **75%** | **11%** | **14%** |

Table 4.1: $\mathbf{RQ}_1$: Overlap between $R\langle TF\rangle$ and $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle DF\rangle$.

In addition, we estimated the magnitude of the observed differences by using the Cliff's Delta ($d$), a non-parametric effect size measure for ordinal data [62]. Cliff's $d$ is considered negligible for $d < 0.148$ (positive as well as negative values), small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [62].

### 4.3.3 RQ1: Complementarity of the Readability Features

Table 4.1 reports the overlap metrics computed between the results of the readability models using textual, structural, and visual features. Across the three datasets, the $R\langle TF\rangle$ model exhibits an overlap of code snippets correctly classified as readable/non-readable included between 68% ($R\langle TF\rangle \cap R\langle BWF\rangle$) and 75% ($R\langle TF\rangle \cap R\langle DF\rangle$). This means that, despite the competitive model considered, about 30% of the code snippets are differently assessed as readable/non-readable when only relying on textual features. Indeed, (i) between 11% ($R\langle TF\rangle \setminus R\langle DF\rangle$) and 17% ($R\langle TF\rangle \setminus R\langle PF\rangle$) of code snippets are correctly classified only by $R\langle TF\rangle$ and (ii) between 13% ($R\langle PF\rangle \setminus R\langle TF\rangle$) and 17% ($R\langle BWF\rangle \setminus R\langle TF\rangle$) are correctly classified only by the competitive models.

These results highlight a high complementarity between state-of-the-art and textual features when used for readability assessment. An example of a snippet

```
/**
 * 1. Recreate the collection key -> collection map
 * 2. rebuild the collection entries
 * 3. call Interceptor.postFlush()
 */
protected void postFlush(SessionImplementor session) throws HibernateException {

    LOG.trace( "Post flush" );

    final PersistenceContext persistenceContext = session.getPersistenceContext();
    persistenceContext.getCollectionsByKey().clear();

    // the database has changed now, so the subselect results need to be invalidated
    // the batch fetching queues should also be cleared - especially the collection
     batch fetching one
    persistenceContext.getBatchFetchQueue().clear();

    for ( Map.Entry<PersistentCollection, CollectionEntry> me : IdentityMap.
     concurrentEntries( persistenceContext.getCollectionEntries() ) ) {
        CollectionEntry collectionEntry = me.getValue();
        PersistentCollection persistentCollection = me.getKey();
        collectionEntry.postFlush(persistentCollection);
        if ( collectionEntry.getLoadedPersister() == null ) {
            //if the collection is dereferenced, remove from the session cache
            //iter.remove(); //does not work, since the entrySet is not backed by the
     set
            persistenceContext.getCollectionEntries()
                              .remove(persistentCollection);
        }
        else {
            //otherwise recreate the mapping between the collection and its key
            CollectionKey collectionKey = new CollectionKey(
                              collectionEntry.getLoadedPersister(),
                              collectionEntry.getLoadedKey()
            );
            persistenceContext.getCollectionsByKey().put(collectionKey,
     persistentCollection);
        }
    }

}
```

Figure 4.7: Code snippet from the new dataset correctly classified as "non-readable" **only** when relying on **state-of-the-art features** and missed when using **textual features**.

```
protected void scanAnnotatedMembers(Map<Class<? extends Annotation>, List<
     FrameworkMethod>> methodsForAnnotations, Map<Class<? extends Annotation>, List<
     FrameworkField>> fieldsForAnnotations) {
    for (Class<?> eachClass : getSuperClasses(fClass)) {
        for (Method eachMethod : MethodSorter.getDeclaredMethods(eachClass)) {
            addToAnnotationLists(new FrameworkMethod(eachMethod), methodsForAnnotations)
     ;
        }
        // ensuring fields are sorted to make sure that entries are inserted
        // and read from fieldForAnnotations in a deterministic order
        for (Field eachField : getSortedDeclaredFields(eachClass)) {
            addToAnnotationLists(new FrameworkField(eachField), fieldsForAnnotations);
        }
    }
}
```

Figure 4.8: Code snippet from the new dataset correctly classified as "readable" **only** when relying on **textual features** and missed by the competitive techniques.

for which the textual features are not able to provide a correct assessment of its readability is reported in Figure 4.7. Such a method (considered "unreadable" by human annotators) has a pretty high average textual coherence (0.58), but, above all, it has a high comment readability and comment-identifiers consistency, *i.e.,* many terms co-occur in identifiers and comments (*e.g.,* "batch" and "fetch"). Nevertheless, some lines are too long, resulting in a high maximum and average line length (146 and 57.3, respectively), both impacting negatively the perceived readability [21].

Figure 4.8 reports, instead, a code snippet correctly classified as "readable" only when exploiting textual features. The snippet has suboptimal structural characteristics, such as a high average/maximum line length (65.4 and 193, respectively) and a high average number of identifiers (2.7), both negatively correlated with readability. Nevertheless, the method has high average textual coherence ($\sim$ 0.73) and high comments readability (100.0). The source code can be read almost as natural language text and the semantic of each line is pretty clear, but such an aspect is completely ignored by the state-of-the-art features.

| Dataset | Snippets | $R\langle BWF\rangle$ | $R\langle PF\rangle$ | $R\langle DF\rangle$ | $R\langle TF\rangle$ | $R\langle SVF\rangle$ | $R\langle$ *All-features* $\rangle$ |
|---|---|---|---|---|---|---|---|
| $D_{b\&w}$ | 100 | 81.0% | 78.0% | 81.0% | 74.0% | 83.0% | 87.0% |
| $D_{dorn}$ | 360 | 78.6% | 72.8% | 80.0% | 78.1% | 80.6% | 83.9% |
| $D_{new}$ | 200 | 72.5% | 66.0% | 75.5% | 76.5% | 77.0% | 84.0% |
| **Overall**$_{wm}$ | **660** | **77.1%** | **71.5%** | **78.8%** | **77.0%** | **79.9%** | **84.4%** |
| **Overall**$_{am}$ | **660** | **77.4%** | **72.3%** | **78.8%** | **76.2%** | **80.2%** | **85.0%** |

Table 4.2: **RQ**$_2$: Average accuracy achieved by the readability models in the three datasets.

**Summary for** $RQ_1$. A code readability model solely relying on textual features exhibits complementarity with models mainly exploiting structural and visual feature. On average, the readability of 11%-17% code snippets is correctly assessed only when using textual features. Similarly, 13-17% of code snippets are correctly assessed only when using structural and visual features.

### 4.3.4 RQ2: Accuracy of the Readability Model

Table 4.2 shows the accuracy achieved by (i) the comprehensive readability model, namely the model which exploits structural, visual, and textual features (*All-Features*), (ii) the model solely exploiting textual features (*TF*), (iii) the three state-of-the-art models mainly based on structural features (*BWF*, *PF*, and *DF*) and (iv) the model based on all the state-of-the-art features. We report the overall accuracy achieved by each model using two different proxies: overall$_{wm}$ and overall$_{am}$. Overall$_{wm}$ is computed as the weighted mean of the accuracy values for each dataset, where the weights are the number of snippets in each dataset. Overall$_{am}$ is computed as the arithmetic mean of the accuracy values for each dataset.

When comparing all the models, it is clear that textual features achieve an accuracy comparable and, on average, higher than the one achieved by the model proposed by Posnett et al. ($R\langle PF\rangle$). Nevertheless, as previously pointed out, textual-based features alone are not sufficient to measure readability: indeed, the models $R\langle BWF\rangle$ and $R\langle DF\rangle$ always achieve a higher accuracy than $R\langle TF\rangle$.

| Dataset | Snippets | $R\langle BWF\rangle$ | $R\langle PF\rangle$ | $R\langle DF\rangle$ | $R\langle TF\rangle$ | $R\langle SVF\rangle$ | $R\langle$ *All-features* $\rangle$ |
|---|---|---|---|---|---|---|---|
| $D_{b\&w}$ | 100 | 0.874 | 0.880 | 0.828 | 0.762 | 0.850 | 0.867 |
| $D_{dorn}$ | 360 | 0.828 | 0.781 | 0.826 | 0.830 | 0.842 | 0.874 |
| $D_{new}$ | 200 | 0.791 | 0.746 | 0.792 | 0.800 | 0.782 | 0.853 |
| **Overall**$_{wm}$ | 660 | **0.824** | **0.785** | **0.816** | **0.811** | **0.825** | **0.867** |
| **Overall**$_{am}$ | 660 | **0.831** | **0.802** | **0.815** | **0.797** | **0.825** | **0.865** |

Table 4.3: **RQ**$_2$: Average AUC achieved by the readability models in the three datasets.

On the other hand, if we use a model which combines all the features, the combined model achieves an accuracy higher than the other models when analyzing each dataset individually. In addition, we obtained an overall accuracy (*i.e.,* using all the accuracy samples as a single dataset) higher than all the compared models for both the proxies, *i.e.,* overall$_{am}$ (from 6.2% with respect to $R\langle DF\rangle$ to 12.7% with respect to $R\langle PF\rangle$) and overall$_{wm}$ (from 5.6% with respect to $R\langle DF\rangle$ to 12.9% with respect to $R\langle PF\rangle$). It is also worth noting that $R\langle$ *All-features* $\rangle$ achieves an higher accuracy also compared to a model containing all the state-of-the-art features together. This further shows that textual features have an important role.

Since the results in terms of accuracy may depend on a specific threshold, we also report in Table 4.3 the Area Under the Curve (AUC) achieved by all the readability models. Also in this case, we report the overall accuracy achieved by each model using the two proxies previously defined, *i.e.,* overall$_{wm}$ (weighted average) and overall$_{am}$ (arithmetic average). The AUC values, overall, confirm that the combined model outperforms the other models. Nevertheless, we can see that the overall$_{wm}$ AUC achieved by $R\langle TF\rangle$ is comparable to the overall$_{wm}$ AUC achieved by $R\langle DF\rangle$, and slightly minor than the one achieved by $R\langle BWF\rangle$. While in terms of accuracy $R\langle DF\rangle$ seems to be slightly better than $R\langle BWF\rangle$, in terms of AUC, the opposite is true. Furthermore, there is a high difference in terms of accuracy between $R\langle$ *All-features* $\rangle$ and all the other models on the dataset by Buse and Weimer, but in terms of AUC this difference is less evident and, instead, other models achieve higher AUC (*e.g.,* $R\langle PF\rangle$).

Table 4.4 shows the p-values after correction and the Cliff's delta for the pairwise comparisons performed between the model that combines structural, visual,

| Dataset | $R\langle BWF\rangle$ | $R\langle PF\rangle$ | $R\langle DF\rangle$ | $R\langle TF\rangle$ | $R\langle SVF\rangle$ |
|---|---|---|---|---|---|
| $D_{b\&w}$ | $0.70(d = 0.27)$ | $0.70(d = 0.44)$ | $0.70(d = 0.31)$ | $0.21(d = 0.65)$ | $0.70(d = 0.21)$ |
| $D_{dorn}$ | $0.10(d = 0.53)$ | $\mathbf{0.03}(d = 0.85)$ | $0.22(d = 0.31)$ | $0.22(d = 0.49)$ | $0.22(d = 0.30)$ |
| $D_{new}$ | $0.09(d = 0.55)$ | $\mathbf{0.04}(d = 0.77)$ | $0.15(d = 0.45)$ | $0.09(d = 0.43)$ | $0.15(d = 0.39)$ |
| $D_{all}$ | $\mathbf{0.01}(d = 0.43)$ | $\mathbf{0.00}(d = 0.64)$ | $\mathbf{0.01}(d = 0.33)$ | $\mathbf{0.00}(d = 0.51)$ | $\mathbf{0.01}(d = 0.28)$ |

Table 4.4: $\mathbf{RQ_2}$: Comparisons between the accuracy of $R\langle$*All-features*$\rangle$ and each one of state-of-the-art models.

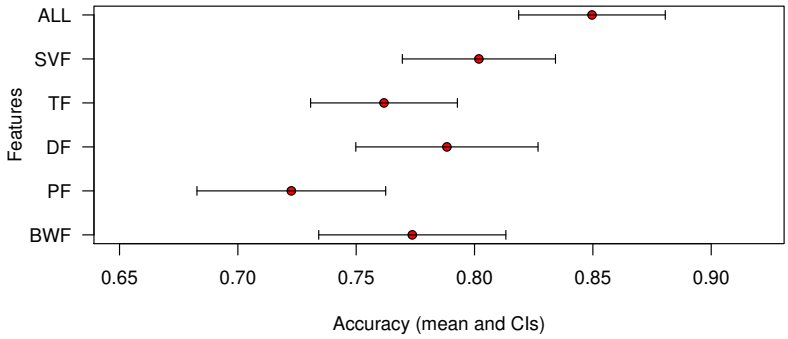



Figure 4.9: Mean accuracy and confidence intervals (CIs) of the compared models.

and textual features and the other models (in bold statistically significant values). When analyzing the results at dataset granularity, we did not find significant differences between *All-Features* and the other models. However, the effect size is medium-large (*i.e.,* $d \geq 0.33$) in most of the comparisons. This issue of no statistical significance with large effect size is an artifact of the size of the samples used with the test, which has been reported previously by Cohen [33] and Harlow *et al.* [67]; in fact, the size of the samples used in our tests for each dataset is 10 measurements (note that we performed 10-fold cross validation). In that sense, we prefer to draw conclusions (conservatively) from the tests performed on the set $D_{all}$, which has a larger sample (30 measurements). When using the datasets as a single one (*i.e.,* $D_{all}$), there is significant difference in the accuracy when comparing $R\langle$*All-features*$\rangle$ to the other models; the results are confirmed with the Cliff's $d$ that suggest a medium-large difference (*i.e.,* $d \geq 0.33$) in all the cases except for $R\langle SVF\rangle$, for which the difference is, overall, small (0.28). It is worth

| | $D_{b\&w}$ | | | $D_{dorn}$ | | | $D_{new}$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Rank** | **Feature** | **Weight** | **Rank** | **Feature** | **Weight** | **Rank** | **Feature** | **Weight** |
| BWF | 5 | #identifiers$_{max}$ | 0.07 | 3 | #comments$_{avg}$ | 0.05 | 19 | Indentation length$_{avg}$ | 0.02 |
| | 8 | #identifiers$_{avg}$ | 0.06 | 8 | #identifiers$_{max}$ | 0.03 | 27 | Identifiers length$_{max}$ | 0.02 |
| | 11 | Line length$_{max}$ | 0.05 | 14 | #operators$_{avg}$ | 0.02 | 31 | #comments$_{avg}$ | 0.02 |
| PF | 10 | Volume | 0.05 | 9 | Entropy | 0.03 | 8 | Lines | 0.02 |
| | 26 | Entropy | 0.03 | 18 | Volume | 0.02 | 10 | Volume | 0.02 |
| | 36 | Lines | 0.02 | 50 | Lines | 0.01 | 58 | Entropy | 0.01 |
| DF | 2 | Area (Strings/Comments) | 0.08 | 2 | #comments (Visual Y) | 0.05 | 1 | #comments (Visual Y) | 0.04 |
| | 3 | Area (Operators/Comments) | 0.08 | 5 | #numbers (Visual Y) | 0.03 | 3 | #conditionals (DFT) | 0.04 |
| | 4 | Area (Identifiers/Comments) | 0.08 | 6 | #comments (Visual X) | 0.03 | 5 | #numbers (DFT) | 0.03 |
| TF | 1 | *CR* | 0.09 | 1 | *CR* | 0.09 | 2 | $TC_{min}$ | 0.04 |
| | 38 | $TC_{avg}$ | 0.02 | 4 | $ITID_{avg}$ | 0.03 | 4 | *CR* | 0.04 |
| | 39 | *NOC* | 0.02 | 12 | $TC_{max}$ | 0.02 | 7 | *NOC* | 0.02 |

Table 4.5: $\mathbf{RQ}_2$: Evaluation of the single features using ReliefF.

noting that we considered the union of the results obtained on the three datasets rather than running a new 10-fold cross validation on a dataset containing all the data points from the three dataset. We did this because such datasets are very different from one another as for number of evaluators, number of snippets, and data collection procedure.

Figure 4.9 illustrates the difference in the accuracy achieved with each model by using the mean accuracy and confidence intervals (CIs) on $D_{all}$. There is a 95% of confidence that the mean accuracy of $R\langle$*All-features*$\rangle$ is larger than $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle TF\rangle$ (*i.e.,* there is no overlap between the CIs). Although the mean accuracy of $R\langle$*All-features*$\rangle$ is the largest one in the study, there is an overlap with the CIs for $R\langle DF\rangle$ and $R\langle SVF\rangle$. Combining $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle DF\rangle$, improves the accuracy on average when compared to $R\langle TF\rangle$. Therefore, including the proposed textual features in state-of-the-art models, overall, improves the accuracy of the readability model, with significant difference when compared to the other models. The statistical tests also confirm that using only textual features is not the best choice for a code readability model.

Regarding individual features, we investigated the most relevant ones for each combination dataset-model. Table 4.5 reports the importance (*i.e.,* weight) of single features, using the ReliefF attribute selection algorithm [76, 78]. Specifically, we report the three best features for each pair dataset-model, specifying also their ranking in the complete list of features for the same dataset and their importance weight. We use ReliefF instead of the feature selection technique we used to build and test our model because it provides a weight for each feature and it allows us to make a ranking. The textual features that, overall, show the best ReliefF

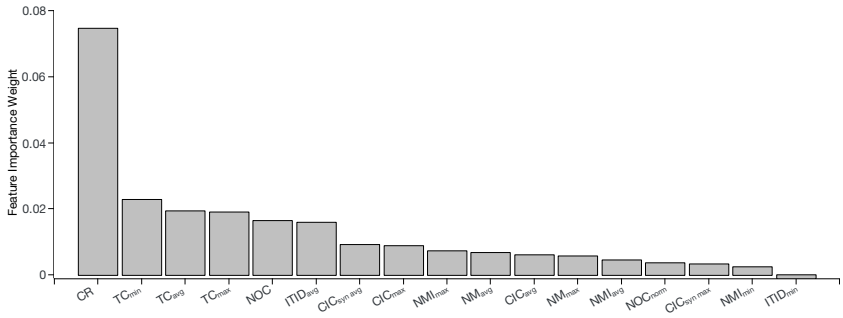



Figure 4.10: Average importance weights of all the textual features.

values (*i.e.,* weight and ranking) are *Comments Readability*, *Textual Coherence* and *Number of Concepts*, since they are in the top-three positions for the three datasets. Besides, the ranking values confirm what Table 4.2 previously hinted, *i.e.,* that textual features are useful in Dorn's dataset and in the new dataset, but they are less useful in Buse and Weimer's dataset; indeed, besides CR, the other features have a low ReliefF value. Finally, Figure 4.10 shows the average attribute importance weight of all the textual features: it is clear that *Comments Readability* is the best predictor of code readability among the textual features, achieving an average ReliefF which is higher than the double of the second best predictor (*i.e.,* $TC_{min}$).

**Summary for** $RQ_2$. A comprehensive model of code readability that combines structural, visual, and textual features is able to achieve a higher accuracy than all the state-of-the-art models. The magnitude of the difference, in terms of accuracy, is mostly medium-to large when considering structural, visual, and textual models. The minimum improvement is of 6.2% and, the difference is statistically significant when compared to the other models (*i.e.,* Buse and Weimer, Postnet et al., Dorn, and Textual features).

| | *ML Technique* | *BWF* | *PF* | *DF* | *TF* | *All-Features* |
|---|---|---|---|---|---|---|
| $D_{b\&w}$ | **BayesNet** | 76.0% | 76.0% | 68.0% | 52.0% | 74.0% |
| | **ML Perceptron** | 76.0% | 77.0% | 78.0% | 72.0% | 83.0% |
| | **SMO** | 82.0% | 78.0% | 79.0% | 74.0% | 77.0% |
| | **RandomForest** | 78.0% | 78.0% | 73.0% | 70.0% | 75.0% |
| $D_{dorn}$ | **BayesNet** | 75.0% | 68.1% | 74.7% | 68.1% | 75.8% |
| | **ML Perceptron** | 74.2% | 70.3% | 72.5% | 69.4% | 76.9% |
| | **SMO** | 79.7% | 71.9% | 76.7% | 71.7% | 83.6% |
| | **RandomForest** | 75.8% | 68.9% | 71.7% | 74.2% | 76.4% |
| $D_{new}$ | **BayesNet** | 63.5% | 70.5% | 64.0% | 69.5% | 71.5% |
| | **ML Perceptron** | 67.5% | 67.0% | 68.5% | 71.5% | 74.0% |
| | **SMO** | 65.5% | 66.0% | 72.5% | 73.0% | 82.0% |
| | **RandomForest** | 65.5% | 60.0% | 63.0% | 65.5% | 74.5% |

Table 4.6: Accuracy achieved by *All-Features*, *TF*, *BWF*, *PF*, and *DF* in the three data sets with different machine learning techniques.

## 4.4 Threats to Validity

Possible threats to validity are related to the methodology in the construction of the new dataset, to the machine learning technique used and to the feature selection technique adopted. In this section we discuss such threats, grouping them into *construct*, *internal* and *external* validity.

**Construct Validity**. The results could depend on the machine learning technique used for computing the accuracy of each model. Table 4.6 shows the accuracy achieved by each model using different machine learning techniques. While different techniques achieve different levels of accuracy, some results are still valid when using other classifiers, *e.g.,* the combined model achieves a better accuracy than any other model on all the data sets, except for the dataset by Buse and Weimer when using *BayesNet* and *RandomForest*.

**Internal validity**. To mitigate the over-fitting problem of machine learning techniques, we used 10-fold cross-validation, and we performed statistical analysis (Wilcoxon test, effect size, and confidence intervals) in order to measure the significance of the differences among the accuracies of different models. Also, feature selection could affect the final results on each model. Finding the best set of features in terms of achieved accuracy is infeasible when the number of features is large. Indeed, the number of subsets of a set of $n$ elements is $2^n$; while

an exhaustive search is possible for models with a limited number of features, like *BWF*, *PF* and *TF*, it is unacceptable for *DF* and *All-Features*. Such a search would require, respectively, $1.2 \times 10^{18}$ and $3.2 \times 10^{34}$ subset evaluations. Thus, we used a linear forward selection technique [64] in order to reduce the number of evaluations and to obtain a good subset in a reasonable time.

Comparing models obtained with exhaustive search to models obtained with a sub-optimal search technique could lead to biased results; therefore, we use the same feature selection technique for all the models to perform a fairer comparison. It is worth noting that the likelihood of finding the best subset remains higher for models with less features.

Another threat to internal validity is the use of datasets of different sizes: Buse and Weimer involved 120 participants and they collected 12,000 evaluations; Dorn involved over 5,000 participants and he collected 76,741 evaluations (each snippet was evaluated, on average, by about 200 participants); we involved nine participants and we collected 1,800 evaluations. Besides, each data set implies also its own risks. The main problem of the data set by Buse and Weimer is that it contains also not compilable snippets; one of the textual features we introduced, *Textual Coherence*, can only be computed on syntactically correct snippets. In the data set by Dorn, each participant evaluated a small subset of snippets, 14/360 on average; in this case, there could be the risk that the difference in rating is a matter of the difference among evaluators more than the difference among snippets. Finally, the main threat to validity related to our data set is the small number of evaluators. Therefore, since each data set complements the others, to reduce the risks we report the results on all of them. However, since the number of evaluators are different, we compare the models on the three data sets separately.

**External validity**. In order to build the new data set, we had to select a set of snippets that human annotators would evaluate. The set of snippets selected may not be representative enough and, thus, could not help to build a generic model. We limited the impact of such a threat by selecting the set of the most distant snippets as for the features used in this study through a greedy center selection technique. Other threats regarding the human evaluation of the readability of snippets, also pointed out by Buse and Weimer [21], are related

to the experience of human evaluators and to the lack of a rigorous definition of readability. However, the human annotators for $D_{new}$ showed a high agreement on the readability of snippets.

## 4.5 Final Remarks

State-of-the-art code readability models mostly rely on syntactic metrics, and as of today they consider the impact of textual aspects on code readability in a very limited way. In this chapter we present a set of textual features that are based on the source code vocabulary analysis and aim at improving the accuracy of code readability models. The proposed textual features measure the consistency between source code and comments, specificity of the identifiers, usage of complete identifiers, among the others. To validate our hypothesis, stating that combining structural, visual, and textual features improves the accuracy of readability models, we used the features proposed by the state-of-the art models as a baseline, and measured (i) to what extent the proposed textual-based features complement the features proposed in the literature for predicting code readability, and (ii) the accuracy achieved when including textual features into the state-of-the-art models.

In summary, the lessons learned from this chapter are the following:

- textual feature complement structural and visual features in capturing code readability;
- textual features help increasing the accuracy of readability assessment models.

However, there are still some open issues:

- all the readability prediction models are based on datasets in which readability is measured asking developers to rate snippets on a Likert scale. This subjective evaluation could be biased in some ways (*e.g.,* by the skills with some APIs used in the code). Future work should be aimed at trying to define proper ways of determining if a snippet is readable or unreadable for a given developer.

- all the models from the state-of-the-art introduce some features. However, it is not completely clear what developers should do to make their code readable. Future work should address such an issue by using the collected data to define some code readability guidelines.

# CHAPTER 5

## Using Code Readability to Predict Quality Issues

**Contents**



## 5.1 Introduction

Buse and Weimer [21] conducted a study in which they tried to correlate code readability with a different measure of code quality. Specifically, they chose to use the number of warnings reported by FindBugs, a static code analysis tool, as an indicator of code quality: the higher the number of warnings, the lower the quality. They showed that readability is negatively correlated with the number

of warnings raised by FindBugs. Such a correlation shows that unreadable code usually also presents other quality issues. However, FindBugs performs many different analyses and it raises different types of warnings. It is still unclear which categories of warnings appear more frequently in unreadable code.

In this chapter we present a replication of the study performed by Buse and Weimer [21]. Our hypothesis is that, if readability is actually correlated with FindBugs warnings, an improvement in the accuracy of readability prediction should imply an improvement of such a correlation. We analyzed 20 open-source Java software systems, totaling in 3M lines of codes and 7K methods and we show that using the readability predicted by a model which uses structural, visual and textual features (*i.e.,* the one which achieves the best readability prediction accuracy, as shown in Chapter 4) we have a higher correlation with the presence of FindBugs warnings. We also try to explain why such a correlation is present, providing examples and a more in-depth analysis. Specifically, we perform such an analysis considering (i) all the warnings and (ii) warnings divided by category. We show that code readability is correlated with the presence of some categories of FindBugs warnings (*e.g.,* Dodgy code) but not with others (*e.g.,* Malicious code).

This chapter is organized as follows. Section 5.2 describes the empirical study in which we computed the correlation between code readability and quality warnings as captured by FindBugs; Section 5.3 describes the threats to validity of such a study; Section 5.4 concludes this chapter.

## 5.2 Empirical Study

This study is a replication of the study performed by Buse and Weimer [21], in which the authors used readability as a proxy for quality, in particular, using warnings reported by the FindBugs tool [1] as an indicator of quality. Specifically, the *goal* of the study is to understand if the model which achieves the best accuracy in readability prediction (*i.e.,* the *All-features* model) can predict FindBugs warnings with a higher accuracy compared to the model originally proposed by Buse and Weimer [21]. It is worth noting that we are not directly using the met-

[1] `http://findbugs.sourceforge.net/`

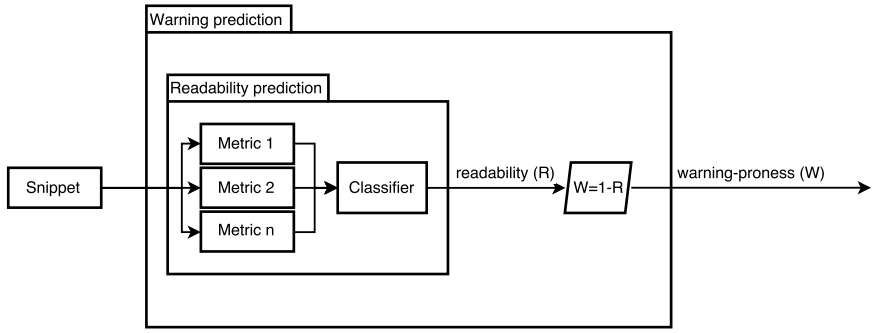



Figure 5.1: Workflow followed to predict readability by relying on FindBugs warnings.

rics defined in Section 4.2 as predictors of FindBugs warnings: instead, we first use some of the features previously defined to predict readability, and then we use readability to predict warnings. It is not the goal of this study to assess the FindBugs warnings prediction power of the metrics used to predict readability. The *quality focus* is to improve the prediction of quality warnings by considering readability metrics, while the *perspective* of the study is of a researcher interested in analyzing whether the proposed approach can be used for the prediction of quality problems in source code.

### 5.2.1 Empirical Study Design

This study was steered by the following research question:

- **$RQ_1$:** *Is the combined readability model able to improve the prediction accuracy of quality warnings?* With this question we want to understand if a higher accuracy in readability prediction helps to improve the correlation between FindBugs warnings and readability. In other words, we want to re-assess the FindBugs warnings prediction power of readability models.

The *context* of this study is comprised of 20 Java programs: 11 of the 15 systems analyzed by Buse and Weimer [21] and nine systems introduced in study. We did not include four of the 15 system cited in the study by Buse and Weimer [21] (*i.e.,* Gantt Project, SoapUI, Data Crow and Risk) because the snapshots of the specific versions of those systems were not available at the time when this study was performed. In order to select the nine new systems, we first randomly chose from SourceForge some software categories which were not represented by the other systems and, for each of them, we chose one of the most downloaded ones. We started from the most downloaded project, and we selected the first one which had the following characteristics:

- developed in Java: this was necessary because FindBugs is only able to analyze bytecode binaries, resulting from the compilation of Java and few other languages;
- source code was available, *i.e.,* there was a public repository or it was released as a zip file: this was necessary in order to compute the readability score of the methods;
- either a build automation tool was used, such as Ant, Maven or Gradle, or it was available as a compiled jar file of the exact same version of the source code: this was necessary to have a reasonably easy way to provide FindBugs with compiled programs to analyze.

Table 5.1 depicts the selected systems, which accounts for 103,000 methods and about 3 million lines of code. The star symbol indicates software systems added in this study. "Methods with warnings" indicates the number of methods with at least a warning.

In order to answer $\mathbf{RQ}_1$, we followed the process depicted in Figure 5.1. First, we trained a Logistic classifier on the dataset defined in the previous chapter and we computed the readability score of all the methods of all the systems using our combined model. The readability score is defined as the probability that a method belongs to the class "readable" according to the classifier. Such a value ranges between 0 (surely unreadable) and 1 (surely readable). For each method, we also computed the unreadability score, which is the probability

that a snippet belongs to the class "unreadable". Such a score is computed as $unreadability(M) = 1 - readability(M)$. As a second step, we ran the FindBugs tool on all the compiled versions of the analyzed systems. Then, we extracted from the FindBugs report only the warnings reported at method level: indeed, FindBugs warnings can also concern lines of code which belong to other parts of a class (*e.g.,* field declarations). We discarded such warnings, so that we have a readability score (computed at method level) for each warning.

Given a system $S$ having a set $X$ of methods, we split $X$ in two sub-sets: $X_b$, methods with at least a warning, and $X_c$, warning-free methods. In order to avoid the bias derived from the different size of the sets, we sub-sample $X_b$ and $X_c$: we consider $m = min(|X_b|, |X_c|)$ and we randomly pick, from each set, $m$ elements. At the end, we have two sets $X_{bs} \subseteq X_b$ and $X_{cs} \subseteq X_c$, so that $|X_{bs}| = |X_{cs}|$. This sub-sampling procedure was the same applied by Buse and Weimer [21].

Finally, we used the unreadability score ($unreadability(M)$) to predict FindBugs warnings. In order to evaluate how accurate is the unreadability score to predict FindBugs warnings we first plotted the Receiving Operating Curve (ROC) obtained using unreadability as a continuous predictor of warning/no warning: such a curve shows the true-positive rate (TFP) against the false-positive rate (FPR) considering different thresholds for the predictor (unreadability). Then, we computed the area under such a curve (Area Under the Curve - AUC). We preferred AUC over F-measure, originally used by Buse and Weimer [21], because AUC does not require the choice of a threshold, which may not be the same for all the models. To answer $\mathbf{RQ}_1$, we compared three readability models: (i) the original model proposed by Buse and Weimer trained on their dataset ( $R\langle BWF\rangle \circ BW$ )[1]; (ii) the model by Buse and Weimer trained on the new dataset ( $R\langle BWF\rangle \circ New$); (iii) our model containing all the features trained on the new dataset ( $R\langle$ *All-features* $\rangle \circ New$). We included the first model as a sanity check and we used the tool provided by the authors to compute the readability score; then we trained both $R\langle BWF\rangle$ and $R\langle$ *All-features* $\rangle$ on the same dataset, so that there is no bias caused by the different training set.

[1]We use the operator $\circ$ to denotate that a model $M$ is trained with dataset $X$: $M \circ X$

| Project name | LOC | Methods analyzed | Methods with warnings | SourceForge category |
|---|---|---|---|---|
| Azureus: Vuze 4.0.0.4 | 651k | 30,161 | 2,508 | Internet file sharing |
| JasperReports 2.04 | 269k | 11,256 | 367 | Dynamic content |
| StatSVN 0.7.0 * | 244k | 441 | 21 | Documentation |
| aTunes 3.1.2 * | 216k | 11,777 | 501 | Sound |
| Hibernate 2.1.8 | 189k | 4,954 | 192 | Database |
| jFreeChart 1.0.9 | 181k | 7,517 | 410 | Data representation |
| FreeCol 0.7.3 | 167k | 4,270 | 283 | Game |
| TV Browser 2.7.2 | 162k | 7,517 | 862 | TV guide |
| Neuroph 2.92 * | 160k | 2,067 | 179 | Frameworks |
| jEdit 4.2 | 140k | 5,192 | 518 | Text editor |
| Logisim 2.7.1 * | 137k | 5,771 | 232 | Education |
| JUNG 2.1.1 * | 74k | 3,559 | 156 | Visualization |
| Xholon 0.7 | 61k | 3,489 | 338 | Simulation |
| DavMail 4.7.2 * | 52k | 1,793 | 80 | Calendar |
| Portecle 1.9 * | 27k | 532 | 37 | Cryptography |
| Rachota 2.4 * | 23k | 791 | 112 | Time tracking |
| JSch 0.1.37 | 18k | 603 | 170 | Security |
| srt-translator 6.2 * | 8k | 103 | 26 | Speech |
| jUnit 4.4 | 5k | 660 | 18 | Software development |
| jMencode 0.64 | 3k | 253 | 80 | Video encoding |
| *Total* | 3M | 103k | 7k | |

Table 5.1: Software systems analyzed for the study.

### 5.2.2 Empirical Study Results

Figure 5.2 shows the AUC achieved by the three readability models on all the analyzed systems. $R\langle$ *All-features* $\rangle \circ New$ is able to predict FindBugs warnings more accurately than the baselines on 12 systems out of 20. The AUC achieved by such a model ranges between 0.573 (Neuroph) and 0.900 (aTunes). Figure 5.3 shows three box plots which indicate, for each model, the AUC achieved on the 20 systems analyzed. Here it is clear that $R\langle$ *All-features* $\rangle \circ New$ generally achieves a higher AUC as compared to the other models. Specifically, the mean AUC achieved by $R\langle BWF\rangle \circ BW$ is 0.717, the AUC achieved by $R\langle BWF\rangle \circ New$ is 0.724 while the AUC achieved by $R\langle$ *All-features* $\rangle \circ New$ is 0.770. We also report in Figure 5.4 the box plot relative only to the 11 projects also considered in the original experiment by Buse and Weimer [21].

Furthermore, we checked if the difference is statistically significant (p=0.05) performing a paired Wilcoxon test [42] with p-values adjusted using the Holm's correction procedure for multiple pairwise comparisons [70]: the adjusted p-values resulting from such a test are 0.006 (comparison with $R\langle BWF\rangle \circ BW$) and 0.004 (comparison with $R\langle BWF\rangle \circ New$) with a medium effect size (0.375 and 0.360 correspondingly), which suggest that $R\langle$ *All-features* $\rangle \circ New$ has a significantly higher AUC compared to the two baselines. Figure 5.5 illustrates the difference

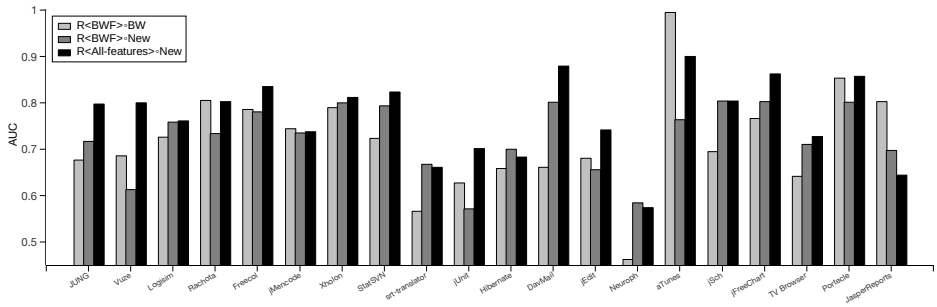



Figure 5.2: AUC achieved using readability models to predict FindBugs warnings (by system).

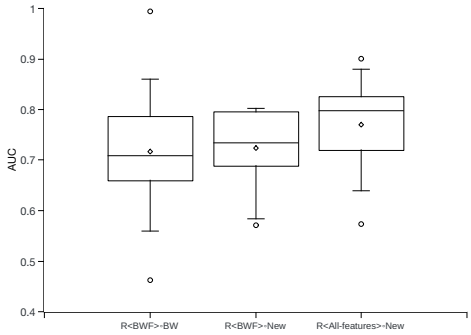



Figure 5.3: AUC achieved using readability models to predict FindBugs warnings (overall).

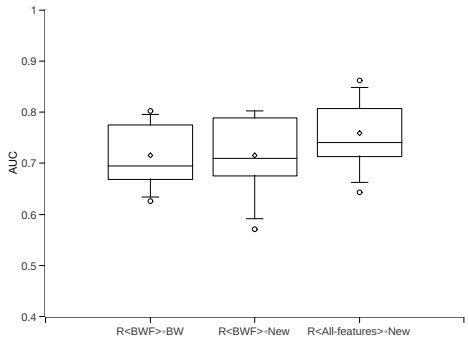



Figure 5.4: AUC achieved using readability models to predict FindBugs warnings (projects from the original study).

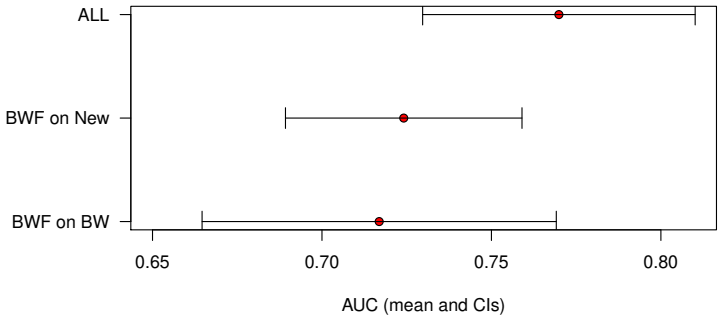



Figure 5.5: Mean accuracy and confidence intervals (CIs).

in the AUC achieved with each model by using the mean AUC and confidence intervals (CIs). The CIs show how there is overlap between the three models, however there is a region of the CI of $R\langle$ *All-features* $\rangle \circ New$ that is higher than the other CIs, which confirms the medium effect size of the significant difference between $R\langle$ *All-features* $\rangle \circ New$ and the other two models. There is a 95% of confidence that the mean AUC achieved by $R\langle$ *All-features* $\rangle \circ New$.

The results suggest that the answer to $\mathbf{RQ}_1$ is *positive*: an improvement in the prediction accuracy of readability results in a better prediction of FindBugs

warnings. Such a result further corroborates the findings by Buse and Weimer [21] about the correlation between readability and FindBugs warnings.

Furthermore, we wanted to understand which categories of FindBugs warnings correlated with readability the most. We selected a set of six categories of FindBugs warnings, *i.e.,* "Performance", "Correctness", "Bad Practices", "Malicious code", "Dodgy code" and "Internationalization". Categories described on the official FindBugs website[2], which are not represented for many of the analyzed systems, such as "Security", were excluded.

Figure 5.7 shows the AUC achieved by the best model (the $R\langle$ *All-features* $\rangle \circ New$) on different categories of FindBugs warnings. "Dodgy code" is the best predicted category for seven systems out of 20, "Correctness" is the best one for seven systems out of 20, while the others are the best predicted more rarely ("Internationalization" and "Performance" for 5 systems and "Bad practice" for four systems). "Malicious code" is the category with the lowest prediction accuracy. It is worth noting that any of those categories is directly related to code readability.

Analyzing the results more in depth, Figure 5.8 shows the box plots of the AUC achieved by $R\langle$ *All-features* $\rangle \circ New$ on the analyzed categories of FindBugs warnings. Except for "Malicious code", for which the mean AUC is 0.470, the warning belonging to all the other categories are predicted with a mean AUC above 0.7. The main reason why "Malicious code" is not correlated with readability is that the most frequent warnings belonging to such a category can be found in very simple and short snippets. Figure 5.6 shows an example class with a method (getKey) with the warning "May expose internal representation by returning reference to mutable object". Such a warning is raised when "Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object.". In many cases this warning can be found in "getter" methods, such as the one in the example, which have, obviously, a higher level of readability.

Therefore, the first finding is that possible malicious/vulnerable code, but specifically code which involve the exposure of internal representation, is not correlated with readability and, on the other hand, all other kind of possible

[2] `http://findbugs.sourceforge.net/bugDescriptions.html`

```
public class KeyParameter
  implements CipherParameters
{
    private byte[]  key;

    [...]

    public byte[] getKey()
    {
        return key;
    }
}
```

Figure 5.6: Example of code that FindBugs classifies as "Malicious code".

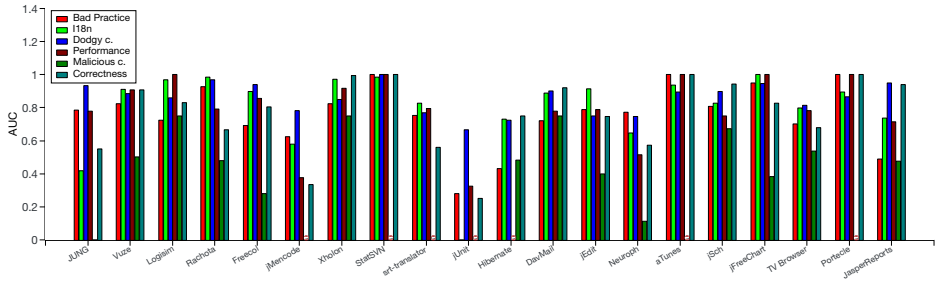



Figure 5.7: AUC achieved using readability models to predict FindBugs warnings (by warning category and system).

programming mistakes detected by FindBugs, like stylistic issues or performance problems, could be predicted reasonably well with readability. The differences between the means of the AUC achieved on all the categories is not significant. Nevertheless, if we take into account the minimum AUC achieved for each category, "Dodgy code" is the category more reliably predicted by readability. The minimum AUC achieved for such a category is 0.667 (the only case in which it is less than 0.7) on jUnit, but, on the same system, all other categories are predicted with a very low AUC.

While the correlation between unreadability and FindBugs warnings is strong and the former can be used to predict the latter, it is not trivial to understand why FindBugs warnings are more frequent in methods with lower readability. Indeed, it is worth noting that, in some cases, it is possible to rearrange the code so that it is more readable and it still has the same FindBugs warning.

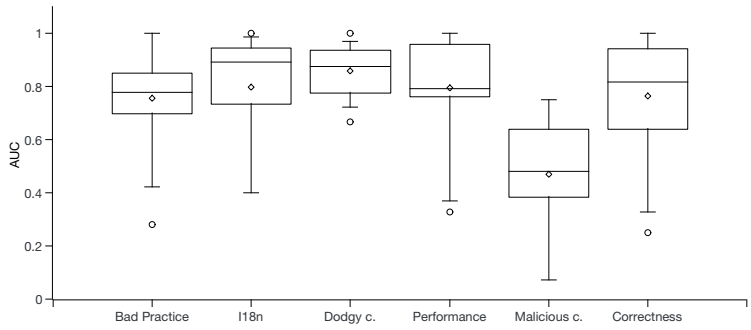



Figure 5.8: AUC achieved using readability models to predict FindBugs warnings (by warning category).

The cause of the correlation could be that unreadable code is more likely to have hidden mistakes, which may not be fixed until the system fails or a tool warns the developers about it. Consider the snippet in Figure 5.9. FindBugs shows a warning belonging to the category "Dodgy code", specifically "Useless object created". According to the official documentation, this warning is reported when an object is "created and modified, but its value never go outside of the method or produce any side-effect.". In this case, the variable declared in line 6 is used in lines 13, 26, 38 and 49, but it has no effect on the outside of the method, so it can be removed, together with the lines in which it is used. Noticing this kind of issues on an unreadable method such as the one proposed in the example could be very hard, and this may be the reason why it is introduced and it remains in the code. In readable code, instead, such warnings may be less frequent because they would be clearly visible either to the developer who writes it or to any other developer who maintains the source code.

**Summary for** $RQ_1$. Our study confirms that the correlation between warnings and readability is high and it suggests that a model which predicts readability with a higher accuracy is also able to predict FindBugs warnings better. Specifically, all the categories of warnings taken into account are well-predicted, except for "Malicious code", which is more frequent in small and readable methods, like "getter" methods.

```
static protected LocaleUtilDecoderCandidate[] getTorrentCandidates(TOTorrent torrent)
        throws TOTorrentException, UnsupportedEncodingException {
    long lMinCandidates;
    byte[] minCandidatesArray;

    Set cand_set = new HashSet();
    LocaleUtil localeUtil = LocaleUtil.getSingleton();

    List candidateDecoders = localeUtil.getCandidateDecoders(torrent.getName());
    lMinCandidates = candidateDecoders.size();
    minCandidatesArray = torrent.getName();

    cand_set.addAll(candidateDecoders);
    TOTorrentFile[] files = torrent.getFiles();

    for (int i = 0; i < files.length; i++) {
        TOTorrentFile file = files[i];
        byte[][] comps = file.getPathComponents();

        for (int j = 0; j < comps.length; j++) {
            candidateDecoders = localeUtil.getCandidateDecoders(comps[j]);
            if (candidateDecoders.size() < lMinCandidates) {
                lMinCandidates = candidateDecoders.size();
                minCandidatesArray = comps[j];
            }
            cand_set.retainAll(candidateDecoders);
        }
    }

    byte[] comment = torrent.getComment();

    if (comment != null) {
        candidateDecoders = localeUtil.getCandidateDecoders(comment);
        if (candidateDecoders.size() < lMinCandidates) {
            lMinCandidates = candidateDecoders.size();
            minCandidatesArray = comment;
        }
        cand_set.retainAll(candidateDecoders);
    }

    byte[] created = torrent.getCreatedBy();

    if (created != null) {
        candidateDecoders = localeUtil.getCandidateDecoders(created);
        if (candidateDecoders.size() < lMinCandidates) {
            lMinCandidates = candidateDecoders.size();
            minCandidatesArray = created;
        }
        cand_set.retainAll(candidateDecoders);
    }

    List candidatesList = localeUtil.getCandidatesAsList(minCandidatesArray);
    LocaleUtilDecoderCandidate[] candidates;
    candidates = new LocaleUtilDecoderCandidate[candidatesList.size()];
    candidatesList.toArray(candidates);

    Arrays.sort(candidates, new Comparator() {
        public int compare(Object o1, Object o2) {
            LocaleUtilDecoderCandidate luc1 = (LocaleUtilDecoderCandidate) o1;
            LocaleUtilDecoderCandidate luc2 = (LocaleUtilDecoderCandidate) o2;
            return (luc1.getDecoder().getIndex() - luc2.getDecoder().getIndex());
        }
    });

    return candidates;
}
```

Figure 5.9: Example of unreadable code with a “Dodgy code” warning.

## 5.3 Threats to Validity

The threats to validity for our study are mainly related to the analyzed systems and to the metrics used to evaluate the correlation between readability and FindBugs warnings. In this section we discuss such threats, grouping them into *internal* and *external* validity.

**Internal Validity**. The results strongly depend on the actual implementation of the models. An error in the computation of the features introduced by Buse and Weimer [21] may negatively affect the performance of their model. To minimize such a risk, we used the original implementation of the metrics provided by the authors, and we used Weka to re-train the logistic classifier using the larger dataset. We used AUC for evaluating the prediction power of readability to predict warnings. We did this because other metrics would have implied the use of a specific threshold, while we wanted to compute the inherent correlation between a continuous metric (readability) and a discrete value (presence of FindBugs warnings). Finally, it could be argued that the relationship between unreadability and the presence of FindBugs warnings is caused by the size of the methods: larger methods could more likely make FindBugs raise a warning and they could be usually less readable than shorter ones. We used the Spearman's rank correlation coefficient $\rho$ to compute the correlation between number of FindBugs warnings and number of lines of code. Such a correlation is $\sim 0.240$: while larger methods are clearly more prone to cause FindBugs warnings, the correlation is only *weak*.

**External Validity**. We had to select a set of systems for computing the correlation between readability and FindBugs warnings. We selected a subset of the systems analyzed in the previous study by Buse and Weimer [21] and we introduced new systems for such a study. The main threats are that (i) the systems may not be representative enough and (ii) some of the systems may use FindBugs, and thus the use of such a tool may influence the natural correlation with readability. We limited the first threat by selecting systems belonging to different categories and having different sizes in terms of methods and lines of

code. Besides, we limited the second threat by checking if the number of FindBugs warnings was not too low (*e.g.,* similar to 0) on each system.

## 5.4 Final Remarks

In this chapter we reported a replication of a study by Buse and Weimer [21] on the correlation between readability and FindBugs warnings, in order to check if an improvement in readability prediction achieved using the comprehensive model introduced in Chapter 4 causes an improvement in the correlation with FindBugs warnings.

In summary, the lessons learned from this chapter are the following:

- the model with the highest readability prediction accuracy also predicts FindBugs warnings more accurately than the other models;
- unreadable code is more prone to having issues, which may become bugs, and it is more likely that such problems would stay in the code, as it is more difficult to notice and correct them; this finding confirms we what Buse and Weimer observed in their original study [21]
- some categories of FindBugs warnings are more correlated with readability than others; for example, *Malicious Code* shows a very low correlation.

There is also a main open issue: FindBugs warnings do not represent real bugs, therefore it is still unclear if unreadable code is also more bug-prone. Future work should fill this gap.

# CHAPTER 6

## Renaming Identifiers to Improve Code Readability

**Contents**



## 6.1 Introduction

In programming languages, identifiers are used to name program entities; *e.g.,* in Java, identifiers include names of packages, classes, interfaces, methods, and variables. Identifiers account for ∼30% of the tokens and ∼70% of the characters in the source code [41]. Naming identifiers in a careful, meaningful, and consistent

manner likely eases program comprehension and supports developers in building consistent and coherent conceptual models [41].

Instead, poorly chosen identifiers might create a mismatch between the developers' cognitive model and the intended meaning of the identifiers, thus ultimately increasing the risk of fault proneness. Indeed, several studies have shown that bugs are more likely to reside in code with low quality identifiers [22, 2]. Arnaoudova *et al.* [9] also found that methods containing identifiers with higher physical and conceptual dispersion are more fault-prone. This suggests the important role played by a specific class of identifiers, *i.e.,* local variables and method parameters, in determining the quality of methods. We showed in Chapter 4 that identifiers quality have an important role in code readability.

Naming conventions can help to improve the quality of identifiers. However, they are often too general, and cannot be automatically enforced to ensure consistent and meaningful identifiers. For example, the Java Language Specification[1] indicates three rules for naming local variables and parameter names: (i) *"should be short, yet meaningful"*; (ii) *"one-character identifiers should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type"*; (iii) *"identifiers that consist of only two or three lowercase letters should not conflict with the initial country codes and domain names that are the first component of unique package names"*.

All these requirements do not guarantee that developers will name variables in a consistent way. For example, developers might use "`localVar`" and "`varLocal`" in different source code locations even if these two names are used in the same context and with the same meaning. Also, synonyms might be used to name the same objects, such as "`car`" and "`auto`". Finally, developers might not completely adhere to the rules defined in project-specific naming conventions.

Researchers have presented tools to support developers in the consistent use of identifiers. Thies and Roth [140] analyzed variable assignments to identify pairs of variables likely referring to the same object but named with different identifiers. Allamanis *et al.* [5] pioneered the use of NLP techniques to support identifiers renaming. Their NATURALIZE tool exploits a language model to infer from a code base the naming conventions and to spot *unnatural* identifiers (*i.e.,*

[1] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html`

unexpected identifiers used to name variables, parameters, and methods), that should be renamed to promote consistency.

To obtain a reliable evaluation of approaches supporting automatic identifier renaming, the original authors of the source code should be involved in assessing the meaningfulness of the suggested refactorings. However, running such evaluations is expensive. Therefore, refactoring techniques are often evaluated in "artificial scenarios" (*e.g.,* injecting a meaningless identifier in the code and check whether the tool is able to recommend a rename refactoring for it) and/or by relying on the manual evaluation of a limited number of recommended rename refactorings. For example, Thies and Roth [140] manually assessed the meaningfulness of 32 recommendations generated by their tool. Instead, Allamanis *et al.* [5] firstly analyzed 33 rename recommendations generated by NATURALIZE, and then opened pull requests in open source projects to evaluate, with the help of the projects' contributors, the meaningfulness of 18 renaming recommended by NATURALIZE (for a total of 51 data points).

We aim at assessing the meaningfulness of the rename refactorings recommended by state-of-the-art approaches on a larger scale (922 evaluations in total) and by only relying on developers having a first-hand experience on the object systems of our study.

Also, we directly compare the following techniques and study their complementarity:

- The approach proposed by Thies and Roth [140] is used as representative of the techniques exploiting static code analysis to recommend rename refactoring;
- The NATURALIZE tool [5] is the only one in the literature using NLP techniques to support identifier renaming;
- Finally, we propose a variation of the NATURALIZE tool, named LEAR (LExicAl Renaming), exploiting a different concept of language model more focused on the lexical information present in the code.

This chapter is organized as follows: Section 6.2 illustrates LEAR; in Section 6.3 we report the evaluation study and the comparison with the baseline

approach; Section 6.4 discusses the threats to validity; Section 6.5 concludes this chapter.

## 6.2 LEAR: LExicAl Renaming

LEAR takes as input a Java system for which the developer is interested in receiving rename refactoring recommendations. Note that LEAR does only recommend renaming operations related to (i) variables declared in methods, and (ii) method parameters. The renaming of methods/classes as well as of instance/-class variables is currently not supported. In the following we describe in detail the main steps of LEAR.

**Identifying methods and extracting the vocabulary.** LEAR parses the source code of the input system by relying on the `srcML` infrastructure [34]. The goal of the parsing is to extract (i) the complete list of methods, and (ii) the identifiers' vocabulary, defined as the list of all the identifiers used to name parameters and variables (declared at both method and class level) in the whole project. From now on we refer to the identifiers' vocabulary simply as the *vocabulary*. Once the vocabulary and the list of methods have been extracted, the following steps are performed for each method $m$ in the system. We use the method in Listing 6.1 as a running example.

**N-gram Extraction from $m$.** We extract all textual tokens from the method $m$ under analysis, by removing (i) comments and string literals, (ii) all non-textual content, *i.e.,* punctuation and (iii) non-interesting words, such as Java keywords and the name of method $m$ itself. Basically, we only keep tokens referring to identifiers, excluding the name of $m$, and non-primitive types, which are Java keywords. This is one of the main differences with respect to NATURALIZE, the approach LEAR is inspired from.

Indeed, while NATURALIZE uses all textual tokens in the $n$-gram language model (including, *e.g.,* Java keywords), we only focus on tokens containing *lexical* information. The reason is that we expect sequences of only lexical tokens to better capture and characterize the context in which a given identifier is used.

From the `printUser` method in the example, the list of identifiers will include: `uid`, `String`, `q`, `uid`, `User`, `user`, `runQuery`, `q`, `System`, `out`, `println`, `user`.

Listing 6.1: Example of method analyzed

```
public void printUser(int uid) {
    String q = "SELECT * WHERE user_id = " + uid;
    User user = runQuery(q);
    System.out.println(user);
}
```

Again, our conjecture is that such a list of tokens captures the *context*—referred to method `printUser`—where an identifier (*e.g.,* `q`) is used. Once obtained such a list of tokens, we extract a set of $n$-grams from it that will be used by the language model to estimate the probability that a specific identifier should be used in a given context.

Lin *et al.* [89] found that the $n$-gram language model achieves the best accuracy in supporting code completion tasks when setting $n = 3$ (they experimented with values of $n$ going from 3 to 15 at steps of one). The same value has also been used in the original work by Hindle *et al.* [69] proposing the usage of the language model for code completion. For these reasons, we build 3-grams from the extracted list of tokens. In our running example, this would result in the extraction of ten 3-grams, including: ⟨`uid`, `String`, `q`⟩, ⟨`String`, `q`, `uid`⟩, ⟨`q`, `uid`, `User`⟩, ⟨`uid`, `User`, `user`⟩, *etc.*

**Generating candidate rename refactoring.** For each variable/parameter identifier in $m$ (in the case of `printUser`: `uid`, `q`, and `user`), LEAR looks for its possible renaming by exploiting the *vocabulary* built in the first step. Given an identifier under analysis *id*, LEAR extracts from the *vocabulary* all the identifiers $id_s$ which meet the following constraints:

- $C_1$: $id_s$ is used to name a variable/parameter of the same type as the one referred by *id*. For example, if *id* is a parameter of type `int`, $id_s$ must be used at least once as an `int` variable/parameter;
- $C_2$: $id_s$ is not used in $m$ to name any other variables/parameters. Indeed, in such a circumstance, it would not be possible to rename *id* in $id_s$ in any case;
- $C_3$: $id_s$ is not used to name any attribute of the class $C_k$ implementing $m$ nor in any class $C_k$ extends, for the same reason explained in $C_2$.

The checking of these constraints represent another difference between LEAR and NATURALIZE (*i.e.,* they are not considered in NATURALIZE).

We refer to the list of valid identifiers fulfilling the above criteria as $VI_{id}$. Then, LEAR uses a customized version of the 3-gram language model to compute the probability that each identifier $id_s$ in $VI_{id}$ appears, instead of $id$, in all the 3-grams of $m$ including $id$.

Let $TP_{id}$ be the set of *3-gram patterns* containing at least once $id$, and $tp_{id \to id_s}$ be a 3-gram obtained from a pattern $tp_{id} \in TP_{id}$ where the variable $id$ is replaced with a valid identifier $id_s \in VI_{id}$. We define the probability of a given substitution to a variable as:

$$P(tp_{id \to id_s}) = \frac{count(tp_{id \to id_s})}{\sum_{y \in VI_{id}} count(tp_{id \to y})}$$

When the pattern is in the form of $\langle id_1, id_2, id \rangle$, the probability of a substitution corresponds to the classic probability as computed by a 3-gram language model, that is:

$$P(\langle id_1, id_2, id \rangle_{id \to id_s}) = P(id_s | id_1, id_2) = \frac{count(\langle id_1, id_2, id_s \rangle)}{count(\langle id_1, id_2 \rangle)}$$

To better understand this core step of LEAR, let us discuss what happens in our running example when LEAR looks for possible renaming of the `uid` parameter identifier. The 3-grams of `printUser` containing `uid` are: ⟨ `uid`, `String`, `q` ⟩, ⟨ `String`, `q`, `uid` ⟩, ⟨ `q`, `uid`, `User` ⟩, and ⟨ `uid`, `User`, `user` ⟩.

Assume that the list of identifiers $VI_{id}$ (*i.e.,* the list of valid alternative identifiers for `uid`) includes `userId` and `localCount`. LEAR uses the language model to compute the probability that `userId` occurs in each of the 3-grams of `printUser` containing `uid`. For example, the probability of observing `userId` in the 3-gram ⟨`q`, `uid`, `User`⟩ is:

$$p(\texttt{q}, \texttt{userId}, \texttt{User}) = \frac{count(\texttt{q}, \texttt{userId}, \texttt{User})}{count(\texttt{q}, \texttt{y}, \texttt{User})}$$

where $count(\texttt{q}, \texttt{userId}, \texttt{User})$ is the number of occurrences of the 3-gram ⟨`q`, `userId`, `User`⟩ in the system, and $count(\texttt{q}, \texttt{y}, \texttt{User})$ is the number of occurrences

of the corresponding 3-gram, where `y` represents any possible identifier (including `userId` itself). Note that the *count* function only considers $n$-grams where $id_s$ has the same type as $id$. Also, it does not take into account $n$-grams extracted from the method under analysis. This is done to avoid favoring the probability of the current identifier name used in the method under analysis as compared to the probability of other identifiers.

How the probability for a given identifier to appear in a $n$-gram is computed also differentiates LEAR from NATURALIZE. In the example reported above, NATURALIZE in fact computes the probability of observing User following $\langle$q, userId$\rangle$:

$$p(\texttt{User}|\texttt{q}, \texttt{userId}) = \frac{count(\texttt{q}, \texttt{userId}, \texttt{User})}{count(\texttt{q}, \texttt{userId})}$$

The two probabilities (*i.e.,* the one computed by LEAR and by NATURALIZE), while based on similar intuitions, could clearly differ. Our probability function is adapted from the standard language model (*i.e.,* the one used by NATURALIZE) in an attempt to better capture the context in which an identifier is used. This can be noticed in the way our denominator is defined: it keeps intact that identifiers' context in which we are considering injecting `userId` instead of `uid`.

The average probability across all these 3-grams is considered as the probability of $id_s$ being used instead of $id$ in $m$.

This process results in a ranked list of $VI_{id}$ identifiers having on top the identifier with the highest average probability of appearing in all the 3-grams of $m$ as a replacement (*i.e.,* rename) of $id$. We refer to this top-ranked identifier as $T_{id}$.

Finally, LEAR uses the same procedure to compute the average probability that the identifier $id$ itself appears in the 3-grams where it currently is (*i.e.,* that no rename refactoring is needed). If the $T_{id}$ probability of appearing in the $id$ 3-grams is higher than $id$ probability, a candidate rename refactoring has been found (*i.e.,* rename $id$ in $T_{id}$). Otherwise, no rename refactoring is needed.

**Assessing the confidence and the reliability of the candidate recommendations.** LEAR uses two indicators acting as proxies for the *confidence* and the *reliability* of the recommended refactoring. Given a rename refactoring recommendation $id \rightarrow T_{id}$ in the method $m$, the confidence indicator is the av-

erage probability of $T_{id}$ to occur instead of $id$ in the tri-grams of $m$ where $id$ appears, computed via the customized language model detailed in the previous point. We refer to this indicator as $C_p$, and it is defined in the [0, 1] interval. The higher $C_p$, the higher the confidence of the recommendation. We study how the $C_p$ value influences the quality of the recommendations generated by LEAR in the following.

The "reliability" indicator, named $C_c$, is the number of distinct 3-grams used by the language model in the computation of $C_p$ for a given recommendation $id \rightarrow T_{id}$ in the method $m$. Given $\langle id_1, id_2, id \rangle$ a 3-gram where $id$ appears in $m$, we count the number of 3-grams in the system in the form $\langle id_1, id_2, x \rangle$, where $x$ can be any possible identifier. This is done for all the 3-grams of $m$ including $id$, and the sum of all computed values is represented by $C_c$. The conjecture is that the higher $C_c$, the higher is the reliability of the $C_p$ computation. Indeed, the higher $C_c$, the higher the number of 3-grams from which the language model learned that $T_{id}$ is a good substitution for $id$. $C_c$ is unbounded on top. We study what is the minimum value of $C_c$ allowing reliable recommendations in the following.

Note that while NATURALIZE does also provide a scoring function based on the probability derived by the $n$-gram language model to indicate the confidence of the recommendation (*i.e.,* the equivalent of our $C_p$ indicator), it does not implement a "reliability" indicator corresponding to $C_c$.

**Tuning of the $C_c$ and $C_p$ indicators.** To assess the influence of the $C_p$ (confidence) and $C_c$ (reliability) indicators on the quality of the rename refactorings generated by LEAR, we conducted a study on one system, named SMOS. We asked one of the SMOS developers (having nowadays six years of industrial experience) to assess the meaningfulness of the LEAR recommendations. SMOS is a Java Web-based application developed by a team of Master students (including the participant), and composed by 121 classes for a total of ∼23 KLOC. We used the SMOS system only for the tuning of the indicators $C_p$ and $C_c$, *i.e.,* to identify for both of them minimum values needed to receive meaningful recommendations.

SMOS is not used in the actual evaluation of our approach, presented in Section 6.3.

| **Original name** | **Rename** | $C_p$ | $C_c$ |
|---|---|---|---|
| mg | managerUser | 1.00 | 146 |
| e | invalidValueException | 0.90 | 356 |
| buf | searchBuffer | 0.89 | 5 |
| result | classroom | 0.87 | 15 |
| managercourseOfStudy | managerCourseOfStudy | 0.67 | 12 |

Table 6.1: Five rename refactoring tagged with a *yes*.

We ran LEAR on the whole system and asked the participant to analyze the 146 rename refactoring generated by LEAR and to answer, for each of them, the question *Would you apply the proposed refactoring?*, assigning a score on a three-point Likert scale: 1 (yes), 2 (maybe), and 3 (no). We clarified with the participant the meaning of the three possible answers:

**1** (yes) must be interpreted as "*the recommended renaming is meaningful and should be applied*", *i.e.,* the recommended identifier name is better than the one currently used;

**2** (maybe) must be interpreted as "*the recommended renaming is meaningful, but should not be applied*", *i.e.,* the recommended identifier is a valid alternative to the one currently used, but is not a better choice;

**3** (no) must be interpreted as "*the recommended rename refactoring is not meaningful*".

The participant answered *yes* to 18 (12%) of the recommended refactoring, *maybe* to 15, and *no* to 113. This negative trend is something expected, considering the fact that we asked the participant to assess the quality of the recommended refactoring independently from the values of the $C_p$ and the $C_c$ indicators. In other words, given the goal of this study, also recommendations having very low values for both indicators (*e.g.,* $C_p = 0.1$ and $C_c = 1$) were inspected, despite we expect these recommendations to not be meaningful. Table 6.1 reports five representative examples of rename refactoring tagged with a *yes* by the developer.

By inspecting the assessment performed by the participant, the first thing we noticed is that recommendations having $C_c < 5$ (*i.e.,* less than five distinct 3-grams have been used by the language model to learn the recommended rename

refactoring) are generally unreliable, and should not be considered. Indeed, out of the 28 rename refactoring having $C_c < 5$, one (3%) was accepted (answer "*yes*") by the developer and three (10%) were classified as *maybe*, despite the fact that 22 of them had $C_p = 1.0$ (*i.e.,* the highest possible confidence for the generated recommendation). Thus, when $C_c < 5$ even recommendations having a very high confidence are simply not reliable. When $C_c \geq 5$, we noticed that its influence on the quality of the recommended renames is limited. In other words, there is no other clear trend in the quality of the recommended refactoring that can be observed for different values of $C_c$. Thus, we excluded the 28 refactoring recommendations having $C_c < 5$ and studied the role played by $C_p$ in the remaining 118 recommendations (17 *yes*, 12 *maybe*, and 89 *no*).

Fig. 6.1 reports the recall and precision levels of our approach when excluding the recommendations having $C_p < t$, with $t$ varying between 1.0 and 0.1 at steps of 0.1. Note that in the computation of the recall and precision we considered the 29 recommendations accepted with a *yes* (17) or assessed as meaningful with a *maybe* (12) as correct (*i.e.,* the *maybe* answers are equated to the *yes* answers, and considered correct). This choice was dictated by the fact that we see the meaningful recommendations tagged with *maybe* as valuable for the developer, since she can then decide whether the alternative identifier name provided by our approach is valid or not. For a given value of $t$, the recall is computed as the number of correct recommendations having $C_p \geq t$ divided by 29 (the number of correct recommendations). This is an "approximation" of the *real recall* since we do not know the actual number of correct renamings that are needed in SMOS. In other words, if a correct rename refactoring was not recommended by LEAR, it was not evaluated by the participant and thus is not considered in the computation of the recall.

The precision is computed as the number of correct recommendations having $C_p \geq t$ divided by the number of recommendations having $C_p \geq t$. For example, when considering recommendations having $C_p = 1.0$, we only have three recommended renames, two of which have been accepted by the developer. This results in a recall of 0.07 (2/29) and a precision of 0.67 (2/3)—see Fig. 6.1.

Looking at Fig. 6.1, we can see that both recall and precision increase moving from $C_p = 1.0$ to $C_p = 0.8$, reaching recall=0.42 (12/29) and precision=0.92

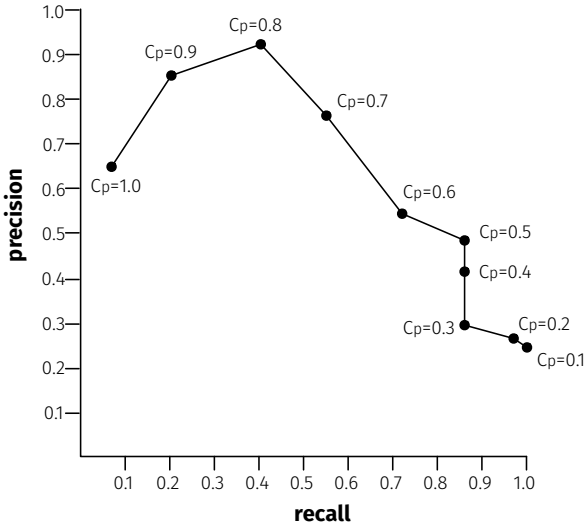



Figure 6.1: Precision and recall of the LEAR recommendations when varying $C_p$.

(12/13). This means that only one among the top-13 recommendations ranked by $C_p$ has been considered as not meaningful by the developer. Moving towards lower values of $C_p$, the recall increases thanks to the additional recommendations considered, while the precision decreases, indicating that the quality of the generated recommendations tend to decrease with lower $C_p$ values (*i.e.,* there are higher chances of receiving a meaningless recommendation for low values of $C_p$). It is quite clear in Fig. 6.1 that the likelihood of receiving good rename recommendations when $C_p < 0.5$ is very low. On the other hand, using a $C_p$ too high (*e.g.,* 0.8) results in a low recall, *i.e.,* many good rename suggestions would be excluded.

Based on the results of the performed tuning, we modified our tool in order to generate refactoring recommendations only when $C_c \geq 5$ *and* $C_p \geq 0.5$. In the evaluation reported in Section 6.3 we will further study the meaningfulness of the generated recommendations of rename refactorings for different values of $C_p$ in the significant range, *i.e.,* varying between 0.5 and 1.0.

| System | Type | LOCs | Participants | Experience (avg.) | Occupation |
|---|---|---|---|---|---|
| Therio | Web App | 13k | 1/2 | 7+ years | Stud. (PhD) |
| LifeMipp | Web App | 7k | 2/2 | 7+ years | Pro./Stud. (PhD) |
| MyUnimolAndroid | Android App | 27k | 1/4 | 5+ years | Pro. |
| MyUnimolServices | Web Services | 8k | 2/7 | 3+ years | Stud. (BS) |
| OCELOT | Desktop App | 22k | 1/2 | 7+ years | Stud. (PhD) |

Table 6.2: Context of the study (systems and participants).

## 6.3 Evaluation

This section presents the design and the results of the empirical study we carried out to compare the three previously introduced approaches for rename refactoring.

### 6.3.1 Study Design

The *goal* of the study is to assess the meaningfulness of the rename refactorings recommended by CA-RENAMING, NATURALIZE, and LEAR.

The *perspective* of the study is of researchers who want to investigate the applicability of approaches based on static code analysis (*i.e.,* CA-RENAMING) and on the $n$-gram language model (*i.e.,* NATURALIZE and LEAR) to recommend rename refactorings. The *context* is represented by *objects*, *i.e.,* five software projects on which we invoked the three experimented tools to generate recommendations for rename refactorings, and *subjects*, *i.e.,* seven developers of the object systems assessing the meaningfulness of the recommended rename refactorings.

To limit the number of refactoring recommendations to be evaluated by the developers, we applied the following "filtering policy" to the experimented techniques:

- LEAR: Given the results of the tuning of the $C_p$ and the $C_c$ indicators, we only consider the recommendations having $C_c \geq 5$ and $C_p \geq 0.50$.

- NATURALIZE: We used the original implementation made available by the authors with the recommended $n = 5$ in the $n$-gram language model. To limit the number of recommendations, and to apply a similar filter with respect to the one used in LEAR, we excluded all recommendations having

a probability lower than 0.5. Moreover, since NATURALIZE is also able to recommend renamings for identifiers used for method names (as opposed to the other two competitive approaches), we removed these recommendations, in order to have a fair comparison.

- CA-RENAMING: No filtering of the recommendations was applied (*i.e.,* all of them were considered). This is due to the fact that, as it will be shown, CA-RENAMING generates a much lower number of recommendations as compared to the other two techniques.

Despite these filters, our study involves a total of 922 manual evaluations of recommendations for rename refactoring.

**Research Questions and Context**

Our study is steered by the following research question:

- **RQ**$_1$ *Are the rename refactoring recommendations generated by approaches exploiting static analysis and NLP meaningful from a developer's point of view?*

The *object* systems taken into account are five Java systems developed and actively maintained in the University of Molise in the context of research projects or as part of its IT infrastructure. As *subjects*, we involved seven of the developers currently maintaining these systems. Table 6.2 shows size attributes (number of classes and LOCs) of the five systems, the number of developers actively working on them (column "Developers"), the number of developers we were able to involve in our study (column "Participants"), the average experience of the involved participants, and their occupation[2].

As it can be seen we involved a mix of professional developers and Computer Science students at different levels (Bachelor, Master, and PhD). All the participants have at least three years of experience in Java and they are directly involved in the development and maintenance of the object systems.

*Therio* is a Web application developed and maintained by Master and PhD students. It is currently used for research purposes to collect data from re-

[2]Here "Professional" indicates a developer working in industry.

| Approach | Confidence | # recomm. | | # yes | | # maybe | | # no | | $Prec_{yes \cup maybe}$ | $Prec_{yes}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | overall | mean | overall | mean | overall | mean | overall | mean | | |
| CA-RENAMING | N/A | 80 | 11.43 | 21 | 3.00 | 30 | 4.29 | 29 | 4.14 | 63.75% | 26.25% |
| NATURALIZE | >=0.5 | 459 | 65.57 | 76 | 10.86 | 99 | 14.14 | 284 | 40.57 | 38.13% | 16.56% |
| NATURALIZE | >=0.6 | 319 | 45.57 | 59 | 8.43 | 67 | 9.57 | 193 | 27.57 | 39.50% | 18.50% |
| NATURALIZE | >=0.7 | 185 | 26.43 | 35 | 5.00 | 43 | 6.14 | 107 | 15.29 | 42.16% | 18.92% |
| NATURALIZE | >=0.8 | 88 | 12.57 | 20 | 2.86 | 47 | 6.71 | 47 | 6.71 | 76.14% | 22.73% |
| LEAR | >=0.5 | 380 | 54.29 | 111 | 15.86 | 140 | 20.00 | 129 | 18.43 | 66.05% | 29.21% |
| LEAR | >=0.6 | 296 | 42.29 | 99 | 14.14 | 112 | 16.00 | 85 | 12.14 | 71.28% | 33.45% |
| LEAR | >=0.7 | 186 | 26.57 | 67 | 9.57 | 69 | 9.86 | 50 | 7.14 | 73.12% | 36.02% |
| LEAR | >=0.8 | 130 | 18.57 | 55 | 7.86 | 50 | 7.14 | 25 | 3.57 | 80.77% | 42.31% |

Table 6.3: Participants' answers to the question *Would you apply the proposed rename refactoring?*

searchers from all around the world. *LifeMipp* is a Web application developed and maintained by a professional developer and a PhD student. *LifeMipp* has been developed in the context of an European project and it is currently used by a wide user base. *MyUnimolAndroid* is an Android application developed and maintained by students and professional developers. Such an app is available on the Google PlayStore, it has been downloaded more than 1,000 times, and it is mostly used by students and faculties. *MyUnimolServices* is an open-source software developed and maintained by students and professional developers. Such a system is the back-end of the MyUnimolAndroid app. Finally, *OCELOT* is a Java desktop application developed and maintained by PhD students. At the moment, it is used by researchers in an academic context.

### Data Collection and Analysis

We run the three experimented approaches (*i.e.,* CA-RENAMING, NATURALIZE, and LEAR) on each of the five systems to recommend rename refactoring operations. Given $R$ the set of refactoring recommended by a given technique on the $P$ system, we asked the $P$'s developers involved in our study to assess the meaningfulness of each of the recommended refactorings. We adopted the same question/answers template previously presented for the tuning of the LEAR's $C_c$ and $C_p$ indicators. In particular, we asked the developers the question: *Would you apply the proposed refactoring?* with possible answers on a three-point Likert scale: 1 (yes), 2 (maybe), and 3 (no). Again, we clarified the meaning of the three possible answers as detailed in Section 6.2.

Overall, participants assessed the meaningfulness of 725 rename refactorings, 66 recommended by CA-RENAMING, 357 by NATURALIZE, and 302 by LEAR across the five systems. Considering the number of participants involved (*e.g.,* two participants evaluated independently the recommendations generated for LifeMipp), this accounts for a total of 922 refactoring evaluations, making our study the largest empirical evaluation of rename refactoring tools performed with developers having first-hand experience on the object systems.

To answer our research question we report, for the three experimented techniques, the number of rename refactoring recommendations tagged with *yes* (*i.e.,* the recommended rename refactoring is meaningful and should be applied), *maybe* (*i.e.,* the recommended rename refactoring is meaningful, but should not be applied) and *no* (*i.e.,* the recommended rename refactoring is not meaningful). We also report the precision of each technique computed in two different variants. In particular, given $R$ the set of refactorings recommended by an experimented technique, we compute:

- $Prec_{yes}$, computed as the number of recommendations in $R$ tagged with a *yes* divided by the total number of recommendations in $R$. This version of the precision considers as meaningful only the recommendations that the developers would actually implement.

- $Prec_{yes \cup maybe}$, computed as the number of recommendations in $R$ tagged with a *yes* or with a *maybe* divided by the total number of recommendations in $R$. This version of the precision considers as meaningful also the recommendations indicated by the developers as a valid alternative to the original variable name but not calling for a refactoring operation.

We discuss the results aggregated by technique (*i.e.,* by looking at the overall performance across all systems and as assessed by all participants).

Finally, we analyze the complementarity of the three techniques by computing, for each pair of techniques $(T_i, T_j)$, the following overlap metrics:

$$correct_{T_i \cap T_j} = \frac{|correct_{T_i} \cap correct_{T_j}|}{|correct_{T_i} \cup correct_{T_j}|}$$

$$correct_{T_i \setminus T_j} = \frac{|correct_{T_i} \setminus correct_{T_j}|}{|correct_{T_i} \cup correct_{T_j}|}$$

$$correct_{T_j \setminus T_i} = \frac{|correct_{T_j} \setminus correct_{T_i}|}{|correct_{T_i} \cup correct_{T_j}|}$$

The formulas above use the following metrics:

- $correct_{T_i}$ represents the set of meaningful refactoring operations recommended by technique $T_i$;
- $correct_{T_i \cap T_j}$ measures the overlap between the set of meaningful refactorings recommended by both techniques;
- $correct_{T_i \setminus T_j}$ measures the meaningful refactoring operations recommended by $T_i$ only and missed by $T_j$.

The latter metric provides an indication on how a rename refactoring tool contributes to enrich the set of meaningful refactorings identified by another tool. Such an analysis is particularly interesting for techniques relying on totally different strategies (*e.g.,* static code analysis *vs* NLP) to identify different rename refactoring opportunities. We report the three overlap metrics when considering both the recommendations tagged with *yes* and *maybe* as correct.

### 6.3.2 Results

Table 6.3 reports the answers provided by the developers to the question *"Would you apply the proposed rename refactoring?"*. Results are presented by approach, starting with the technique based on static code analysis (*i.e.,* CA-RENAMING [140]) followed by four different variations of NATURALIZE and of LEAR using different thresholds for the confidence of the generated recommendations. Table 6.3 does also report the $Prec_{yes}$ and $Prec_{yes \cup maybe}$ computed as described in Section 6.3.1.

**General Trends.** Before discussing in detail the performance of the experimented techniques, it is worthwhile to comment on some general trend reported in Table 6.3. First of all, *the approaches based on NLP generate more recommendations than* CA-RENAMING. This holds as well when considering the highest

confidence threshold we experimented with (*i.e.,* 0.8). Indeed, in this case LEAR generates a total of 130 rename refactorings (on average 18.57 per system) and NATURALIZE 88 (12.57 on average), as compared to the 80 recommended by CA-RENAMING (11.43 on average).

Another consideration is that LEAR *recommends a higher number of refactorings that are accepted by the developers with respect to* NATURALIZE *and to* CA-RENAMING. Overall, 111 rename refactorings recommended by LEAR have been fully accepted with a *yes*, as compared to the 76 recommendations provided by NATURALIZE, and the 21 suggested by CA-RENAMING.

Also, the higher number of accepted refactorings does not result in a lower precision. Indeed, LEAR does also achieve a higher $Prec_{yes}$ with respect to CA-RENAMING (29.21% *vs* 26.25%) and to NATURALIZE (16.56%). The precision of NATURALIZE is negatively influenced by the extremely high number of recommendations it generates when considering all those having confidence $\geq 0.5$ (*i.e.,* 459 recommendations).

Finally, LEAR*'s and* NATURALIZE*'s precision is strongly influenced by the chosen confidence threshold.* The values on Table 6.3 show an evident impact of the confidence threshold on $Prec_{yes}$ and $Prec_{yes \cup maybe}$ for both the approaches. Indeed, going to the least to the most conservative configuration for the confidence level, $Prec_{yes \cup maybe}$ increases by ∼14% (from 66.05% to 80.77%) for LEAR and by ∼38% for NATURALIZE (from 38.13% to 76.14%), while $Prec_{yes}$ increases by ∼13% for LEAR (from 29.21% to 42.31%) and by ∼6% for NATURALIZE (from 16.56% to 22.73%).

These results indicate one important possibility offered by these two approaches based on a similar underlying model: Depending on the time budget developers want to invest in rename refactoring, they can decide whether to have a higher or a lower number of recommendations, being informed of the fact that the most restrictive threshold is likely to just generate very few false positives, but also to potentially miss some good recommendations.

**Per-project analysis.** Table 6.4 reports examples of recommendations generated by the three approaches and tagged with *yes*, *maybe*, and *no*.

| | System | Original name | Rename | *Conf.* | Tag |
|---|---|---|---|---|---|
| CA-RENAMING | LifeMipp | `i` | `insect` | N/A | *yes* |
| | Therio | `pk` | `idCollection` | N/A | *yes* |
| | MyUnimolAndroid | `data` | `result` | N/A | *maybe* |
| | OCELOT | `hash` | `md5final` | N/A | *maybe* |
| | OCELOT | `navigator` | `this` | N/A | *no* |
| | MyUnimolAndroid | `fullname` | `fullnameOk` | N/A | *no* |
| NATURALIZE | OCELOT | `callString` | `macro` | 0.92 | *yes* |
| | MyUnimolAndroid | `factory` | `inflater` | 0.75 | *yes* |
| | OCELOT | `declaration` | `currentDeclaration` | 0.79 | *maybe* |
| | MyUnimolServices | `moduleName` | `name` | 0.69 | *maybe* |
| | LifeMipp | `species` | `t` | 0.64 | *no* |
| | MyUnimolServices | `username` | `token` | 0.91 | *no* |
| LEAR | LifeMipp | `image` | `photo` | 1.00 | *yes* |
| | MyUnimolServices | `careerId` | `pCareerId` | 0.63 | *yes* |
| | OCELOT | `type` | `realType` | 0.91 | *maybe* |
| | LifeMipp | `file` | `fileFullName` | 0.67 | *maybe* |
| | Therio | `pUsername` | `pName` | 0.59 | *no* |
| | MyUnimolAndroid | `info` | `o` | 1.00 | *no* |

Table 6.4: Examples of rename refactorings generated by the experimented tools and tagged with *yes*, *maybe*, and *no*.

Moving to the assessment performed by participants on each project, we found that the accuracy of the recommendations generated by the three tools substantially varies across the subject systems.

For example, on the LifeMipp project, CA-RENAMING is able to achieve very high values of precision, substantially better than the ones achieved by the approaches based on NLP. The refactoring recommendations for the LifeMipp project have been independently evaluated by two developers. Both of them agreed on the meaningfulness of all eight recommendations generated by CA-RENAMING. Indeed, the first developer would accept all of them, while the second tagged five recommendations with *yes* and three with *maybe*. NATURALIZE and LEAR, instead, while able to recommend a higher number of *yes* and *maybe* recommendations as opposed to CA-RENAMING (on average 19 for NATURALIZE and 22 for LEAR *vs* the 8 for CA-RENAMING), present a high price to pay in terms of false positives to discard (0 false positives for CA-RENAMING as compared to

49 for NATURALIZE and 19 for LEAR). Such a cost is strongly mitigated when increasing the confidence threshold. Indeed, when only considering recommendations having confidence $\geq 0.8$, the number of false positives drops to 1 (first developer) or 0 (second developer) for LEAR and to 8 or 6 for NATURALIZE, while still keeping an advantage in terms of number of *yes* and *maybe* generated recommendations (13 and 14—depending on the developer—for LEAR, and 12, for both developers, for NATURALIZE). A trend similar to the one discussed for LifeMipp has also been observed for MyUnimolServices.

When run on MyUnimolAndroid, CA-RENAMING only recommends three rename refactorings, two tagged with a *maybe* and one discarded (*no*). NATURALIZE generates 65 recommendations, with nine *yes*, 14 *maybe*, and 42 *no*. Finally, LEAR generates 35 suggestions, with six *yes*, twelve *maybe*, and 17 *no*.

This is the only system in which we did not observe a clear trend between the quality of the refactoring recommended by LEAR and the value used for the $C_p$ threshold. Indeed, the precision of our approach is not increasing with the increase of the $C_p$ value. This is due to the fact that the developer involved in the evaluation of the refactoring for the MyUnimolAndroid rejected with a *no* seven recommendations having $C_p \geq 0.8$.

We asked further comments to the developer, *i.e.,* to check what went "wrong" for this specific system, and in particular we asked to comment on each of these seven cases. Some of the explanations seemed to indicate more a *maybe* recommendation rather than the assigned *no*. For example, our approach recommended with $C_p = 0.9$ and $C_c = 54$ the renaming *activity* $\rightarrow$ *navigationDrawer*. The developer explained that the *activity* identifier refers to an object of `FragmentActivity` that is casted as a `NavigationDrawer` and, for this reason, he prefers to keep the *activity* name rather than the recommended one. Another false positive indicated by the developer was renaming *info* $\rightarrow$ *o*, where *info* is a method parameter of type `Object`. LEAR learned from the MyUnimolAndroid's trigrams that the developers tend to name a parameter of type `Object` with *o*. This is especially true in the implementation of `equals` methods. Thus, while the renaming would have been consistent with what present in the system, the developer preferred to keep the original name as being "*more descriptive*", rejecting the recommendation. MyUnimolAndroid is also the only system in which

NATURALIZE achieves a higher precision with respect to LEAR when considering the most restrictive confidence (*i.e.,* $\geq 0.8$).

Finally, on the Therio and on the OCELOT projects, LEAR substantially outperforms the two competitive approaches. On Therio, CA-RENAMING achieves $Prec_{yes} = 0.33$ and $Prec_{yes \cup maybe} = 0.47$, as compared to the $Prec_{yes} = 0.37$ and $Prec_{yes \cup maybe} = 0.74$ achieved by LEAR when considering only recommendations having $C_p \geq 0.6$. LEAR also generates a much higher number of *yes* (35 *vs* 5) and *maybe* (13 *vs* 2) recommendations. Examples of recommendations generated by LEAR and accepted by the developers include $pk \rightarrow idTaxon$ and $o \rightarrow occurrences$, while an example of rejected recommendation is $pUsername \rightarrow pName$. NATURALIZE also achieves its best performance on Therio when considering all recommendations having confidence $\geq 0.6$ ($Prec_{yes} = 0.35$ and $Prec_{yes \cup maybe} = 0.74$), but with a lower number of *yes* (23) and *maybe* (8) recommendations with respect to LEAR. A similar trend is also observed on OCELOT, where LEAR is able to recommend 89 renamings with a $Prec_{yes \cup maybe} = 0.93$.

**Overlap Metrics Analysis.** Table 6.5 reports the three overlap metrics between the experimented techniques, as described in Section 6.3.1.

| $T_i$ | $T_j$ | $correct_{T_i \cap T_j}$ | $correct_{T_i \setminus T_j}$ | $correct_{T_j \setminus T_i}$ |
|---|---|---|---|---|
| CA-RENAMING | LEAR | 1.00% | 16.05% | 82.94% |
| CA-RENAMING | NATURALIZE | 0.00% | 22.57% | 77.43% |
| LEAR | NATURALIZE | 4.16% | 57.21% | 38.63% |

Table 6.5: Overlap metrics

The overlap in terms of meaningful recommendations provided by the different tools is extremely low; 1% between CA-RENAMING and LEAR, 0% between CA-RENAMING and NATURALIZE, and 4% between LEAR and NATURALIZE. While the low overlap between the techniques using static code analysis and NLP is somehow expected, the 4% overlap observed between LEAR and NATURALIZE is surprising considering the fact that LEAR is inspired by the core idea behind NATURALIZE. This means that the differences between the two techniques described in Section 6.2 (*e.g.,* only considering the lexical tokens in the language model as opposed to use all tokens) have a strong impact on the generated recommenda-

tions. While this was already clear by the different performance provided by the two approaches (see Table 6.3), it is even more evident from Table 6.5.

LEAR is able to recommend 82.94% of meaningful renamings that are not identified by CA-RENAMING, and 57.21% that are not recommended by NATURALIZE. However, there is also a high percentage of meaningful rename refactorings recommended by CA-RENAMING (16.05%) and by NATURALIZE (38.63%) but not identified by LEAR. This confirms the extremely high complementarity of the different techniques, paving the way to novel rename refactoring approaches based on their combination, that we plan to investigate as part of our future research agenda.

## 6.4 Threats to Validity

**Threats to *construct validity*** are mainly related to how we assessed the developers' perception of the refactoring meaningfulness. We asked developers to express on a three-point Likert scale the meaningfulness of each recommended refactoring making sure to carefully explain the meaning of each possible answer from a practical point of view.

**Threats to *internal validity*** are represented, first of all, by the calibration of the LEAR confidence $C_p$ and $C_c$ indicators. We performed the calibration of these indicators on one project (SMOS) not used in the LEAR's evaluation, by computing the recall *vs* precision curve for different possible values of the $C_p$ indicator. This was not really needed for the $C_c$ indicator, for which we just observed the unreliability of the recommendations having $C_c < 5$. Concerning the other approaches, for the NATURALIZE's $n$-gram model parameter we adopted the one used by its authors (*i.e.,* $n = 5$) and we relied on their implementation of the approach. To limit the number of refactoring recommendations, we excluded the ones having a probability lower than 0.5. This choice certainly does not penalize NATURALIZE, since we are only considering the best recommendations it generates. As for CA-RENAMING, we used our own implementation.

Another threat to internal validity is related to the participants involved in our study. We just targeted original developers of the systems used in our evaluation, without taking into account the possibility of involving people with no experience

on the object systems. Still, we cannot ensure that the involved developers had a good knowledge of the specific code components subject of the refactoring.

However, we are confident that the original developers had *at least* a very good knowledge of the application domain and of the projects' naming conventions. Thus, we are confident that the selected developers were able to properly judge the quality of the recommended refactoring.

**Threats to *external validity*** can be related to the size of the set of chosen objects and to the pool of the participants to the study. Concerning the chosen objects, we are aware that our study is based on rename refactorings recommended on five Java systems only and that the considered systems, while not trivial, are generally of small-medium size (between 7 and 27 KLOC). Also, we were only able to involve in our study seven developers. Still, as previously said, (i) we preferred to limit our study to developers having a first-hand experience with the object systems, rather than inviting also external developers to take part in our study, and (ii) despite the limited number of systems and developers, our results are still based on a total of 922 manual inspections performed to assess the quality of the recommended refactoring.

## 6.5 Final Remarks

We reported an empirical investigation assessing the meaningfulness of recommendations generated by three approaches—two existing in the literature (*i.e.,* CA-RENAMING [140] and NATURALIZE [5]) and one presented in this chapter (*i.e.,* LEAR)—that aim to promote a consistent use of identifiers in source code. The manual evaluation of 922 rename refactoring operations performed by seven original developers of five software systems highlighted that:

In summary, the lessons learned from this chapter are the following:

- overall, LEAR achieves a higher precision and it is able to recommend a higher number of meaningful refactoring operations with respect to the competitive techniques;

- while being the best performing approach, LEAR still generates a high number of false positives, especially when just considering as meaningful the

recommendations tagged with a *yes* by the developers (*i.e.,* the ones they would actually implement). Indeed, in this scenario, $\sim 58\%$ of the renamings generated by LEAR in its most precise configuration ($C_p \geq 0.8$) are discarded by developers. This means that there is large room for improvement in state-of-the-art tools for rename refactoring;

- the experimented approaches have unstable performance across the different systems. Indeed, even if LEAR is, overall, the approach providing the most accurate recommendations, it is not the clear winner on all the object systems. This indicates that there are peculiarities of the software systems that can influence the performance of the three techniques.

# CHAPTER 7

# From Code Readability to Code Understandability

## Contents

## 7.1 Introduction

In the previous chapters we presented code readability models and we showed how introducing textual features can improve their accuracy. Even if readability models have been shown to be quite accurate in distinguishing readable from unreadable code, the literature lacks of a similar model for automatically assessing code understandability. Indeed, previous attempts to define a code understandability model [24, 91, 135] have not been empirically evaluated, consider understandability as a factor in a quality model [3, 13], or measure understandability at the level of a whole software system [28]. As we previously mentioned, reading and understanding are two related but different concepts. Therefore, we argue that readability models are not necessarily sufficient to also assess code understandability.

To fill this gap, in this chapter we investigate 121 metrics to determine the extent to which they correlate with code understandability. These metrics can be categorized into three types: (i) code-related metrics (105 metrics), (ii) documentation-related metrics (11), and (iii) developer-related metrics (5). The code-related metrics are comprised of classic code metrics, like LOC and cyclomatic complexity, and readability metrics, like text coherence (see Chapter 4) and code indentation [21]. A developer may be able to read some code snippet, but it may use unknown code. Thus, we included existing documentation-related metrics, like the availability of external documentation, and introduced nine new documentation-related metrics. Finally, we included developer-related metrics to understand the extent to which the developer's experience and background might influence code comprehension.

To run our investigation, we performed a study with 63 participants using the 121 metrics to determine their correlation with the understandability of code snippets. Each participant was required to understand up to eight code snippets, leading to a total of 444 evaluations. We consider understandability from two

perspectives: *perceived* and *actual* understandability of the code snippets. The participants were presented a code snippet to understand and they were asked whether they could understand it (*perceived understandability*). If their answer was positive, they were asked to answer three verification questions (*actual understandability*). We also monitored the time spent understanding each snippet to estimate the effort.

By performing an extensive statistical analysis, we obtained a **negative empirical result**: none of the considered metrics exhibit a significant correlation with either the *perceived* or the *actual* understandability. This result was quite surprising to us, especially considering the involvement in our study of complexity and readability metrics generally thought to influence code understandability.

Trockman *et al.* [143] re-analyzed part of the dataset presented in this chapter (*i.e.,* the one we released as part of our ASE'17 work [124] that this chapter extends). They showed that it is possible to define classification models with some discriminatory power by combining the 121 metrics considered in our study. Specifically, they use LASSO regression [143] to classify evaluations as *understandable* or *not understandable*. Inspired by such a work, we also took into account combinations of metrics and, to do this, we exploited *classification* and *regression* models. While these models represent a step ahead as compared to the single metrics taken in isolation, their prediction accuracy is still too low to make them useful in practice. Finally, we interviewed five developers to better understand their perspective when it comes to understanding a code snippet and to infer factors that could be exploited in future metrics to automatically assess code understandability.

This chapter is organized as follows: Section 7.2 describes the 121 metrics used in our empirical study, while Section 7.3 presents the proxies we used to assess the developers' understandability of a given code snippet. The design and results of the study are presented in Sections 7.4 and 7.4.4, respectively. Section 7.5 discusses the threats that could affect our findings, while Section 7.6 concludes the chapter.

## 7.2 Candidate Predictors for Code Understandability

Understandability is a multifaceted property of source code and, as well as readability, is subjective in nature. In readability, the subjectivity is represented by personal taste and habits, while in understandability it lies in the previous knowledge of a developer and in her mental models [136]. Consider a method of an Android activity in a mobile app; its understandability might be high for an Android developer, while it could be low for a Java developer with no experience in Android. In this section, we briefly discuss the 121 metrics considered in our study aimed at assessing their ability to capture the understandability of a given piece of code. Table 7.1 shows the complete list of metrics: rows contain the basic metrics and columns indicate how the metrics are aggregated (*e.g.,* the *Identifiers length* for a given code snippet is computed, as suggested by previous work [21], as the average and as the maximum length of the identifiers used in the snippet). We report with "[C4]" the features introduced in this thesis (Chapter 4). We report in boldface the new metrics introduced in this chapter. It is worth noting that the number of metrics shown in Table 7.1 does not add up to the 121 metrics considered in our study. This is due to the fact that some forms of aggregation, *e.g.,* "Visual" and "Area", include multiple ways of aggregating the same measure. For example, each metric aggregated as "Visual" should be counted twice. Indeed, for these metrics, a "virtual" color to each character in the code is assigned, based on the type of the token it belongs to (*e.g.,* characters of identifiers have color 1, while characters of keywords have color 2), thus creating a matrix of colors for the snippet. Then, the variation of colors is computed both on the $X$ and on the $Y$ axis of such a matrix [43], thus resulting in two different forms of aggregation. In the following subsections, we discuss the considered metrics grouped by their type.

### 7.2.1 Code-Related Metrics

Most of the metrics considered in our study assess source code properties. We include the five metrics used by Kasto and Whalley [74]: *cyclomatic complex-*

*ity* [97], which estimates the number of linear independent paths of the snippet *average number of nested blocks*, which measures the average code-block nesting in the snippet, *number of parameters*, *number of statements* and *number of operands*, *i.e.,* number of identifiers. It is worth noting that we **do not** include some other understandability metrics from the literature discussed in Chapter 2 [24, 104, 141, 135, 28, 13, 12] since they are defined at system-level (*i.e.,* they provide an overall indication of the system understandability), while we are interested in studying whether it is possible to measure the understandability of a given code snippet, as already done in the literature for code readability.

We also include in this category all the code-related readability metrics defined in the literature [21, 119, 43] and in Chapter 4. These include the ones by Buse and Weimer [21], assessing properties for a single line of code (*e.g., number of identifiers* or *line length*) and then aggregated (with the maximum and/or the average) to work at the level of "code snippet".

*Lines of code* (LOC), *token entropy* and *Halstead's volume* are used by Posnett *et al.* [119] in the context of readability prediction. Dorn [43] presents a variation to the basic metrics introduced by Buse and Weimer [21], measuring the bandwidth of the Discrete Fourier Transform (DFT) of the metrics, the *absolute* and the *relative* area of characters belonging to different token categories (*e.g.,* identifiers, keywords or comments), the alignment of characters through different lines, and the number of identifiers containing words belonging to an English dictionary. Note that the area-related metrics defined by Dorn are computed both in an absolute way (*e.g.,* total area of comments) and in a relative way (*e.g.,* area of comments divided by area of strings).

In Chapter 4, we introduced *Narrow Meaning Identifiers* (NMI), *Number of Meanings* (NM), *Identifiers Terms In Dictionary* (ITID) and *Textual Coherence* (TC) to capture the readability of a code snippet. Such metrics are computed line-by-line (ITID), identifier-by-identifier (NMI and NM) or block-by-block (TC); the authors aggregate the measures using minimum, average and maximum, in order to have a single measure for the snippet. We also use code readability, using the comprehensive model introduced in Chapter 4, as a separate metric, combining together the previously listed metrics. Specifically, we defined the readability

model by training a logistic classifier on the 420 Java snippets available in the literature [21, 43] and provided by the new dataset presented in Chapter 4.

We also introduce a new code-related metric, the *Invoked Methods Signature Quality* (*IMSQ*), which measures the quality of the signature of the internal methods invoked by a given code snippet $s$ (*i.e.,* methods belonging to the same system of $s$) in terms of readability and representativeness. We define the Method Signature Quality (*MSQ*) of an invoked method $m$ as:

$$MSQ(m) = \frac{1}{|IS(m)|} \sum_{id \in IS(m)} IQ(id)$$

where *IS(m)* is the set of identifiers used in the $m$'s signature (*i.e.,* method name and parameters) and *IQ(id)* is defined as:

$$IQ(id) = \begin{cases} \frac{1}{2}(Rd(id) + Rp(id)), & id \text{ is a method name} \\ Rd(id), & id \text{ is a parameter name} \end{cases}$$

*IQ(id)* captures the quality of an identifier in terms of its readability (*Rd*) and its representativeness (*Rp*). The idea behind the readability is that an identifier should be composed of a (possibly small) set of meaningful words. To measure *Rd* for an identifier (*id*), we (i) split *id* into the words composing it, (ii) expand each word to bring it in its original form (*e.g.,* `ptr` → `pointer`), (iii) create a new identifier $id_{exp}$ composed by the expanded words separated by a "_", and (iv) measure the Levenshtein distance between *id* and $id_{exp}$. The Levenshtein distance between two strings $a$ and $b$ measures the minimum number of single-character changes needed to transform $a$ into $b$. The conjecture behind *IQ(id)* is that the higher the Levenshtein distance between *id* and $id_{exp}$, the higher the mental effort required for the developer to understand the meaning of the identifier by mentally splitting and expanding it during program comprehension. Note also that we separate the expanded terms in $id_{exp}$ by using "_" in order to penalize, by increasing the Levenshtein distance, identifiers composed by several words. For example, the identifier `printOnStdOut` is first split into `print`, `on`, `std`, `out`; then, each word is expanded, which has no effect on the first two words,

but expands `std` into `standard` and `out` into `output`. Therefore, `printOnStdOut` is transformed in `print_on_standard_output`.

To have *Rd(id)* defined in [0, 1], we normalize the Levenshtein distance ($L$) between $id$ and $id_{exp}$ as follows:

$$Rd(id) = 1 - \frac{L(id, id_{exp})}{\max(|id|, |id_{exp}|)}$$

where $\max(|id|, |id_{exp}|)$ represents the longest identifier among the two. When the distance equals zero, the readability of the identifier equals one, indicating no need for expansion/splitting (*i.e.,* $id$ is composed by a single expanded word).

Note that in the implementation of *Rd(id)*, we used a semi-automatic approach to split/expand identifiers. We first used a naive automatic splitting technique, based on camel case and underscores; then, we automatically checked the presence of each resulting word in an English dictionary. If the word was not found, we manually expanded/further split the specific word. For example, for the word "cmdline" there would not be automatic split. Since the word "cmdline" does not exist in the dictionary, we manually convert it to "command" and "line". We save all the manual substitutions in order to minimize the human effort. In the literature, there are many automatic approaches for identifier splitting/expansion [84, 63, 82, 36], but we preferred to implement a simpler and more effective strategy at this stage, since the number of identifiers to split/expand was limited and our goal was to assess the correlation of the defined metrics with the understandability effort. Thus, we wanted to be sure to avoid introducing imprecision while computing the metrics.

When dealing with the identifier used to name a method, we also verify whether it is representative of what the method does ($Rp$). We compute the textual overlap between the terms used in the identifier and in the method body. We tokenize the method body to define its dictionary. Then, we count the number of times each word from the identifier (expanded or not) is contained in the dictionary extracted from the method body. We consider only names and verbs from the identifiers, ignoring other parts of speech such as conjunctions, since they do not carry semantic information. Following the `printOnStdOut` example, we check whether the method body contains the words `print`, `standard`, `std`,

`output`, and `out`. We measure the representativeness as the ratio between the number of words from the identifier (*i.e.,* method name) contained in the method body, and the total number of words in the identifier. If all the words from the identifier are used in $m$'s body, we assume that the method name is representative of $m$ and thus, should ease the understanding of methods invoking $m$. If, instead, words are not found in the method body, this could hinder the understandability of the methods invoking $m$.

In our study, we consider the minimum, the average, and the maximum values of the *MSQ* metric for a given code snippet (*e.g.,* the average *MSQ* of all methods invoked in the code snippet).

### 7.2.2 Documentation-Related Metrics

In Chapter 4 we introduced three metrics to capture the quality of the internal documentation of a snippet: *Comments Readability* (CR) measures the readability of the comments in a snippet using the Flesch reading-ease test [50]; *Comments and Identifiers Consistency* (CIC) measures the consistency between comments and code; and $\text{CIC}_{syn}$, a variant of CIC which takes into account the synonyms of the words in the identifiers.

We also introduce two new metrics aimed at capturing the quality of both the *internal* (*MIDQ*) and *external* (*AEDQ*) documentation available for code components used in a given snippet. The Methods Internal Documentation Quality (*MIDQ*) for a snippet $s$ acts as a proxy for the internal documentation (*i.e.,* Javadoc) available for the internal methods (the ones belonging to the same project as $s$) invoked in $s$. Given $m$ an internal invoked method, we compute *MIDQ(m)* using a variation of the approach proposed by Schreck *et al.* [126]:

$$MIDQ(m) = \frac{1}{2}(DIR(m) + readability_D(m))$$

where *DIR(m)* is the Documented Items Ratio computed as the number of documented items in $m$ divided by the number of documentable items in $m$. We consider as *documentable items* for $m$ (i) its parameters, (ii) the exceptions it throws, and (iii) its return value. Such items are considered as *documented* if there is an explicit reference to them in the Javadoc through the tags `@param`,

`@throws` and `@returns`. $readability_D(m)$ represents, instead, the readability of the Javadoc comments assessed using the Flesch reading-ease test [50]. The higher *MIDQ* the higher the internal documentation quality for $m$. We consider the minimum, the average, and the maximum values of the *MSQ* metric for a given code snippet.

Concerning the API External Documentation Quality (*AEDQ*), it tries to capture the amount of information about APIs used in the given snippet $s$ that can be acquired from external sources of documentation, such as Q&A websites. The conjecture is that if external documentation is available, it is more likely that developers are able to understand the usage of an API in a code snippet $s$. We compute the availability of external documentation for each external class $c$ used in $s$ via the *AEDQ(c)* metric. First, we identify all Stack Overflow discussions related to $c$ by running the following query:

```
title:"how to" <c> hasaccepted:yes [java]
```

In other words, we select all Stack Overflow discussions that (i) contain "how to" and the class name in the title, (ii) have an accepted answer, and (iii) concern Java (since our study has been performed on Java snippets). Then, we sum the votes assigned by the Stack Overflow users to the question in each retrieved discussion, in order to have a quantitative information about the interest of the developers' community in such a class. We assume that higher interest in a given API class implies a higher availability of external sources of information (*e.g.,* discussions, code examples, *etc.*). We consider in our study the minimum, the average, and the maximum values of the *AEDQ* metric for the external classes used in $s$.

### 7.2.3 Developer-Related Metrics

Since understandability is a very subjective feature of code, we introduced three developer-related metrics. We measure the programming experience of the developer who is required to understand a snippet ($PE_{gen}$ and $PE_{spec}$) and the popularity of the API used in the snippet (*EAP*).

The common wisdom is that the higher the programming experience of developers, the higher their capability of understanding code. $PE_{gen}$ measures the

programming experience (in years) of a developer in general (*i.e.,* in any programming language). $PE_{spec}$ assesses instead the programming experience (in years) of a developer in the programming language in which a given snippet $s$ is implemented. The higher $PE_{spec}$, the higher the developer's knowledge about the libraries available for such a programming language.

With External API Popularity (*EAP*), we aim at capturing the popularity of the external APIs used in a given snippet. The assumption is that the lower the popularity, the lower the probability that a typical developer knows the API. If the developer is not aware of the APIs used in a snippet, it is likely that she has to look for its documentation or to inspect its source code, thus spending more effort in code understanding.

We rely on an external base of Java classes $E$ to estimate the popularity of an external class. We chose as $E$ a 10% random sample of classes from Java/Android projects hosted on GitHub in 2016, totaling ∼2M classes from ∼57K Java projects. We used Google BigQuery to extract all the imports of all the classes belonging to such projects using a regular expression. Then, we counted the number of times each class imported in $E$ occurred in the `import` statements. Note that in Java it is possible to import entire packages (*e.g.,* `import java.util.`$*$). In this case, it is difficult to identify the actual classes imported from the package. For this reason, we applied the following strategy. Let us assume that a class, `Foo`, is imported only once with the statement `import bar.Foo`, but it is part of a quite popular package, `bar`, that is imported 100 times in $E$ through the statement `import bar.`$*$. The class `Foo2`, belonging to the same package, is imported 99 times with the statement `import bar.Foo2`. In this case, we increase the number of occurrences of classes belonging to imported package in a proportional way. In the presented example, we add 1 to the number of `Foo`'s imports, and 99 to the number of `Foo2` imports. We found that imports of entire packages represent only 2.6% of all the imports and, therefore, their impact is very low. *EAP(c)* is defined as the number of $c$ imports normalized over the number of imports of $c_{max}$, where $c_{max}$ is the most imported class we found in $E$ (*i.e.,* `java.util.List`).

| | Metric | Non-aggregated | Min | Avg | Max | DFT | Visual | Area |
|---|---|---|---|---|---|---|---|---|
| Code | Cyclomatic comp. | [74] | | | | | | |
| | #nested blocks | | | [74] | | | | |
| | #parameters | [74] | | | | | | |
| | #statements | [74] | | | | | | |
| | #assignments | | | [21] | | [43] | | |
| | #blank lines | | | [21] | | | | |
| | #characters | | | | [21] | | | |
| | #commas | | | [21] | | [43] | | |
| | #comments | | | [21] | | [43] | [43] | [43] |
| | #comparisons | | | [21] | | [43] | | |
| | #conditionals | | | [21] | | [43] | | |
| | #identifiers | [74] | | [21] | [21] | [43] | [43] | [43] |
| | #keywords | | | [21] | [21] | [43] | [43] | [43] |
| | #literals | | | | | | [43] | [43] |
| | #loops | | | [21] | | [43] | | |
| | #numbers | | | [21] | [21] | [43] | [43] | [43] |
| | #operators | | | [21] | | [43] | [43] | [43] |
| | #parenthesis | | | [21] | | [43] | | |
| | #periods | | | [21] | | [43] | | |
| | #spaces | | | [21] | | [43] | | |
| | #strings | | | | | | [43] | [43] |
| | #words | | | | [21] | | | |
| | Indentation length | | | [21] | [21] | [43] | | |
| | Identifiers length | | | [21] | [21] | | | |
| | Line length | | | [21] | [21] | [43] | | |
| | #aligned blocks | [43] | | | | | | |
| | Ext. of alig. blocks | [43] | | | | | | |
| | Entropy | [119] | | | | | | |
| | LOC | [119] | | | | | | |
| | Volume | [119] | | | | | | |
| | NMI | | [C4] | [C4] | [C4] | | | |
| | NM | | | [C4] | [C4] | | | |
| | ITID | | [C4] | [C4][43] | | | | |
| | TC | | [C4] | [C4] | [C4] | | | |
| | Readability | [C4] | | | | | | |
| | **IMSQ** | | [C7] | [C7] | [C7] | | | |
| Docs | CR | [C4] | | | | | | |
| | CIC | | | [C4] | [C4] | | | |
| | $CIC_{syn}$ | | | [C4] | [C4] | | | |
| | **MIDQ** | | [C7] | [C7] | [C7] | | | |
| | **AEDQ** | | [C7] | [C7] | [C7] | | | |
| Devs | **EAP** | | [C7] | [C7] | [C7] | | | |
| | $\mathbf{PE}_{gen}$ | [C7] | | | | | | |
| | $\mathbf{PE}_{spec}$ | [C7] | | | | | | |

Table 7.1: Candidate predictors for code understandability.

## 7.3 Proxies for Code Understandability

Code understandability can affect two aspects of code understanding: the *correctness* (*i.e.,* how well the developer is able to understand a snippet), and the *time* needed to understand the snippet. Moreover, developers might *perceive* that they understand a given code without *actually* understanding it. Since understandability is composed by several facets, we introduce six proxies of code understandability. These proxies can be used to (i) study the correlation between the candidate predictor variables introduced in Section 7.2, and (ii) as dependent variables in techniques aimed at predicting code understandability:

1. **Perceived Binary Understandability (*PBU*)**. This is a binary categorical variable that is *true* if a developer perceives that she understood a given code, and *false* otherwise.

2. **Time Needed for Perceived Understandability (*TNPU*)**. This is a continuous variable in $\Re^+$, measuring the time spent by the developer to comprehend a given code before having a clear idea on whether she understood it or not.

3. **Actual Understandability (*AU*)**. This is a continuous variable in $[0, 1]$, measuring the actual understanding of the inspected code. A possible way of measuring actual understandability is through verification questions. For example, the developer understanding the code could be required to answer three questions, and the percentage of correct answers is used to assess $AU$.

4. **Actual Binary Understandability ($ABU_{k\%}$)**. This is a binary categorical variable derived from $AU$. It is *true* if $AU$ is greater than $k$, *false* otherwise. $ABU_{k\%}$ is basically a proxy to classify snippets of code as understandable or not based on the level of actual understanding developers are able to achieve while inspecting it.

5. **Timed Actual Understandability (*TAU*)**. This is a continuous variable in $[0, 1]$, derived from $AU$ and $TNPU$. It gets a value of 0 if the developer perceives that she did not understand the snippet. Otherwise, it

is computed as:

$$TAU = AU\left(1 - \frac{TNPU}{\max TNPU}\right)$$

where $AU$ and $TNPU$ are the variables previously defined. The higher $AU$, the higher $TAU$, while the higher $TNPU$, the lower $TAU$. We take into account the relative time ($\frac{TNPU}{\max TNPU}$) instead of the absolute time, so that $TAU$ gives the same importance to both the correctness achieved ($AU$) and the time needed ($TNPU$). $\max TNPU$ is, indeed, the maximum $TNPU$ measured on the snippet.

6. **Binary Deceptiveness ($BD_{k\%}$).** This is a binary categorical variable derived from $PBU$ and $ABU_{k\%}$, which is *true* if $PBU$ is *true* and $ABU_{k\%}$ is *false*, and *false* otherwise. $BD_{k\%}$ indicates whether a developer can be deceived by a method in terms of its understandability (*i.e.,* she incorrectly thinks she understood the method).

## 7.4 Evaluation

The *goal* of our study is to assess the extent to which the considered 121 metrics are related to code understandability and what developers consider as understandable/not understandable. The *perspective* is of researchers interested in (i) analyzing whether *code-related*, *documentation-related*, and *developer-related* metrics can be used to assess the understandability level of a given piece of code, and (ii) investigating characteristics of code considered as important for developers during program comprehension.

### 7.4.1 Empirical Study Design

This study aims at answering the following research questions:

$RQ_1$ *What is the correlation between the 121 considered metrics and the understandability level of a given developer for a specific code snippet?* Given the wide and heterogeneous set of considered metrics, answering this research

| System | Java KLOC | Category | Description |
|---|---|---|---|
| *ANTLR* | 178 | Desktop | Lexer-parser |
| *Car-report* | 45 | Mobile | Car costs monitoring |
| *Hibernate* | 948 | Framework | ORM framework |
| *Jenkins* | 231 | Web | Continuous integration |
| *K9 mail* | 121 | Mobile | Mail client |
| *MyExpenses* | 101 | Mobile | Budget monitoring |
| *OpenCMS* | 1059 | Web | Content Management System |
| *Phoenix* | 352 | Framework | Relational database engine |
| *Spring* | 197 | Framework | Generic application framework |
| *Weka* | 657 | Desktop | Machine-learning toolkit |

Table 7.2: Systems used in our study.

question would allow us and, in general, the research community to understand how far we are from defining a set of metrics capable of automatically and objectively assessing code understandability;

$RQ_2$ *Is it possible to define understandability models able to predict code understandability?* Given a snippet of code, we want to determine whether combining metrics in a model can effectively capture the level of understandability of that code;

$RQ_3$ *How do developers determine the understandability of code?* While the first two questions relate to metrics to assess understandability, it is also important to consider the perspective of developers when trying to understand code. To this end, we aim to deepen our analysis by asking experienced developers what makes a certain code snippet *understandable* or *not understandable*.

### 7.4.2 Data Collection

The *context* of the study consists of 50 Java/Android methods extracted from ten popular systems listed in Table 7.2 (five methods from each system). We first extracted all the methods having $50 \pm 20$ ELOCs (*i.e.,* Effective Lines Of Code, excluding blank and comment lines) from the systems. The choice of the

methods' size (*i.e.,* $50 \pm 20$ ELOCs) was driven by the decision of assessing the understandability of mid-size code snippets.

Afterwards, we computed all the metrics described in Section 7.2 for the selected methods[1]. Then, we used a greedy algorithm for center selection [77] to select the 50 most representative methods based on the defined metrics. Given a set of candidate methods $M$ and a set of already selected centers $C$, the algorithm chooses, in each iteration, $\arg\max_{m \in M} dist(C, m)$, *i.e.,* the candidate method which is the farthest possible (in terms of considered metrics) from the already selected centers of which the first center is randomly selected. In order to select exactly five snippets from each system, we used the set of candidate methods from a specific system as $M$ until the five methods for such a system were selected; then, we changed $M$ with the set of candidate methods from another system, and so on, until $|C| = 50$. Note that (i) we did not empty the $C$ set when changing the candidate methods (*i.e.,* when moving from one system to another) to always keep track of the methods selected up to that moment, thus avoiding the risk of adding to $C$ methods similar to the ones already in $C$; (ii) we did not run the algorithm on the union of all candidate methods to ensure the selection of five methods per system (thus increasing the heterogeneity of the final sample).

After selecting the 50 methods and computing the values of the metrics for each of these 50 methods, we needed to define a ground-truth, which reports the understandability of each method. To this aim, we invited 63 Java developers and CS students to participate in a survey, where they were required to understand the selected methods. The survey was implemented in a Web application and featured the following steps. First, we collected demographic data about participants: (i) years of experience in programming and more specifically in Java, and (ii) current position (*e.g.,* CS student, developer *etc.*). This information was used in part to compute the *developer-related metrics*. We asked participants for consent to anonymously use the gathered data. We do not report developers' names and e-mail addresses for privacy reasons. After this preliminary step, each participant was required to understand a subset of eight methods randomly selected from the 50 methods. The Web application was designed to automatically balance the number of evaluations for each of the 50 methods (*i.e.,* the number of participants

[1]Excluding the "Developer programming experience" and the "Developer Java experience"

understanding each method was roughly the same). In total, we collected 444 evaluations across the 50 methods (∼8.9 evaluations per method on average), since not all participants completed the survey.

The eight methods were presented individually (*i.e.,* each method on a different page) to participants, and the Web application allowed navigation of the method and access to the methods/classes invoked/used by it. Also, participants were allowed to browse the Web to collect information about types, APIs, data structures, *etc.* used in the method. This was done to simulate the typical understanding process performed by developers. We asked participants to carefully read and fully understand each method. Participants could, at any moment, click on the button "I understood the method" or the button "I cannot understand the method". In both cases, the Web application stored the time spent, in seconds, by the developer for the method's understanding before clicking on one of the two buttons. If the participant clicked on "I understood the method", the method was hidden and she was required to answer three *verification questions* about the method she just inspected. The provided answers were stored for future analysis.

In particular, one of the questions we asked was about the identifiers used in the method (*e.g.,* what does "CMS" mean?); a second question was about the purpose of a call to an internal method (*e.g.,* What does the invoked method `X` do?); and the third question was about the purpose of using an external component in the snippet (*e.g.,* JDBC APIs).

### 7.4.3 Analysis Method

In the context of our study, we measure the understandability level using the previously defined proxies of code understandability (Section 7.3).

We measure **Perceived Binary Understandability (*PBU*)** using the initial declaration of the participants: if they clicked on "I cannot understand the method" button, *PBU* is *false*, while it is *true* otherwise (*i.e.,* the participant clicked on "I understood the method").

We measure **Time Needed for Perceived Understandability (*TNPU*)** as time, in seconds, spent by the participant while inspecting the method before clicking on "I understood the method". This metric cannot be computed when the participant clicked on "I cannot understand the method".

We measure **Actual Understandability ($AU$)** as the percentage of correct answers given by the participants to the three verification questions. If the participant clicked on the "I cannot understand the method" button, $AU$ is 0.

We measure **Actual Binary Understandability ($ABU_{k\%}$)**, with $k = 50$. Therefore, $ABU_{50\%}$ is *true* when participants correctly answer at least two of the three verification questions, and *false* otherwise.

To measure **Timed Actual Understandability ($TAU$)**, we use the previously defined formula, which combines $AU$ and $TNPU$. $TAU$ gets value 0 if the participant clicked on the "I cannot understand the method" button. In this context, we used a modified version of $TNPU$ in which outliers (detected using the Tukey's test [144], with $k = 3$) are replaced with the maximum value of $TNPU$ which is not an outlier. We did this because the maximum value of $TNPU$ in our dataset is 1,649 seconds, much greater than the third quartile (164 seconds). Using the real maximum value would have flattened down all the relative times.

Finally, we measure **Binary Deceptiveness ($BD_{k\%}$)**, with $k = 50$, using the previously defined formula which combines $PBU$ and $ABU_{50\%}$.

We computed these six variables for each of the 444 evaluations performed by participants (*i.e.,* for each method that each participant tried to understand). We excluded 2 of the 121 considered metrics (*i.e.,* $\text{NMI}_{min}$ and $\text{ITID}_{min}$), because the value of such metrics was 0 for all the snippets.

To answer $RQ_1$, we first verified which metrics strongly correlate among the 121. This was done to exclude redundant metrics, which capture the same information in different ways, from our analysis. We compute the Kendall rank correlation coefficient (*i.e.,* Kendall's $\tau$) [75] to determine whether there are pairs exhibiting a strong correlation. We adopted the Kendall's $\tau$, since it does not assume the data to be normally distributed nor the existence of a straight linear relationship between the analyzed pairs of metrics. Cohen [32] provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 \leq |\tau| < 0.1$, small correlation when $0.1 \leq |\tau| < 0.3$, medium correlation when $0.3 \leq |\tau| < 0.7$, and strong correlation when $0.7 \leq |\tau| \leq 1$. For each pair of metrics exhibiting a strong correlation (*i.e.,* with a Kendall's $|\tau| \geq 0.7$), we excluded the ones which presented the highest

number of missing values[2] or one at random, when the number of missing values were equal. This allowed us to reduce the number of investigated metrics from 121 to 73. Finally, we computed the Kendall correlation between each of the remaining 73 metrics and $PBU$, $TNPU$, $AU$, $BD_{50\%}$, and $TAU$ to verify whether some of them are able to capture the (actual and perceived) understandability of code. We did not compute the correlation with $ABU_{50\%}$, since, in this case, it would have been redundant, because we already compute the correlation with $AU$.

To answer $RQ_2$, we tried to combine the metrics defined in Section 7.2 to predict the six proxies of understandability previously defined. Since the number of metrics is high as compared to the number of instances, we performed a preliminary phase of feature selection. First, we removed the features highly correlated among each others (as previously done for $RQ_1$); then, we removed Area Keywords/Comments, because of the high number of missing values (124, more than 30% of the instances), which could be problematic for some of the used machine learning techniques.

To build a model for predicting $PBU$, $ABU_{50\%}$, and $BD_{50\%}$, we use a broad selection of classifiers defined in the literature, since such variables are nominal. Specifically, we use (i) Logistic Regression [87], (ii) Bayes Networks [73], (iii) Support Vector Machines (SMO algorithm [115]), (iv) Neural Networks (Multilayer Perceptron), (v) k-Nearest-Neighbors [4], and (vi) Random Forest [20]. As the first step, we use 10% of each dataset to tune the hyper-parameters and we removed such instances from the dataset used in the experiment. We focus on the main parameters of each technique and, specifically, we tune: (i) *number of hidden layers*, *learning rate*, and *momentum* for Multilayer Perceptron; (ii) $k$, *weighting method*, and *distance metric* for kNN; (iii) *kernel*, *exponent*, and *complexity* for SMO; (iv) *number of features* for Random Forest. To do this, we use an exhaustive search approach on a reduced search space, *i.e.,* we defined discrete values for each parameters (at most 10 values) and then we try all of the possible combinations. We search for the combination of parameters that achieves the best AUC using leave-one-out cross-validation on the tuning set. Note we

[2]Some metrics cannot be computed in some cases. For example, "Area of comments/literals" cannot be computed if the method does not contain literals.

did not tune hyper-parameters for Logistic Regression and Bayes Networks, since they do not rely on any particular parameter.

Both $PBU$ and $BD_{50\%}$ were unbalanced: $PBU$ presented a high number of positive instances (∼69%), while $BD_{50\%}$ a high number of negative instances (∼80%). To have balanced models, we use the SMOTE filter [30] on the training sets to generate artificial instances for the minority classes. Moreover, while Random Forest is designed to automatically select the best features, for the other algorithms it is important to have an adequate number of features in relation to the number of instances. To achieve this goal, for those techniques, we performed a second round of feature selection using linear floating forward selection with a wrapper strategy. We used Logistic Regression as the classifier for the wrapper and AUC (Area Under the Curve) as the metric for the evaluation of the effectiveness of each subset of features, computed with a 5-fold cross validation. We report the average F-Measure of the classes of each variable and the AUC of all the classifiers for the three variables to show the effectiveness of combinations of metrics defined in the literature in the prediction of the nominal proxies of understandability. Additionally, we discuss the cases in which precision and recall are very different (*e.g.,* precision is much higher than recall).

For the other proxies, *i.e.,* $TNPU$, $AU$ and $TAU$, we use several regression techniques defined in the literature, since such variables are numeric. Specifically, we use (i) Linear Regression, (ii) Support Vector Machines (SMOreg [131]), (iii) Neural Networks (Multilayer Perceptron), (iv) k-Nearest-Neighbors[4] and (v) Random Forest[20]. In this case, we report the correlation of the predicted values with the actual values and the MAE (Mean Absolute Error) of the prediction, computed as $\frac{\sum_i^n (|x_i - x_i^*|)}{n}$. We use the same approach that we used for classification to tune the parameters of the regressors and to select the best features. In this case, we look for the parameters and the features that minimize the mean absolute error. We use linear regression as regressor in the wrapper strategy.

Both classifiers and regression models need to be trained and tested on different datasets to avoid overfitting (*i.e.,* the model would fit the specific data, but not generalize). For this reason, we performed leave-one-out cross-validation where we divided the dataset in 444 folds, *i.e.,* each fold contains exactly one

instance. For each iteration, one of the folds was used as *test set* and the union of the other folds as *training set* for our models. Therefore, for each evaluated instance, we train our models on the whole dataset without the test instance. This solution is ideal in this context for two reasons: (i) since our dataset may be small for regression techniques to be effective, we cannot afford to reduce the number of instances for the training phase; (ii) this type of validation allows us to use all the evaluations of the same developer (except for the one that has to be tested) in the training phase. This would allow machine learning techniques to define rules specific for the developer itself. It is worth highlighting that we do not aim at comparing different machine learning techniques. Instead, our goal is to understand if any technique is able to effectively combine the considered metrics to capture code understandability.

Finally, to answer $RQ_3$, we conducted semi-structured interviews with five experienced developers, listed in Table 7.4. The developers provided consent to report their names in this thesis. We do not directly associate the names to the performance in the tasks. To guide the interviews, we selected 4 methods among the 50 that we used to answer our first two research questions. Such methods where (i) the one with the highest mean $TAU$, *i.e.,* the most understandable one, (ii) the one with the lowest mean $TAU$, *i.e.,* the least understandable one, (iii) the one with the highest standard deviation in $TAU$, *i.e.,* the one for which the understandability seems to be most subjective, and (iv) the one that has the highest $TNPU$ and a number of $BD_{50\%} = true$ greater than zero, *i.e.,* the method that, despite being analyzed for the longest time, still makes some developers incorrectly believe that they understood it.

We use $TAU$ as a proxy for understandability to select the first three snippets as it takes into account both the actual understandability and the time taken to understand the snippet. Table 7.3 shows the four methods we selected. For each method, we asked the five developers to read and understand it, thinking aloud, if they wanted. Before the participants started to read and understand the snippet, we asked them how familiar they were with the system to which the snippet belongs and with the APIs used in the snippet. For each snippet, after the participants concluded the understanding phase, we asked precise questions: $Q_1$: "*do you think this snippet is understandable?*"; $Q_2$: "*what makes this method under-*

| System | Class | Method | Type |
|---|---|---|---|
| OpenCMS | CmsHoverbarContextMenuButton | createContextMenu | The most understandable ($\overline{TAU} = 0.74$) |
| Phoenix | MetaDataEndpointImpl | doDropSchema | The least understandable ($\overline{TAU} = 0.06$) |
| Hibernate | TimesTenDialect | TimesTenDialect | The most subjective ($sd(TAU) = 0.46$) |
| MyExpenses | LazyFontSelector | processChar | The most deceptive ($\overline{TNPU} = 391.0$, $\#BD_{50\%} = 2$) |

Table 7.3: Methods used during the interviews with developers.

| Name | Position | Experience |
|---|---|---|
| Salvatore Geremia | PhD Student @ Unimol | 8 years |
| Giovanni Grano | PhD Student @ Uzh | 8 years |
| Stefano Dalla Palma | Android developer @ Datasound | 5 years |
| Carlo Branca | Front-end developer @ Gatelab | 8 years |
| Matteo Merola | Back-end developer @ Bunq | 8 years |

Table 7.4: Position and experience of the interviewed developers.

*standable/not understandable to you?*"; $Q_3$: "*is it possible to modify the method to make it more understandable? If yes, how?*". If the participants understood the snippet, we asked the purpose of the snippet to ensure they actually understood it, unless they explained it while they thought aloud during the interview. Finally, we registered the time the participants took to understand the snippets. We report the answers of the participants to each question and, above all, we report interesting insights that we could catch during the interviews.

### 7.4.4 Empirical Study Results

In this section, we present the results of our empirical study. Fig. 7.1 provides information about the participants involved in $RQ_1$ (same dataset is used for $RQ_2$ as well). The majority of them ($\sim 60\%$) are CS bachelor's students—mixed in terms of years of programming experience. The sample of participants also includes nine master's students, three Ph.D. students, and thirteen professional developers.

### 7.4.5 $RQ_1$: Correlation With Single Metrics

Fig. 7.2 reports a heatmap representing the correlation between metrics and our six proxies for code understandability. It is clear that **very few metrics**

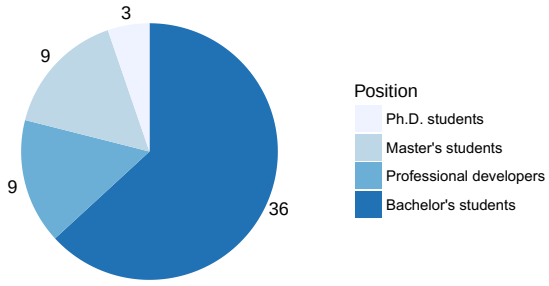



Figure 7.1: Participants to the study.

**have a correlation with understandability higher than |0.1|**. Specifically, 8 metrics have a weak correlation with $PBU$, only one with $TNPU$, 13 with $AU$, 13 with $TAU$, and 2 with $BD_{50\%}$. Note that, since we only observed weak correlations, these metrics are very unlikely to be appropriate proxies for code understandability. 51 out of the 73 metrics considered showed no correlation at all with any of the proxies.

The metric which has the highest correlation with $PBU$ are two: (i) *maximum line length* ($\tau \approx -0.13$), which is one of the metrics introduced by Buse and Weimer [21] for readability prediction; (ii) $PE_{spec}$ ($\tau \approx 0.13$), which measures the Java experience of the developer. Note that Buse and Weimer also found that *Maximum line length* is the most important one for readability prediction in their model [21]. Therefore, this reinforces the fact that, generally, developers tend to perceive code with long lines as less pleasant. The correlation with $PE_{spec}$, instead, shows that developers with more experience in the specific programming language tend to have a slightly higher confidence and they tend to perceive snippets of code as understandable more frequently than developers with less experience. Finally, we observed other low correlations with $PBU$: $NM_{avg}$ (-0.12), *i.e.,* when words used for identifiers have many meanings, they make developers perceive the snippet as slightly less understandable; $MIDQ_{min}$ (0.12), *i.e.,* the higher the minimum quality of the internal documentation, the higher


Figure 7.2: Correlation between metrics and proxies for understandability.

the perceived understandability; *average identifiers' length* (-0.12), *i.e.,* shorter identifiers slightly help perceiving the code as more understandable.

It is worth noting that only a single metric has a slight correlation with $TNPU$, *i.e., DFT of conditionals* (0.11).

The metric that has the highest correlation with $AU$ is average *Textual Coherence* ($\tau \approx -0.16$). The fact that such a correlation is negative is surprising, because we expected a higher Textual Coherence to imply a higher understandability. Also, we found that *number of parameters* negatively correlates with $AU$ ($\tau \approx -0.13$) (*i.e.,* the larger the number of parameters, the lower the actual understandability of the method). We found a similar result also in $RQ_3$ when interviewing developers. Other examples of metrics correlated with $AU$ are $PE_{spec}$ ($\tau \approx 0.13$) and *DFT of conditionals* ($\tau \approx -0.13$).

The metric with the highest correlation with $TAU$ is *DFT of conditionals* ($\tau \approx -0.16$). This suggests that high complexity reduces the understandability of a snippet. It should be noted that for $TAU$, which is expression of *actual* understandability, we observe only a slight correlation with the programming experience ($\tau \approx 0.11$ for $PE_{spec}$).

Finally, only two metrics, *i.e., number of literals (visual X)* and *DFT of conditionals*, are slightly correlated with $BD_{50\%}$ ($\tau \approx 0.11$ and 0.1, respectively). All the other metrics show a negligible correlation.

**Summary of $RQ_1$.** None of the metrics we considered achieve a medium/strong correlation with any of the proxies of code understandability we defined.

### 7.4.6 $RQ_2$: Performance of Multi-Metrics Models

We report the performance of models that combine metrics dividing them as *classification*, for $PBU$, $ABU_{50\%}$ and $BD_{50\%}$, and *regression*, for $TNPU$, $AU$ and $TAU$.

| Classifier | $PBU$ | | $ABU_{50\%}$ | | $BD_{50\%}$ | |
|---|---|---|---|---|---|---|
| | *F-Measure* | *AUC* | *F-Measure* | *AUC* | *F-Measure* | *AUC* |
| *Logistic* | 0.68 | 0.71 | 0.63 | 0.71 | 0.72 | 0.71 |
| *kNN* | 0.62 | 0.63 | 0.59 | 0.66 | 0.74 | 0.66 |
| *SMO* | 0.64 | 0.63 | 0.63 | 0.63 | 0.71 | 0.63 |
| *Naive Bayes* | 0.60 | 0.65 | 0.66 | 0.68 | 0.63 | 0.66 |
| *Random Forest* | 0.65 | 0.63 | **0.67** | **0.72** | **0.77** | 0.64 |
| *ML Perceptron* | **0.66** | **0.69** | 0.63 | 0.70 | 0.70 | **0.70** |

Table 7.5: Classification results of $PBU$, $ABU_{50\%}$, and $BD_{50\%}$.

### Classification

Table 7.5 shows the F-Measure and AUC of the classification of $PBU$, $ABU_{50\%}$, and $BD_{50\%}$. Since we use Logistic Regression for feature selection, we do not include it directly in the comparison among the techniques. It is worth noting that the AUC achieved by Random Forest and Multilayer Perceptron for the classification of $ABU_{50\%}$ seems to suggest that it is possible to classify with a good level of confidence snippets as *actually understandable* or *actually not understandable*. Looking at the F-Measure, however, it is clear that we are quite far from having a practical classifier for actual understandability. Also, looking at the classification accuracy, 33% of the instances are wrongly classified by the best model.

Such results are even more negative for $PBU$ and, above all, $BD_{50\%}$. For $PBU$ the maximum F-Measure is $\sim$0.66. On the other hand, the best F-measure for $BD_{50\%}$ is 0.77, achieved by Random Forest. However, this positive result hides the fact that such a classifier has good results only on *negative* instances. Both precision and recall for the positive class are, indeed, very low (mean 0.31 and 0.51 for precision and recall, respectively). In general, looking at the F-Measure of the minority classes, which are *perceived as not understandable* and *deceptive*, such values are much lower (*i.e.,* 0.48 and 0.37, respectively), despite the fact that we used SMOTE to balance the training sets. We can conclude that the combination of the considered metrics shows a slight discriminatory power for actual binary understandability ($ABU_{50\%}$); however, we are quite far from a practically useful prediction model of actual/perceived understandability and deceptiveness.

| Regressor | *TNPU* | | *AU* | | *TAU* | |
|---|---|---|---|---|---|---|
| | *Correlation* | *MAE* | *Correlation* | *MAE* | *Correlation* | *MAE* |
| *Linear Regression* | 0.14 | 132.9 | 0.35 | 0.30 | 0.36 | 0.27 |
| *kNN* | 0.11 | 137.1 | 0.26 | 0.31 | 0.21 | 0.28 |
| *SMOreg* | 0.17 | **114.4** | 0.36 | 0.29 | 0.29 | **0.27** |
| *Random Forest* | 0.09 | 147.0 | 0.34 | **0.29** | 0.29 | 0.28 |
| *ML Perceptron* | **0.18** | 124.8 | **0.37** | 0.30 | **0.36** | **0.27** |

Table 7.6: Regression results of $TNPU$, $AU$, and $TAU$.

**Regression**

In Table 7.6, we report the performance of the regression models for $TNPU$, $AU$ and $TAU$. The first thing that is very clear is that our models are not able to predict $TNPU$ (Time Needed for Perceived Understandability). The highest correlation for $TNPU$ is only 0.18, higher than the correlation achieved by single metrics, but still very low. The Mean Absolute Error is also very high. On average, we can expect a prediction of $TNPU$ to be wrong by about 2 minutes. Considering that the average $TNPU$ is 143.4 seconds, without excluding outliers, it is clear that any prediction of $TNPU$ made with the state of the art metrics is practically useless.

On the other hand, it can be seen that there is a good improvement in the correlation for both $AU$ and $TAU$, when using combinations of metrics rather than single metrics. The maximum correlations are 0.37 and 0.36, respectively, leading to a *medium* correlation. However, it is worth noticing that the Mean Absolute Error is quite high. In our context, the MAE of 0.29 for $AU$ means that we should expect our model to be wrong by one answer, on average ($0.29 \approx 0.33$). If our model predicts that $AU$ is 0.66 (the participant gives two correct answers out of three), it may be that she gives one or three correct answers, thus making this prediction poorly actionable except for corner cases (predicted $AU$ very low or very high).

**Summary of $RQ_2$.** Combining the state-of-the-art metrics can result in models that show some discriminatory power in the prediction of some proxies of code understandability (*i.e.,* $ABU_{50\%}$, $AU$ and $TAU$). However, such predictions are not sufficiently good yet to be used in practice.

### 7.4.7 $RQ_3$: Structured Interviews with Developers

We report the results of the interviews by presenting the developer responses grouped by each type of code snippet. The interviews lasted roughly between one and two hours each.

#### The Most Understandable Method

All the developers correctly described the functionality implemented in the method, and all of them answered positively to $Q_1$ (*i.e.,* they think the method is understandable). The average time needed to understand the method was about 2.5 minutes. As expected, this was the method understood most quickly.

The single aspect that makes this method highly understandable, according to the developers, is the easy/clear task it implements. Other positive aspects of the method as for understandability ($Q_2$) are that it is very tidy (good readability, in general) and it implements a single task. Carlo highlighed that the names are very well chosen. However, both Salvatore and Giovanni put emphasis on the many repetitions in the code. Salvatore said that, to some extent, repetition and alignment are positive, because they help the brain ignoring part of the code that is not useful to read. On the other hand, Giovanni thinks that repetitions make the method "poorly maintainable and less elegant"; however, he does not think that repetitions hinder understandability.

Three developers agreed that there is no negative aspect in terms of understandability, and it would not be possible to have a more understandable version of the method ($Q_3$). On the other hand, Salvatore thinks that the repetition of the actual parameter "hoverbar" for all the constructors and the lack of alignment forces the reader to check if it is actually true that all the constructors are called with such an actual parameter, and this slightly increases the time needed

```
//Actual code
result.add(new CmsGotoMenuEntry(hoverbar));
result.add(new CmsGotoExplorerMenuEntry(hoverbar));
//Desired code
result.add(new CmsGotoMenuEntry        (hoverbar));
result.add(new CmsGotoExplorerMenuEntry(hoverbar));
```

Figure 7.3: Actual code vs aligned code (first snippet).

to understand the method. He would have aligned all the actual parameters of the constructors in the list (Fig. 7.3). Matteo thinks that the name "hoverbar" is not very clear, and documentation is lacking for it. He also thinks that the abstract return type makes the method slightly less understandable, because it is necessary to understand what kinds of concrete types can be returned. He said that, from his experience, there is often a trade-off between *understandability* and *maintainability*. In this case, using abstract types is necessary to make the method more maintainable (*e.g.,* all the classes that implement the same interface have the same signature, and it is easier to extend this system). However, this makes the implemented mechanism (and the system) harder to understand.

While reading and understanding the method, the developers used different approaches. Some of the developers looked at class and internal dependencies of the method (*i.e.,* the context in which such method exists); some looked as well at some of the classes used in the method and they inferred that they are very similar, which was helped by the names; some focused on the method itself. One of them searched on the internet for "hoverbar" to understand its meaning in this context. One of the developers also looked for information about the entire system and the repository (*e.g.,* number of developers) and he looked at the documentation of the implemented interface to get information about the method to understand.

#### The Least Understandable Method

Four developers out of five did not understand the second method. They admitted that fully understanding the method would have required much longer, and they asked to stop after about 5 minutes. One of them, on the other hand, took some extra time to analyze the classes used in the method (about 8 minutes,

in total). In the end, he said he understood the method and he was able to describe what the method did. Three developers said that the method is not understandable ($Q_1$), while the other two, surprisingly, said that the method is understandable. The developer, who did not *fully* understand the method but *perceived* it as understandable, explained that he roughly understood the method, but the lack of knowledge about the system made him say that it would take more time to properly understand it.

Two developers highlighted that the positive aspects of the method ($Q_2$) are that it implements just one task. One of them also generally liked the names used for the identifiers. On the other hand, all the developers listed many negative aspects. The most negative aspect highlighted is the complete lack of comments and internal documentation (all the developers). Also, the fact that the method belongs to a big and untidy class makes the method itself less understandable (2 out of 5 developers). The developers also generally complained about the low readability of the method (4 out of 5 developers). Matteo and Carlo pointed out that another negative aspect is that the method is highly dependent on some internal classes/methods that are hard to understand. Stefano, Salvatore and Matteo did not like the presence of many exit points for the method; Salvatore and Matteo also said that the high number of parameters strongly reduces its understandability (confirming what we found answering $RQ_1$), and Matteo precised that some of the parameters are not used at all. Stefano and Matteo did not like some identifiers names (*e.g.,* `areTablesIdentifiers`), which they considered potentially misleading. Matteo thinks that this method has too many responsibilities. For example, the first thing the method does is to load a schema, based on the value of one of the parameters, but he thinks that it would be better to directly have a parameter with the loaded schema. Also, he thinks that the application domain is quite complex.

All the developers answered $Q_3$ saying that they would improve the understandability of the method by adding comments and improving its readability. Matteo said that he would (i) change the order of the parameters (*e.g.,* putting the schema as the first parameter), (ii) use exceptions instead of `SCHEMA_NOT_FOUND` instances, and (iii) start with a check for exceptional behaviors (that lead to `SCHEMA_NOT_FOUND`) and the normal behavior after that.

Also, in this case, the developers used different approaches to read and understand the snippet. One developer tried to look for the internal documentation of the used classes (which is lacking); two developers looked at the code of the used classes; one developer strongly relied on the names of the identifiers. Finally, one developer said that he would have used the debugger to understand the snippet (*i.e.,* he would have analyzed the dynamic behavior of the code). He explained that understanding this snippet only looking at code statically would take longer.

#### The Most Controversial Method

Four out of five developers were able to understand the method with a low effort (∼2 minutes). One of the developers took longer (5 minutes). He explained that he had no experience with SQL dialects. However, in the end, all of them were able to understand the method. Three of the developers were familiar with SQL dialects and Hibernate, and they took shorter than the average to understand the snippet. All the developers agreed that the method is understandable ($Q_1$). However, three out of five developers explicitly said that a good knowledge of SQL dialects and, partially, of Hibernate is necessary to understand this method; conversely, one developer said that the lack of knowledge on SQL dialects would have just increased the time needed to understand, but the method would have been easy to understand anyway.

All the developers appreciated the good names and the good quality of comments and Javadoc ($Q_2$) and four out of five developers said that there are no negative aspects of this method in terms of understandability and that they would not change it ($Q_3$). Matteo, on the other hand, said that, in general, he would have divided the method in three private methods: one for the columns, one for the SQL properties, and one for the functions. However, he said that it is not strictly necessary in this specific case, because the dialect is easy.

The interviews suggest that this method was the most controversial in our study due to differences in background among the developers. Such subjectivity may have strongly influenced the time needed to understand the snippet. While the subjectivity of this method emerged from the survey we conducted, on the other hand the five developers we interviewed seemed to agree that the method is

understandable. This is most likely due to the fact that they are all professional and experienced developers.

### The Most Deceiving Method

Three out of five developers said that they understood the method and that they found the method understandable ($Q_1$). Two developers said that it would take longer to fully understand it. On average, the developers took about 5 minutes to understand the method. Interestingly, when we asked them to precisely describe what the method does, one of the developers noticed that he actually did not understand the reason why a for loop was in the method. Therefore, in the end, only two developers actually understood the method.

Differently from the other methods for which developers often highlighted more or less the same positive/negative aspects in terms of its understandability ($Q_2$), for this method, each of them talked about a different positive aspect. Salvatore said that, despite the nesting, the conditions are quite simple and the blocks of the conditions are short. According to him, this facilitates the understanding of causes and effects in the code. Conversely, Matteo said that he did not like the many nested if controls, which make it hard to understand, according to him. Giovanni liked the names used, and he found them quite clear. Carlo, instead, did not like them, but he said that the fact that all the internal methods called in this snippet were in the same class improved the understandability of the method. Stefano did not find any positive aspects about this method. On the other hand, the developers said that some names, such as "surrogate", are uncommon and abstract, and they may hinder the understandability of the method (2 out of 5 developers); Matteo points out that the name of the method itself (`processChar`) is too generic and ambiguous. The redundancy in some parts of the method is definitely a negative aspect. The method contains two very similar blocks of code (4 out of 5 developers). They all suggest to remove this repetition ($Q_3$).

To understand this snippet, three developers looked only at the code of the method, while two of them found it useful to look at the only method in the class that calls the snippet's method.

Table 7.7: Factors mentioned by the participants.

| Factor | MM | SG | GG | SD | CB | Study |
|---|---|---|---|---|---|---|
| Quality of Identifiers | × | × | × | × | × | ✓ |
| Code Readability | × | × | × | × | | ✓ |
| Presence of Comments | × | × | × | | × | ✓ |
| # Exit Points | × | × | | × | | |
| Documentation Quality | × | | × | × | | ✓ |
| # Responsibilities | × | | | × | × | |
| Quality of the Class | × | | | × | × | |
| # Parameters | × | × | | | | ✓ |
| Nesting | × | × | | | | ✓ |
| Quality of Dependencies | × | | | | × | |
| Application Domain | × | | × | | | |
| Code repetitions | | × | × | | | |
| Broken Lines | | × | | | × | |
| Control Flow | | | × | × | | |

**Factors Mentioned by the Developers**

Table 7.7 provides a summary of the factors that the developers mentioned at least once during the interview and it shows the ones that we used as potential predictors to answer $RQ_1$ and $RQ_2$. The quality of the identifiers is mentioned by all of them, but also code readability and comments seem to be valuable, according to most of them. We took into account all the factors that at least four out of five interviewed developers agreed on: *quality of identifiers* is captured by some of the textual features we introduced to improve the readability assessment, such as *ITID* and *CIC*; we estimated *code readability* using the comprehensive model introduced in Chapter 4; *documentation quality* is captured by two new metrics we introduced in this chapter, *i.e., MIDQ* and *AEDQ*; finally, we measure *number of parameters* of a method and the average *number of nested blocks*, both mentioned by two developers out of five.

**Summary of $RQ_3$.** There is a diverse set of aspects considered by developers during code understanding. The interviewed developers (mostly) agree on a subset of aspects that are desirable (*e.g.,* good identifiers, readability, and presence of comments) or undesirable (*e.g.,* too many exit points) for having understandable code. However, we found no agreement on many other aspects and, above all, on the adopted understanding process.

### 7.4.8 Discussion

Our results show that no single metric has a non-weak correlation with any proxy of understandability.

We also tried to combine these metrics in *classification* and *regression* models to predict different aspects of code understandability. In a previous study, Trockman *et al.* [143] used LASSO regression to classify $ABU_{50\%}$, and they achieved an AUC of 0.64. We achieved a higher AUC for the classification of $ABU_{50\%}$ (0.72) and a comparable AUC for $PBU$ (0.69) and $BD_{50\%}$ (0.70). However, looking at the F-Measure, it is clear that the prediction model would not be useful in practice. Compared to the readability models, understandability models are much less effective and practically unusable. Combining metrics in regression models to predict $TNPU$, $AU$ and $TAU$ also shows the limits of the metrics, which can achieve a maximum correlation of 0.37 (with $AU$). Therefore, we have a clear negative result: **the metrics we investigated are not enough to capture code understandability**. However, as previously hinted by Trockman *et al.* [143], combining metrics helps to achieve better results as compared to single metrics for most of the understandability proxies.

In the interviews that we conducted, we found that developers perceive readability as an aspect that highly influences code understandability. However, this contradicts our quantitative results. It is possible that we did not capture any correlation between readability and understandability because we measured the understandability effort only by using the time spent to understand a snippet as a proxy. A factor that we ignore here is the mental effort actually needed to understand a snippet. It could be that unreadable code makes developers more tired in the long run, but when focusing on a single snippet this does not result

in noticeable differences in terms of comprehension time and it does not affect the correctness of the understandability process. Experienced developers, who had the chance of working both with readable and unreadable code for longer periods, consider this as an important aspect because they feel it affects their overall performance. The same may be true for other aspects associated to code readability (such as the quality of identifiers, as previously seen in Chapter 4). Most importantly, our interviews with developers show that each developer uses her own understanding process. When code is harder to understand, some look at the related classes, while others say they would run the code and use the debugger to understand it. One of the most surprising facts is that some developers found the least understandable snippet to be *understandable*. Finally, we found that the personal background of developers plays a crucial role in understanding: when the knowledge of a concept is lacking, it takes time to acquire such a knowledge. This may be the main limit of the metrics we introduced: We measure the "knowledge" contained in a snippet (*e.g.,* with MIDQ and AEDQ), but we do not measure the knowledge that the developer already has. Our metrics may be valid for the "average" developer, but when we try to measure understandability at a personal level, they are not enough.

To further confirm this, we tried to check how the values of our proxies for code understandability vary among different evaluators (for the same snippet) and different snippets (for the same evaluator). For each proxy $P$, we computed the mean variance of $P$ among different evaluators (for the same snippet) using the formula $V_s(P) = \frac{\sum_{i=1}^{n} var(P_i)}{n}$, where $n$ is the number of snippets in our dataset and $P_i$ is the vector containing the values of $P$ for the snippet $i$. The higher $V_s$, the larger the differences in terms of understandability among different evaluators for the same snippets. Similarly, we computed the variations among different snippets for the same evaluator as $V_d(P) = \frac{\sum_{i=1}^{n} var(P_i)}{n}$, where $n$ is the number of developers and $P_i$ is the vector containing the evaluations of the developer $i$. Again, the higher $V_d$, the higher the differences in understandability among different snippets for the same evaluator.

We report in Table 7.8 both $V_s$ and $V_d$ for $TNPU$, $AU$, and $TAU$, *i.e.,* the numeric proxies we previously defined. We also report the values of $V_s$ for the different categories of participants. The table shows that $V_s$ is slightly higher than

| | *AU* | *TNPU* | *TAU* |
|---|---|---|---|
| $\mathbf{V_P^s}$ **(BsC)** | 0.11 | 61,683 | 0.08 |
| $\mathbf{V_P^s}$ **(MsC)** | **0.06** | **4,209** | **0.03** |
| $\mathbf{V_P^s}$ **(PhD)** | 0.09 | 24,924 | 0.10 |
| $\mathbf{V_P^s}$ **(Professional)** | 0.07 | 42,425 | 0.07 |
| $\mathbf{V_P^s}$ | 0.13 | 34,294 | 0.10 |
| $\mathbf{V_P^d}$ | **0.12** | **26,109** | **0.09** |

Table 7.8: Mean variance of the proxies among snippets and developers (lower values imply more similar scores).

$V_d$ for all the proxies. This shows that the understandability depends more on the developer than on the snippet he/she is evaluating, even if such a difference is not very high. Also, it is worth noting that $V_s(AU)$ decreases if we divide the developers in categories based on their professional position. This means that different categories of developers achieve more similar levels of correctness. Specifically, professional developers and Master's students seem to be the most cohesive groups in terms of correctness (*i.e.,* the groups that exhibit the lowest variance). The same happens (but with lower differences) for $V_s(TAU)$. On the other hand, for $V_s(TNPU)$ there are categories with lower inter-group variations (*i.e.,* Master's students and PhD students), while others have higher variations (*i.e.,* Bachelor's students and professional developers).

Finally, because of this result, we tried to use the professional *position* of the developer as an additional feature in our combined models ($RQ_2$). We observed a slight improvement in the regression performance of $TAU$ (Correlation: +0.07; MAE: -0.02) and $AU$ (Correlation: +0.02; MAE: +0.00), while we achieved lower classification performance for $BD_{50\%}$ (F-measure: -0.06; AUC: -0.02), and comparable regression and classification performance for $PBU$ (F-measure: -0.02; AUC: +0.01), $ABU_{(\%}$ F-measure: +0.07; AUC: -0.02), and $TNPU$ (Correlation: +0.03; MAE: +7.5). However, the improvement relates only to the maximum scores achieved: not all the machine learning techniques achieve better results.

Therefore, we can conclude that the effort in the prediction of code understandability should be directed in capturing subjective aspects of developers – their background knowledge and their experience – not only in *quantitative* terms (*i.e.,* years of experience) but also in *qualitative* terms: the interviews suggest that when developers are not familiar with the topics in the code that they need to

understand, they need to spend some time to search for information about them. Introducing new developer-related metrics considering their experience with specific topics (*e.g.,* JDBC APIs) or design patterns (*e.g.,* bridge design-pattern) could be useful to capture such aspects.

## 7.5 Threats to Validity

**Threats to construct validity**, concerning the relation between theory and observation, are mainly due to the measurements we performed, both in terms of the 121 metrics that we studied as well as when defining the six dependent variables for the understandability level. Concerning the 121 metrics, we tested our implementation and, when needed (*e.g.,* for the $IMSQ$ metric during the identifiers splitting/expansion), relied on manual intervention to ensure the correctness of the computed metrics. We measured the developers' experience using the years of experience: this is only a facet of the actual experience of a developer and it may not always represent the actual experience [48, 133]. As for the dependent variables, we tried to capture both the *perceived* and the *actual* code understandability in many ways. However, different results might be achieved combining *correctness* and *time* in different ways.

**Threats to internal validity** concern external factors that we did not consider that could affect the variables and the relations being investigated. Since two of the understandability proxies are time-related (*i.e.,* they are based on the time participants spent while understanding the code), it is possible that some participants were interrupted by external events while performing the comprehension task. For this reason, we replaced outliers for $TNPU$ in the computation of $TAU$ with the maximum $TNPU$ that was not an outlier. An outlier was a participant requiring more than $Q_3 + (3 \times IQR)$ seconds to understand a code snippet, where $Q_3$ is the third quartile and $IQR$ is the Inter Quartile Range. We used leave-one-out cross-validation to evaluate all the models used to answer $RQ_2$. This means that some of the evaluations of the same developer were used in the training set. This could allow the models to learn some peculiarities about the preferences of the developer. It is worth noting that such evaluations represent a large minority of the training instances ($< 2\%$) and they are unlikely to heavily affect

the trained model. Also, our assumption is that, in a real use-case scenario, developers might contribute understandability evaluations to the training set. We acknowledge that this assumption may not hold in all the contexts. Finally, as for $RQ_3$, the think-aloud strategy we used to get qualitative data could have affected the performance of the developers. We did not ask questions while the developers were reading and understanding the code to minimize such a threat.

**Threats to conclusion validity** concern the relation between the treatment and the outcome. The results of $RQ_2$ may depend on the used machine learning techniques. To limit this threat, we used the most common and widespread machine learning techniques, being careful to choose them from different families, such as tree-based, bayesian, and neural networks. Also, such results may depend on the parameters used for the machine learning techniques. We always used the standard parameters provided by Weka [66] for all the machine learning techniques.

**Threats to external validity** concern the generalizability of our findings. Our study has been performed on a large, but limited, set of metrics and by involving 63 participants comprehending a subset of 50 methods extracted from 10 Java systems. All of the results hold for the considered population of participants and for Java code. Larger studies involving more participants and code snippets written in other languages should be performed to corroborate or contradict our results. The same is true for the developers involved in $RQ_3$ (*i.e.,* they were from Italy with a similar background). We tried to select developers with diverse specializations: The three professional developers work in different areas (Android, front-end, back-end); one of the two PhD students had a previous experience in industry, while the other one did not. It is worth noting that it is very hard involving developers in such a long interview (more than an hour each). Since we observed differences in their evaluation of code understandability, a more comprehensive study with a more diverse set of developers would be needed to generalize our results, and it may highlight other factors that underline the subjectivity of code understandability.

## 7.6 Final Remarks

We presented an empirical study investigating the correlation between code understandability and 121 metrics related to the code itself, to the available documentation, and to the developer who is understanding the code. We asked 63 developers to understand 50 Java snippets, and we gathered a total of 444 evaluations. We assessed the participants' *perceived* and *actual* understanding for each snippet they inspected and the time they needed for the comprehension process. Our results demonstrate that, in most of the cases, there is no correlation between the considered metrics and code understandability. In the few cases, where we observed a correlation, its magnitude was very small. Combining metrics generally results in models with some discriminatory power (classification) and with a higher correlation, compared to single metrics (regression). However, such models are still far from being usable in practice for the prediction of understandability. Finally, we reported interviews with software developers, which provide useful insights about what makes code *understandable* or *not understandable*. We noticed that each developer puts emphasis on some aspects of understandability, and they give a different level of importance to each aspect. Note that we used shallow/classic models in our study. Deep learning models should be used as part of future work, because of their power to abstract complex relationships between input and output data, similarly to the cognitive processes in the human brain. As suggested by the interviews, developers exhibited some commonalities but also many differences in the understandability process that might not be captured by classic models, thus, shallow models are not the best choice to assess understandability.

Our study lays the foundations for future research on new metrics actually able to capture facets of code understandability.

In summary, the lessons learned from this chapter are the following:

- no single state-of-the-art metric (including readability, LOCs and Cyclomatic Complexity) is able to capture code understandability;
- combining state-of-the-art metric helps achieving relatively good performances, but we are still far from automatically assessing code understandability for practical usage;

- developers use different processes to understand code and different developers care about different aspects of source code.

Finally, there is still a main open issue: understandability seems to depend equally on code and developers, but most of the state-of-the-art metrics are about code. Future work should be aimed at defining new developer-related metrics to successfully assess code understandability.

# Part III

# Understandability From the Testing Perspective

*"Nam vitiis nemo sine nascitur; optimus ille est,*
*qui minimis urgetur."*
*"For no one is born without faults, and the best is the one*
*who has the fewest"*

Horace, Satires

CHAPTER 8

# Assessing Test Understandability with Coverage Entropy

## Contents



## 8.1 Introduction

The quality problems present in test cases have been defined as *test smells* [146, 109]. *Eager Test* is one of the smells that affects the most automatically generated tests [107] and that hinders the most their understandability [146]. A test is *eager* if it exercises more than a behavior of a specific class [146]. However,

the effort that developers make to understand a test does not simply depend on the number of called methods, but rather on the number of possible paths a developer can go down while debugging. Lawrance *et al.* [80] explained debugging using information foraging: when they are required to find a bug, developers tend to behave like predators searching for a prey, *i.e.,* they follow the path with the strongest "scent". When there are many paths and the scent is not strong enough, however, it is more likely that predators get lost.

**Motivational Example.** In Listing 8.1 we give an example of a *class under test* (CUT) that implements a simple parser for arithmetic expressions.

```
public class ArithmeticParser {
    public double add(Expression a, Expression b) {
        return evaluate(a) + evaluate(b);
    }

    public double sub(Expression a, Expression b)  {
        return evaluate(a) - evaluate(b);
    }

    public double evaluate(Expression x) {
        if (x.isSubtract())
            return sub(x.left, x.right);
        else if (x.isAdd())
            return add(x.left, x.right);
        else
            return x.value;
    }
}
```

Listing 8.1: ArithmeticParser class under test

The method `evaluate` dispatches the operations `add` and `sub` of the correspondent left and right sub expression. Thus, the method `evaluate` is recursively called to evaluate sub expressions until they are reduced to simple numbers. Assume a test that calls `evaluate(new Expression("10+5-100+2+23+43-2+32"))`: by definition, it would not be eager because it tests a single method. However, if such a test fails, the developer may need to manually simulate the execution of the test until she finds the problem. If the path that she needs to follow is long and twisted, like in the example, the effort is higher, regardless of the number of methods called by the original test.

Conversely, a *focused* test—a test that covers only branches belonging to few methods (possibly one)—presents a lower number of potential paths a developer

can walk and, therefore, it reduces the potential debugging effort. In the example, the perfect test suite including focused tests would be composed of three tests that call `evaluate`: one with a simple subtraction (*e.g.,* `new Expression("12-3")`), one with a simple sum (*e.g.,* `new Expression("7+2")`), and one with a simple number (*e.g.,* `new Expression("9")`). The lack of focus in tests, *i.e.,* the coverage of code from different methods, increases the number of context-switch operations that developers make to analyze the code. As a result, the effort needed to find a bug or to update a test is higher. Test focus is, therefore, an important property of understandable tests: unfocused tests are, inevitably, less understandable.

In this chapter we introduce *coverage entropy*, a metric that measures the focus of test cases: tests with low entropy are more focused, while entropic tests are less focused. *Coverage entropy* aims at capturing more accurately the potential effort that developers will make to understand a test case.

We conducted an empirical study to investigate the relationship between *coverage entropy* and *eager test*, to understand whether entropy and test eagerness capture two different aspects of quality. Our results show that eager tests are significantly more entropic compared to non-eager tests. However, the two metrics are only moderately correlated. Thus, we conclude that they capture two different shades of test quality.

This chapter is organized as follows: Section 8.2 defines coverage entropy; in Section 8.3 we report the comparison between coverage entropy and test eagerness; Section 8.4 discusses the threats to validity; Section 8.5 concludes this chapter.

## 8.2 Measuring Test Focus with Coverage Entropy

A test case is *focused* when it covers only branches belonging to few methods (possibly one) and thus follows the *Single-Condition Tests* principle [100]. To capture such a characteristic of test cases, we define a novel metric, *coverage entropy*, based on the classical concept of information entropy proposed by Shannon [130] and already used in SE studies [68, 27].

Let's suppose that a test case $t$ covers a set of branches $cov(t, C) = \{b_1, \ldots, b_n\}$ of a component $C$. Each branch $b_i$ belongs to a method $m_j \in methods(C)$. Using

the concept of information entropy we calculate coverage entropy as:

$$H(t,C) = -\sum_{m_j \in methods(C)} \left(\frac{|cov(t,m_j|)}{|cov(t,C)|}\right) \cdot \log\left(\frac{|cov(t,m_j)|}{|cov(t,C)|}\right) \tag{8.1}$$

where $cov(t,m_j) = \{b_\alpha, \ldots, b_\omega\}$ represents the covered branches of $m_j$.

Coverage entropy measures the entropy of the relative percentage of covered branches of the CUT. If a test is focused, *i.e.,* it covers branches belonging to a single method, its coverage entropy is 0. On the other hand, when many methods are called and the test covers the exact same amount of branches for all of them, the entropy is maximum (*i.e.,* 1). Considering only branches related to conditional statements in the code would make coverage entropy blind as for the call to branchless methods. Therefore, we take into account the virtual call branches that connect the invoker to the invoked method and we consider them as part of the called method. Such a branch is covered when the method is called. Consider the `ArithmeticParser` class in Listing 8.1 and the following test case:

```
ArithmeticParser parser = new ArithmeticParser();
int result = parser.evaluate(new Expression("3+11-5"))
assertEquals(9, result);
```

Such a test covers all the branches of the class. To compute coverage entropy, we first compute the total number of branches: in this case we have a virtual call branch for each method, *i.e.,* 4 (including the constructor), plus two branches for each `if` statement in the `evaluate` method (*i.e.,* 4). In total, we have 8 branches. Then, for each method, we compute the relative coverage: `evaluate` will have as relative coverage $\frac{5}{8}$, while all the other three methods will have relative coverage of $\frac{1}{8}$ each. We compute the information entropy of such values, which results to be ∼1.07. If we use `"3+6"` as the expression in the previous test, instead, we cover just the 3 branches of `evaluate` (excluding the `true` branch of the first `if`) and 3 virtual call branches of the 3 called methods (excluding `sub`). In this case, the total number of covered branches is 6, the relative coverage would be $\frac{4}{6}$ for `evaluate` and $\frac{1}{6}$ for the two others. The entropy, in this case, would be ∼0.88, lower than the other.

Coverage entropy depends not only on the test, but also on the CUT: for example, the optimal coverage entropy for the tests of a CUT containing a set of utility functions could be 0, while for CUTs with many methods that call each

other, such as the one previously analyzed, could be higher (*i.e.,* 0.88, in the example).

Coverage entropy has two interesting properties that make it a good metric to measure the effort that developers would devote to understand and maintain a test:

- *Setup-Methods Awareness:* If a test $t$ contains many method calls, but most of the covered branches are from a single method, the entropy is still low, because the test is focused on that method. This property is useful because it gives a lower importance to methods used to setup the status of the class (*e.g.,* setters).
- *Eager-Test Awareness*: like entropy in general, coverage entropy is lower-bounded, but it is not upper-bounded. In other words, the higher the number of called methods with the same number of covered branches, the higher the coverage entropy. Therefore, if a test calls many different methods the entropy very likely is high. We show the presence of this effect in Section 8.3.

## 8.3 Evaluation

The *goal* of our first exploratory study is to investigate the relationship between coverage entropy and the presence of a conceptually similar test smell, *i.e.,* eager test.

### 8.3.1 Empirical Study Design

Such a study is steered by the following research question:

$RQ_0$ *What is the relationship between coverage entropy and eager tests?*

#### Context of the Study

The *context* of this study consists of a random selection of classes from the SF110 corpus [56]. The SF110 benchmark is a set of Java classes extracted from the SourceForge repository that have been widely exploited in literature [111,

122, 128], consisting of 23,886 testable classes from 110 different projects. From such a set, we randomly select 100 Java classes. As suggested by previous work [112], we discard the trivial classes, *i.e.,* the ones having cyclomatic complexity lower than 5, for which generated test may be not useful.

We run EvoSuite [53] and we automatically generate test suites for the selected classes. In total, we generated 2,708 test cases. Test Smell Detector, a heuristic-based automatic test-smell detection tool introduced by Bavota *et al.* [14] was then used to detect eager tests. Test Smell Detector classifies tests as *eager* if they call more than a method of the class under test, and as *non-eager* otherwise. Such a tool is able to detect all the instances of eager tests (*i.e.,* 100% of recall), but it sometimes classifies as *eager* also non-eager tests (*i.e.,* precision of 88%) [14]. This happens when the call is done to setup the state of the class rather than to test a specific behavior. For this reason, we considered as *non-eager* all the negative instances reported by Test Smell Detector (*i.e.,* the test cases classified as *non-eager* by such a tool); we also manually analyzed a significant sample (99% confidence level with ±5% confidence interval) of positive instances (*i.e.,* tests classified as *eager* by the tool). Three evaluators (the author, a PhD student, and a professional developer) manually labeled 336 tests as *eager* or *not eager*. In the first phase, each evaluator independently labeled about two thirds of the test cases each, so that each instance had exactly two evaluations. A third evaluation was added by another evaluator when there was no agreement. The evaluators performed a total of 710 manual evaluations; 61% of the evaluated tests were actually eager, while 39% of them where false-positives. In total, our dataset is composed by 206 eager tests and 1,411 non-eager tests, totaling 1,614 test cases.

**Experimental Procedure**

To answer $RQ_0$, we check if there is a significant difference between eager and non-eager tests in terms of coverage entropy. We use a non-parametric two-tailed Mann-Whitney U test to check if the difference of coverage entropy between eager and non-eager tests is significant. We reject the null hypothesis (*i.e.,* there is no difference) if the p-value is lower than 0.05. We also use the Vargha-Delaney ($\hat{A}_{12}$) [147] test to measure the magnitude of such a difference. Such a test

has the following interpretation: when $\hat{A}_{12} > 0.50$ the values for a given metric achieved by one of the baselines are higher than the ones achieved by TERMITE, while $\hat{A}_{12} < 0.50$ indicates the opposite. Vargha-Delaney ($\hat{A}_{12}$) test also classifies the effect size into four different levels, *i.e., negligible*, *small*, *medium* and *large* [147].

Finally, we compute the correlation between coverage entropy and test eagerness (0 for non-eager tests and 1 for eager tests). We rely on the Kendall rank correlation coefficient (*i.e.,* Kendall's $\tau$) [75].

### 8.3.2 Empirical Study Results

As expected, we find that eager tests are significantly more entropic than non-eager tests (p-value $< 0.001$). The Vargha-Delaney $\hat{A}_{12}$ is 0.84 (large). However, the correlation between coverage entropy and test eagerness is $\sim$0.33, which is only *moderate*. This shows that, even if there is a clear relationship between eager tests and coverage entropy, the two concepts are different, and coverage entropy captures a new dimension of test quality.

Listing 8.2 shows an example of eager test with low entropy. Such a test exercises two distinct methods of the CUT, *i.e.,* `isAzStyle` and `decodeMnemonic`. However, since the coverage is unbalanced among the two methods, the resulting coverage entropy is still quite low. On the other hand, we report in Listing 8.3 an example of non-eager test with very high coverage entropy: in this case, the test just calls a single method, but such a method indirectly calls plenty of other methods. If the behavior of such a method changes, *i.e.,* a check for null is added in a specific method to avoid the exception, developers not aware of the change may still struggle finding the cause of the problem and adapting such a test to the new behavior.

Listing 8.2: Eager Test with low entropy (0.45)

```
BTPeerIDByteDecoderDefinitions.VER_AZ_LAST_THREE_DIGITS = "1.2";
BTPeerIDByteDecoderDefinitions.VER_AZ_THREE_DIGITS = "1.2(34)";
BTPeerIDByteDecoderUtils.isAzStyle("RM");
BTPeerIDByteDecoderUtils.decodeMnemonic('x');
```

Listing 8.3: Non-eager test with high entropy (3.43)

```
try {
    Attribute.main((String[]) null);
    fail("Expecting exception: NoSuchElementException");
} catch(NoSuchElementException e) {
    verifyException("java.util.LinkedList", e);
}
```

**Summary of $RQ_0$.** Eager tests generally suffer from higher coverage entropy; However, eager tests are not always focused, and focused tests are not always non-eager. Therefore, coverage entropy needs to be used as a distinct quality indicator.

## 8.4 Threats to Validity

**Construct Validity**. The results of our study mostly depend on the classification of a set of test cases as *eager* or *non-eager*. We automatically performed part of such classification using a tool. Since such a tool achieves 100% of recall [14], we focused our manual validation on a sample of the tests classified as *eager*, while we assumed that the other tests were *non-eager*. The manual labeling we performed could depend on the evaluator. To reduce the possibility of human errors, we made sure that at least two of the evaluators labeled each test cases, and, in case of disagreement, the third evaluator helped in the final decision. We observed disagreement in 38 cases, but full consensus was reached after a brief discussion.

**Internal Validity**. In our study we have an unbalanced set of eager/non-eager tests (∼13% eager and ∼87% non-eager tests). This was due to the unbalanced effort required to classify tests as non-eager (fully automated) and eager (partially manual). Statistical tests show that such a sample is sufficient to conclude that the difference in terms of coverage entropy among the two population is significant. We tried to replicate the results with 1,000 random subsamples of the non-eager tests of the same size of the sample of eager tests we considered. We observed no relevant difference compared to what we observed in our study as for the significance of the comparison. However, we observed a higher correlation (on average $\sim 0.48$), probably due to the fact that the sample is smaller.

**External Validity**. It is possible that the conclusion of our empirical study is not valid for all the Java classes. We generated test cases for 100 Java classes using MOSA. In total, our study took into account 1,614 test cases. Since we randomly selected the 100 subject classes, we believe that there is no bias in such a sample.

## 8.5 Final Remarks

A test case is *focused* when it covers only branches belonging to few methods (possibly one). We introduced *coverage entropy*, a metric that captures the focus of test cases and we conducted an empirical study in which we compare coverage entropy with another similar concept, *i.e.,* test eagerness.

The lesson learned from this chapter is that coverage entropy is only moderately correlated with the presence of eager tests. Besides, we show that, as expected, eager tests are significantly more entropic than non-eager tests.

There is still a main open issue: we conclude that test focus and test understandability are related; test cases are less abstract than source code, *i.e.,* they provide specific inputs and they result in the coverage of specific paths in the source code. Our conjecture is that code coverage focused on few methods helps reducing the number of potential "travels" the developers do in the source code and, therefore, increases its understandability. However, we still lack evidence that developers find tests with lower coverage entropy more understandable. Future work should be aimed at empirically proving this conjecture.

CHAPTER 9

# Improving the Understandability of Generated Tests

## Contents



## 9.1 Introduction

The lack of quality of automatically generated tests is well known in the literature [61, 107]. Previous work tried to tackle this problem: Daka *et al.* [38] defined an approach to improve the readability of generated tests; however,

they conclude that tests improved by using their approach are not significantly more understandable than other generated tests. More recently, Palomba *et al.* [108] proposed QUALITY-BASED MOSA, an approach that optimizes cohesion and coupling as a secondary criterion of the evolutionary search to improve test quality. However, the quality of generated test cases could still suffer from other types of problems, such as the presence of eager tests and low understandability.

In this chapter we introduce TERMITE, an approach that integrates *coverage entropy* optimization in the test case generation process to produce more focused test cases. TERMITE exploits program slicing to break generated tests into simpler (potentially less entropic) tests. Then, it uses *coverage entropy* (introduced in Chapter 8) as a preference criterion to give an evolutionary advantage to focused test cases.

We conducted an empirical study to evaluate the quality of the tests generated with TERMITE. We used QUALITY-BASED MOSA as a baseline and we compared coverage entropy, cohesion, and coupling (along with the achieved branch coverage) of the tests generated by QUALITY-BASED MOSA and TERMITE. The results achieved show that TERMITE generates test cases with a significantly higher quality as for coverage entropy (for ∼88% of the classes) and cohesion (for ∼78% of the classes), while QUALITY-BASED MOSA generates slightly less coupled test cases. Finally, we show that TERMITE is significantly more efficient (it takes 57% less time) and effective (it achieves higher branch coverage for about 73% of the classes).

This chapter is organized as follows: Section 9.2 introduces TERMITE; in Section 9.3 we report the evaluation of TERMITE and its comparison with QUALITY-BASED MOSA; Section 9.4 discusses the threats to validity; Section 9.5 concludes this chapter.

## 9.2 TERMITE: Focused Test Case Generation

In order to improve the focus of automatically generated test cases, we introduce TERMITE (Test Entropy Reduction for MOSA), a novel test case generation strategy outlined in Algorithm 1. The core part of the algorithm is the same as MOSA [111]. We introduce a new step with the *SLICING* routine (line 12 of

**Algorithm 1** Test Entropy Reduction for MOSA (TERMITE)

**Input:** $B = \{b_1, \ldots, b_n\}$ set of branches of a program
**Result:** A test suite T

```
 1: t ← 0                                                    ▷ current generation
 2: P_t ← RANDOM-POPULATION(M)
 3: archive ← UPDATE-ARCHIVE(P_t, B)
 4: E ← 0
 5: while not stop_condition() do
 6:     Q_t ← GENERATE-OFFSPRING(P_t, B)
 7:     R_t ← P_t ∪ Q_t
 8:     Δ_E ← E− COVERAGE-ENTROPY(R_t)
 9:     if Δ_E > 0 then
10:         E ← COVERAGE-ENTROPY(R_t)
11:         R_t ← R_t ∪ SLICING(R_t)
12:     end if
13:     F ← PREFERENCE-SORTING(R_t)
14:     P_{t+1} ← ∅
15:     d ← 0
16:     while | P_{t+1} | + | F_d | ≤ M do
17:         CROWDING-DISTANCE-ASSIGNMENT(F_d)
18:         P_{t+1} ← P_{t+1} ∪ F_d
19:         d ← d + 1
20:     end while
21:     Sort(F_d)                                           ▷ according to the crowding distance
22:     P_{t+1} ← P_{t+1} ∪ F_d [1 : (M− | P_{t+1} |)]
23:     archive ← UPDATE-ARCHIVE(archive ∪ P_{t+1}, B)
24:     t ← t + 1
25: end while
26: T ← archive
```

Algorithm 1), in which the individuals in the population are divided in simpler pieces in case high coverage entropy is detected. Then, we use coverage entropy as the secondary criterion—instead of test size, as MOSA [111]—to prefer low-entropy test cases when the same level of branch coverage is achieved. When coverage entropy is equal, we prefer shorter tests.

The slicing procedure takes place on each test case belonging to $R_t$, that is, the union of the current population and offspring population. The output $S$ of the slicing procedure is combined with $R_t$ and the *PREFERENCE-SORTING* function is executed on $R_t$ itself (line 13 of Algorithm 1). Therefore, TERMITE keeps in the population the original test cases. In fact, it might happen that a larger test case has some side effects (*e.g.,* set of a static variable) that can result in the coverage of specific branches that cannot be covered with simpler tests. In these cases, complete tests are more useful and, therefore, they are preferred by the genetic algorithm. In the opposite case, small pieces may be preferred, since they contain a lower number of method calls, *i.e.,* tend to be less entropic.

**Algorithm 2** SLICING procedure in TERMITE

**Input:** A test case $tc$, the CUT $C$
**Result:** A set $S$ of slices

```
1: S_c ← ∅                                              ▷ complete sequence (CS) slices
2: S_p ← ∅                                              ▷ partial sequence (PS) slices
3: S_sup ← ∅                                            ▷ support slices
4: M ← ∅                                                ▷ slice seeds
5: for all st ∈ statements(tc) do
6:     d ← ∅
7:     for all v ∈ use(st) do
8:         d ← {s ∈ (S_c ∪ S_sup) s.t. v ∈ defs(s) }    ▷ dependencies of st
9:     end for
10:    s* ← merge(d, st)
11:    if def(st) is of type C then
12:        S_c ← S_c ∪ {s*}                             ▷ s* is a CS slice
13:        M ← M ∪ { def(st) }
14:    else
15:        if | use(st) ∩M| > 0 then
16:            S_p ← S_p ∪ (d ∩ S_c)                    ▷ CS slices become PS slices
17:            S_c ← S_c ∪ {s*} − (d ∩ S_c)             ▷ s* added to CS slices
18:        else
19:            S_sup ← S_sup ∪ {s*} − (d ∩ S_sup)       ▷ s* is a support slice
20:        end if
21:    end if
22: end for
23: S ← S_c ∪ S_p
```

### 9.2.1 Test Slicing

When using coverage entropy as a secondary criterion, still the average entropy of the population may increase during the evolutionary process. For example, it may happen that complex tests that cover some branches are found and kept in the population. At each iteration, TERMITE checks whether the coverage entropy is increased from the previous generation (line 10 of Algorithm 1). Whether the entropy has increased, the approach tries to reduce the overall entropy of the population by performing slicing on all the test cases. The aim of this step is to introduce simpler individuals, *i.e.,* less entropic ones, in the current population. Especially, two types of slices are generated: *complete sequence (CS) slices* and *partial sequence (PS) slices.*

If a test contains more than an instance of the CUT, we want to divide such instances into different tests. For example, if the CUT is the class `Stack` and a test $T$ contains two independent instances of such a class, we want to separate the lines involving the first instance from the lines involving the second one (*i.e.,* perform a *complete sequence slices*). On the other hand, if $T$ invokes several methods on a single instance, *e.g.,* `push` and `pop`, we want to keep slices of $T$ in which only parts

of the methods are invoked (*i.e.,* perform a partial sequence slices). Since the test case representation used for object-oriented code in evolutionary algorithms does not contain control statements [142], slicing them is much simpler than slicing normal programs.

The pseudo-code of the algorithm TERMITE that extracts all the slices from a candidate test case is outlined in Algorithm 2. For each statement *st* in a given test, TERMITE considers all the slices ($d$) on which such a statement depend, *i.e.,* the slices in which the variables used by *st* are defined (lines 7-9 of Algorithm 2). Thus, a new slice, $s^*$ is created, merging all such slices $d$ together with *st* (line 10 of Algorithm 2). If *st* does not depend on any other variable, $s^*$ contains only such a statement. Then, three scenarios are possible: (i) *st* defines a new instance of the CUT: in this case, $s^*$ becomes a new complete sequence slice (lines 11-13 of Algorithm 2); (ii) *st* uses at least an instance of the CUT: $s^*$ becomes an complete sequence slice (line 17 of Algorithm 2); the complete sequence slices in which the instances of the CUT used in the statement *st* were defined are added to the set of partial sequence slices (line 16 of Algorithm 2); thus, these slices are deleted from the set of the complete sequence slices; (iii) no instances of the CUT are used: in this case, $s^*$ is a support slice, *i.e.,* a slice in which other variables are defined (line 19 of Algorithm 2). At the end, the final set of slices is composed by the union of the complete sequence and partial sequence slides (line 23 of Algorithm 2): they are the only ones that contain at least a call to a method in the CUT.

Listing 9.1 shows an example of a test case generated for the class `Stack`. To clarify the slicing process, we report both *complete sequence slices* (reported in Listing 9.2) and *partial sequence slices* (reported in Listing 9.3) extracted from such a class. In the original test case there are several method calls on different instances of `Stack` in a completely chaotic way. Using slicing it is possible to drastically reduce the complexity of the test. In this case, we extract only three complete sequence slices (since there are three independent instances of `Stack`); besides, we extract several partial sequence slices (we report only the first 5 of them). Note that partial sequence slices may help to introduce in the population

```
Stack<Object> stack0 = new Stack<Object>();
stack0.isEmpty();
Stack<String> stack1 = new Stack<String>();
stack1.pop();
Stack<Integer> stack2 = new Stack<Integer>();
int int0 = 1;
Integer integer0 = new Integer(int0);
stack2.push(integer0);
int int1 = 205;
Integer integer1 = new Integer(int1);
stack2.push(integer1);
stack0.pop();
stack1.pop();
Object object0 = new Object();
stack0.push(object0);
String string0 = "f5d24r:";
stack1.push(string0);
```

Listing 9.1: Test Case generated by EvoSuite

```
// Complete Sequence Slice 1
Stack<Object> stack0 = new Stack<Object>();
stack0.isEmpty();
stack0.pop();
Object object0 = new Object();
stack0.push(object0);

// Complete Sequence Slice 2
Stack<String> stack1 = new Stack<String>();
stack1.pop();
stack1.pop();
String string0 = "f5d24r:";
stack1.push(string0);

// Complete Sequence Slice 3
Stack<Integer> stack2 = new Stack<Integer>();
int int0 = 1;
Integer integer0 = new Integer(int0);
stack2.push(integer0);
int int1 = 205;
Integer integer1 = new Integer(int1);
stack2.push(integer1);
```

Listing 9.2: Complete Sequence (CS) Slices

tests that exercise a single behavior of the production code (*i.e.*, less entropic ones). A typical example is Partial Sequence Slice 1.2 of Listing 9.3.

```
// Partial Sequence Slice 1.1
Stack<Object> stack0 = new Stack<Object>();

// Partial Sequence Slice 1.2
Stack<Object> stack0 = new Stack<Object>();
stack0.isEmpty();

// Partial Sequence Slice 1.3
Stack<Object> stack0 = new Stack<Object>();
stack0.isEmpty();
stack0.pop();

// Partial Sequence Slice 2.1
Stack<String> stack1 = new Stack<String>();

// Partial Sequence Slice 2.2
Stack<String> stack1 = new Stack<String>();
stack1.pop();
```

Listing 9.3: Some of the Partial Sequence Slices

## 9.3 Evaluation

The *goal* of this empirical study is to evaluate the *quality* of the tests generated by TERMITE and their effectiveness (*i.e.,* branch coverage).

### 9.3.1 Empirical Study Design

We formalize our research questions as follow:

$RQ_1$ *Does entropy optimization produce more focused test cases?*. With this first research question we investigate whether the proposed approach produces higher quality test cases as for focus (measured with coverage entropy), cohesion, and coupling compared to QUALITY-BASED MOSA [108].

$RQ_2$ *Does entropy optimization affect the effectiveness of test generation?*. In our second research question we check what is the impact of our approach on branch coverage.

#### Prototype Tool

We implemented TERMITE in a prototype tool extending the EVOSUITE test-data generation tool (version 1.0.6). The delta of our implementation is explained in Section 9.2. We also implemented QUALITY-BASED MOSA [108] in our version of EVOSUITE to evaluate our approach. Especially, the authors of the

original paper kindly provided us with the APIs needed to compute the metrics of cohesion and coupling used in their work [108]. All the experimental results reported in this chapter are obtained by using such a prototype tool.

### Context of the Study

The *context* of the second study is again a random selection of classes from the SF110 corpus [56]. The subject classes have been selected as follows: (i) we take 43 classes used in the work by Palomba *et al.* [108]; and (ii) we randomly select 17 additional subjects from the SF110 benchmark. Also in this study, we did not take into account any trivial class [112]. We end up with 60 classes in total. There is no overlap between subject classes we use for the first and the second study.

### Experimental Procedure

We compare TERMITE using QUALITY-BASED MOSA as a baseline to answer both our research questions. Indeed QUALITY-BASED MOSA has been demonstrated to produce test cases of better quality with slightly higher coverage, compared to MOSA [108]. These results have driven the adoption of QUALITY-BASED MOSA as a baseline for the evaluation of TERMITE.

We run the two test generation strategies for each class in our dataset 30 times each [26], to deal with the non-deterministic nature of the employed genetic algorithms. In this step, we collect (i) the coverage entropy, and (ii) the resulting branch coverage. We use as stop criterion for the evolutionary process the total number of generations instead of the execution time. We do this to avoid biases due to the efficiency of the implementation. A generation in the genetic algorithm may take a few seconds for some classes and even minutes for other classes. For this reason, we use different number of generations for different classes. Specifically, we preliminary run MOSA on all the classes with a fixed time budget of 3 minutes [26]. Then, we check how many generations were evolved for each class and we use such values for our experiment. We used MOSA instead of one of the two concurrent approaches in this phase to avoid possible biases. Finally, we set a global timeout of 10 minutes for each class.

As for the other parameters related to the genetic algorithm configuration (*e.g.,* population size or mutation score) we adopt the default parameters used by EvoSuite, since such values provide good results [58].

We compare coverage entropy, cohesion and coupling of the generated test cases among the three approaches to answer $RQ_1$. Instead, we compare branch coverage and the time needed by EvoSuite to evolve the selected classes for the number of generations previously determined to answer $RQ_2$. We compute coverage entropy on the generated test cases in EvoSuite right at the end of the evolutionary process. Indeed, such a metric can only be computed using runtime information (*e.g.,* executed methods). On the other hand, we statically compute cohesion and coupling on the resulting test suites by using the the APIs provided by Palomba *et al.* [108].

We adopt the non-parametric Wilcoxon Rank Sum Test [35] with significance level $\alpha = 0.05$ to compare both quality metrics and effectiveness metrics. Significant *p-values* indicate that the corresponding null hypothesis can be rejected in favor of the alternative one, *i.e.,* one of the approaches is able to generate significantly better test cases as for their quality ($RQ_1$) or their effectiveness ($RQ_2$). Moreover, we use the Vargha-Delaney ($\hat{A}_{12}$) [147] test to measure the magnitude of the differences between the results achieved by the two experimented approaches.

### 9.3.2 Empirical Study Results

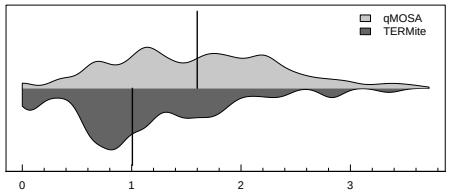
This section presents and discusses the results of the second study answering the RQs formulated in Section 9.3.

**$RQ_1$: Quality of Entropy Optimized Tests** In Table 9.1 we report the mean coverage entropy, lack of cohesion and coupling achieved by both TERMite and quality-based MOSA. In all the cases, lower is better. We show in boldface the cases in which one of the approaches is significantly better than the other ($p < 0.05$); for those cases, we also report the magnitude of the difference (**L**arge, **M**edium, **S**mall, or **N**egligible). In such cases, TERMite achieves better results when $\hat{A}_{12}$ is lower than 0.5, while the opposite happens otherwise. As for coverage entropy, amongst the 60 classes we analyze, we obtain significant differences for

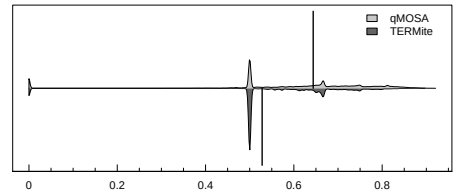
| | | Branch Coverage | | | Coverage Entropy | | | Lack of Cohesion | | | Coupling | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Class** | **#Branches** | **QMOSA** | **TERMITE** | **$\hat{A}_{12}$** | **QMOSA** | **TERMITE** | **$\hat{A}_{12}$** | **QMOSA** | **TERMITE** | **$\hat{A}_{12}$** | **QMOSA** | **TERMITE** | **$\hat{A}_{12}$** |
| Product | 30 | 0.10 | 0.10 | | 0.65 | **0.64** | **S** | 0.50 | 0.50 | | 0.05 | **0.04** | **S** |
| FileUtil | 124 | 0.42 | **0.55** | **L** | 2.40 | **1.15** | **L** | **0.42** | 0.52 | **L** | **0.26** | 0.27 | **N** |
| MessageFormatter | 38 | 0.85 | **0.88** | **L** | 1.26 | **0.85** | **L** | 0.59 | **0.49** | **L** | 0.44 | **0.37** | **S** |
| ArchiveScanner | 44 | 0.65 | **0.87** | **L** | 1.01 | **0.88** | **L** | 0.53 | **0.51** | **S** | **0.43** | 0.49 | **M** |
| OutputFormat | 38 | 0.98 | **0.99** | **L** | 2.04 | **1.42** | **L** | 0.76 | **0.72** | **S** | **0.30** | 0.37 | **M** |
| SAXReader | 60 | 0.87 | **0.98** | **L** | 2.10 | **1.49** | **L** | 0.60 | **0.56** | **S** | 0.33 | **0.31** | **N** |
| Services | 8 | 0.73 | 0.73 | | 2.01 | **0.60** | **L** | 0.49 | **0.14** | **L** | 0.15 | 0.09 | |
| EncodingFactory | 194 | 0.79 | **0.96** | **L** | 1.18 | **0.16** | **L** | 0.57 | **0.50** | **L** | **0.15** | 0.18 | **S** |
| AbstractJavaGDSImpl | 998 | 0.19 | **0.30** | **L** | 1.14 | **0.76** | **L** | NaN | NaN | | NaN | NaN | |
| FBCachedFetcher | 108 | 0.50 | **0.60** | **L** | 1.95 | **1.17** | **L** | 0.61 | **0.49** | **L** | 0.41 | **0.36** | **S** |
| FBProcedureCall | 90 | 0.81 | **0.91** | **L** | 1.83 | **0.95** | **L** | 0.58 | **0.55** | **M** | 0.42 | **0.42** | **N** |
| Library | 32 | 0.55 | 0.60 | | 1.15 | **0.81** | **L** | 0.59 | **0.51** | **L** | **0.32** | 0.45 | **L** |
| JDayChooser | 170 | 0.67 | **0.84** | **L** | 2.94 | **2.84** | **L** | 0.70 | **0.64** | **M** | 0.28 | **0.26** | **S** |
| Controller | 298 | 0.03 | **0.04** | **L** | 0.70 | **0.60** | **L** | 0.24 | 0.25 | | 0.41 | **0.29** | **M** |
| Player | 14 | **0.99** | 0.98 | **M** | 2.24 | **1.62** | **L** | 0.78 | **0.71** | **L** | **0.30** | 0.31 | **N** |
| OracleIdentiteDao | 74 | 0.47 | 0.47 | | 1.37 | **0.94** | **L** | NaN | NaN | | NaN | NaN | |
| ClientMsgReceiver | 12 | 0.46 | **0.47** | **N** | 0.84 | **0.79** | **L** | 0.51 | **0.50** | **N** | **0.47** | 0.53 | **S** |
| PhdBuilder | 22 | 0.85 | **0.99** | **L** | 0.67 | **0.58** | **L** | 0.31 | 0.32 | | 0.39 | 0.39 | |
| Range | 322 | 0.53 | **0.84** | **L** | 3.27 | **2.81** | **L** | 0.63 | 0.62 | | 0.25 | **0.17** | **M** |
| DefaultNucleotideCodec | 60 | 0.98 | **0.99** | **M** | 2.27 | **1.75** | **L** | 0.00 | 0.00 | | 0.38 | **0.33** | **S** |
| FileIterator | 36 | 0.97 | **0.99** | **L** | 2.71 | **2.37** | **L** | **0.54** | 0.55 | **N** | 0.29 | 0.29 | |
| SQLUtil | 174 | 0.86 | **0.98** | **L** | 1.33 | **0.60** | **L** | 0.63 | **0.53** | **L** | 0.13 | **0.08** | **M** |
| MethodWriter | 816 | 0.19 | **0.53** | **L** | **0.82** | 1.13 | **L** | 0.54 | **0.56** | **N** | 0.33 | **0.30** | **S** |
| JavaCharStream | 198 | 0.71 | **0.95** | **L** | 0.97 | **0.64** | **L** | 0.58 | **0.55** | **S** | 0.34 | 0.32 | |
| JavaParserTokenManager | 1,696 | 0.26 | **0.67** | **L** | **1.56** | 1.83 | **L** | 0.50 | **0.55** | **S** | **0.35** | 0.37 | **S** |
| SimpleNode | 58 | 0.86 | **0.97** | **L** | 1.26 | **0.87** | **L** | 0.65 | **0.59** | **M** | **0.28** | 0.30 | **S** |
| UsernamePasswordToken | 14 | 0.92 | 0.92 | | 1.86 | **1.49** | **L** | 0.68 | **0.65** | **S** | **0.27** | 0.30 | **S** |
| DefaultWebSecurityManager | 52 | 0.73 | **0.74** | **S** | 1.64 | **1.46** | **L** | 0.55 | **0.51** | **S** | 0.37 | 0.36 | |
| LagoonCLI | 64 | 0.30 | **0.35** | **L** | 0.35 | **0.07** | **L** | 0.50 | **0.45** | **N** | **0.31** | 0.34 | **S** |
| SupportingDocument | 20 | 0.98 | **1.00** | **S** | 2.46 | **2.19** | **L** | 0.77 | **0.74** | **S** | 0.41 | **0.39** | **N** |
| Variable | 96 | 0.76 | **0.99** | **L** | 2.79 | **1.94** | **L** | 0.82 | **0.70** | **L** | **0.25** | 0.27 | **S** |
| Base64Decoder | 58 | 0.38 | **0.55** | **M** | 0.50 | 0.49 | | 0.50 | 0.50 | | **0.38** | 0.48 | **M** |
| ExpressionMatrixImpl | 54 | 0.96 | **0.99** | **S** | 1.63 | 1.66 | | 0.69 | **0.68** | **S** | **0.35** | 0.37 | **S** |
| ConnectionFactories | 72 | 0.99 | 0.99 | | 1.34 | **0.92** | **L** | 0.55 | **0.53** | **S** | 0.49 | 0.48 | |
| VisualListModel | 138 | 0.73 | **0.85** | **L** | 1.68 | **0.77** | **L** | 0.62 | **0.52** | **L** | **0.36** | 0.39 | **S** |
| TheClient | 2 | 0.98 | 0.96 | | 2.28 | **1.49** | **L** | 0.71 | **0.70** | **N** | **0.33** | 0.36 | **S** |
| AdvancedSettings | 24 | 0.99 | 0.99 | | 2.72 | **2.19** | **L** | 0.81 | **0.78** | **M** | 0.50 | 0.50 | |
| RIFClassLoader | 0 | 1.00 | 1.00 | | 0.96 | **0.69** | **L** | 0.55 | **0.50** | **L** | **0.21** | 0.38 | **L** |
| XPathLexer | 458 | 0.55 | **0.88** | **L** | 2.14 | **1.57** | **L** | **0.46** | 0.49 | **S** | **0.25** | 0.37 | **L** |
| ForeignKeyConstraint | 62 | 0.82 | 0.83 | | 1.64 | **0.89** | **L** | 0.64 | **0.49** | **L** | **0.25** | 0.35 | **L** |
| JSJshop | 104 | 0.39 | 0.38 | | 1.08 | **0.73** | **L** | 0.51 | **0.43** | **M** | 0.25 | **0.23** | **N** |
| JSPredicateForm | 84 | 0.67 | **0.91** | **L** | 1.27 | **0.61** | **L** | 0.42 | **0.39** | **N** | 0.36 | **0.32** | **S** |
| JSTerm | 182 | 0.64 | **0.83** | **L** | 1.73 | **1.04** | **L** | 0.59 | **0.50** | **S** | **0.37** | 0.40 | **S** |
| Session | 158 | 0.00 | 0.00 | | 0.00 | 0.00 | | 0.47 | **0.00** | **L** | 0.00 | 0.00 | |
| DBUtil | 448 | 0.18 | **0.28** | **L** | 0.48 | **0.06** | **L** | 0.57 | **0.50** | **L** | **0.27** | 0.28 | **N** |
| HomeEnvironment | 52 | 0.94 | 0.93 | | 2.25 | **0.86** | **L** | 0.78 | **0.57** | **L** | **0.24** | 0.41 | **L** |
| EWrapperMsgGenerator | 34 | 0.94 | **0.99** | **L** | 1.22 | **0.10** | **L** | 0.69 | **0.51** | **L** | **0.12** | 0.19 | **M** |
| Evaluation | 778 | 0.34 | **0.49** | **L** | 1.84 | **1.46** | **L** | 0.74 | **0.68** | **M** | 0.22 | 0.21 | |
| NaiveBayesMultinomialText | 154 | 0.64 | **0.93** | **L** | 2.30 | **1.64** | **L** | 0.81 | **0.71** | **L** | **0.20** | 0.29 | **L** |
| JRip | 362 | 0.24 | **0.32** | **L** | 1.72 | **1.12** | **L** | 0.64 | **0.55** | **S** | **0.26** | 0.30 | **S** |
| Optimization | 434 | 0.08 | 0.08 | | 0.72 | **0.33** | **L** | 0.36 | **0.28** | **S** | 0.30 | **0.24** | **M** |
| LovinsStemmer | 422 | 0.45 | **0.70** | **L** | **1.09** | 1.28 | **L** | 0.79 | **0.67** | **L** | **0.23** | 0.27 | **S** |
| ResultMatrix | 374 | 0.74 | **0.98** | **L** | 3.52 | **3.36** | **L** | 0.75 | **0.73** | **S** | **0.12** | 0.14 | **S** |
| Discretize | 212 | 0.35 | **0.62** | **L** | 2.09 | **1.84** | **L** | 0.71 | **0.63** | **M** | **0.32** | 0.36 | **S** |
| FieldWriter | 56 | 0.97 | 0.97 | | 1.01 | **0.82** | **L** | 0.63 | **0.55** | **M** | 0.36 | 0.36 | |
| Frame | 686 | 0.37 | **0.78** | **L** | **0.76** | 0.82 | **M** | 0.52 | 0.52 | | **0.57** | 0.59 | **N** |
| Component | 310 | 0.28 | **0.80** | **L** | 2.19 | **1.73** | **L** | **0.59** | 0.65 | **N** | **0.31** | 0.36 | **M** |
| JSONObject | 270 | 0.58 | **0.88** | **L** | 1.96 | **1.13** | **L** | 0.56 | **0.53** | **S** | 0.26 | **0.17** | **M** |
| DynamicSelectModel | 24 | 0.73 | **0.96** | **L** | 1.62 | **1.01** | **L** | 0.64 | **0.54** | **L** | 0.40 | 0.42 | |
| RecordTypeMessage | 44 | 0.54 | 0.54 | | 1.82 | **1.08** | **L** | 0.55 | **0.50** | **L** | **0.35** | 0.45 | **L** |

Table 9.1: Mean Branch Coverage and Quality Metrics Comparison.

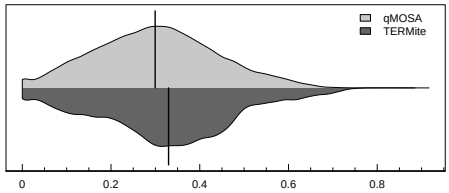
57 subjects. In detail, the test suites generated by TERMITE have lower entropy for 53 out of those 57 cases—with 37 *large* effect sizes— while the contrary only happens in 4 cases. The distribution of the coverage entropy is shown in Figure 9.1a, plotted (like the others in this chapter) with the `DistDiff` R package [90]. The vertical lines represent the median of the distributions, *i.e.,* 1.6 and 1.0 for



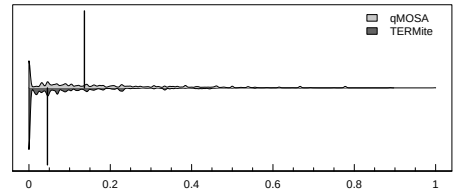

(a) Distribution of the Coverage Entropy.




(b) Distribution of the Lack of Cohesion metric.




(c) Distribution of the Coupling.




(d) Distribution of the Test-Eager severity.

Figure 9.1: Distribution of four quality metrics. The vertical bars indicate the median of the distribution.

QUALITY-BASED MOSA and TERMITE, respectively. On average, the entropy of the test cases generated by TERMITE is about 29% lower; the maximum reduction is about 70% for the class `Services`.

The results show also that TERMITE generates significantly more cohesive tests, even if slightly more coupled, compared to QUALITY-BASED MOSA. The median lack of cohesion of generated tests is 0.53 for TERMITE and 0.64 for QUALITY-BASED MOSA. We show that in 47 classes out of 60 TERMITE achieves a significantly lower lack of cohesion (*i.e.,* higher cohesion). Such a big difference is surprising: while QUALITY-BASED MOSA explicitly tries to minimize such an attribute, TERMITE does not take it into account. The reason why this happens may be that TERMITE uses slicing to reduce entropy, and this also affects the number of test methods and, therefore, the cohesion. On the other hand, we observe that there are more cases in which QUALITY-BASED MOSA generates less coupled test cases (32 classes) than the opposite (16 classes). This is probably due to the fact that TERMITE tends to generate plenty of tests, much more than QUALITY-BASED MOSA (21.7 vs 49.7, *i.e.,* ∼128% more), since

each test case is smaller, on average. Considering the higher number of test cases, the difference in terms of coupling is rather small: the average coupling for TERMITE is 0.32, while it is 0.30 for QUALITY-BASED MOSA. Therefore, such a difference is a price to pay for having more focused and cohesive tests.

We also investigated more in depth the difference among the approaches in terms of the quality of the generated test cases. Specifically, we report the eager test smell intensity [110], calculated for a test case $TC$ as the ratio between the number of CUT methods called by $TC$ and the total number of methods in the CUT; the intensity is 0 if $TC$ calls only 1 CUT method. It lies in the interval $[0, 1)$: 0 means that the test is not smelly, while other values indicate the smell severity. Figure 9.1d shows the distribution of *eager test* intensity for both QUALITY-BASED MOSA and TERMITE. We observe that the test generated by TERMITE have lower values of severity: the median of the distribution is 0.13 and 0.04 for QUALITY-BASED MOSA and TERMITE, respectively. Such results strengthen the findings we observed for coverage entropy and quality metrics: despite the metrics are not strongly correlated, the generation of more cohesive and less entropic tests is also beneficial in addressing the *eager test* smell.

In Listings 9.4 and 9.5 we report two test cases generated by QUALITY-BASED MOSA and TERMITE, respectively. It is clear that the latter is able to generate tests that are much easier to understand, just using a single method, in this case.

```
Services.SIMPLE_Result sr0 = new Services.SIMPLE_Result();
sr0.time = (-784L);
Services.HTTP_Result hr0 = Services.testHTTPS((String) null, 0);
hr0.toString();
hr0.reset();
hr0.reset();
sr0.toString();
sr0.works = false;
hr0.page_weight = 3000;
hr0.works = false;
Services.SIMPLE_Result sr1 = new Services.SIMPLE_Result();
hr0.reset();
hr0.toString();
Services.testPOP3("", 0);
String string0 = hr0.toString();
assertEquals("...", string0);
```

Listing 9.4: Test generated by QUALITY-BASED MOSA.

```
Services.HTTP_Result sr0 = new Services.HTTP_Result();
String string0 = sr0.toString();
assertEquals("...", string0);
```

Listing 9.5: Test generated by TERMITE.

**Summary for $RQ_1$**: TERMITE generates higher quality tests compared to QUALITY-BASED MOSA. Specifically, we observe an improvement in coverage entropy, cohesion, and eager test severity. On the other hand, QUALITY-BASED MOSA generates slightly less coupled tests.

**$RQ_2$: Branch Coverage of TERMITE** We report the comparison of branch coverage in Table 9.1, similarly as we do for the quality attributes analyzed in $RQ_1$. Since in this case higher is better, TERMITE achieves better results when $\hat{A}_{12}$ is higher than 0.5, while the opposite happens otherwise. In this analysis, we observe statistically significant differences in 44 out of 60 subjects (73%). In particular, in 43 cases out of 60 (about 72%), the test suite generated by TERMITE has higher coverage than the one generated by QUALITY-BASED MOSA. This improvement ranges from 1% in branch coverage (for the class `DefaultNucleotideCodec`) up to +52% (for the class `Component`). Indeed, the average branch coverage for the 60 subjects is 62% and 74%, respectively for TERMITE and QUALITY-BASED MOSA. In the majority of the statistically significant cases, *i.e.,* for 32 out of 43 subjects, the effect size of the difference is *large*, while *medium* for other 2 subjects. QUALITY-BASED MOSA is significantly better than TERMITE in 1 subjects, with a *medium* effect size.

**Summary for $RQ_2$**: TERMITE achieves a higher branch coverage compared to QUALITY-BASED MOSA for 72% of the classes we investigated.

## 9.4 Threats to Validity

**Internal Validity**. The main threat to internal validity of our study is that search-based test case generation is intrinsically random. To reduce the

effects of randomness, we run all the approaches 30 times. We used Wilcoxon Rank Sum test to check the significance of the differences and Vargha-Delaney $\hat{A}_{12}$ to measure the magnitude of such differences. Some studies compare test generation approaches using time as the stop criterion. Indeed, in a real usage scenario, developers specify the time they want to spend generating test cases. Despite this, we chose to use the number of generations of the genetic algorithm as the stop criterion. Using time would have not been fair, since the approaches that optimize quality aspects require extra computation compared to normal approaches (such as MOSA). This could have resulted in a difference in terms of branch coverage due to how the approaches use their time. Moreover, sub-optimal implementations of some algorithms or metrics may impact the whole efficiency of the approaches. Therefore, we prefer to analyze *effectiveness* and *efficiency* as two separate aspects: if we used time as the stop criterion, we would have mixed the two aspects.

**External Validity**. It is possible that the conclusion of our study does not generalize to all the Java classes. We focused on 60 randomly selected non-trivial Java classes. Similar work that compare test generation approaches use similar number of classes in their experiments [111, 108]. The conclusion of such a study may be only valid for non-trivial classes. However, we think that automated test case generation benefits developers the most for big and complex classes [112].

## 9.5 Final Remarks

In this chapter we presented TERMite, an approach aimed at automatically generating focused test cases. We conducted a study to evaluate the effectiveness and the efficiency of our tool. The results show that TERMite generates higher-quality test cases compared to QUALITY-BASED MOSA both in terms of coverage entropy and cohesion of the tests, at the expense of slightly more coupled tests. In addition, the severity of eager tests is significantly lower in the tests generated by TERMite. We also show that TERMite does not affect the branch coverage, but it slightly affects the time needed to generate tests. QUALITY-BASED MOSA, the baseline that tries to optimize test cases for their quality, takes longer to optimize test case.

In summary, the lessons learned from this chapter are the following:

- optimizing the coverage entropy of generated tests with TERMITE results in higher cohesion and lower prevalence of eager tests;
- computing cohesion and coupling of tests is a more time-consuming task compared to computing coverage entropy. We found that integrating the latter in test case generation (TERMITE) allows to achieve higher branch coverage compared to the former (QUALITY-BASED MOSA).

Finally, there is a main open issue: it is still unclear if test cases generated with TERMITE are significantly more understandable compared to the ones generated with QUALITY-BASED MOSA. Future work should be aimed at verifying this conjecture.

# CHAPTER 10

# Conclusion

Readability and understandability are key aspects of source code. Readable and understandable code is more maintainable and, therefore, it is desirable. While readability regards the *form*, *i.e.,* how code conveys information to the developer, understandability regards its *substance*, *i.e.,* the information itself that the developer has to process. Measuring such aspects is the first step needed in order to improve them. Previous work introduced code readability models able to automatically distinguish readable from unreadable code. However, such models are mainly focused on structural and visual features and they mostly ignore the textual aspects: since code is largely composed by text, it is reasonable to think that such aspects are important for automtic code readability assessment. As for code understandability, previous studies tried to measure it at system-level. However, the literature lacks approaches for automatically assessing the understandability of smaller components, such as classes or methods.

In this thesis we presented approaches and tools to automatically assess and improve code readability and understandability. We did this focusing both on source code and test code.

The contributions made to automatically assess and improve source code readability and understandability are the following:

- we defined textual features to improve the accuracy of readability models, and we used them to automatically assess code readability;
- we conducted a study in which we tried to correlate code readability with the presence of FindBugs warnings;
- we defined an approach that suggests renaming operations in order to improve the quality of the identifiers in the source code;
- we tried, for the first time, to automatically assess code understandability.

We also made the following contributions to automatically assess and improve the understandability of automatically generated tests:

- we introduced a new metric, namely Coverage Entropy, to estimate the focus and, thus, the understandability of test cases;
- we integrated Coverage Entropy and slicing in search-based test case generation to improve the focus of automatically generated tests.

The first lesson learned is that textual features are important to automatically assess code readability. Such features, indeed, help to improve by 6.2% the accuracy of readability models. Also, we showed that readability is strongly correlated with the presence of FindBugs warnings, confirming the findings by Buse and Weimer [21]. We performed a more in-depth analysis, and we showed that only some categories of FindBugs warnings show a high correlation (*e.g., Dodgy Code*), while others do not (*e.g., Malicious Code*).

The second lesson learned is that, at the moment, we cannot automatically assess source code understandability. Specifically, we showed that (i) single state-of-the-art metrics exhibit a low or not-existing correlation with understandability, and (ii) combining such metrics using machine learning allows to achieve interesting results, but not practically useful yet. We conclude that it is necessary to define new developer-related metrics.

The third lesson learned is that the focus of test cases (measured with Coverage Entropy) is an important quality attribute: eager tests are significantly more

entropic than non-eager tests. Test focus, however, exhibits only a moderate correlation with test eagerness: this shows that it captures a different dimension of test quality.

The fourth lesson learned is that it is possible to generate more focused test cases by (i) using Coverage Entropy as a secondary objective in MOSA and (ii) introducing slicing in the evolutionary process. Specifically, we showed that TERMite, our approach, achieves higher focus, higher coherence and lower test-eager severity compared to the state of the art, without sacrificing the achieved branch coverage.

# Appendices

# APPENDIX A

## Publications

J2 **S. Scalabrino**, G. Bavota, C. Vendome, M. Linares-Vàsquez, D. Poshyvanyk and R. Oliveto. *Automatically Assessing Code Understandability.* Transactions on Software Engineering, 2019. To Appear.

J1 **S. Scalabrino**, M. Linares-Vàsquez, D. Poshyvanyk and R. Oliveto. *A Comprehensive Model for Code Readability.* Journal of Software: Evolution and Process (Special Issue on ICPC). DOI 10.1002/smr.1958.

C4 G. Grano, **S. Scalabrino**, H. C. Gall and R. Oliveto. *An Empirical Investigation on the Readability of Manual and Generated Test Cases.* Proceedings of the International Conference on Program Comprehension (ICPC), 2018. To appear.

C3 B. Lin, **S. Scalabrino**, A. Mocci, R. Oliveto, G. Bavota and M. Lanza. *Investigating the Use of Code Analysis and NLP to Promote a Consistent Usage of Identifiers.* Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017, pages 81-90. ISBN 9781538632383.

C2 **S. Scalabrino**, G. Bavota, C. Vendome, M. Linares-Vàsquez, D. Poshyvanyk and R. Oliveto. *Automatically Assessing Code Understandability: How Far Are We?.* Proceedings of the International Conference on Automated Software Engineering (ASE), 2017, pages 417-427. ISBN 9781538626849.

C1 **S. Scalabrino**, M. Linares-Vàsquez, D. Poshyvanyk and R. Oliveto. *Improving Code Readability Models with Textual Features.* Proceedings of the International Conference on Program Comprehension (ICPC), 2016, pages 1-10. ISBN 9781509014286.

## A.1 Other Publications

J3 **S. Scalabrino**, G. Bavota, B. Russo, M. Di Penta, R. Oliveto. *Listening to the Crowd for the Release Planning of Mobile Apps.* Transactions on Software Engineering, 2017. DOI 10.1109/TSE.2017.2759112.

C9 **S. Scalabrino**, G. Grano, D. Di Nucci, M. Guerra, A. De Lucia, H. C. Gall and R. Oliveto. *OCELOT: a Search-Based Test-Data Generation Tool for C.* Proceedings of the International Conference on Automated Software Engineering (ASE) - Tool Demo track, 2018. To appear.

C8 F. Zampetti, **S. Scalabrino**, R. Oliveto, G. Canfora, M. Di Penta. *How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines.* Proceedings of the International Conference on Mining Software Repositories (MSR), 2017, pages 334-344. ISBN 9781538615447.

C7 **S. Scalabrino**. *On Software Odysseys and How to Prevent Them.* Proceedings of the the International Conference on Software Engineering, Student Research Competition (ICSE-SRC), 2017, pages 91-93. ISBN 9781538615898.

C6 Luca Ponzanelli, **S. Scalabrino**, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta and M. Lanza. *Supporting Software Developers with a Holistic Recommender System.* Proceedings of the International Conference on Software Engineering (ICSE), 2017, pages 94-105. ISBN 9781538638682.

C5 **S. Scalabrino**, G. Grano, D. Di Nucci, R. Oliveto and A. De Lucia. *Search-based Testing of Procedural Programs: Iterative Single-Target or Multi-Target Approach?.* Proceedings of the International Symposium on Search Based Software Engineering (SSBSE), 2016, pages 64-79. ISBN 9783319471068.

BC1 **S. Scalabrino**, S. Geremia, R. Pareschi, M. Bogetti and R. Oliveto. *Freelancing in the Economy 4.0 - How information technology can (really) help.* Book Chapter of "Social Media for Knowledge Management Applications in Modern Organizations" by F. Di Virgilio, pages 290-314. ISBN 9781522528975.

# Bibliography


[1] Collins american dictionary. http://www.collinsdictionary.com/dictionary/american/syllable. Accessed: 2019-02-24.

[2] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc. Can lexicon bad smells improve fault prediction? In *Proceedings of the Working Conference on Reverse Engineering*, pages 235–244. IEEE, 2012.

[3] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 235–241. IEEE, 2002.

[4] D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66. Springer, 1991.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[6] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Transactions on Software Engineering*, 28(10):970–983. IEEE, 2002.

[7] V. Arnaoudova, M. Di Penta, and G. Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158. Springer, 2016.

[8] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *Proceedings of the European*

*Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE, 2013.

[9] V. Arnaoudova, L. M. Eshkevari, R. Oliveto, Y. Guéhéneuc, and G. Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *Proceedings of the International Conference on Software Maintenance*, pages 1–5. IEEE, 2010.

[10] E. Avidan and D. G. Feitelson. Effects of variable names on comprehension: An empirical study. In *Proceedings of the International Conference on Program Comprehension*, pages 55–65. IEEE, 2017.

[11] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[12] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *Transactions on Software Engineering*, 28(1):4–17. IEEE, 2002.

[13] M. Bartsch and R. Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal*, 16(1):23–44. Springer, 2008.

[14] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094. Springer, 2015.

[15] K. Beck. *Implementation Patterns*. Addison Wesley, 2007.

[16] B. Beizer. *Software Testing Techniques*. Dreamtech Press, 2003.

[17] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 73–87. ACM, 2000.

[18] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To CamelCase or Under score. In *Proceedings of the International Conference on Program Comprehension*. IEEE, 2009.

[19] D. Binkley, H. Feild, D. J. Lawrie, and M. Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82(11):1793–1803. Elsevier, 2009.

[20] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32. Springer, 2001.

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

[22] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the Working Conference on Reverse Engineering*, pages 31–35. IEEE, 2009.

[23] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: an empirical study. In *Proceedings of the European*

*Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[24] J. c. Lin and K. c. Wu. A model for measuring software understandability. In *Proceedings of the International Conference on Computer and Information Technology*, pages 192–192. IEEE, 2006.

[25] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 55–66. ACM, 2014.

[26] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 33–48. Springer, 2017.

[27] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta. How changes affect software entropy: An empirical study. *Empirical Software Engineering*, 19(1):1–38. Springer, 2014.

[28] A. Capiluppi, M. Morisio, and P. Lago. Evolution of understandability in OSS projects. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 58–66. IEEE, 2004.

[29] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance*, pages 97–107. IEEE, 2000.

[30] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357. AAAI, 2002.

[31] C. Chen, R. Alfayez, K. Srisopha, L. Shi, and B. Boehm. Evaluating human-assessed software maintainability metrics. In *Proceedings of the National Software Application Conference on Software Engineering and Methodology for Emerging Domains*, pages 120–132. Springer, 2016.

[32] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 2nd edition, 1988.

[33] J. Cohen. The earth is round (p<.05). *American Psychologist*, 49(12):997–1003. APA, 1994.

[34] M. L. Collard, M. J. Decker, and J. I. Maletic. SRCML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the International Conference on Software Maintenance*, pages 516–519. IEEE, 2013.

[35] W. Conover. *Practical Nonparametric Statistics*. Wiley series in probability and statistics. Wiley, 3rd edition, 1999.

[36] A. Corazza, S. Di Martino, and V. Maggio. LINSEN: An approach to split identifiers and expand abbreviations with linear complexity. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012.

[37] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 275–284. IEEE, 2014.

[38] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 107–118. ACM, 2015.

[39] E. Daka, J. M. Rojas, and G. Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 57–67. ACM, 2017.

[40] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Labeling source code with information retrieval methods: An empirical study. *Empirical Software Engineering*, 19(5):1383–1420. Springer, 2014.

[41] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282. Springer, 2006.

[42] S. D.J. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & All, 4th edition, 2007.

[43] J. Dorn. A general software readability model. Master's thesis, University of Virginia, Department of Computer Science, 2012.

[44] J. L. Elshoff and M. Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8):512–521. ACM, 1982.

[45] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 71–80. IEEE, 2009.

[46] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23. IEEE, 2000.

[47] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the International Conference on Knowledge Discovery and Data mining*, pages 226–231. AAAI, 1996.

[48] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *Proceedings of the International Conference on Program Comprehension*, pages 73–82. IEEE, 2012.

[49] R. Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221. APA, 1948.

[50] R. F. Flesch. *How to Write Plain English: A Book for Lawyers and Consumers*. Harpercollins, 1979.

[51] B. Fluri, M. Wursch, and H. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the Working Conference on Reverse Engineering*, pages 70–79. IEEE, 2007.

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

[53] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[54] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 362–369. IEEE, 2013.

[55] G. Fraser and A. Arcuri. Whole test suite generation. *Transactions on Software Engineering*, 39(2):276–291. IEEE, 2013.

[56] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *Transactions on Software Engineering and Methodology*, 24(2):8:1–8:42. ACM, 2014.

[57] G. Fraser and A. Arcuri. Evosuite at the SBST 2016 Tool Competition. In *Proceedings of the International Workshop on Search-Based Software Testing*, pages 33–36. ACM, 2016.

[58] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the conference on Object-Oriented Programming Systems and Applications*, pages 57–76. ACM, 2007.

[59] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[60] D. E. Goldberg. *Genetic Algorithms*. Pearson Education India, 2006.

[61] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the International Conference on Program Comprehension*, pages 348–351. ACM, 2018.

[62] R. J. Grissom and J. J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Earlbaum Associates, 2nd edition, 2005.

[63] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *Proceedings of the Working Conference on Reverse Engineering*, pages 103–112. IEEE, 2012.

[64] M. Gütlein, E. Frank, M. Hall, and A. Karwath. Large-scale attribute selection using wrappers. In *Proceedings of the Symposium on Computational Intelligence and Data Mining*, pages 332–339. IEEE, 2009.

[65] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Proceedings of the International Conference on Program Comprehension*, pages 113–122. IEEE, 2008.

[66] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18. ACM, 2009.

[67] L. L. Harlow, S. A. Mulaik, and J. H. Steiger. *What If There Were No Significance Tests?* Psychology Press, 1997.

[68] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the International Conference on Software Engineering*, pages 78–88. IEEE, 2009.

[69] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847. IEEE, 2012.

[70] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal of Statistics*, 6:65–70. 1979.

[71] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304. Elsevier, 2012.

[72] ISO/IEC. ISO/IEC 9126 software engineering — product quality — part 1: Quality model.

[73] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 338–345. ACM, 1995.

[74] N. Kasto and J. Whalley. Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the Australasian Computing Education Conference*, pages 59–65. Australian Computer Society, 2013.

[75] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):81. Oxford University Press, 1938.

[76] K. Kira and L. A. Rendell. A practical approach to feature selection. In *Proceedings of the International Workshop on Machine Learning*, pages 249–256. Elsevier, 1992.

[77] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson Education India, 2016.

[78] I. Kononenko, E. Šimec, and M. Robnik-Šikonja. Overcoming the myopia of inductive learning algorithms with RELIEFF. *Applied Intelligence*, 7(1):39–55. Springer, 1997.

[79] J. Lawrance, R. Bellamy, and M. Burnett. Scents in programs: Does information foraging theory apply to program maintenance? In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 15–22. IEEE, 2007.

[80] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *Transactions on Software Engineering*, 39(2):197–215. IEEE, 2013.

[81] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart. Reactive information foraging for evolving goals. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 25–34. ACM, 2010.

[82] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the International Conference on Software Maintenance*, pages 113–122. IEEE, 2011.

[83] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE, 2006.

[84] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 213–222. IEEE, 2007.

[85] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proceedings of the International Conference on Program Comprehension*, pages 3–12. IEEE, 2006.

[86] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Journal Innovations in Systems and Software Engineering*, 3(4):303–318. Springer, 2007.

[87] S. Le Cessie and J. C. Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201. Wiley, 1992.

[88] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. Kraft. Automatically documenting unit test cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2016.

[89] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza. On the uniqueness of code redundancies. In *Proceedings of the International Conference on Program Comprehension*. IEEE, 2017.

[90] D. Lin, C.-P. Bezemer, and A. E. Hassan. An empirical study of early access games on the steam platform. *Empirical Software Engineering*, 23(2):771–799. Springer, 2018.

[91] J.-C. Lin and K.-C. Wu. Evaluation of software understandability based on fuzzy matrix. In *Proceedings of the International Conference on Fuzzy Systems*, pages 887–892. IEEE, 2008.

[92] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. ChangeScribe: A tool for automatically generating commit messages. In *Proceedings of the International Conference on Software Engineering*, pages 709–712. IEEE, 2015.

[93] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. How do developers document database usages in source code? In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2015.

[94] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2016.

[95] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *Transactions on Software Engineering*, 34(2):287–300. IEEE, 2008.

[96] R. C. Martin. *Clean Code: A Handbook of Agile Sofware Craftsmanship*. Prentice Hall, 2009.

[97] T. J. McCabe. A complexity measure. *Transactions on Software Engineering*, (4):308–320. IEEE, 1976.

[98] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156. Wiley, 2004.

[99] P. McMinn. Search-based software testing: Past, present and future. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[100] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[101] G. A. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38(11):39–41. ACM, 1995.

[102] R. Minelli, A. Mocci, and M. Lanza. I know what you did last summer – an investigation of how developers spend their time. In *Proceedings of the International Conference on Program Comprehension*, pages 25–35. IEEE, 2015.

[103] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Automatic test case evolution. *Software Testing, Verification and Reliability*, 24(5):386–411. Wiley, 2014.

[104] S. Misra and I. Akman. Comparative study of cognitive complexity measures. In *Proceedings of the International Symposium on Computer and Information Sciences*, pages 1–4. IEEE, 2008.

[105] A. Oram and G. Wilson. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly, 2007.

[106] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the proceedings of the Conference on Object-Oriented Programming Systems and Applications*, pages 815–816. ACM, 2007.

[107] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the International Workshop on Search-Based Software Testing*, pages 5–14. ACM, 2016.

[108] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 130–141. ACM, 2016.

[109] F. Palomba, A. Zaidman, and A. De Lucia. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2018.

[110] F. Palomba, M. Zanoni, F. Arcelli Fontana, A. De Lucia, and R. Oliveto. Toward a amell-aware bug prediction model. *Transactions on Software Engineering*. IEEE, 2017.

[111] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[112] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *Transactions on Software Engineering*, 44(2):122–158. IEEE, 2018.

[113] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the International Conference on Software Engineering*, pages 547–558. ACM, 2016.

[114] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[115] J. Platt. *Fast Training of Support Vector Machines Using Sequential Minimal Optimization*, pages 185–208. MIT Press, 1999.

[116] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137. Emerald Publishing, 1980.

[117] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the International Conference on Software Maintenance*, pages 469–478. IEEE, 2006.

[118] D. Poshyvanyk and D. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the International Conference on Program Comprehension*, pages 37–48. IEEE, 2007.

[119] D. Posnett, A. Hindle, and P. T. Devanbu. A simpler model of software readability. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 73–82. IEEE, 2011.

[120] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69. IEEE, 2004.

[121] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the International Conference on Software Engineering*, pages 255–265. IEEE, 2012.

[122] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893. Springer, 2017.

[123] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194. Springer, 1998.

[124] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: How far are we? In *Proceedings of the International Conference on Automated Software Engineering*, pages 417–427. IEEE, 2017.

[125] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, and A. De Lucia. Search-based testing of procedural programs: Iterative single-target or multi-target approach? In *Proceedings of the International Symposium on Search-Based Software Engineering*, pages 64–79. Springer, 2016.

[126] D. Schreck, V. Dallmeier, and T. Zimmermann. How documentation evolves over time. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 4–10. ACM, 2007.

[127] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30(5):263–272. ACM, 2005.

[128] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374. ACM, 2015.

[129] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 250–261. IEEE, 2018.

[130] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55. ACM, 2001.

[131] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the SMO algorithm for SVM regression. *Transactions on Neural Networks*, 11(5):1188–1193. IEEE, 2000.

[132] K. Shima, Y. Takemura, and K. Matsumoto. An approach to experimental evaluation of software understandability. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 48–55. IEEE, 2002.

[133] J. Siegmund and J. Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, 20(4):1159–1192. Springer, 2015.

[134] D. Spinellis. *Code Quality: The Open Source Perspective*. Adobe Press, 2006.

[135] D. Srinivasulu, A. Sridhar, and D. P. Mohapatra. Evaluation of software understandability using rough sets. In *Proceedings of the International Conference on Advanced Computing, Networking, and Informatics*, pages 939–946. Springer, 2014.

[136] M. A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the International Workshop on Program Comprehension*, pages 181–191. IEEE, 2005.

[137] M. A. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the Working Conference on Reverse Engineering*, pages 12–21. IEEE, 1997.

[138] A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: An experiential study. *Journal of Program Languages*, 4(3):143–167. 1996.

[139] T. Tenny. Program readability: Procedures versus comments. *Transactions on Software Engineering*, 14(9):1271–1279. IEEE, 1988.

[140] A. Thies and C. Roth. Recommending rename refactorings. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 1–5. ACM, 2010.

[141] M. Thongmak and P. Muenchaisri. *Measuring Understandability of Aspect-Oriented Code*, pages 43–54. Springer, 2011.

[142] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128. ACM, 2004.

[143] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu. "Automatically assessing code understandability" reanalyzed: Combined metrics matter. In *Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2018.

[144] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[145] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 33–42. IEEE, 2010.

[146] A. Van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering*. Addison-Wesley, 2001.

[147] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, 25(2):101–132. SAGE Publishing, 2000.

[148] E. J. Weyuker. Evaluating software complexity measures. *Transactions on Software Engineering*, 14(9):1357–1365. IEEE, 1988.

[149] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120. Wiley, 2012.