Received: 15 October 2016 | Revised: 10 January 2018 | Accepted: 23 April 2018

DOI: 10.1002/smr.1958

SPECIAL ISSUE PAPER

WILEY Journal of Software: Evolution and Process

A comprehensive model for code readability

Simone Scalabrino[1] | Mario Linares-Vásquez[2] | Rocco Oliveto[1] | Denys Poshyvanyk[3]

[1]University of Molise, Pesche (IS), Italy

[2]Universidad de los Andes, Bogotá, Colombia

[3]The College of William and Mary, Williamsburg, Virginia, USA

Correspondence
Simone Scalabrino, University of Molise, Pesche (IS), Italy.
Email: simone.scalabrino@unimol.it

Abstract

Unreadable code could compromise program comprehension, and it could cause the introduction of bugs. Code consists of mostly natural language text, both in identifiers and comments, and it is a particular form of text. Nevertheless, the models proposed to estimate code readability take into account only structural aspects and visual nuances of source code, such as line length and alignment of characters. In this paper, we extend our previous work in which we use textual features to improve code readability models. We introduce 2 new textual features, and we reassess the readability prediction power of readability models on more than 600 code snippets manually evaluated, in terms of readability, by 5K+ people. We also replicate a study by Buse and Weimer on the correlation between readability and FindBugs warnings, evaluating different models on 20 software systems, for a total of 3M lines of code. The results demonstrate that (1) textual features complement other features and (2) a model containing all the features achieves a significantly higher accuracy as compared with all the other state-of-the-art models. Also, readability estimation resulting from a more accurate model, ie, the combined model, is able to predict more accurately FindBugs warnings.

KEYWORDS

code readability, quality warning prediction, textual analysis

1 | INTRODUCTION

Software developers read code all the time. The very first step in each software evolution and maintenance task is to carefully read and understand the code; this step needs to be done even when the maintainer is the author of the code. Developers spend much time reading code, far more than writing it from scratch.[1] Therefore, if code is readable, it is pretty easy to start changing it; instead, modifying unreadable code is like assembling a piece of furniture with instructions written in a foreign language the one does not speak: The task is not impossible, but difficult, and a few screws still may remain unused.

Furthermore, incremental change,[2-4] which is required to perform concept location, impact analysis, and the corresponding change implementation/propagation, needs a prior code reading step before it can take place. This is why "readable code" is a fundamental and highly desirable at any stage during software maintenance and evolution.

Yet code readability remains to be a very subjective concept. Several facets, like complexity, usage of design concepts, formatting, source code lexicon, and visual aspects (eg, syntax highlighting) have been widely recognized as elements that impact program understanding.[5-7] Only recently automatic code readability estimation techniques started to be developed and used in the research community.[8-10]

As of today, 3 models for source code readability prediction have been proposed.[8-10] Such models aim at capturing how the source code has been constructed and how developers perceive it. The process consists of (1) measuring specific aspects of source code, eg, line length and number of white lines, and (2) using these metrics to train a binary classifier that is able to tell if a code snippet is "readable" or "non-readable." State-of-the-art readability models define more than 80 features, which can be divided in 2 categories: structural and visual features. The metrics belonging to the former category aim at capturing bad practices such as *lines too long* and good practices such as the *presence of white lines*; the ones

belonging to the latter category are designed to capture bad practices such as *code indentation issues* and good practices such as *alignment of characters*. However, despite a plethora of research that has demonstrated the impact of source code lexicon on program understanding,[11-17] state-of-the-art code readability models are still syntactic in nature and do not consider textual features that reflect the quality of source code lexicon.

In this paper, we extend our previous work[18] in which we proposed a set of textual features that can be extracted from source code to improve the accuracy of state-of-the-art code readability models. Indeed, we hypothesize that source code readability should be captured using both syntactic and textual aspects of source code. Unstructured information embedded in the source code reflects, to a reasonable degree, the concepts of the problem and solution domains, as well as the computational logic of the source code. Therefore, textual features capture the domain semantics and add a new layer of semantic information to the source code, in addition to the programming language semantics. To validate the hypothesis and measure the effectiveness of the proposed features, we performed a twofold empirical study: (1) We measured to what extent the proposed textual features complement the structural ones proposed in the literature for predicting code readability, and (2) we computed the accuracy of a readability model based on structural and textual features as compared to the state-of-the-art readability models. Both parts of the study were performed on a set of more than 600 code snippets that were previously evaluated, in terms of readability, by more than 5000 participants. We also replicated the study performed by Buse and Weimer,[8] in which the authors correlated the readability with the warnings raised by FindBugs, a static analysis tool. Our hypothesis is that, if readability is correlated with FindBugs warnings, an improvement in readability prediction should imply an improvement in the correlation with FindBugs warnings. We analyzed 20 open-source Java software systems, totaling in 3M lines of codes and 7K methods, and we show that using the readability predicted by the model which contains all the features (ie, the one which achieves the best readability prediction accuracy), we have a higher correlation with FindBugs warnings. We also try to explain why such correlation is present, providing examples and a more in-depth analysis.

Summarizing, the specific contributions of this paper as compared with our previous paper[18] are the following:

- The definition of 2 new textual features that enrich the set of previously proposed textual features. [18] The new textual features improve the accuracy of both a readability model based only on textual features (up to 3%) and of a comprehensive model that uses both structural and textual features (about 3%);
- An empirical study conducted on 3 data sets of snippets aimed at analyzing the effectiveness of the proposed approach while measuring the code readability. The results indicate that the model based on both structural and textual features is able to outperform the state-of-the-art code readability metrics;
- The replication of an empirical study originally performed by Buse and Weimer,[8] conducted on 20 software systems, totaling in 3M lines of code, in which we wanted to check if readability predicted by a model which achieves higher accuracy (ie, the model based on both textual and structural features) is more correlated to FindBugs warnings as compared with the baselines. The results confirm the hypothesis that readability is correlated with FindBugs warnings and that, if readability is predicted with a higher accuracy, the correlation is stronger.

The rest of the paper is organized as follows. Section 2 provides background information and discusses the related literature. Section 3 presents in details the textual features defined for the estimation of the source code readability (for the sake of completeness, we reported also the features defined in our previous paper [18]). Sections 4 and 5 describe the 2 empirical studies we conducted to evaluate the accuracy of a readability model based on both structural and textual features and the correlation between code readability and quality warnings as captured by FindBugs. Finally, Section 7 concludes the paper after a discussion of the threats that could affect the validity of the results achieved in our empirical studies (Section 6).

2 | BACKGROUND AND RELATED WORK

In the next subsections, we highlight the importance of source code lexicon (ie, terms extracted from identifiers and comments) for software quality; in addition, we describe state-of-the-art code readability models. To the best of our knowledge, 3 different models have been defined in the literature for measuring the readability of source code.[8-10] Besides estimating the readability of source code, readability models have been also used for defect prediction.[8,10] Recently, Daka et al[19] proposed a specialized readability model for test cases, which is used to improve the readability of automatically generated test suites.

2.1 | Software quality and source code lexicon

Identifiers and comments play a crucial role in program comprehension and software quality since developers express domain knowledge through the names they assign to the elements of a program (eg, variables and methods).[11-13,15,16] For example, Lawrie et al[15] showed that identifiers containing full words are more understandable than identifiers composed of abbreviations. From the analysis of source code identifiers and comments it is also possible to glean the "semantics" of the source code. Consequently, identifiers and comments can be used to measure the conceptual cohesion and coupling of classes[20,21] and to recover traceability links between documentation artifacts (eg, requirements) and source code.[22]

Although the importance of meaningful identifiers for program comprehension is widely accepted, there is no agreement on the importance of the presence of comments for increasing code readability and understandability. Also, while previous studies have pointed out that comments

make source code more readable,[23-25] the more recent study by Buse and Weimer[8] showed that the number of commented lines is not necessarily an important factor in their readability model. However, the consistency between comments and source code has been shown to be more important than the presence of comments, for code quality. Binkley et al[26] proposed the QALP tool for computing the textual similarity between code and its related comments. The QALP score has been shown to correlate with human judgements of software quality and is useful for predicting faults in modules. Specifically, the lower the consistency between identifiers and comments in a software component (eg, a class), the higher its fault-proneness.[26] Such a result has been recently confirmed by Ibrahim et al[27]; the authors mined the history of 3 large open source systems observing that when a function and its comment are updated inconsistently (eg, the code is modified, whereas the related comment is not updated), the defect proneness of the function increases. Unfortunately, such a practice is quite common since developers often do not update comments when they maintain code.[28-33]

2.2 | Source code readability models

Buse and Weimer[8] proposed the first model of software readability and provided evidence that a subjective aspect like readability can be actually captured and predicted automatically. The model operates as a binary classifier, which was trained and tested on code snippets annotated manually (based on their readability). Specifically, the authors asked 120 human annotators to evaluate the readability of 100 small snippets (for a total of 12 000 human judgements). The features used by Buse and Weimer to predict the readability of a snippet are reported in Table 1. Note that the features consider only structural aspects of source code. The model succeeded in classifying snippets as "readable" or "not readable" in more than 80% of the cases. From the 25 features, *average number of identifiers*, *average line length*, and *average number of parentheses* were reported to be the most useful features for differentiating between readable and non-readable code. Table 1 also indicates, for each feature, the predictive power and the direction of correlation (positive or negative).

Posnett et al[9] defined a simpler model of code readability as compared with the one proposed by Buse and Weimer.[8] The approach by Posnett *et al.* uses only 3 structural features: *lines of code*, *entropy*, and *Halstead's Volume metric*. Using the same dataset from Buse and Weimer,[8] and considering the area under the curve (AUC) as the effectiveness metric, the model Posnett et al was shown to be more accurate than the one by Buse and Weimer.

TABLE 1 Features used by Buse and Weimer's readability model[8a]

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

[a]The triangles indicate if the feature is positively (up) or negatively (down) correlated with high readability, and the color indicates the predictive power (green = "high," yellow = "medium," red = "low").

Dorn introduced a "generalizable" model, which relies on a larger set of features for code readability (see Table 2), which are organized into 4 categories: *visual*, *spatial*, *alignment*, and *linguistic*.[10] The rationale behind the 4 categories is that a better readability model should focus on how the code is read by humans on screens. Therefore, aspects such as syntax highlighting, variable naming standards, and operators alignment are considered by Dorn[10] as important for code readability, in addition to structural features that have been previously shown to be useful for measuring code readability. The 4 categories of features used in Dorn's model are described as follows:

- **Visual features**: To capture the visual perception of the source code, 2 types of features are extracted from the source code (including syntax highlighting and formatting provided by an IDE) when represented as an image: (1) a ratio of characters by color and colored region (eg, comments), and (2) an average bandwidth of a single feature (eg, indentation) in the frequency domain for the vertical and horizontal dimensions. For the latter, the discrete Fourier transform (DFT) is computed on a line-indexed series (one for each feature), for instance, the DFT is applied to the function of indentation space per line number.
- **Spatial features**: Given a snippet S, for each feature A marked in Table 2 as "Spatial," it is defined as a matrix $M^A \in \{0,1\}^{L \times W}$, where W is the length of the longest line in S and L is the number of lines in S. Each cell $M^A_{i,j}$ of the matrix assumes the value 1 if the character in line i and column j of S plays the role relative to the feature A. For example, if we consider the feature "comments," the cell $M^C_{i,j}$ will have the value "1" if the character in line i and column j belongs to a comment; otherwise, $M^C_{i,j}$ will be "0." The matrices are used to build 3 kind of features:
  - Absolute area (*AA*): it represents the percentage of characters with the role A. It is computed as: $AA = \frac{\sum_{i,j} M^A_{i,j}}{L \times W}$;
  - Relative area (*RA*): for each couple of features A_1, A_2, it represents the quantity of characters with role A_1 with respect to characters with role A_2. It is computed as: $RA = \frac{\sum_{i,j} M^{A_1}_{i,j}}{\sum_{i,j} M^{A_2}_{i,j}}$;
  - Regularity: it simulates "zooming-out" the code "until the individual letters are not visible but the blocks of colors are, and then measuring the relative noise or regularity of the resulting view."[10] Such a measure is computed using the 2-dimensional discrete Fourier transform on each matrix M^A.
- **Alignment features**: Aligning syntactic elements (such as "=" symbol) is very common, and it is considered a good practice in order to improve the readability of source code. Two features, namely operator alignment and expression alignment, are introduced in order to measure, respectively, how many times the operators and entire expressions are repeated on the same column/columns.
- **Natural-language features**: For the first time, Dorn introduces a textual-based factor, which simply counts the relative number of identifiers composed by words present in an English dictionary.

The model was evaluated by conducting a survey with 5K+ human annotators judging the readability of 360 code snippets written in 3 different programming languages (ie, Java, Python, and CUDA). The results achieved on this dataset showed that the model proposed by Dorn achieves a higher accuracy as compared with the Buse and Weimer's model re-trained on the new dataset. [10]

Summarizing, existing models for code readability mostly rely on structural properties of source code. Source code lexicon, while representing a valuable source of information for program comprehension, has been generally ignored for estimating source code readability. Some structural features, such as the ones that measure the number of identifiers, indirectly measure lexical properties of code, such as the vocabulary size.

TABLE 2 Features defined by Dorn[10a]

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

[a]The table maps categories (ie, visual perception, spatial perception, alignment or natural language analysis) to individual features.

However, only Dorn provides an initial attempt to explicitly use such valuable source of information[10] by considering the number of identifiers composed of words present in a dictionary. We conjecture that more pertinent aspects of source code lexicon can be exploited aiming at extracting useful information for estimating source code readability.

3 | TEXT-BASED CODE READABILITY FEATURES

Well-commented source code and high-quality identifiers, carefully chosen and consistently used in their contexts, are likely to improve program comprehension and support developers in building consistent and coherent conceptual models of the code.[11,12,17,34-36] Our claim is that the analysis of source code lexicon cannot be ignored when assessing code readability. Therefore, we propose 7 textual properties of source code that can help in characterizing its readability. In the next subsections we describe the textual features introduced to measure code readability.

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

1. Remove nontextual tokens from the corpora, eg, operators, special symbols, and programming language keywords;
2. Split the remaining tokens into separate words by using the under score or camel case separators; eg, *getText* is split into *get* and *text*;
3. Remove words belonging to a stop-word list (eg, articles, adverbs).[37]
4. Extract stems from words by using the Porter algorithm.[38]

3.1 | Comments and identifiers consistency

This feature is inspired by the QALP model proposed by Binkley et al[26] and aims at analyzing the consistency between identifiers and comments. Specifically, we compute the *Comments and Identifiers Consistency* (*CIC*) by measuring the overlap between the terms used in a method comment and the terms used in the method body:

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

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

Note that we chose to compute the simple overlap between terms instead of using more sophisticated approaches such as information retrieval (IR) techniques (as done in the QALP model), since the 2 pieces of text compared here (ie, the method body and its comment) are expected to have a very limited verbosity, thus making the application of IR techniques challenging.[39] Indeed, the QALP model measures the consistency at file level, thus focusing on code components having a much higher verbosity.

One limitation of *CIC* (but also of the QALP model) is that it does not take into account the use of synonyms in source code comments and identifiers. In other words, if the method comment and its code contain 2 words that are synonyms (EG, car and automobile), they should be considered consistent. Thus, we introduce a variant of *CIC* aimed at considering such cases:

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

where *Syn* is the set of all the synonyms of the terms in *Ids*. With such a variant the use of synonyms between comments and identifiers contributes to improving the value of *CIC*.

3.2 | Identifier terms in dictionary

Empirical studies have indicated that full-word identifiers ease source code comprehension.[11] Thus, we conjecture that the higher the number of terms in source code identifiers that are also present in a dictionary, the higher the readability of the code. Thus, given a line of code *l*, we measure the feature *Identifier terms in dictionary* (*ITID*) as follows:

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

where *Terms*(*l*) is the set of terms extracted from a line *l* of a method and *Dictionary* is the set of words in a dictionary (eg, English dictionary). As for the *CIC*, the higher the value of *ITID*, the higher the readability of the line of code *l*. To compute the feature *Identifier terms in dictionary* for an entire snippet *S*, it is possible to aggregate the $ITID(l), \forall_l \in S$—computed for each line of code of the snippet—by considering the min, the max or the average of such values. Note that the defined *ITID* is inspired by the *Natural Language Features* introduced by Dorn.[10]

3.3 | Narrow meaning identifiers

Terms referring to different concepts may increase the program comprehension burden by creating a mismatch between the developers' cognitive model and the intended meaning of the term.[34,40] Thus, we conjecture that a readable code should contain more *hyponyms*, ie, terms with a specific meaning, than *hypernyms*, ie, generic terms that might be misleading. Thus, given a line of code *l*, we measure the feature *Narrow meaning identifiers* (*NMI*) as follows:

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

where *t* is a term extracted from the line of code *l* and *particularity*(*t*) is computed as the number of hops from the node containing *t* to the root node in the hypernym tree of *t*. Specifically, we use hypernym/hyponym trees for English language defined in WordNet.[41] Thus, the higher the *NMI*, the higher the particularity of the terms in *l*, ie, the terms in the line of code *l* have a specific meaning allowing a better readability. Figure 1 shows an example of hypernyms/hyponyms tree: considering the word "state," the distance between the node that contains such a term from the root node, which contains the term "entity," is 3, so the particularity of "state" is 3. To compute the *NMI* for an entire snippet *S*, it is possible to aggregate the $NMI(l), \forall_l \in S$, by considering the min, the max or the average of such values.

3.4 | Comments readability

While many comments could surely help to understand the code, they could have the opposite effect if their quality is low. Indeed, a maintainer could start reading the comments, which should ease the understanding phase. If such comments are inadequate, the maintainer will waste time before starting to read the code. Thus, we introduced a feature that calculates the readability of comments (*CR*) using the Flesch-Kincaid[42] index, commonly used to assess readability of natural language texts. Such an index considers 3 types of elements: words, syllables, and phrases. A *word* is a series of alphabetical characters separated by a space or a punctuation symbol; a *syllable* is "a word or part of a word pronounced with a single, uninterrupted sounding of the voice [...] consisting of a single sound of great sonority (usually a vowel) and generally 1 or more sounds of lesser sonority (usually consonants)"[43]; a *phrase* is a series of words that ends with a new-line symbol, or a strong punctuation point (eg, a full stop). The Flesch-Kincaid (FK) index of a snippet *S* is empirically defined as

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

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

We calculate the *CR* by (1) putting together all commented lines from the snippet *S*; (2) joining the comments with a "." character, in order to be sure that different comments are not joined creating a single phrase; and (3) calculating the Flesch-Kincaid index on such a text.

3.5 | Number of meanings

All the natural languages contain polysemous words, ie, terms which could have many meanings. In many cases the context helps to understand the specific meaning of a polysemous word, but, if many terms have many meanings it is more likely that the whole text (or code, in this case) is ambiguous. For this reason, we introduce a feature which measures the number of meanings (*NM*), or the level or polysemy, of a snippet. For each

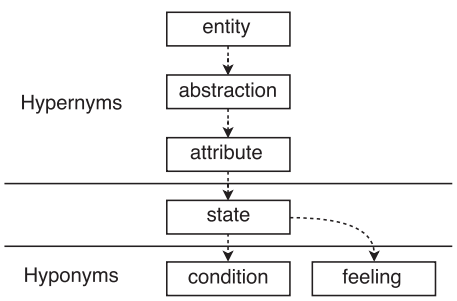

FIGURE 1 Example of hypernyms and hyponyms of the word "state"

term in the source code, we measure its number of meanings derived from WordNet.[41] To compute the feature *Number of Meanings* for an entire snippet *S*, it is possible to aggregate the *NI(l)* values—computed for each line of code *l* of the snippet—considering the max or the average of such values. We do not consider the minimum but still consider the maximum, because while it is very likely that a term with few meanings is present, and such a fact does not help in distinguishing readable snippets from not-readable ones, the presence of a term with too many meanings could be crucial in identifying unreadable snippets.

3.6 | Textual coherence

The lack of cohesion of classes negatively impacts the source code quality and correlates with the number of defects.[20,44] Based on this observation, our conjecture is that when a snippet has a low cohesion (ie, it implements several concepts), it is harder to comprehend than a snippet implementing just 1 "concept." The textual coherence of the snippet can be used to estimate the number of "concepts" implemented by a source code snippet. Specifically, we considered the syntactic blocks of a specific snippet as documents. We parse the source code and we build the abstract syntax tree (AST) in order to detect syntactic blocks, which are the bodies of every control statement (eg, `if` statements). We compute (as done for *Comments and Identifiers Consistency*) the vocabulary overlap between all the possible pairs of distinct syntactic blocks. The *Textual coherence (TC)* of a snippet can be computed as the max, the min or the average overlap between each pairs of syntactic blocks. For instance, the method in Figure 2 has 3 blocks: B_1 (lines 2-11), B_2 (lines 5-8), and B_3 (lines 8-10); for computing *TC*, first, the vocabulary overlap is computed for each pair of blocks, (B_1 and B_2, B_1 and B_3, B_2 and B_3); then, the 3 values can be aggregated by using the average, the min, or the max.

3.7 | Number of concepts

Textual Coherence tries to capture the number of implemented topics in a snippet at block level. However, its applicability may be limited when there are few syntactic blocks. Indeed, if a snippet contains just a single block, such a feature is not computable at all. Besides, textual coherence is a coarse-grain feature, and it works under the assumption that syntactic blocks are self-consistent. Therefore, we introduced a measurement which is able to directly capture the number of concepts implemented in a snippet at line-level. It is worth noting that such features can be computed also on snippets that may not be syntactically correct. In order to measure the number of concepts, as a first step, we create a document for each line of a given snippet. All the empty documents, resulting from empty lines or lines containing only non-alphabetical characters, are deleted. Then, we use a density-based clustering technique, DBSCAN,[45,46] in order to create clusters of similar documents (ie, lines). We measure the distance between 2 documents (represented as sets of terms) as

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

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

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

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

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

It is worth noting that *NOC* and NOC_{norm} measure something that has an opposite meaning with respect to textual coherence. While textual coherence is *higher* if different blocks contain the many similar words, Number of Concepts is *lower* if different lines contain many similar words. This happens because when several lines contain similar words, they are put in the same cluster and, thus, the number of clusters is lower, as well as the whole *NOC* and NOC_{norm} features.

Figure 3 shows an example of a method with just a block. In this case, *TC* cannot be computed. On the other hand, *NOC* and NOC_{norm} are computed as follows. As a first step, 4 documents are extracted from the snippet in Figure 3, namely, "public boolean is playing TG measure measure," "thread safe," "TG measure play measure this play measure," "return is playing play measure null measure equals play measure."

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

FIGURE 2 An example of computing textual coherence for a code snippet

```
1 public boolean isPlaying(TGMeasure measure) {
2   // thread safe
3   TGMeasure playMeasure = this.playMeasure;
4
5   return (isPlaying() && playMeasure != null && measure.equals(playMeasure));
6 }
7
```

FIGURE 3 Example of a small method

Assuming that such documents are clustered all together, except for "thread safe", which constitutes a cluster on its own, we have that $NOC(isPlaying) = 2$ and $NOC_{norm}(isPlaying) = \frac{2}{4} = 0.5$.

DBSCAN does not need to know the number of clusters, which is, actually, the result of the computation that we use to define *NOC* and NOC_{norm}. Instead, this algorithm needs the parameter ϵ, which represents the maximum distance at which 2 documents need to be in order to be grouped in the same cluster. We did not choose ϵ arbitrarily; instead, we tuned such a parameter, by choosing the value that allows the features *NOC* and NOC_{norm} to achieve, alone, the highest readability prediction accuracy. In order to achieve this goal, we considered all the snippets and the oracles from the 3 data sets described in Section 4 and we trained and tested 9 classifiers, each of which contained just a feature, NOC^{ϵ}, where NOC^{ϵ} is *NOC* computed using a specific ϵ parameter for DBSCAN. Since the distance measure we use ranges between 0 and 1, also ϵ can range between such values and, thus, the values we used as candidate ϵ for the 9 NOC^{ϵ} features are {0.1,0.2,...,0.9}; we discarded the extreme values, 0 and 1, because in these cases each document would have been in a separate cluster or all documents would have been in the same cluster, respectively. We use each classifier containing a single NOC^{ϵ} feature to predict readability, and we pick the value of ϵ that leads to the best classification accuracy with 10-fold cross-validation. The classification technique used for tuning ϵ was Logistic Regression, also used in Section 4. We repeated the same procedure for NOC_{norm}. Figure 4 shows the accuracy achieved by each classifier containing different NOC^{ϵ} (in blue) or $NOCnorm\epsilon$ (in red). The best ϵ value for *NOC* is 0.1, while for NOC_{norm} it is 0.3, as the chart shows.

3.8 | Readability vs understandability

Posnett et al[9] compared the difference between readability and understandability to the difference between syntactic and semantic analysis. Readability measures the effort of the developer to access the information contained in the code, while understandability measures the complexity of such information. We defined a set of textual features that still capture aspects of code related to the difficulty of accessing the information contained in a snippet. For example, *NOC* estimates the number of concepts implemented in a snippet. A snippet with a few concepts, potentially more readable, can still be hard to understand if a few concepts are not easy to understand. In our opinion, textual features, which do not take into account semantics, like the ones we defined, can be used to measure readability.

4 | CASE STUDY 1: IMPROVING READABILITY ESTIMATION

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

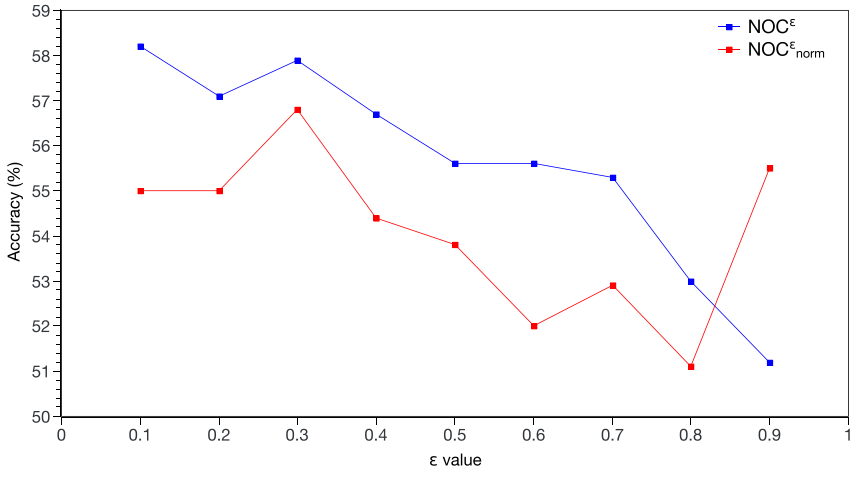

FIGURE 4 Accuracy of different classifiers based only on NOC^{ϵ} (blue) and $NOCnorm\epsilon$ (red)

We formulated the following research questions (RQs):

- **RQ$_1$**: *To what extent the proposed textual features complement the structural ones proposed in the literature for predicting code readability?* With this preliminary question we are interested in verifying whether the proposed textual features complement structural ones when used to measure code readability. This represents a crucial prerequisite for building an effective comprehensive model considering both families of features.
- **RQ$_2$**: *What is the accuracy of a readability model based on structural and textual features as compared to the state-of-the-art readability models?* This research question aims at verifying to what extent a readability model based on both structural and textual features overcomes readability models mainly based on structural features, such as the model proposed by Buse and Weimer,[8] the one presented by Posnett et al,[9] and the most recent one introduced by Dorn.[10]

4.1 | Data collection

An important prerequisite for evaluating a code readability model is represented by the availability of a reliable oracle, ie, a set of code snippets for which the readability has been manually assessed by humans. This allows measuring to what extent a readability model is able to approximate human judgment of source code readability. All the datasets used in the study are composed of code snippets for which the readability has been assessed via human judgement. In particular, each snippet in the data sets is accompanied by a flag indicating whether it was considered readable by humans (ie, binary classification). The first dataset (in the following $D_{b\&w}$) was provided by Buse and Weimer,[8] and it is composed of 100 Java code snippets having a mean size of 7 lines of code. The readability of these snippets was evaluated by 120 student annotators. The second dataset (in the following D_{dorn}) was provided by Dorn[10] and represents the largest dataset available for evaluating readability models. It is composed of 360 code snippets, including 120 snippets written in CUDA, 120 in Java, and 120 in Python. The code snippets are also diverse in terms of size including for each programming language the same number of small- (~10 LOC), medium- (~30 LOC) and large- (~50 LOC) sized snippets. In D_{dorn}, the snippets' readability was assessed by 5468 humans, including 1800 industrial developers.

The main drawback of the aforementioned datasets ($D_{b\&w}$ and D_{dorn}) is that some of the snippets are not complete code entities (eg, methods); therefore, some of the data instances in $D_{b\&w}$ and D_{dorn} datasets are code fragments that only represent a partial implementation (and thus they may not be syntactically correct) of a code entity. This is an impediment for computing one of the new textual features introduced in this paper: *textual coherence (TC)*; it is impossible to extract code blocks from a snippet if an opening or closing bracket is missing. For this reason, we built an additional dataset (D_{new}), by following an approach similar to the one used in the previous work to collect $D_{b\&w}$ and D_{dorn}.[8,10] Firstly, we extracted all the methods from 4 open source Java projects, namely, *jUnit*, *Hibernate*, *jFreeChart*, and *ArgoUML*, having a size between 10 and 50 lines of code (including comments). We focused on methods because they represent syntactically correct and complete code entities of code.

Initially, we identified 13 044 methods for D_{new} that satisfied our constraint on the size. However, the human assessment of all the 13K+ methods is practically impossible, since it would require a significant human effort. For this reason, we evaluated the readability of only 200 sampled methods from D_{new}. The sampling was not random, but rather aimed at identifying the most representative methods for the features used by all the readability models defined and studied in this paper. Specifically, for each of the13 044 methods, we calculated all the features (ie, the structural features proposed in the literature and textual ones proposed in this paper) aiming at associating each method with a feature vector containing the values for each feature. Then, we used a greedy algorithm for center selection[47] to find the 200 most representative methods in D_{new}. The distance function used in the implementation of such algorithm is represented by the Euclidean distance between the feature vector of 2 snippets. The adopted selection strategy allowed us (1) to enrich the diversity of the selected methods avoiding the presence of similar methods in terms of the features considered by the different experimented readability models and (2) to increase the generalizability of our findings.

After selecting the 200 methods in D_{new}, we asked 30 computer science students from the College of William and Mary to evaluate the readability r of each of them. The participants were asked to evaluate each method using a 5-point Likert scale ranging between 1 (*very unreadable*) and 5 (*very readable*). We collected the rankings through a web application (Figure 5) where participants were able to (1) read the method (with syntax highlighting); (2) evaluate its readability; and (3) write comments about the method. The participants were also allowed to complete the evaluation in multiple rounds (eg, evaluate the first 100 methods in 1 day and the remaining after 1 week). Among the 30 invited participants, only 9 completed the assessment of all the 200 methods. This was mostly due to the large number of methods to be evaluated; the minimum time spent to complete this task was about 2 hours. In summary, given the 200 methods in $m_i \in D_{new}$ and 9 human taggers $t_j \in T$, we collected readability rankings $r(m_i,t_j), \forall_{i,j}, i \in [1,200], j \in [1,9]$.

After having collected all the evaluations, we computed, for each method $m \in D_{new}$, the mean score that represents the final readability value of the snippet, ie, $\bar{r}(m) = \frac{\Sigma_1^9 r(m,j)}{9}$. We obtained a high agreement among the participants with Cronbach α = .98, which is comparable to the one achieved in $D_{b\&w}$ = 0.96. This confirms the results reported by Buse and Weimer in terms of humans agreement when evaluating/ranking code readability: "*humans agree significantly on what readable code looks like, but not to an overwhelming extent*".[8] Note that code readability evaluation by using crisp categories (eg" *readable* and *non-readable*) is required to build a readability model over the collected snippets; therefore, for the methods in D_{new}, we used the mean of the readability score among all the snippets as a cut-off value. Specifically, methods having a score below 3.6 were classified as *non-readable*, while the remaining methods (ie, $\bar{r}(m) \geq 3.6$) as *readable*. A similar approach was also used by Buse and Weimer.[8]

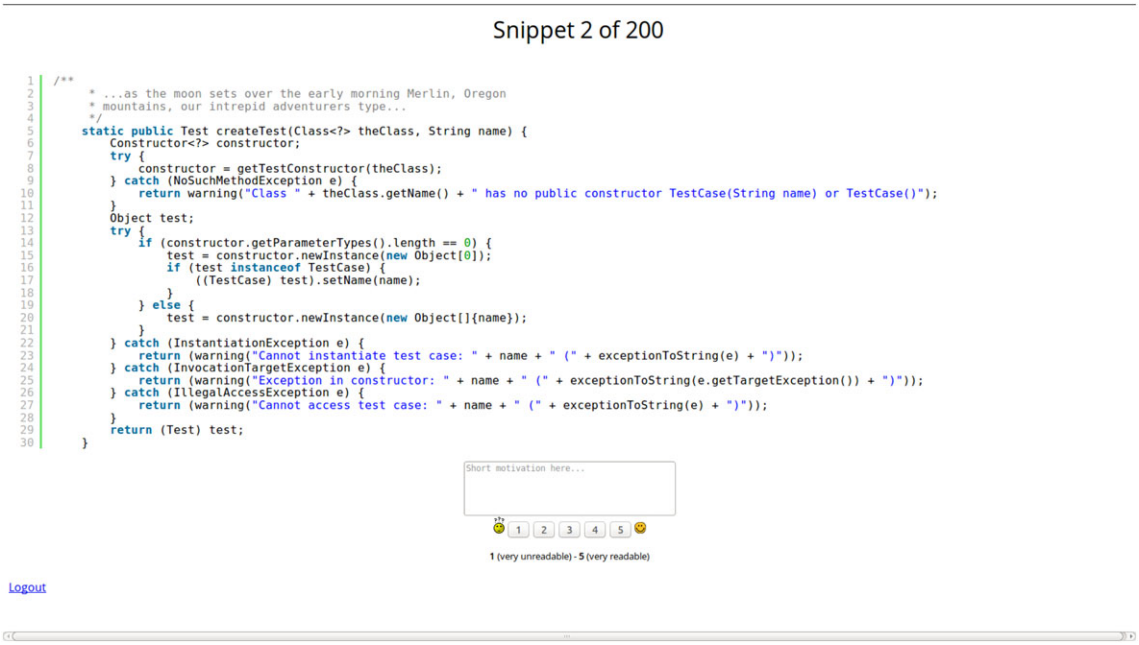

FIGURE 5 Web application used to collect the code readability evaluation for our new dataset D_{new}

4.2 | Analysis method

To answer **RQ$_1$**and **RQ$_2$**, we built readability models (ie, binary classifiers) for each dataset (ie, $D_{b\&w}$, D_{dorn}, and D_{new}) by using different sets of structural and (our) textual features: Buse and Weimer's (*BWF*),[8] Posnett's (*PF*),[9] Dorn's (*DF*),[10] our textual features (*TF*), and all the features (*All-Features* = *BWF*∪*PF*∪*DF*∪*TF*). With notational purposes, we will use $R\langle F eatures\rangle$ to denote a specific readability model R we built using a set of *Features*. For instance, $R\langle TF\rangle$ denotes the textual features-based readability model. It is worth noting that with our experiments we are not running the same models proposed in the prior works, but, we are using the same features proposed by previous works.

As for the classifier used with the models, we relied on logistic regression because it has been shown to be very effective in binary-classification and it was used by Buse and Weimer for their readability model.[8] To avoid over-fitting, we performed feature selection by using linear forward selection with a wrapper strategy[48] available in the Weka machine learning toolbox. In the wrapper selection strategy each candidate subset of features is evaluated through the accuracy of the classifier trained and tested using only such features. The final result is the subset of features which obtained the maximum accuracy. With respect to our previous study,[18] we increased the "search termination" parameter from 5 to 10, in order to search more deeply in the possible features subsets. Such a parameter indicates the amount of backtracking of the algorithm. Such a modification resulted in a little improvement (2˜%) in the accuracy of some of the classifiers.

In the case of **RQ$_1$**, we analyzed the complementarity of the textual features-based model with the models trained with structural features, by computing overlap metrics between $R\langle TF\rangle$ and each of the 3 competitive models (ie, $R\langle BWF\rangle$, $R\langle PF\rangle$, $R\langle DF\rangle$). Specifically, given 2 readability models under analysis, $R\langle TF\rangle$ a model based on textual features, and $R\langle SF\rangle$ a model based on structural features (ie, $SF \in \{BWF, PF, DF\}$) *, the metrics are defined as in the following:

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

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

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

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

*Note that later in this paper we will replace $R\langle SF\rangle$ with $R\langle BWF\rangle$, $R\langle PF\rangle$, or $R\langle DF\rangle$.

Turning to the second research question (**RQ$_2$**), we compared the accuracy of a readability model based on both all the structural and textual features ($R\langle All\text{-}Features\rangle$) with the accuracy of the e baselines, ie, $R\langle BWF\rangle$, $R\langle PF\rangle$ and $R\langle DF\rangle$. To further show the importance of textual features, we also compared $R\langle All\text{-}Features\rangle$ to an additional baseline, namely a model based on all the state-of-the-art structural and visual features ($R\langle SVF\rangle = R\langle BWF + PF + DF\rangle$). To compute the accuracy, we fist compute

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

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

In addition, we report the accuracy achieved by the readability model only exploiting textual features (ie, $R\langle TF\rangle$). In particular, we measured the percentage of code snippets correctly classified as readable/non-readable by each technique on each of the 3 datasets. We also report the AUC achieved by all the models, in order to compare them taking into account an additional metric, widely used for evaluating the performance of a classifier.

Each readability model was trained on each dataset individually and a 10-fold cross-validation was performed. The process for the tenfold cross-validation is composed of 5 steps: (1) randomly divide the set of snippets for a dataset into 10 approximately equal subsets, regardless of the projects they come from; (2) set aside 1 snippet subset as a test set, and build the readability model with the snippets in the remaining subsets (ie, the training set); (3) classify each snippet in the test set using the readability model built on the snippet training set and store the accuracy of the classification; (4) repeat this process, setting aside each snippet subset in turn; (5) compute the overall average accuracy of the model.

Finally, we used statistical tests to assess the significance of the achieved results. In particular, since we used tenfold cross validation, we considered the accuracy achieved on each fold by all the models. We used the Wilcoxon test[49] (with $\alpha = .05$) in order to estimate whether there are statistically significant differences between the classification accuracy obtained by $R\langle TF\rangle$ and the other models. Our decision for using the Wilcoxon test, is a consequence of the usage of the tenfold cross-validation to gather the accuracy measurements. During the cross-validation, each fold is selected randomly, but we used the same seed to have the same folds for all the experiments. For example, the fifth testing fold used for $R\langle BWF\rangle$ is equal to the fifth testing fold used with $R\langle All\text{-}Features\rangle$. Consequently, pairwise comparisons are performed between related samples.

Because we performed multiple pairwise comparisons (ie, *All-features* vs. the rest), we adjusted our *P* values using the Holm's correction procedure.[50] In addition, we estimated the magnitude of the observed differences by using the Cliff's Delta (*d*), a non-parametric effect size measure for ordinal data.[51] Cliff's *d* is considered negligible for $d < 0.148$ (positive as well as negative values), small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$. [51]

4.3 | Replicability

We make our study fully replicable providing an online appendix for this paper.[52] Such an online appendix contains: (1) the new dataset and the links to the other 2 dataset used in this study; (2) the ARFF files containing all the features computed on all the snippets in the 3 datasets; (3) our readability tool, which uses the combined dataset trained on all the dataset to compute the readability level of a snippet.

4.4 | RQ1: complementarity of readability features

Table 3 reports the overlap metrics computed between the results of the readability models using textual and structural features. Across the 3 datasets, the $R\langle TF\rangle$ model exhibits an overlap of code snippets correctly classified as readable/non-readable included between 68% ($R\langle TF\rangle \cap R\langle BWF\rangle$) and 75% ($R\langle TF\rangle \cap R\langle DF\rangle$). This means that, despite the competitive model considered, about 30% of the code snippets are differently assessed as readable/non-readable when only relying on textual features. Indeed, (1) between 11% ($R\langle TF\rangle \setminus R\langle DF\rangle$) and 17% ($R\langle TF\rangle \setminus R\langle PF\rangle$) of code snippets are correctly classified only by $R\langle TF\rangle$ and (2) between 13% ($R\langle PF\rangle \setminus R\langle TF\rangle$) and 17% ($R\langle BWF\rangle \setminus R\langle TF\rangle$) are correctly classified only by the competitive models exploiting structural information.

These results highlight a high complementarity between structural and textual features when used for readability assessment. An example of a snippet for which the textual features are not able to provide a correct assessment of its readability is reported in Figure 6. Such a method (considered "unreadable" by human annotators) has a pretty high average textual coherence (0.58), but, above all, it has a high comment readability and comment-identifiers consistency, ie, many terms co-occur in identifiers and comments (eg, "batch" and "fetch"). Nevertheless, some lines are too long, resulting in a high maximum and average line length (146 and 57.3, respectively), both impacting negatively the perceived readability. [8]

Figure 7 reports, instead, a code snippet correctly classified as "readable" only when exploiting textual features. The snippet has suboptimal structural characteristics, such as a high average/maximum line length (65.4 and 193, respectively) and a high average number of identifiers (2.7), both negatively correlated with readability. Nevertheless, the method has high average textual coherence (~ 0.73) and high comments readability

TABLE 3 RQ_1: Overlap between $R\langle TF\rangle$ and $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle DF\rangle$

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

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

    LOG.trace( "Post flush" );

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

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

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

}
```

FIGURE 6 Code snippets correctly classified as "non-readable" only when relying on structural features and missed when using textual features

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

FIGURE 7 Code snippets correctly classified as "readable" only when relying on textual features and missed by the competitive techniques

(100.0). The source code can be read almost as natural language text and the semantic of each line is pretty clear, but such an aspect is completely ignored by structural features.

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

4.5 | RQ2: accuracy of readability model

Table 4 shows the accuracy achieved by (1) the comprehensive readability model, namely, the model which exploits both structural and textual features (*All-Features*); (2) the model solely exploiting textual features (*TF*); (3) the 3 state-of-the-art models mainly based on structural features (*BWF*, *PF*, and *DF*) ;and (4) the model based on all the state-of-the-art structural and visual features. We report the overall accuracy achieved by each model using 2 different proxies: overall_{wm} and overall_{am}. Overall_{wm} is computed as the weighted mean of the accuracy values for each dataset, where the weights are the number of snippets in each dataset; we used such a proxy also in our previous work.[18] Overall_{am} is computed as the arithmetic mean of the accuracy values for each dataset.

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

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

Since the results in terms of accuracy may depend on a specific threshold, we also report in Table 5 the AUC achieved by all the readability models. Also in this case, we report the overall accuracy achieved by each model using the 2 proxies previously defined, ie, overall_{wm} (weighted average) and overall_{am} (arithmetic average). The AUC values, overall, confirm that the combined model outperforms the other models. Nevertheless, we can see that the overall_{wm} AUC achieved by $R\langle TF\rangle$ is comparable with the overall_{wm} AUC achieved by $R\langle DF\rangle$, and slightly minor than the one achieved by $R\langle BWF\rangle$. While in terms of accuracy $R\langle DF\rangle$ seems to be slightly better than $R\langle BWF\rangle$, in terms of AUC, the opposite is true. Furthermore, there is a high difference in terms of accuracy between $R\langle \textit{All-features}\rangle$ and all the other models on the dataset by Buse and Weimer, but in terms of AUC this difference is less evident and, instead, other models achieve higher AUC (eg, $R\langle PF\rangle$).

Table 6 shows the *P* values after correction and the Cliff's delta for the pairwise comparisons performed between the model that combines structural and textual features and the other models. When analyzing the results at dataset granularity, we did not find significant differences between *All-Features* and the other models. However, the effect size is medium-large (ie, $d \geq 0.33$) in most of the comparisons. This issue of no statistical significance with large effect size is an artifact of the size of the samples used with the test, which has been reported previously by Cohen[53] and Harlow *et al.*[54]; in fact, the size of the samples used in our tests for each dataset is 10 measurements (note that we performed tenfold cross validation). In that sense, we prefer to draw conclusions (conservatively) from the tests performed on the set D_{all}, which has a larger sample (30 measurements). When using the datasets as a single one (ie, D_{all}), there is significant difference in the accuracy when comparing $R\langle \textit{All-features}\rangle$ to the other models; the results are confirmed with the Cliff's *d* that suggest a medium-large difference (ie, $d \geq 0.33$) in all the cases except for $R\langle SVF\rangle$, for which the difference is, overall, small (0.28).

TABLE 4 RQ_2: Average accuracy achieved by the readability models in the 3 datasets

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

TABLE 5 RQ_2: Average AUC achieved by the readability models in the 3 datasets

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

TABLE 6 RQ_2: *P* values (corrected with the Holm procedure) of the Wilcoxon test and Cliff's delta (*d*), for the pairwise comparisons between the accuracy of $R\langle All\text{-}features\rangle$ and each one of state-of-the-art models[a]

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

[a]In bold, statistically significant values.

Figure 8 illustrates the difference in the accuracy achieved with each model by using the mean accuracy and confidence intervals (CIs). There is a 95% of confidence that the mean accuracy of $R\langle All\text{-}features\rangle$ is larger than $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle TF\rangle$ (ie, there is no overlap between the CIs). Although the mean accuracy of $R\langle All\text{-}features\rangle$ is the largest one in the study, there is an overlap with the CIs for $R\langle DF\rangle$ and $R\langle SVF\rangle$. Combining $R\langle BWF\rangle$, $R\langle PF\rangle$, and $R\langle DF\rangle$, improves the accuracy on average when compared to $R\langle TF\rangle$. Therefore, including the proposed textual features in state-of-the-art models, overall, improves the accuracy of the readability model, with significant difference when compared to the other models. The statistical tests also confirm that using only textual features is not the best choice for a code readability model.

Regarding individual features, we investigated the most relevant features for each combination dataset-model. Table 7 reports the importance (ie, weight) of single features, using the ReliefF attribute selection algorithm.[55,56] Specifically, we report the 3 best features for each pair dataset-model, specifying also their ranking in the complete list of features for the same dataset and their importance weight. The textual features that, overall, show the best ReliefF values (ie, weight and ranking) are *Comments Readability*, *Textual Coherence* and *Number of Concepts*, since they are in the top-3 positions for the 3 datasets. Besides, the ranking values confirm what Table 4 previously hinted, ie, that textual features are useful in Dorn's dataset and in the new dataset, but they are less useful in Buse and Weimer's dataset; indeed, besides CR, the other features have a low ReliefF value. Finally, Figure 9 shows the average attribute importance weight of all the textual features: It is clear that *Comments Readability* is the best predictor of code readability among the textual features, achieving an average ReliefF which is higher than the double of the second best predictor (ie, TC_{min}).

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

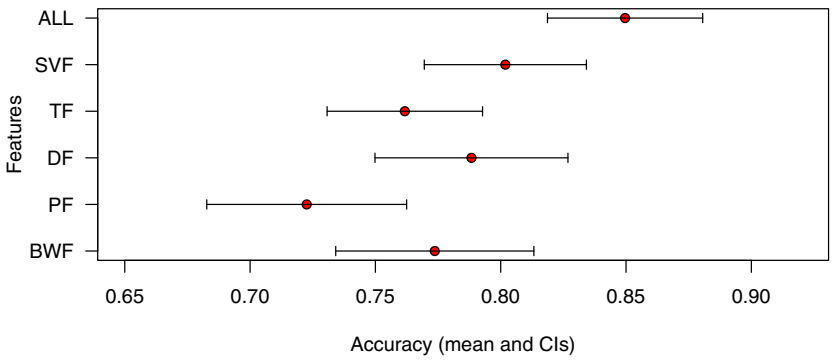

FIGURE 8 Mean accuracy and confidence intervals (CIs) with 95% of confidence for the accuracy of each one of the models analyzed for RQ_2

TABLE 7 RQ_2: Evaluation of the single features using ReliefF

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

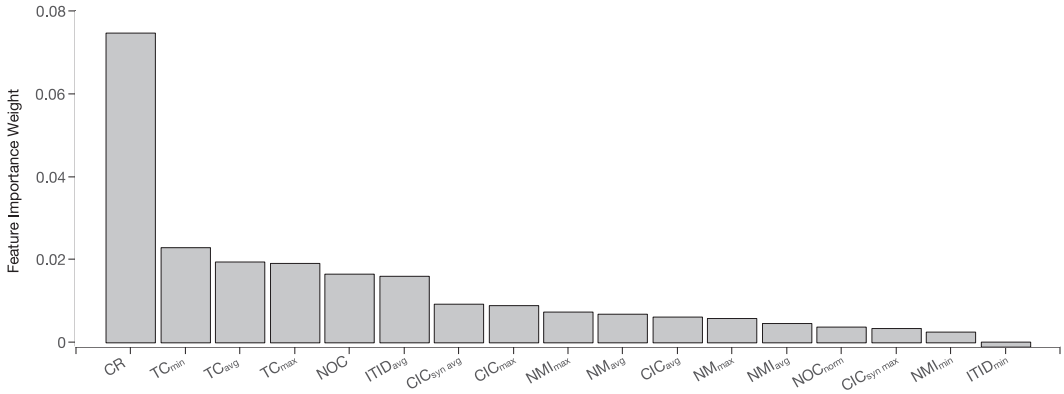

FIGURE 9 Average importance weights (computed with the ReliefF methods) of all the textual features

5 | CASE STUDY 2: PREDICTION OF QUALITY WARNINGS

This second study is a replication of the study performed by Buse and Weimer,[8] in which the authors used readability as a proxy for quality, in particular, using warnings reported by the FindBugs tool[†] as an indicator of quality. Specifically, the *goal* of the second study is to understand if the model which achieves the best accuracy in readability prediction (ie, the *All-features* model) can predict FindBugs warnings with a higher accuracy compared with the model originally proposed by Buse and Weimer.[8] It is worth noting that we are not directly using the metrics defined in Section 3 as predictors of FindBugs warnings; instead, we first use some of the features previously defined to predict readability, and then we use readability to predict warnings. It is not the goal of this study to assess the FindBugs warnings prediction power of the metrics used to predict readability. The *quality focus* is to improve the prediction of quality warnings by considering readability metrics, while the *perspective* of the study is of a researcher interested in analyzing whether the proposed approach can be used for the prediction of quality problems in source code.

5.1 | Research question and study context

In the context of the second study, we formulated the following research question:

- **RQ_3:** *Is the combined readability model able to improve the prediction accuracy of quality warnings?*With this question we want to understand if a higher accuracy in readability prediction helps to improve the correlation between FindBugs warnings and readability. In other words, we want to re-assess the FindBugs warnings prediction power of readability models.

The *context* of this study is composed of 20 Java programs: 11 of the 15 systems analyzed by Buse and Weimer[8] and 9 systems introduced in study. We did not include 4 of the 15 system cited in the study by Buse and Weimer[8] (ie, Gantt Project, SoapUI, Data Crow, and Risk) because the snapshots of the specific versions of those systems were not available at the time when this study was performed. To select the 9 new systems, we first randomly chose from SourceForge some software categories which were not represented by the other systems and, for each of them, we chose one of the most downloaded ones. We started from the most downloaded project, and we selected the first one which had the following characteristics:

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

Table 8 depicts the selected systems, which accounts for 103 000 methods and about 3 million lines of code.

To answer **RQ_3**, we followed the process depicted in Figure 10. First, we trained a Logistic classifier on the dataset defined in our previous study,[18] and we computed the readability score of all the methods of all the systems using our combined model. The readability score is defined as the probability that a method belongs to the class "readable" according to the classifier. Such a value ranges between 0 (surely unreadable) and 1 (surely readable). For each method, we also computed the unreadability score, which is the probability that a snippet belongs to the class "unreadable." Such a score is computed as $unreadability(M) = 1 - readability(M)$. As a second step, we ran the FindBugs tool on all the compiled

[†]http://findbugs.sourceforge.net/

TABLE 8 Software systems analyzed[a]

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

[a]The star symbol indicates software systems added in this study. "Methods with warnings" indicates the number of methods with at least a warning.

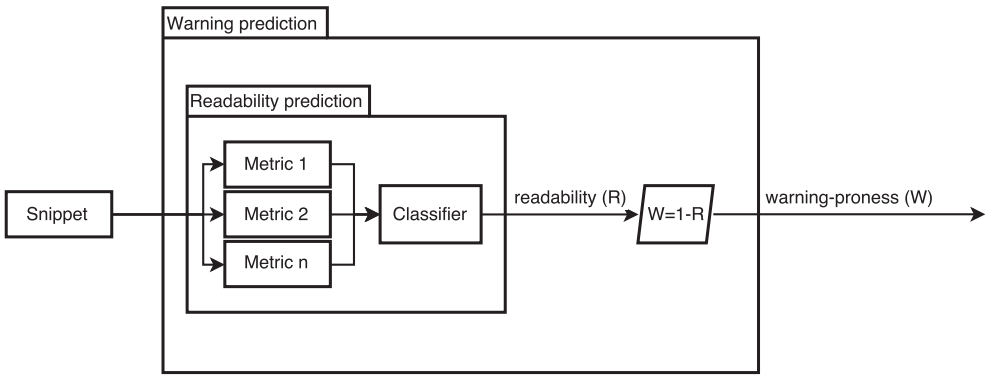

FIGURE 10 Workflow followed to predict readability by relying on FindBugs warnings

versions of the analyzed systems. Then, we extracted from the FindBugs report only the warnings reported at method level; indeed, FindBugs warnings can also concern lines of code which belong to other parts of a class (eg, field declarations). We discarded such warnings, so that we have a readability score (computed at method level) for each warning.

Given a system S having a set X of methods, we split X in 2 subsets: X_b, methods with at least a warning, and X_c, warning-free methods. To avoid the bias derived from the different size of the sets, we subsample X_b and X_c: We consider $m = min(|X_b|,|X_c|)$, and we randomly pick, from each set, m elements. At the end, we have 2 sets $X_{bs} \subseteq X_b$ and $X_{cs} \subseteq X_c$, so that $|X_{bs}| = |X_{cs}|$. This subsampling procedure was the same applied by Buse and Weimer.[8]

Finally, we used the unreadability score (*unreadability*(M)) to predict FindBugs warnings. To evaluate how accurate is the unreadability score to predict FindBugs warnings, we first plotted the receiving operating curve (ROC) obtained using unreadability as a continuous predictor of warning/no warning: such a curve shows the true-positive rate (TFP) against the false-positive rate (FPR) considering different thresholds for the predictor (unreadability). Then, we computed the area under such a curve (AUC). We preferred AUC over F-measure, originally used by Buse and Weimer,[8] because AUC does not require the choice of a threshold, which may not be the same for all the models. To answer **RQ$_3$**, we compared 3 readability models: (1) the original model proposed by Buse and Weimer trained on their dataset ($R\langle BWF\rangle \circ BW$)[‡]; (2) the model by Buse and Weimer trained on the new dataset ($R\langle BWF\rangle \circ New$); (3) our model containing all the features trained on the new dataset ($R\langle All\text{-}features\rangle \circ New$).

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

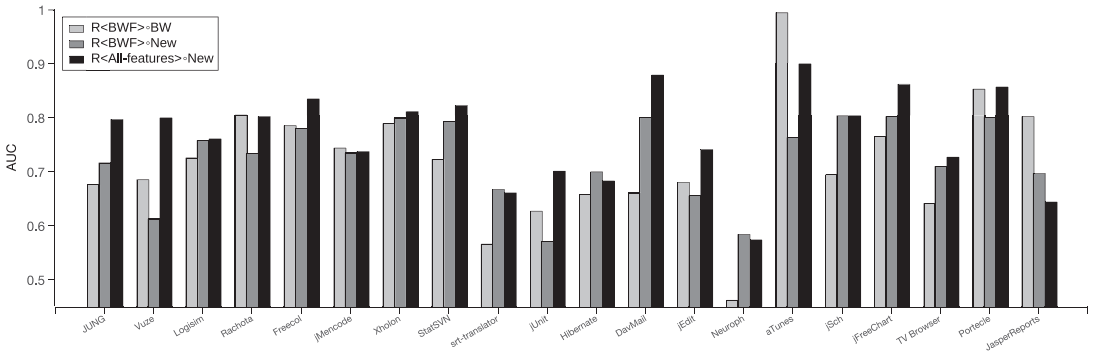

FIGURE 11 AUC achieved using readability models to predict FindBugs warnings for each system

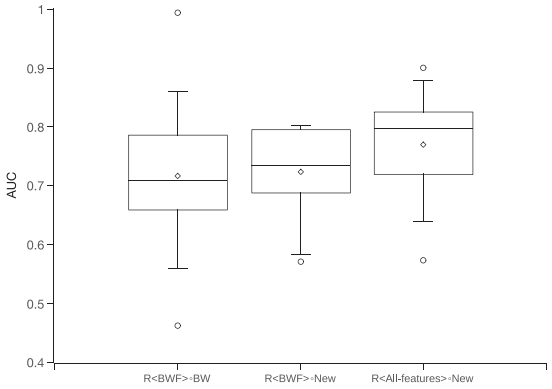

FIGURE 12 Box plots showing the AUC achieved using readability models to predict FindBugs warnings

We included the first model as a sanity check and we used the tool provided by the authors to compute the readability score; then, we trained both $R\langle BW\,F\rangle$ and $R\langle All\text{-}features\rangle$ on the same dataset, so that there is no bias caused by the different training set.

5.2 | RQ3: Improvement of the prediction of quality warnings

Figure 11 shows the AUC achieved by the 3 readability models on all the analyzed systems. $R\langle All\text{-}features\rangle\circ New$ is able to predict FindBugs warnings more accurately than the baselines on 12 systems out of 20. The AUC achieved by such a model ranges between 0.573 (Neuroph) and 0.900 (aTunes). Figure 12 shows 3 box plots which indicate, for each model, the AUC achieved on the 20 systems analyzed. Here, it is clear that $R\langle All\text{-}features\rangle\circ New$generally achieves a higher AUC as compared with the other models. Specifically, the mean AUC achieved by $R\langle BW\,F\rangle\circ BW$ is 0.717, the AUC achieved by $R\langle BW\,F\rangle\circ New$ is 0.724 while the AUC achieved by $R\langle All\text{-}features\rangle\circ New$is 0.770. We also report in Figure 13 the box plot relative only to the 11 projects also considered in the original experiment by Buse and Weimer.[8]

Furthermore, we checked if the difference is statistically significant ($P = .05$) performing a paired Wilcoxon test[49] with P values adjusted using the Holm's correction procedure for multiple pairwise comparisons[50]: The adjusted P values resulting from such a test are .006 (comparison with $R\langle BW\,F\rangle\circ BW$) and .004 (comparison with $R\langle BW\,F\rangle\circ New$) with a medium effect size (.375 and .360, correspondingly), which suggest that $R\langle All\text{-}features\rangle\circ New$has a significantly higher AUC compared with the 2 baselines. Figure 14 illustrates the difference in the AUC achieved with each model by using the mean AUC and confidence intervals (CIs). The CIs show how there is overlap between the 3 models, however there is a region of the CI of $R\langle All\text{-}features\rangle\circ New$ that is higher than the other CIs, which confirms the medium effect size of the significant difference between $R\langle All\text{-}features\rangle\circ New$ and the other 2 models. There is a 95% of confidence that the mean AUC achieved by $R\langle All\text{-}features\rangle\circ New$.

The results suggest that the answer to $\mathbf{RQ_3}$is *positive*: an improvement in the prediction accuracy of readability results in a better prediction of FindBugs warnings. Such a result further corroborates the findings by Buse and Weimer[8] about the correlation between readability and FindBugs warnings.

Furthermore, we wanted to understand which categories of FindBugs warnings correlated with readability the most. We selected a set of 6 categories of FindBugs warnings, ie, "Performance," "Correctness," "Bad Practices," "Malicious code," "Dodgy code," and "Internationalization." Categories described on the official FindBugs website,[§] which are not represented for many of the analyzed systems, such as "Security," were excluded.

Figure 15 shows the AUC achieved by the best model (the $R\langle All\text{-}features\rangle\circ New$) on different categories of FindBugs warnings. "Dodgy code" is the best predicted category for 7 systems out of 20, "Correctness" is the best one for 7 systems out of 20, and the others are the best predicted

[§]http://findbugs.sourceforge.net/bugDescriptions.html

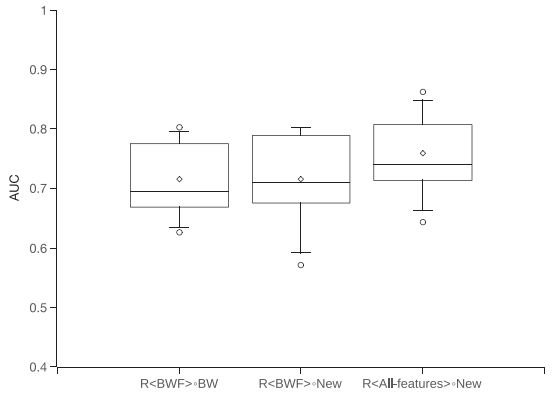

FIGURE 13 Box plots showing the AUC achieved using readability models to predict FindBugs warnings, only on projects also considered in the original experiment by Buse and Weimer

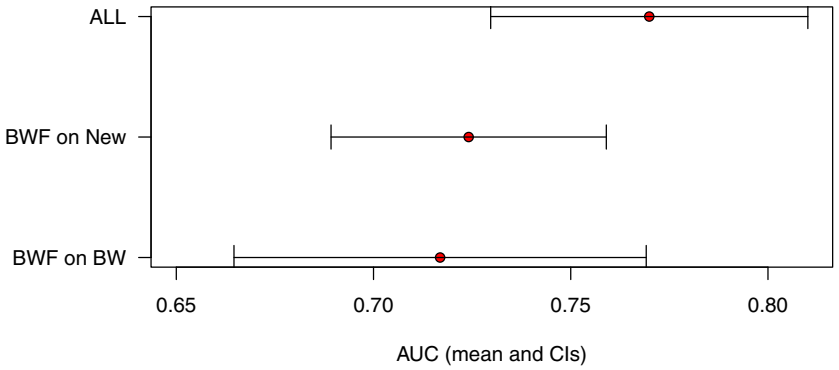

FIGURE 14 Mean accuracy and confidence intervals (CIs) with 95% of confidence for the AUC of each one of the models analyzed for $\mathbf{RQ_3}$

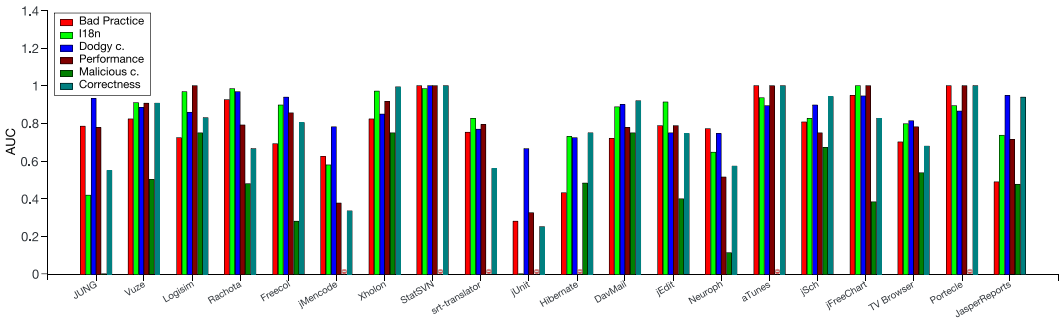

FIGURE 15 AUC achieved using readability models to predict different categories of FindBugs warnings for each system

more rarely ("Internationalization" and "Performance" for 5 systems and "Bad practice" for 4 systems). "Malicious code" is the category with the lowest prediction accuracy.

Analyzing the results more in depth, Figure 16 shows the box plots of the AUC achieved by *R⟨All-features⟩◦New* on the analyzed categories of FindBugs warnings. Except for "Malicious code," for which the mean AUC is 0.470, the warning belonging to all the other categories are predicted with a mean AUC above 0.7. The main reason why "Malicious code" is not correlated with readability is that the most frequent warnings belonging to such a category can be found in very short snippets. Figure 17 shows an example of method with the warning "May expose internal representation by returning reference to mutable object". Such a warning is raised when "Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object." In many cases this warning can be found in "getter" methods, such as the one in the example, which have, obviously, a higher level of readability.

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

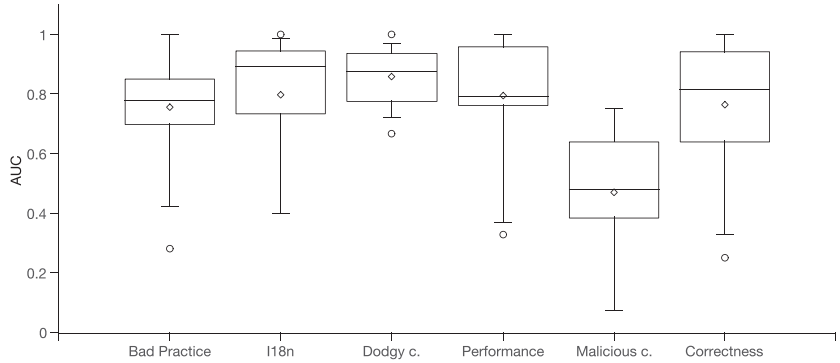

FIGURE 16 Box plots showing the AUC achieved using readability models to predict different categories of FindBugs warnings

```
1  public class KeyParameter
2    implements CipherParameters
3  {
4      private byte[]  key;
5
6      [...]
7
8      public byte[] getKey()
9      {
10         return key;
11     }
12 }
```

FIGURE 17 Excerpt of a class with a method (getKey) for which FindBugs raises a "Malicious code" warning

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

The cause of the correlation could be that unreadable code is more likely to have hidden mistakes, which may not be fixed until the system fails or a tool warns the developers about it. Consider the snippet in Figure 18. FindBugs shows a warning belonging to the category "Dodgy code," specifically "Useless object created." According to the official documentation, this warning is reported when an object is "created and modified, but its value never go outside of the method or produce any side-effect." In this case, the variable declared in line 6 is used in lines 13, 26, 38, and 49, but it has no effect on the outside of the method, so it can be removed, together with the lines in which it is used. Noticing this kind of issues on an unreadable method such as the one proposed in the example could be very hard, and this may be the reason why it is introduced and it remains in the code. In readable code, instead, such warnings may be less frequent because they would be clearly visible either to the developer who writes it or to any other developer who maintains the source code.

Summary for RQ_3. Our study confirms that the correlation between warnings and readability is high and it suggests that a model which predicts readability with an higher accuracy is also able to predict FindBugs warnings better. Specifically, all the categories of warnings taken into account are well-predicted, except for "Malicious code," which is more frequent in small and readable methods, like "getter" methods.

6 | THREATS TO VALIDITY

Possible threats to validity for the first study are related to the methodology in the construction of the new data set, to the machine learning technique used and to the feature selection technique adopted. The threats to validity for the second study are mainly related to the analyzed systems and to the metrics used to evaluate the correlation between readability and FindBugs warnings. In this section we discuss such threats, grouping them into *construct*, *internal* and *external* validity.

6.1 | Construct validity

The main threat is the choice of the metrics used for evaluating (1) the readability models and the correlation between readability and FindBugs warnings and (2) to the machine learning technique used for evaluating the readability models. For the first study, we used accuracy and AUC achieved when using *logistic regression* as the underlying classifier for the readability models, while for the second study, we used AUC for evaluating the prediction power of readability to predict warnings; for both the studies, we could have used different metrics (eg, F-measure), and for the first study, we could have used different machine learning techniques (eg, BayesNet or neural networks). We chose accuracy and AUC because they are widely used in the literature for the evaluation of classifiers. Specifically, we used AUC for the second study because other metrics would have implied the use of a specific threshold, while we wanted to compute the inherent correlation between a continuous metric (readability) and a discrete value (presence of FindBugs warnings).

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

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

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

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

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

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

    byte[] comment = torrent.getComment();

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

    byte[] created = torrent.getCreatedBy();

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

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

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

    return candidates;
}
```

FIGURE 18 Unreadable code with a "Dodgy code" warning

In addition, in the first study, the results could depend on the machine learning technique used for computing the accuracy of each model. Table 9 shows the accuracy achieved by each model using different machine learning techniques. While different techniques achieve different levels of accuracy, some results are still valid when using other classifiers, eg, the combined model achieves a better accuracy than any other model on all the data sets, except for the data set by Buse and Weimer when using *BayesNet* and *RandomForest*.

6.2 | Internal validity

To mitigate the over-fitting problem of machine learning techniques, we used tenfold cross-validation, and we performed statistical analysis (Wilcoxon test, effect size, and confidence intervals) in order to measure the significance of the differences among the accuracies of different models. Also, feature selection could affect the final results on each model. Finding the best set of features in terms of achieved accuracy is infeasible when the number of features is large. Indeed, the number of subsets of a set of n elements is 2^n, while an exhaustive search is possible for models with a limited number of features, like *BWF*, *PF*, and *TF*, it is unacceptable for *DF* and *All-Features*. Such a search would require, respectively, 1.2×10^{18} and 3.2×10^{34} subset evaluations. Thus, we used a linear forward selection technique[48] in order to reduce the number of evaluations and to obtain a good subset in a reasonable time.

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

TABLE 9 Accuracy achieved by *All-Features*, *TF*, *BWF*, *PF*, and *DF* in the 3 data sets with different machine learning techniques

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

Another threat to internal validity is the use of data sets of different sizes: Buse and Weimer involved 120 participants and they collected 12 000 evaluations; Dorn involved over 5000 participants and he collected 76 741 evaluations (each snippet was evaluated, on average, by about 200 participants); we involved 9 participants and we collected 1800 evaluations. Besides, each data set implies also its own risks. The main problem of the data set by Buse and Weimer is that it contains also not compilable snippets; one of the textual features we introduced, *Textual Coherence*, can only be computed on syntactically correct snippets. In the data set by Dorn, each participant evaluated a small subset of snippets, 14/360 on average; in this case, there could be the risk that the difference in rating is a matter of the difference among evaluators more than the difference among snippets. Finally, the main threat to validity related to our data set is the small number of evaluators. Therefore, since each data set complements the others, to reduce the risks we report the results on all of them. However, since the number of evaluators are different, we compare the models on the 3 data sets separately.

6.3 | External validity

In the first study, To build the new data set, we had to select a set of snippets that human annotators would evaluate. The set of snippets selected may not be representative enough and, thus, could not help to build a generic model. We limited the impact of such a threat by selecting the set of the most distant snippets as for the features used in this study through a greedy center selection technique. Other threats regarding the human evaluation of the readability of snippets, also pointed out by Buse and Weimer,[8] are related to the experience of human evaluators and to the lack of a rigorous definition of readability. However, the human annotators for D_{new} showed a high agreement on the readability of snippets.

In the second study, we had to select a set of systems for computing the correlation between readability and FindBugs warnings. We selected a subset of the systems analyzed in the previous study by Buse and Weimer[8] and we introduced new systems for such a study. The main threats are that (1) the systems may not be representative enough and (2) some of the systems may use FindBugs, and thus the use of such a tool may influence the natural correlation with readability. We limited the first threat by selecting systems belonging to different categories and having different sizes in terms of methods and lines of code. Besides, we limited the second threat by checking if the number of FindBugs warnings was not too low (eg, similar to 0) on each system.

7 | CONCLUSION

State-of-the-art code readability models mostly rely on structural metrics, and as of today, they do not consider the impact of source code lexicon on code readability. In this paper, we present a set of textual features that are based on source code lexicon analysis and aim at improving the accuracy of code readability models. The proposed textual features measure the consistency between source code and comments, specificity of the identifiers, usage of complete identifiers, among the others. To validate our hypothesis, stating that combining structural and textual features improves the accuracy of readability models, we used the features proposed by the state-of-the art models as a baseline, and measured (1) to what extent the proposed textual-based features complement the structural features proposed in the literature for predicting code readability, and (2) the accuracy achieved when including textual features into the state-of-the-art models. Our findings show that textual features complement structural ones, and the combination (ie, structural+textual) improves the accuracy of code readability models. Furthermore, we replicated a study by Buse and Weimer on the correlation between readability and FindBugs warnings, in order to check if an improvement in readability prediction causes an improvement in the correlation with FindBugs warnings. The results confirm our hypothesis: the model with the highest readability prediction accuracy also predicts FindBugs warnings more accurately than the other models. We conclude that unreadable

code is more prone to having issues, which may be also bugs, and it is more likely that such problems would stay in the code, as it is more difficult to notice and correct them.

ORCID

Rocco Oliveto http://orcid.org/0000-0002-7995-8582

REFERENCES

1. Erlikh L. Leveraging legacy system dollars for e-business. *IT Professional*. May 2000;2(3):17-23.
2. Bennett KH, Rajlich VT. Software maintenance and evolution: a roadmap. In: Proceedings of the conference on the future of software engineering. Limerick, Ireland; 2000:73-87.
3. Rajlich V, Gosavi P. Incremental change in object-oriented programming. *IEEE Softw*. July 2004;21(4):62-69.
4. Poshyvanyk D, Marcus D. Combining formal concept analysis with information retrieval for concept location in source code. In: Icpc'07; Canada: Banff, Alberta, BC; 2007: 37-48.
5. Martin RC. *Clean code: A handbook of agile sofware craftsmanship*. Upper Saddle River, NJ: Prentice Hall; 2009.
6. Oram A, Wilson G. *Beautiful Code: Leading Programmers Explain How They Think*. Sebastapol, CA, USA: O'reilly; 2007.
7. Beck K. *Implementation Patterns*. Westford, MA, USA: Addison Wesley; 2007.
8. Buse RPL, Weimer Westley. Learning a metric for code readability. *IEEE TSE*. 2010;36(4):546-558.
9. Posnett D, Hindle A, Devanbu PT. A simpler model of software readability. In: Msr'11; Waikiki, Honolulu, HI, USA; 2011:73-82.
10. Dorn J. A general software readability model. *Master's Thesis*; 2012. https://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf.
11. Lawrie D, Morrell C, Feild H, Binkley D. Effective identifier names for comprehension and memory. *ISSE*. 2007;3(4):303-318.
12. Lawrie D, Morrell C, Feild H, Binkley D. What's in a name? A study of identifiers. In: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on. Athens, Greece; 2006:3-12.
13. Caprile B, Tonella P. Restructuring program identifier names. In: Proceedings off the International Conference in Software Maintenance(ICSM 2000); 2000; Washington, DC:97-107.
14. Deissenbock F, Pizka M. Concise and consistent naming. In: Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005); 2005; Los Alamitos:97-106.
15. Lawrie D, Feild H, Binkley D. Syntactic identifier conciseness and consistency. In: Scam'06; 2006; Los Alamitos:139-148.
16. Enslen E, Hill E, Pollock LL, Vijay-Shanker K. Mining source code to automatically split identifiers for software analysis. In: Msr'09; 2009; Los Alamitos:71-80.
17. Takang A, Grubb P, Macredie R. The effects of comments and identifier names on program comprehensibility: an experiential study. *J Program Lang*. 1996;4(3):143-167.
18. Scalabrino S, Vásquez ML, Poshyvanyk D, Oliveto R. Improving code readability models with textual features. In: 24th IEEE international conference on program comprehension, ICPC 2016; 2016; Austin, TX, USA:1-10.
19. Daka E, Campos J, Fraser G, Dorn J, Weimer W. Modeling readability to improve unit tests. In: Esec/fse'15; 2015; ACM, New York:107-118.
20. Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. 2008;34(2):287-300.
21. Poshyvanyk D, Marcus A. The conceptual coupling metrics for object-oriented systems. In: ICSM'06. Philadelphia, PA, USA; 2006:469-478.
22. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *IEEE TSE*. 2002;28(10):970-983.
23. Elshoff JL, Marcotty M. Improving computer program readability to aid modification. *CACM*. 1982;25(8):512-521.
24. Tenny T. Program readability: Procedures versus comments. *IEEE TSE*. 1988;14(9):1271-1279.
25. Spinellis D. *Code Quality: The Open Source Perspective*. Boston MA: Adobe Press; 2006.
26. Binkley D, Feild H, Lawrie DJ, Pighin M. Increasing diversity: natural language measures for software fault prediction. *J Syst Softw*. 2009;82(11):1793-1803.
27. Ibrahim WM, Bettenburg N, Adams B, Hassan AE. On the relationship between comment update practices and software bugs. *J Syst Softw*. 2012;85(10):2293-2304.
28. Fluri B, Würsch M, Gall H. Do code and comments co-evolve? On the relation between source code and comment changes. In: Wcre'07. Vancouver, BC, Canada; 70-79.
29. Linares-Vásquez M, Li B, Vendome C, Poshyvanyk D. How do developers document database usages in source code? In: Ase'15. Lincoln, NE, USA; 2015; 36-41.
30. Li B, Vendome C, Linares-Vásquez M, Poshyvanyk D, Kraft N. Automatically documenting unit test cases. In: ICST'16. Chicago, IL, USA; 2016:341-352.
31. Linares-Vásquez M, Li B, Vendome C, Poshyvanyk D. Documenting database usages and schema constraints in database-centric applications, *ISSTA 2016*; 2016; New York, NY, USA:ACM:270-281.
32. Linares-Vásquez M, Cortés-Coy LF, Aponte J, Poshyvanyk D. Changescribe: A tool for automatically generating commit messages. *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, *ICSE '15*. Piscataway, NJ, USA: IEEE Press; 2015:709-712.
33. Cortés-Coy LF, Linares-Vásquez M, Aponte J, Poshyvanyk D. On automatically generating commit messages via summarization of source code changes. *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, *SCAM '14*. Washington, DC, USA: IEEE Computer Society; 2014:275-284.
34. Deissenboeck F, Pizka M. Concise and consistent naming. *Softw Qual J*. 2006;14(3):261-282.
35. Haiduc S, Marcus A. On the use of domain terms in source code. In: Icpc'08. Amsterdam, Netherlands; 2008:113-122.

36. Binkley D, Davis M, Lawrie D, Morrell C. To CamelCase or Under score. In: Icpc'09. Vancouver, BC, Canad; 2009:158-167.
37. Baeza-Yates R, Ribeiro-Neto B. *Modern Information Retrieval*. Harlow, England, UK: Addison-Wesley; 1999.
38. Porter MF. An algorithm for suffix stripping. *Program*. 1980;14(3):130-137.
39. Lucia AD, Penta MD, Oliveto R, Panichella A, Panichella S. Labeling source code with information retrieval methods: an empirical study. *EMSE*. 2014;19(5):1383-1420.
40. Arnaoudova V, Eshkevari LM, Oliveto R, Guéhéneuc YG, Antoniol G. Physical and conceptual identifier dispersion: measures and relation to fault proneness. In: Icsm'10; 2010:1-5.
41. Miller GA. Wordnet: A lexical database for english. 1995;38(11):39-41.
42. Flesch Rudolph. A new readability yardstick. *J Appl Psychol*. 1948;32(3):221.
43. Collins american dictionary.
44. Ujhazi B, Ferenc R, Poshyvanyk D, Gyimothy T. New conceptual coupling and cohesion metrics for object-oriented systems. *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10*. Washington, DC, USA: IEEE Computer Society; 2010:33-42.
45. Ester M, Kriegel HP, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Kdd, Vol. 96; 1996:226-231.
46. Sander J, Ester M, Kriegel HP, Xu X. Density-based clustering in spatial databases: the algorithm gdbscan and its applications. *Data Min Knowl Discov*. June 1998;2(2):169-194. https://doi.org/10.1023/A:1009745219419
47. Kleinberg J, Tardos É. *Algorithm Design*. Taramani, India: Pearson Education India.
48. Gütlein M, Frank E, Hall M, Karwath A. Large-scale attribute selection using wrappers. In: Cidm'09. Nashville, TN, USA; 2009:332-339.
49. Sheskin DJ. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*: Chapman & All; 2007.
50. Holm S. A simple sequentially rejective Bonferroni test procedure. *Scand J Stat*. 1979;6:65-70.
51. Grissom RJ, Kim JJ. *Effect Sizes for Research: A Broad Practical Approach*. 2nd ed. Mahwah, NJ, USA: Lawrence Earlbaum Associates; 2005.
52. Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D. Online appendix. https://dibt.unimol.it/report/readability.
53. Cohen J. The earth is round (p<.05). *Am Psychol*. 1994;49(12):997-1003.
54. Harlow LL, Mulaik SA, Steiger JH. *What if There Were No Significance Tests?* New York, NY, USA: Psychology Press; 1997.
55. Kira K, Rendell LA. A practical approach to feature selection. In: Proceedings of the ninth international workshop on machine learning. Aberdeen, Scotland, UK; 1992:249-256.
56. Kononenko I, Simec E, Robnik-Sikonja M. Overcoming the myopia of inductive learning algorithms with relieff. *Appl Intell*. 1997;7(1):39-55.

How to cite this article: Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D. A Comprehensive Model for Code Readability. *J Softw Evol Proc*. 2018;1-23. https://doi.org/10.1002/smr.1958