On Automatically Assessing Code Understandability

Simone Scalabrino, *Student Member, IEEE,* Gabriele Bavota, *Member, IEEE,* Christopher Vendome, *Member, IEEE,* Mario Linares-Vásquez, *Member, IEEE,* Denys Poshyvanyk, *Member, IEEE,* Rocco Oliveto, *Member, IEEE*

Abstract—Understanding software is an inherent requirement for many maintenance and evolution tasks. Without a thorough understanding of the code, developers would not be able to fix bugs or add new features timely. Measuring code *understandability* might be useful to guide developers in writing better code, and could also help in estimating the effort required to modify code components. Unfortunately, there are no metrics designed to assess the understandability of code snippets.
In this work, we perform an extensive evaluation of 121 existing and new code-related, documentation-related, and developer-related metrics. We try to (i) correlate each metric with understandability and (ii) build models combining metrics to assess understandability. To do this, we use 444 human evaluations from 63 developers and we got a bold *negative result*: none of the 121 experimented metrics is able to capture code understandability, not even the ones assumed to assess quality attributes apparently related, such as code readability and complexity. While we observed some improvements combining metrics in models, their effectiveness is still far from making them suitable in practice. Finally, we conducted interviews with five professional developers to understand the factors that influence their ability to understand code snippets, aiming at identifying possible new metrics.

Index Terms—Software metrics, Code understandability, Empirical study, Negative result

1 Introduction

Developers spend most of their time (∼70%) understanding code [1]. While such a finding might look surprising, it only highlights the pivotal role that code understanding plays in any code-related activity, such as feature implementation or bug fixing. While the importance of code understandability is undisputed for maintenance-related activities [2], [3], [4], [5], [6], [7], 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 is focused on the cognitive process adopted when understanding code [8], [9]. 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. Similarly, researchers have designed models and proposed metrics to evaluate the readability of code by correlating them with the readability *perceived* by developers [10], [11], [12], [13], [14], [15]. This typically means that the developers participating in the evaluation are required to read a fragment of code and rate its readability on a given scale (*e.g.,* from 1: low to 5: high readability). However, the perceived readability is something different from the actual understandability of the code; a developer could find a piece of code readable while still experiencing difficulties in understanding it, for example due to unknown APIs used. Let us consider the code fragment listed below:

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

A developer, as well as all readability metrics proposed in the literature [10], [11], [12], [13], [14], [15], would consider this snippet of code as readable, since it is concise and utilizes 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.

Having a metric to estimate 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 metric 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 code understandability as a fitness function; or (iii) assess the quality of code changes during code reviews.

- *S. Scalabrino and R. Oliveto are with the University of Molise, Italy. E-mail: {simone.scalabrino, rocco.oliveto}@unimol.it*
- *G. Bavota is with the Università della Svizzera italiana (USI), Switzerland. E-mail: gabriele.bavota@usi.ch*
- *C. Vendome is with Miami University, USA. E-mail: vendomcg@miamioh.edu*
- *D. Poshyvanyk is with the The College of William & Mary, USA. E-mail: denys@cs.wm.edu*
- *M. Linares-Vásquez is with the Universidad de los Andes, Colombia. E-mail: m.linaresv@uniandes.edu.co*

While we have possible proxies for code understandability, such as code complexity and readability, we do not (i) know whether these proxies actually correlate with the effort required to understand a piece of code, and (ii) have a metric able to provide an estimation for code understandability. Previous attempts to define a code understandability model [4], [16], [6] have not been empirically evaluated, consider understandability as a factor in a quality model [2], [17], or measure understandability at the level of a whole software system [18].

In this work, 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 [14], [15] and code indentation [10]. As the aforementioned example illustrates, 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. Then, we exploited *classification* and *regression* models utilizing combinations of metrics to predict proxies of code understandability. 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.

The main contributions of this work as compared to our ASE'17 paper [19] it extends, are the following:

- We increased the size of our dataset from 324 evaluations by 46 developers to 444 evaluations by 63 developers to increase the confidence in our findings;
- We used combinations of feature models to predict the independent variables that we define as proxies of understandability;
- We conducted structured interviews with five experienced developers to understand what makes code understandable/not understandable to them; we used such information to propose possible future directions for code understandability assessment.

Paper structure. Section 2 provides background information about code characteristics possibly related to its understandability. Section 3 describes the 121 metrics used in our empirical study, while Section 4 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 5 and 6, respectively. Section 7 discusses the threats that could affect our findings, while Section 8 concludes the paper and highlights future research directions.

2 Background & Related Work

In this section, we describe metrics and models that have been proposed to measure code readability. All these metrics have been included in our study. Afterwards, we briefly describe related work presenting metrics for measuring understandability (at system level) as a single quality attribute and as part of a quality model.

2.1 Code Readability

Identifiers and comments play a crucial role in program comprehension, since developers express domain knowledge through the names that they assign to the code entities at different levels (*i.e.,* packages, classes, methods, variables) [20], [21], [22], [23], [24]. Thus, source code lexicon impacts the psychological complexity of a program [14], [25]. Another aspect that also contributes to the readability (and potentially understandability) of source code are structural aspects such as indentation, code entity length [10], [11], and visual/spatial aspects such as syntax highlighting, code formatting, and visual areas covered by code entities [12].

All the aforementioned aspects have been used as features in binary classifiers able to predict the readability of code snippets [10], [11], [12], [14], [15]. In the model by Buse and Weimer [10], source code structural aspects (*e.g.,* number of branches, loops, operators, blank lines, comments) represent the underlying features in the classifier. The model was trained a-priori on 100 small snippets; the snippets were tagged manually as readable or non-readable by 120 human annotators. The reported results provide evidence that readability can be estimated automatically.

Posnett *et al.* [11] proposed a model based on a reduced set of the features introduced by Buse and Weimer. An empirical evaluation conducted on the same dataset used by Buse and Weimer [10] indicated that the model by Posnett *et al.* is more accurate than the one by Buse and Weimer.

Dorn introduced a readability model, which relies on a larger set of features grouped in four categories: *visual*, *spatial*, *alignment*, and *linguistic* [12]. This larger set of features highlights the fact that structural aspects are not the only ones that should be considered for code readability; aspects representing and modeling how the code is read on the screen, such as syntax highlighting, variable naming standards, and operators alignment, should be also considered. Dorn trained and validated the model on a new dataset, including programs in Java, Python, and CUDA, for a total of 360 snippets. Such a model achieved a higher accuracy as compared to the one by Buse and Weimer.

Scalabrino *et al.* [14], [15] proposed and evaluated a set of features based entirely on source code lexicon analysis (*e.g.,* consistency between source code and comments, specificity of the identifiers, textual coherence, comments readability). The model was evaluated on the two datasets previously introduced by Buse and Weimer [10] and Dorn [12] and on a new dataset, composed of 200 Java snippets, manually evaluated by nine developers. The results indicated that combining the features (*i.e.,* structural+textual) improves the accuracy of code readability models.

2.2 Software/Code Understandability

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 [9], [4]. There have been several metrics designed to evaluate software understandability by focusing on complexity as well as source-level metrics.

Lin *et al.* [4] 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 [26] 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 [27]. 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.* [3] considered aspect-oriented software dependence graphs to assess understandability of aspect-oriented software, while Srinivasulu *et al.* [6] 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.* [18] 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.* [2] 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.* [7] 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 [28] 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.

It is worth noting that we **do not** consider the above discussed understandability metrics [4], [26], [3], [6], [18], [17], [28] in our study 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. Instead, we included in our study the metrics used by Kasto and Whalley [29] to study the understandability of code snippets in an educational context. Specifically, Kasto and Whalley analyzed the performance of 93 students in their final examination for the Java programming course and they correlated their results with five metrics.

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 [30]. 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 [5]. 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*" [31]; and (ii) the theories, challenges, and models for program understanding at cognitive levels [9], [8]. 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 [4], [16].

More recently, Trockman *et al.* [32] analyzed the dataset we that released as part of our ASE'17 work [19] that this paper 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 [32] to classify evaluations as *understandable* or *not understandable*. In this paper, we also experiment with combinations of metrics to predict the understandability of a given code snippet. However, we (i) rely on our new and larger dataset featuring 22% more data points as compared to the one we previously released; and (ii) we experiment with both classification and *regression* techniques, and use several different proxies for the dependent variable (*i.e.,* code understandability), as compared to the single one used by Trockman *et al.* [32].

3 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 [8]. 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 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 [10], as the average and as the maximum length of the identifiers used in the snippet). We report in boldface the new metrics introduced in the our previous work [19]. It is worth noting that the number of metrics shown in Table 1 does not sum up to the 121 metrics considered in our study.

TABLE 1: Candidate predictors for code understandability.

	Metric	Non-aggregated	Min	Avg	Max	DFT	Visual	Area
Code	Cyclomatic comp.	[29]						
	#nested blocks			[29]				
	#parameters	[29]						
	#statements	[29]						
	#assignments			[10]		[12]		
	#blank lines			[10]				
	#characters				[10]			
	#commas			[10]		[12]		
	#comments			[10]		[12]	[12]	[12]
	#comparisons			[10]		[12]		
	#conditionals			[10]		[12]		
	#identifiers	[29]		[10]	[10]	[12]	[12]	[12]
	#keywords			[10]	[10]	[12]	[12]	[12]
	#literals						[12]	[12]
	#loops			[10]		[12]		
	#numbers			[10]	[10]	[12]	[12]	[12]
	#operators			[10]		[12]	[12]	[12]
	#parenthesis			[10]		[12]		
	#periods			[10]		[12]		
	#spaces			[10]		[12]		
	#strings						[12]	[12]
	#words				[10]			
	Indentation length			[10]	[10]	[12]		
	Identifiers length			[10]	[10]			
	Line length			[10]	[10]	[12]		
	#aligned blocks	[12]						
	Ext. of alig. blocks	[12]						
	Entropy	[11]						
	LOC	[11]						
	Volume	[11]						
	NMI		[14]	[14]	[14]			
	NM			[14]	[14]			
	ITID		[14]	[14][12]				
	TC		[14]	[14]	[14]			
	Readability	[14]						
	IMSQ		[19]	[19]	[19]			
Docs	CR	[14]						
	CIC			[14]	[14]			
	CIC$_{syn}$			[14]	[14]			
	MIDQ		[19]	[19]	[19]			
	AEDQ		[19]	[19]	[19]			
Devs	**EAP**		[19]	[19]	[19]			
	PE$_{gen}$	[19]						
	PE$_{spec}$	[19]						

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 [12], thus resulting in two different forms of aggregation. The complete list of metrics is available in our replication package [33]. In the following subsections, we discuss the considered metrics grouped by their type.

3.1 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 [29]: *cyclomatic complexity* [34], 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.

We also include in this category all the code-related readability metrics defined in the literature [10], [11], [12], [14]. These include the ones by Buse and Weimer [10], 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.* [11] in the context of readability prediction. Dorn [12] presents a variation to the basic metrics introduced by Buse and Weimer [10], 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). These variants are not reported in Table 1 due to space constraints, but are considered in our study and listed in our replication package [33].

Scalabrino *et al.* [14] define *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, as defined by Scalabrino *et al.* [14], as a separate metric, combining together the previously listed metrics. We followed the steps described by Scalabrino *et al.* to define the readability model by using a logistic classifier that we train on the 420 Java snippets available in the literature [10], [12], [14].

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` $\rightarrow$ `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, 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).

3.2 Documentation-Related Metrics

Scalabrino *et al.* [14] 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 [35]; *Comments and Identifiers Consistency* (CIC) measures the consistency between comments and code; and CIC_{syn}, a variant of CIC, which takes synonyms into account.

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.* [36]:

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

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 [35]. 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.

3.3 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`).

4 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, as previously shown [19], 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 3, 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).

5 Empirical Study Design

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. 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 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*.

TABLE 2: Systems used in our study

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

5.1 Data Collection

The *context* of the study consists of 50 Java/Android methods extracted from ten popular systems listed in Table 2 (five methods from each system). We first extracted all the methods having 50 ± 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 ± 20 ELOCs) was driven by the decision of excluding methods that are too trivial or too complex to understand.

Afterwards, we computed all the metrics described in Section 3 for the selected methods[1]. Then, we used a greedy algorithm for center selection [37] 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

1. Excluding the "Developer programming experience" and the "Developer Java experience"

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 understanding each method was roughly the same). In total, we collected 444 evaluations across the 50 methods ($\sim$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. Specifically, one of the questions was about the identifiers used in the method (*e.g.,* what does "CMS" mean?), one was about the purpose of a call to an internal method and another one about the purpose of the use of an external component (*e.g.,* JDBC APIs). The provided answers were stored for future analysis. An example of verification question is: "What does the invoked method `X` do?". To answer such a question, the participant must understand all of the consequences of invoking a specific external method.

5.2 Analysis Method

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

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 [38], 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.,* NMI$_{min}$ and 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 τ) [39] to determine whether there are pairs exhibiting a strong correlation. We adopted the Kendall's τ, 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 [40] 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 3 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 [41], (ii) Bayes Networks [42], (iii) Support Vector Machines (SMO algorithm [43]), (iv) Neural Networks (Multilayer Perceptron), (v) k-Nearest-Neighbors [44], and (vi) Random Forest [45]. As a first step, we use 10% of each dataset to tune the hyperparameters for some machine learning techniques known to be sensitive to parameter tuning and we remove 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: We define discrete values for each parameters (at most 10 values) and 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. We do not tune hyperparameters 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 [46] 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.

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 [47]), (iii) Neural Networks (Multilayer Perceptron), (iv) k-Nearest-Neighbors[44] and (v) Random Forest[45]. 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 minimizes 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 4. The developers provided consent to include their names in this paper. 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

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.

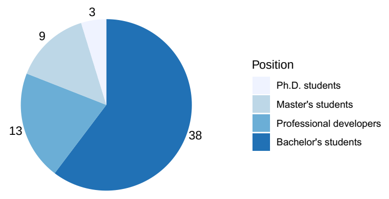


Fig. 1: Participants to the study

the snippet. Table 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 understandable/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.

5.3 Replication Package

The data used in our study is publicly available [33]. We provide the research community with our dataset reporting the (perceived and actual) understandability achieved by the 63 participants, hoping that the availability of such a dataset will help foster research on the definition of a metric to automatically assess code understandability.

6 Empirical Study Results

In this section, we present the results of our empirical study. Fig. 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.

6.1 RQ_1: Correlation With Single Metrics

When evaluating the metrics on a larger number of evaluations and by considering more proxies for code understandability, we still observe the negative results that were shown in our previous study [19]: **Very few metrics 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 [10] 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 [10]. 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 the perceived understandability; *average identifiers' length* (-0.12), *i.e.,* shorter identifiers slightly help perceiving the code as more understandable.

While in our previous study we showed that ten of the metrics have a slight correlation with *TNPU*, replicating this work on a larger number of evaluations reduced the number to just a single metric, *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 for RQ_1. None of the metrics we considered achieve a medium/strong correlation with any of the proxies of code understandability we defined.

6.2 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*.

TABLE 3: Methods used during the interviews with developers

System	Class	Method	Type	URL
OpenCMS	CmsHoverbarContextMenuButton	createContextMenu	The most understandable ($\overline{TAU} = 0.74$)	https://git.io/vpHzL
Phoenix	MetaDataEndpointImpl	doDropSchema	The least understandable ($\overline{TAU} = 0.06$)	https://git.io/vpHzm
Hibernate	TimesTenDialect	TimesTenDialect	The most subjective ($sd(TAU) = 0.46$)	https://git.io/vpHzs
MyExpenses	LazyFontSelector	processChar	The most deceptive ($\overline{TNPU} = 391.0$, $\#BD_{50\%} = 2$)	https://git.io/vpHzZ

TABLE 4: Position and experience of the interviewed developers

Name	Position	Programming 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 5: Classification results of *PBU*, $ABU_{50\%}$, and $BD_{50\%}$

Classifier	***PBU***		$\boldsymbol{ABU_{50\%}}$		$\boldsymbol{BD_{50\%}}$	
	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 6: Regression results of *TNPU*, *AU*, and *TAU*

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**

6.2.1 *Classification*

Table 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 ∼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.

6.2.2 *Regression*

In Table 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 for 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.

6.3 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. For space limitations, we report the complete code snippets in our replication package [33].

6.3.1 *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.

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

Fig. 2: Actual code vs aligned code (first snippet)

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 to understand the method. He would have aligned all the actual parameters of the constructors in the list (Fig. 2). 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.

6.3.2 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.

6.3.3 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.

6.3.4 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 provides a summary of the factors that the developers mentioned at least once during the interview. 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.

TABLE 7: Factors mentioned by the participants

Factor	MM	SG	GG	SD	CB
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			×	×	

Summary for RQ_3. The interviewed developers agree on a subset of aspects that are desirable (*e.g.*, good identifiers) 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.

6.4 Discussion

In our previous study [19], we showed that no single metric has a non-weak correlation with any proxy of understandability. Increasing the size of our dataset further reduced the correlations. Most noticeably, the number of metrics with a correlation with *TNPU* higher than |0.1| were ten in our previous study, and this number has been reduced to just one with the larger dataset.

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.* [32] 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 can confirm the negative result of our previous study: **the metrics we investigated are not enough to capture code understandability**. However, as previously hinted by Trockman *et al.* [32], 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 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 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 [14]). 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 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

TABLE 8: Mean variance of the proxies among snippets and developers (lower values imply more similar scores).

	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**

of V_s for the different categories of participants. The table shows that V_s is slightly higher than 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 $BD_{(\%}$ 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 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. 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 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 [48] 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. We increased the number of observation from our previous study [19] by 22% to improve the generalizability of our findings. 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 a 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.

8 Conclusion and Future Work

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 is 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 models from the deep learning comunity should be used as part of future work, given their power to abstract and model complex relationships between input data and several abstraction layers, similar to the cognitive processes in the humans brain; as suggested by the interviews, developers exhibited some comonalities but also variabilities in the understandability process that might not be captured by classic models, thus, shallow models are not the best choice to predice/measure understandability.

Our study lays the foundations for future research on new metrics actually able to capture facets of code understandability. In our opinion, the state-of-the-art lacks *developer-related* metrics. Therefore, we think that future research should be aimed at defining more of such metrics to properly capture code understandability. To enable the research community to investigate this direction further, we publicly release our new dataset [33].

Acknowledgments

Bavota was supported in part by the SNF project JITRA, No. 172479.

References

[1] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer - an investigation of how developers spend their time," in *23rd IEEE International Conference on Program Comprehension*, 2015, pp. 25–35.

[2] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Annual Reliability and Maintainability Symposium.*, 2002, pp. 235–241.

[3] M. Thongmak and P. Muenchaisri, *Measuring Understandability of Aspect-Oriented Code*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 43–54.

[4] J. c. Lin and K. c. Wu, "A model for measuring software understandability," in *6th IEEE International Conference on Computer and Information Technology*, 2006, pp. 192–192.

[5] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 255–265.

[6] D. Srinivasulu, A. Sridhar, and D. P. Mohapatra, *Evaluation of Software Understandability Using Rough Sets*. New Delhi: Springer India, 2014, pp. 939–946.

[7] C. Chen, R. Alfayez, K. Srisopha, L. Shi, and B. Boehm, *Evaluating Human-Assessed Software Maintainability Metrics*. Singapore: Springer Singapore, 2016, pp. 120–132.

[8] M. A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension*, 2005, pp. 181–191.

[9] M. A. D. Storey, K. Wong, and H. A. Muller, "How do program understanding tools affect how programmers understand programs?" in *4th Working Conference on Reverse Engineering*, 1997, pp. 12–21.

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

[11] D. Posnett, A. Hindle, and P. T. Devanbu, "A simpler model of software readability." in *8th Working Conference on Mining Software Repositories*, 2011, pp. 73–82.

[12] J. Dorn, "A general software readability model," Master's thesis, University of Virginia, Department of Computer Science, https://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf, 2012.

[13] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015, pp. 107–118.

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

[15] ——, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, To Appear.

[16] J.-C. Lin and K.-C. Wu, "Evaluation of software understandability based on fuzzy matrix," in *IEEE International Conference on Fuzzy Systems*, 2008, pp. 887–892.

[17] M. Bartsch and R. Harrison, "An exploratory study of the effect of aspect-oriented programming on maintainability," *Software Quality Journal*, vol. 16, no. 1, pp. 23–44, 2008.

[18] A. Capiluppi, M. Morisio, and P. Lago, "Evolution of understandability in oss projects," in *8th European Conference on Software Maintenance and Reengineering*, 2004, pp. 58–66.

[19] 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 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 417–427.

[20] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory." *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[21] ——, "What's in a name? a study of identifiers," in *14th International Conference on Program Comprehension*, 2006.

[22] B. Caprile and P. Tonella, "Restructuring program identifier names," in *International Conference on Software Maintenance*, 2000, pp. 97–107.

[23] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *6th International Working Conference on Source Code Analysis and Manipulation*, 2006, pp. 139–148.

[24] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *6th Working Conference on Mining Software Repositories*, 2009.

[25] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2015.

[26] S. Misra and I. Akman, "Comparative study of cognitive complexity measures," in *23rd International Symposium on Computer and Information Sciences*, Oct 2008, pp. 1–4.

[27] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[28] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, Jan 2002.

[29] N. Kasto and J. Whalley, "Measuring the difficulty of code comprehension tasks using software metrics," in *15th Australasian Computing Education Conference*. Australian Computer Society, Inc., 2013, pp. 59–65.

[30] K. Shima, Y. Takemura, and K. Matsumoto, "An approach to experimental evaluation of software understandability," in *International Symposium on Empirical Software Engineering*, 2002, pp. 48–55.

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

[32] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, ""automatically assessing code understandability" reanalyzed: Combined metrics matter," 2018.

[33] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Replication package." https://dibt.unimol.it/report/understandability-tse.

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

[35] R. F. Flesch, *How to write plain English: A book for lawyers and consumers*. Harpercollins, 1979.

[36] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in *Ninth International Workshop on Principles of Software Evolution*. ACM, 2007, pp. 4–10.

[37] J. Kleinberg and É. Tardos, *Algorithm design*. Pearson Education India.

[38] J. W. Tukey, "Exploratory data analysis," 1977.

[39] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1-2, p. 81, 1938.

[40] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

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

[42] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1995, pp. 338–345.

[43] J. Platt, "Fast training of support vector machines using sequential minimal optimization. advances in kernel methodsâĂŤsupport vector learning (pp. 185–208)," *AJ, MIT Press, Cambridge, MA*, 1999.

[44] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine learning*, vol. 6, no. 1, pp. 37–66, 1991.

[45] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[46] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[47] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to the smo algorithm for svm regression," *IEEE transactions on neural networks*, vol. 11, no. 5, pp. 1188–1193, 2000.

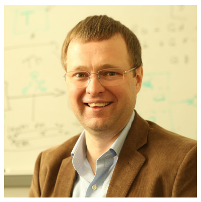
[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

Simone Scalabrino is a Ph.D. student at University of Molise, Italy. He received his Master's Degree in Computer Science from the University of Salerno in 2015, defending a thesis on Search Based Software Testing, advised by Andrea De Lucia. He received the Bachelor's Degree from the University of Molise in 2013, defending a thesis about source code readability, advised by Rocco Oliveto and Denys Poshyvanyk. His research interests include software quality, testing and security. He received two ACM SIGSOFT Distinguished Paper awards at ICPC 2016 and ASE 2017. He served as Local Arrangement Co-Chair for SANER 2018.

Gabriele Bavota is an Assistant Professor at the Università della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is the author of over 100 papers appeared in international journals, conferences and workshops. He received four ACM SIGSOFT Distinguished Paper awards at ASE 2013, ESEC-FSE 2015, ICSE 2015, and ASE 2017, an IEEE TCSE Distinguished Paper Award at ICSME 2018, the best paper award at SCAM 2012, and three distinguished reviewer awards at WCRE 2012, SANER 2015, and MSR 2015. He is the recipient of the 2018 ACM Sigsoft Early Career Researcher Award. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, and others.

Christopher Vendome is an Assistant Professor at Miami University in Ohio. He received a B.S. in Computer Science from Emory University in 2012 and he received his M.S. in Computer Science from The College of William & Mary in 2014. He received his PhD degree in Computer Science from the College of William & Mary in 2018. His main research areas are software maintenance and evolution, mining software repositories, program comprehension, software provenance, and software licensing. His has received an ACM Distinguished Paper Award at ASE 2017. He is a member of ACM, IEEE, and IEEE Computer Society. More information at: http://www.cs.wm.edu/~cvendome/.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. His research interests include software evolution and maintenance, software architecture, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, and mobile development.



Denys Poshyvanyk is the Class of 1953 Term Distinguished Associate Professor of Computer Science at the College of William and Mary in Virginia. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He served as a program co-chair for ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves on the editorial board of IEEE Transactions on Software Engineering (TSE), Empirical Software Engineering Journal (EMSE, Springer) and Journal of Software: Evolution and Process (JSEP, Wiley). His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16 and ASE'17. He also received the Most Influential Paper Awards at ICSME'16 and ICPC'17. He is a recipient of the NSF CAREER award (2013). He is a member of the IEEE and ACM. More information available at: http://www.cs.wm.edu/~denys/.

Rocco Oliveto is Associate Professor in the Department of Bioscience and Territory at University of Molise (Italy). He is the Chair of the Computer Science program and the Director of the Laboratory of Computer Science and Scientific Computation of the University of Molise. He received the PhD in Computer Science from University of Salerno (Italy) in 2008. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He is author of about 100 papers appeared in international journals, conferences and workshops. He serves and has served as organizing and program committee member of international conferences in the field of software engineering. He is a member of IEEE Computer Society and ACM.