# Improving Code Readability Models with Textual Features

Simone Scalabrino*, Mario Linares-Vásquez§, Denys Poshyvanyk§ and Rocco Oliveto*
*University of Molise, Pesche (IS), Italy
§The College of William and Mary, Williamsburg, Virginia, USA

***Abstract*—Code reading is one of the most frequent activities in software maintenance; before implementing changes, it is necessary to fully understand source code often written by other developers. Thus, readability is a crucial aspect of source code that may significantly influence program comprehension effort. In general, models used to estimate software readability take into account only structural aspects of source code, *e.g.,* line length and a number of comments. However, source code is a particular form of text; therefore, a code readability model should not ignore the textual aspects of source code encapsulated in identifiers and comments. In this paper, we propose a set of textual features aimed at measuring code readability. We evaluated the proposed textual features on 600 code snippets manually evaluated (in terms of readability) by 5K+ people. The results demonstrate that the proposed features complement classic structural features when predicting code readability judgments. Consequently, a code readability model based on a richer set of features, including the ones proposed in this paper, achieves a significantly higher accuracy as compared to all of the state-of-the-art readability models.**

## I. Introduction

*Beautiful*, *Clean*, *Great*, or *Good code* [1], [2], [3] are common expressions that describe the type of code that software developers expect/hope to write or read. In fact, having *"great/clean/good/beautiful code"* is more important during software evolution and maintenance tasks, because developers spend a lot of time maintaining code (which can be written by others), far more than writing the code from scratch [4]. Reading code is the very first step during incremental change [5], [6], [7], which is required to perform concept location, impact analysis, and the corresponding change implementation/propagation. Developers need to read and understand the code before changing it. Therefore, *"readable code"* is a fundamental and highly desirable property of source code.

Several facets have been reported as components that contribute to having readable code. For instance, complexity, usage of design concepts, formatting, source code lexicon, and visual aspects (*e.g.,* syntax highlighting) have been widely recognized as elements that impact program understanding [1], [2], [3]. However, metrics for software readability are still under development and have started gaining traction just recently in the research community [8], [9], [10].

As of today, only three models for source code readability have been proposed [8], [9], [10], which extract features from a code snippet, and then report a readability ranking or a binary classification (*i.e.,* "readable", "not-readable"). State-of-the-art code readability models aim at capturing how the source code has been constructed and how it looks to the developers; the models mostly rely on structural properties of the source code (*e.g.,* number of identifiers). However, despite a plethora of research that has demonstrated the impact of source code lexicon on program understanding [11], [12], [13], [14], [15], [16], [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.

Under the hypothesis that source code readability should be captured using both syntactic and textual code features, in this paper we present a set of textual features that can be extracted from source code to improve the accuracy of state-of-the-art code readability models. Unstructured information embedded in the source code reflects to a reasonable degree the concepts of the problem and solution domains, as well as the computational logic of the source code. Therefore, textual features capture the domain semantics and add a new layer of semantic information to the source code, in addition to the programming language semantics. To validate the hypothesis and measure the effectiveness of the proposed features, we performed a two-fold empirical study: (i) we measured to what extent the proposed textual features complement the structural ones proposed in the literature for predicting code readability; and (ii) 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 600 code snippets that were previously evaluated (in terms of readability) by 5K+ participants.

Summarizing, the contributions of this paper are:

- A set of textual features that enrich previous code readability metrics by considering textual (or lexical) aspects of source code, which we claim to be crucial for effectively capturing code readability;
- An empirical study conducted on three 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 structural and textual features is able to outperform the state-of-the-art code readability metrics;
- A new set of 200 code snippets in Java that was manually tagged (in terms of readability) by nine participants. This new dataset is comprised of complete code snippets (existing datasets contain only partial snippets), and was collected to consider textual features. The dataset is available with our online appendix [18].

## II. BACKGROUND AND RELATED WORK

In the next sub-sections we highlight the importance of source code lexicon (*i.e.,* 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, three different models have beed defined in the literature for measuring the readability of source code [8], [9], [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.

### A. 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 (*e.g.,* variables and methods) [11], [12], [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 (*e.g.,* requirements) and source code (*e.g.,* [22]).

While the importance of meaningful identifiers for program comprehension is quite consolidated, there is no agreement on the importance of the presence of comments for increasing code readability and understandability. While the previous studies pointed out that comments make source code more readable [23], [24], [25], the more recent study by Buse and Weimer [8] showed that the number of commented lines is not 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 (*e.g.,* 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 three large open source systems observing that when a function and its comment are updated inconsistently (*e.g.,* the code is modified, whereas the related comment is not updated), the defect proneness of the function increases. Unfortunately, such a practice is quite common since developers often do not update comments when they maintain code [28], [29], [30].

### B. Structural Features as a Proxy of Readability

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 I. 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 I also indicates, for each feature, the predictive power and the direction of correlation (positive or negative).

TABLE I
FEATURES USED BY BUSE AND WEIMER'S READABILITY MODEL [8]. 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").

| 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 | | ▼ |

Posnett *et al.* [9] defined a simpler model of code readability as compared to the one proposed by Buse and Weimer [8]. The approach by Posnett *et al.* uses only three 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, Posnett *et al.*'s model was shown to be more accurate than the one by Buse and Weimer.

### C. A Universal Model of Code Readability

Dorn introduced a "generalizable" model, which relies on a larger set of features for code readability (see Table II), which are organized into four categories: *visual*, *spatial*, *alignment*, and *linguistic* [10]. The rationale behind the four categories is that a better readability model should focus on how the code is read by humans on screens. Therefore, aspects such as

TABLE II
FEATURES DEFINED BY DORN [10]. THE TABLE MAPS CATEGORIES (I.E., VISUAL PERCEPTION, SPATIAL PERCEPTION, ALIGNMENT OR NATURAL LANGUAGE ANALYSIS) TO INDIVIDUAL FEATURES.

| FEATURE | VISUAL | SPATIAL | ALIGNMENT | TEXTUAL |
|---|---|---|---|---|
| Line length | • | | | |
| Indentation length | • | | | |
| Assignments | • | | | |
| Commas | • | | | |
| Comparisons | • | | | |
| Loops | • | | | |
| Parentheses | • | | | |
| Periods | • | | | |
| Spaces | • | | | |
| Comments | • | • | | |
| Keywords | • | • | | |
| Identifiers | • | • | | • |
| Numbers | • | • | | |
| Operators | • | • | • | |
| Strings | | • | | |
| Literals | | • | | |
| Expressions | | | • | |

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. In the following we describe the four categories of features used in Dorn's model.

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

**Spatial features**: Given a snippet $S$, for each feature $A$ marked in Table II as "Spatial", it is defined as a matrix $M^A \in \{0,1\}^{L \times W}$, where $W$ is the length of the longest line in $S$ and $L$ is the number of lines in $S$. Each cell $M^A_{i,j}$ of the matrix assumes the value 1 if the character in line $i$ and column $j$ of $S$ plays the role relative to the feature $A$. For example, if we consider the feature "comments", the cell $M^C_{i,j}$ will have the value "1" if the character in line $i$ and column $j$ belongs to a comment; otherwise, $M^C_{i,j}$ will be "0". The matrices are used to build three kind of features:

- Absolute area ($AA$): it represents the percentage of characters with the role $A$. It is computed as: $AA = \frac{\sum_{i,j} M^A_{i,j}}{L \times W}$;
- Relative area ($RA$): for each couple of features $A_1, A_2$, it represents the quantity of characters with role $A_1$ with respect to characters with role $A_2$. It is computed as: $RA = \frac{\sum_{i,j} M^{A_1}_{i,j}}{\sum_{i,j} M^{A_2}_{i,j}}$;
- Regularity: it simulates "zooming-out" the code "until the individual letters are not visible but the blocks of colors are, and then measuring the relative noise or regularity of the resulting view"[10]. Such a measure is computed using the two-dimensional Discrete Fourier Transform on each matrix $M^A$.

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

**Natural-language features**: For the first time, Dorn introduces a textual-based factor, which simply counts the relative number of identifiers composed by words present in an English dictionary. The model was evaluated by conducting a survey with 5K+ human annotators judging the readability of 360 code snippets written in three different programming languages (*i.e.,* Java, Python and CUDA). The results achieved on this dataset showed that the model proposed by Dorn achieves a higher accuracy as compared to the Buse and Weimer's model re-trained on the new dataset [10].

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

## III. 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], [31], [32], [33]. Our claim is that the analysis of source code lexicon cannot be ignored when assessing code readability. Therefore, we propose seven textual properties of source code that can help in characterizing its readability. In the next subsections we describe these features. Note that we use the word *term* to refer to any word extracted from source code.

### A. Comments and Identifiers Consistency (CIC)

This feature is inspired by the QLAP 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:

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

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 QLAP model), since the two pieces of text compared here (*i.e.,* the method body and its comment) are expected to have a very limited verbosity, thus making the application of IR techniques challenging [34]. Indeed, the QLAP 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 QLAP model) is that it does not take into account the use of synonyms in source code comments and identifiers. In other words, if the method comment and its code contain two words that are synonyms (*e.g.,* car and automobile), they should be considered consistent. Thus, we introduce a variant of *CIC* aimed at considering such cases:

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

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 contribute to improve the value of *CIC*.

### B. Identifier Terms in Dictionary (ITID)

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:

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

where $Terms(l)$ is the set of terms extracted from a line $l$ of a method and $Dictionary$ is the set of words in a dictionary (*e.g.,* English dictionary). As for the *CIC*, the higher the value of *ITID*, the higher the readability of the line of code $l$. In order to compute the feature *Identifier terms in dictionary* for an entire snippet $S$, it is possible to aggregate the $ITID(l), \forall_l \in S$—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].

### C. Narrow Meaning Identifiers (NMI)

Terms referring to different concepts may increase the program comprehension burden by creating a mismatch between the developers' cognitive model and the intended meaning of the term [31], [35]. Thus, we conjecture that a readable code should contain more *hyponyms*, *i.e.,* terms with a specific meaning, than *hypernyms*, *i.e.,* generic terms that might be misleading. Thus, given a line of code $l$, we measure the feature *Narrow meaning identifiers* (*NMI*) as follows:

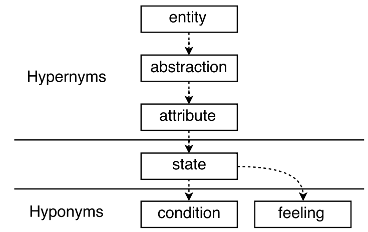


Fig. 1. Example of hypernyms and hyponyms of the word "state".

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

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 [36]. Thus, the higher the *NMI*, the higher the particularity of the terms in $l$, *i.e.,* the terms in the line of code $l$ have a specific meaning allowing a better readability. Fig. 1 shows an example of hypernyms/hyponyms tree: considering the word "state", the distance between the node that contains such a term from the root node, which contains the term "entity", is 3, so the particularity of "state" is 3. In order to compute the *NMI* for an entire snippet $S$, it is possible to aggregate the $NMI(l), \forall_l \in S$, by considering the $\min$, the $\max$ or the average of such values.

### D. Textual Coherence (TC)

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

```java
public void buildModel() {
    if (getTarget() != null) {
        Object target = getTarget();
        Object kind = Model.getFacade().getAggregation(target);
        if (kind == null
                || kind.equals(Model.getAggregationKind().getNone())) {
            setSelected(ActionSetAssociationEndAggregation.NONE_COMMAND);
        } else {
            setSelected(ActionSetAssociationEndAggregation.AGGREGATE_COMMAND);
        }
    }
}
```

Fig. 2. An example of computing textual coherence for a code snippet

### E. Comments Readability (CR)

While many comments could surely help to understand the code, they could have the opposite effect if their quality is low. Indeed, a maintainer could start reading the comments, which should ease the understanding phase. If such comments are 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 [37] index, commonly used to assess readability of natural language texts. Such an index considers three types of elements: words, syllables, and phrases. A *word* is a series of alphabetical characters 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 one or more sounds of lesser sonority (usually consonants)” [38]; a *phrase* is a series of words that ends with a new-line symbol, or a strong punctuation point (*e.g.,* a full-stop). The Flesch-Kincaid (FK) index of a snippet $S$ is empirically defined as:

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

While word segmentation and phrase segmentation are easy tasks, it is a little 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 (i) putting together all commented lines from the snippet $S$; (ii) joining the comments with a “.” character, in order to be sure that different comments are not joined creating a single phrase; (iii) calculating the Flesch-Kincaid index on such a text.

### F. Number of Meanings (NM)

All the natural languages contain polysemous words, *i.e.,* terms which could have many meanings. In many cases the context helps to understand the specific meaning of a polysemous word, but, if many terms have many meanings it is more likely that the whole text (or code, in this case) is ambiguous. For this reason, we introduce a feature which measures the number of meanings (*NM*), or the level or polysemy, of a snippet. For each term in the source code, we measure its number of meanings derived from WordNet [36]. In order to compute the feature *Number of Meanings* for an entire snippet $S$, it is possible to aggregate the *NI(l)* values—computed for each line of code $l$ of the snippet—considering the $\max$ or the average of such values. We do not consider the minimum but still consider the maximum, because while it is very likely that a term with few meanings is present, and such a fact does not help in distinguishing readable snippets from not-readable ones, the presence of a term with too many meanings could be crucial in identifying unreadable snippets.

## IV. Empirical Study Design

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.

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].

### A. Data Collection

An important prerequisite for evaluating a code readability model is represented by the availability of a reliable oracle, *i.e.,* a set of code snippets for which the readability has been manually assessed by humans. This allows measuring to what extent a readability model is able to approximate human judgment of source code readability. All the datasets used in the study are composed of code snippets for which the readability has been assessed via humans judgment. In particular, each snippet in the datasets is accompanied by a flag indicating whether it was considered readable by humans (*i.e.,*, binary classification). The first dataset (in the following $D_{b\&w}$) was provided by Buse and Weimer [8] and it is composed of 100 Java code snippets having a mean size of seven lines of code. The readability of these snippets was evaluated by 120 student annotators. The second dataset (in the following $D_{dorn}$) was provided by Dorn [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- ($\sim$10 LOC), medium- ($\sim$30 LOC) and large- ($\sim$50 LOC) sized snippets. In $D_{dorn}$, the snippets' readability was assessed by 5,468 humans, including 1,800 industrial developers.

The main drawback of the above datasets is that some of the snippets are not complete code entities (*e.g.,* methods) but fragments of code only representing a partial implementation (and thus they may not be syntactically correct). This represents an impediment for computing one of the new features introduced in this paper, *i.e., 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 methods of four open source Java projects, namely *jUnit*, *Hibernate*, *jFreeChart* and *ArgoUML*, having a size between ten and 50 lines of code (including comments). We focused on methods because they represent syntactically correct and complete snippets 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 method (*i.e.,*, 13,044 methods) we calculated all the features (*i.e.,* the structural features proposed in the literature and those 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 [39] to find the 200 most representative methods in $D_{new}$. The distance function used in the implementation of such algorithm is represented by the Euclidean distance between the feature vector of two snippets. The adopted selection strategy allowed us (i) to enrich the diversity of the selected methods avoiding the presence of similar methods in terms of the features considered by the different experimented readability models, and (ii) to increase the generalizability of our findings.

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

After having collected all the evaluations, we computed, for each method $m \in D_{new}$, the mean score that represents the final readability value of the snippet, i.e., $\bar{r}(m) = \frac{\Sigma_1^9 r(m,j)}{9}$. We obtained a high agreement among the participants with Cronbach-$\alpha$=0.98, which is comparable to the one achieved in $D_{b\&w}$=0.96. This confirms the results achieved by Buse and Weimer: "*humans agree significantly on what readable code looks like, but not to an overwhelming extent*" [8]. Note that crisp categories (*e.g.,*, *readable*, *non-readable*) are 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 as *readable*. A similar approach was also used by Buse and Weimer [8].

## B. Analysis Method

In order to answer the RQs, we built readability models (*i.e.,* binary classifiers) for each dataset (*i.e.,* $D_{b\&w}$, $D_{dorn}$, and $D_{new}$) by using different sets of structural and textual (*i.e.,* our proposal) 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*= $BWF \cup PF \cup DF \cup TF$). With notational purposes, we will use $R\langle Features\rangle$ to denote an specific model for a set of features, for instance, $R\langle TF\rangle$ denotes the textual features-based model.

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 using a wrapper strategy [40] 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.

In the case of $\mathbf{RQ}_1$ 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 three competitive models (*i.e.,* $R\langle BWF\rangle$, $R\langle PF\rangle$, $R\langle DF\rangle$). Specifically, given two readability models under analysis, $R\langle TF\rangle$ a model based on textual features, and $R\langle SF\rangle$ a model based on structural features ($SF \in BWF, PF, DF$), the metrics are defined as in the following:

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

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

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

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

Turning to the second research question (**RQ**$_2$), we compared the accuracy of a readability model based on both all the structural and textual features ($R\langle$ *All-Features* $\rangle$) with the accuracy of the three baselines, *i.e.,* $R\langle BWF\rangle$, $R\langle PF\rangle$ and $R\langle DF\rangle$. In order to compute the accuracy, we fist compute:

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

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

In addition, we report the accuracy achieved by the readability model only exploiting textual features (*i.e.,* $R\langle TF\rangle$). In particular, we measured the percentage of code snippets correctly classified as readable/non-readable by each technique on each of the three datasets.

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

Finally, we used statistical tests to assess the significance of the achieved results. In particular, since we used 10-fold cross validation, we consider the accuracy achieved on each fold by all the models. We used the Wilcoxon test [41] (with $\alpha = 0.05$) in order to estimate whether there are statistically significant differences between the classification accuracy obtained by $R\langle TF\rangle$ and the other models. Our decision for using the Wilcoxon test, is a consequence of the usage of the 10-fold cross validation to gather the accuracy measurements. During the cross-validation, each fold is selected randomly, but we used the same seed to have the same folds for all the experiments. For example, the 5$^{\text{th}}$ testing fold used for $R\langle BWF\rangle$ is equal to the 5$^{\text{th}}$ testing fold used with *All-features*. Consequently, the pairwise comparisons are performeed between related samples. Moreover, because we performed multiple pairwise comparisons (*i.e., All-features* vs. the rest), we adjusted our $p$-values using the Holm's correction procedure [42]. 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 [43]. 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 $d \geq 0.474$ [43].

## V. Analysis of the results

In this section we analyze the obtained results aiming at answering the research questions in our study.

### A. ***RQ1**: complementarity of readability features*

Table III reports the overlap metrics computed between the results of the readability models using textual and structural features. Across the three datasets, the $R\langle TF\rangle$ model exhibits an overlap of code snippets correctly classified as readable/non-readable included between 62% ($R\langle TF\rangle \cap R\langle PF\rangle$) and 71% ($R\langle TF\rangle \cap R\langle DF\rangle$). This means that, despite the competitive model considered, almost 30% of the code snippets are differently assessed as readable/non-readable when only relying on textual features. Indeed, (i) between 12% ($R\langle TF\rangle \setminus R\langle DF\rangle$) and 21% ($R\langle TF\rangle \setminus R\langle PF\rangle$) of code snippets are correctly classified only by $R\langle TF\rangle$ and (ii) between 17% ($R\langle PF\rangle \setminus R\langle TF\rangle$) and 18% ($R\langle BWF\rangle \setminus R\langle TF\rangle$) 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 Fig. 3. Such a method (considered "unreadable" by human annotators) has a pretty high average textual coherence (0.58), but, above all, it has a high comment readability and comment-identifiers consistency, *i.e.,* many terms co-occur in identifiers and comments (*e.g.,* "batch" and "fetch"). Nevertheless, some lines are too long, resulting in a high maximum and average line length (146 and 57.3, respectively), both impacting negatively the perceived readability [8].

Fig. 4 reports, instead, a code snippet correctly classified as "readable" only when exploiting textual features. The snippet has suboptimal structural characteristics, such as a high average/maximum line length (65.4 and 193, respectively) and a high average number of identifiers (2.7), both negatively correlated with readability. Nevertheless, the method has high average textual coherence ($\sim$ 0.73) and high comments readability (100.0). The source code can be read almost as natural language text and the semantic of each line is pretty clear, but such an aspect is completely ignored by structural features.

**Summary for RQ**$_1$**.** A code readability model solely relying on textual features exhibits a high degree of complementarity with models mainly exploiting structural feature. On average,

TABLE III
**RQ**$_1$: OVERLAP BETWEEN $TFM$ AND $BWM$, $PM$, AND $DM$.

| Dataset | $TFM \cap BWM$ | $TFM \setminus BWM$ | $BWM \setminus TFM$ | $TFM \cap PM$ | $TFM \setminus PM$ | $PM \setminus TFM$ | $TFM \cap DM$ | $TFM \setminus DM$ | $DM \setminus TFM$ |
|---|---|---|---|---|---|---|---|---|---|
| $D_{b\&w}$ | 76% | 14% | 10% | 73% | 8% | 19% | 72% | 14% | 13% |
| $D_{dorn}$ | 69% | 16% | 15% | 63% | 16% | 21% | 74% | 14% | 12% |
| $D_{new}$ | 54% | 24% | 22% | 55% | 21% | 24% | 66% | 22% | 12% |
| **Overall** | **66%** | **18%** | **16%** | **62%** | **17%** | **21%** | **71%** | **17%** | **12%** |

```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);
        }
    }

}
```

Fig. 3. 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);
        }
    }
}
```

Fig. 4. Code snippets correctly classified as "readable" **only** when relying on **textual features** and missed by the competitive techniques.

the readability of 12%-21% code snippets is correctly assessed only when using textual features.

### B. ***RQ2***: *accuracy of readability model*

Table IV shows the accuracy achieved by (i) the comprehensive readability model, namely the model which exploits both structural and textual features (*All-Features*), (ii) the model solely exploiting textual features (*TF*), and (iii) the three state-of-the-art models mainly based on structural features (*BWF*, *PF*, and *DF*).

TABLE IV
**RQ**$_2$: AVERAGE ACCURACY ACHIEVED BY THE READABILITY MODELS IN THE THREE DATASETS AND CONSIDERING ALL THE SNIPPETS.

| Dataset | Snippets | $R\langle BWF\rangle$ | $R\langle PF\rangle$ | $R\langle DF\rangle$ | $R\langle TF\rangle$ | $R\langle$ *All-features* $\rangle$ |
|---|---|---|---|---|---|---|
| $D_{b\&w}$ | 100 | 81.0% | 78.0% | 80.0% | 74.0% | 79.0% |
| $D_{dorn}$ | 360 | 78.6% | 72.8% | 80.0% | 77.2% | 83.9% |
| $D_{new}$ | 200 | 70.5% | 66.0% | 75.5% | 68.0% | 79.5% |
| **Overall** | **660** | **76.5%** | **71.5%** | **78.6%** | **73.9%** | **81.8%** |

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

On the other hand, if we use a model which combines all the features, we obtain an overall accuracy (*i.e.,* using all the accuracy samples as a single dataset) higher than all the compared models (from 3.2% with respect to $R\langle DF\rangle$ to 10.3% with respect to $R\langle PF\rangle$). On the dataset defined by Buse and Weimer, the combined model achieves an accuracy lower than $R\langle DF\rangle$ and $R\langle PF\rangle$. This result could be a consequence of two characteristics of the snippets used for such a dataset: (i) their size is very limited, thus excluding snippets with large comments; (ii) only few snippets are syntactically correct, thus features like "Textual coherence" cannot be computed. Such a limitation is more clear if we look at the accuracy achieved by $R\langle TF\rangle$: while, on average, such a model achieves an accuracy higher than the one achieved by $R\langle PF\rangle$, in this case it is not true, and, instead, $R\langle PF\rangle$ achieves a higher accuracy.

Table V 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 (*i.e.,* $d \geq 0.33$) in most of the comparisons. This issue of no statistical significance with large effect size is an artifact of the size of the samples used with the test, which has been reported previously by Cohen [44] and Harlow *et al.* [45]; in fact, the size of the samples used in our tests for each dataset is 10 measurements (note that we performed 10-fold cross validation). In that sense, we prefer to draw conclusions (conservatively) from the tests performed on the set $D_{all}$, which has a larger sample (30 measurements). When using

TABLE V

$\mathbf{RQ_2}$: P-VALUES (CORRECTED WITH THE HOLM PROCEDURE) OF THE WILCOXON TEST AND CLIFF'S DELTA ($d$), FOR THE PAIRWISE COMPARISONS BETWEEN $R\langle$ *All-features* $\rangle$ AND EACH ONE OF STATE-OF-THE-ART MODELS.

| Dataset | $R\langle BWF\rangle$ | $R\langle PF\rangle$ | $R\langle DF\rangle$ | $R\langle TF\rangle$ |
|---|---|---|---|---|
| $D_{b\&w}$ | 1 ($d = -0.08$) | 1($d = 0.06$) | 1($d = -0.02$) | 0.36($d = 0.22$) |
| $D_{dorn}$ | 0.13($d = 0.52$) | 0.10($d = 0.75$) | 0.13($d = 0.33$) | 0.10($d = 0.55$) |
| $D_{new}$ | 0.10($d = 0.45$) | 0.06($d = 0.58$) | 0.38($d = 0.24$) | 0.10($d = 0.58$) |
| $D_{all}$ | 0.19($d = 0.27$) | 0.01($d = 0.45$) | 0.36($d = 0.19$) | 0.00($d = 0.43$) |

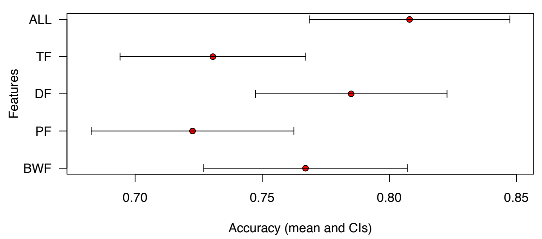


Fig. 5. Mean accuracy and confidence intervals (CIs) with 95% of confidence for each one of the models analyzed for $\mathbf{RQ_2}$

TABLE VI

ACCURACY ACHIEVED BY *All-Features*, *TF*, *BWF*, *PF*, AND *DF* IN THE THREE DATA SETS WITH DIFFERENT MACHINE LEARNING TECHNIQUES.

| | *ML Technique* | *BWF* | *PF* | *DF* | *TF* | *All-Features* |
|---|---|---|---|---|---|---|
| $D_{b\&w}$ | **BayesNet** | 76.0% | 76.0% | 67.0% | 53.0% | 72.0% |
| | **ML Perceptron** | 76.0% | 77.0% | 72.0% | 77.0% | 73.0% |
| | **SMO** | 82.0% | 77.0% | 77.0% | 72.0% | 81.0% |
| | **RandomForest** | 80.0% | 77.0% | 75.0% | 72.0% | 77.0% |
| $D_{dorn}$ | **BayesNet** | 75.0% | 68.1% | 74.7% | 68.1% | 76.1% |
| | **ML Perceptron** | 74.2% | 70.3% | 72.5% | 74.2% | 77.8% |
| | **SMO** | 79.7% | 71.1% | 76.7% | 71.7% | 80.6% |
| | **RandomForest** | 78.1% | 70.3% | 74.4% | 74.2% | 78.9% |
| $D_{new}$ | **BayesNet** | 62.5% | 69.5% | 64.0% | 64.0% | 71.0% |
| | **ML Perceptron** | 66.5% | 65.5% | 68.5% | 66.5% | 70.0% |
| | **SMO** | 67.0% | 68.0% | 72.5% | 65.0% | 77.0% |
| | **RandomForest** | 68.0% | 60.5% | 69.0% | 65.5% | 69.0% |

the datasets as a single one (*i.e.,* $D_{all}$), there is significant difference in the accuracy when comparing $R\langle$*All-features*$\rangle$ to $R\langle PF\rangle$ and $R\langle TF\rangle$; the results are confirmed with the Cliff's $d$ that suggest a medium-large difference (*i.e.,* $d \geq 0.4$) in both cases.

Fig. 5 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-features*$\rangle$ is larger than $R\langle PF\rangle$ and $R\langle TF\rangle$ (*i.e.,* there is no overlap between the CIs). Although the mean accuracy of $R\langle$*All-features*$\rangle$ is the largest one in the study, there is an overlap with the CIs for $R\langle BWF\rangle$ and $R\langle DF\rangle$. 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 ones used in the Posnett et al. model. The statistical tests also confirm that using only textual features is not the best choice for a code readability model.

**Summary for $\mathbf{RQ_2}$.** 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 3.9% and, the difference is statistically significant when compared to the model based on the features by Posnett et al.

## VI. THREATS TO VALIDITY

Possible threats to validity 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. In this section we discuss such threats, grouping them into *construct*, *internal* and *external* validity.

**Construct Validity.** The main threat is the choice of a proper metric for evaluating the models. We used the accuracy achieved when using *logistic regression* as the underlying classifier for the readability models, however, we could have used other metrics (*e.g.,* AUC) or machine learning techniques (*e.g.,* BayesNet or neural networks). We chose accuracy because it is widely used in the literature [46], and in particular for readability metrics [8]. In addition, the results could depend on the machine learning technique used for computing the accuracy of each model. Table VI 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; the combined model achieves a better accuracy than any other model on the new data set and on the one defined by Dorn, while $R\langle BWF\rangle$ outperforms the other models on the data set defined by Buse and Weimer.

**Internal validity.** To mitigate the over-fitting problem of machine learning techniques, we used 10-fold cross-validation, and we performed statistical analysis (Wilcoxon test, effect size, and confidence intervals) in order to measure the significance of the differences among the accuracies of different models. Also, feature selection could affect the final results on each model. Finding the best set of features in terms of achieved accuracy is infeasible when the number of features is large. Indeed, the number of subsets of a set of $n$ elements is $2^n$; while an exhaustive search is possible for models with a limited number of features, like *BWF*, *PF* and *TF*, it is unacceptable for *DF* and *All-Features*. Such a search would require, respectively, $1.2 \times 10^{18}$ and $3.2 \times 10^{34}$ subset evaluations. Thus, we used a linear forward selection technique [40] in order to reduce the number of evaluations and to obtain a good subset in a reasonable time. Comparing models obtained with exhaustive search to models obtained with a sub-optimal search technique could lead to biased results; therefore, we use the same feature selection technique for all the models to perform a fairer comparison. It is worth noting that the likelihood of finding the best subset remains higher for models with less features.

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

## VII. Conclusion and Future Work

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 (*i.e.,* 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 (i) to what extent the proposed textual-based features complement the structural features proposed in the literature for predicting code readability, and (ii) the accuracy achieved when including textual features into the state-of-the-art models. Our findings show that textual features complement structural ones, and the combination (*i.e.,* structural+textual) improves the accuracy of code readability models. Our future work will focus on designing more advanced textual features and identifying whether the proposed features can be used for defect prediction.

## References

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

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

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

[4] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, May 2000.

[5] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 73–87.

[6] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Softw.*, vol. 21, no. 4, pp. 62–69, Jul. 2004.

[7] D. Poshyvanyk and D. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *ICPC'07*, 2007.

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

[9] D. Posnett, A. Hindle, and P. T. Devanbu, "A simpler model of software readability." in *MSR'11*, 2011, pp. 73–82.

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

[11] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory." *ISSE*, vol. 3, no. 4, pp. 303–318, 2007.

[12] ——, "What's in a name? a study of identifiers," in *ICPC'06*, 2006.

[13] B. Caprile and P. Tonella, "Restructuring program identifier names," in *ICSM*, 2000, pp. 97–107.

[14] F. Deissenbock and M. Pizka, "Concise and consistent naming," 2005.

[15] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *SCAM'06*, 2006, pp. 139–148.

[16] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *MSR'09*.

[17] A. Takang, P. Grubb, and R. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[18] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Appendix: Improving code readability models with textual features." [Online]. Available: https://dibt.unimol.it/icpc2016/appendix.html

[19] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *ESEC/FSE'15*, 2015, pp. 107–118.

[20] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," vol. 34, no. 2, pp. 287–300, 2008.

[21] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *ICSM'06*, 2006, pp. 469–478.

[22] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE TSE*, vol. 28, no. 10, pp. 970–983, 2002.

[23] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *CACM*, vol. 25, no. 8, pp. 512–521, 1982.

[24] T. Tenny, "Program readability: procedures versus comments," *IEEE TSE*, vol. 14, no. 9, pp. 1271–1279, 1988.

[25] D. Spinellis, *Code Quality: The Open Source Perspective*. Adobe Press.

[26] D. Binkley, H. Feild, D. J. Lawrie, and M. Pighin, "Increasing diversity: Natural language measures for software fault prediction." *Journal of Systems and Software*, vol. 82, no. 11, pp. 1793–1803, 2009.

[27] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[28] B. Fluri, M. Würsch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *WCRE'07*.

[29] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *ASE'15*, 2015.

[30] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. Kraft, "Automatically documenting unit test cases," in *ICST'16*, 2016.

[31] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[32] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *ICPC'08*, 2008, pp. 113–122.

[33] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To CamelCase or Under score," in *ICPC'09*, 2009.

[34] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *EMSE*, vol. 19, no. 5, pp. 1383–1420, 2014.

[35] V. Arnaoudova, L. M. Eshkevari, R. Oliveto, Y. Guéhéneuc, and G. Antoniol, "Physical and conceptual identifier dispersion: Measures and relation to fault proneness," in *ICSM'10*, 2010, pp. 1–5.

[36] G. A. Miller, "Wordnet: A lexical database for english," vol. 38, no. 11, pp. 39–41, 1995.

[37] R. Flesch, "A new readability yardstick." *Journal of applied psychology*, vol. 32, no. 3, p. 221, 1948.

[38] Collins american dictionary. [Online]. Available: http://www.collinsdictionary.com/dictionary/american/syllable

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

[40] M. Gütlein, E. Frank, M. Hall, and A. Karwath, "Large-scale attribute selection using wrappers," in *CIDM'09*, 2009, pp. 332–339.

[41] S. D.J., *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

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

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

[44] J. Cohen, "The earth is round (p<.05)," *American Psychologist*, vol. 49, no. 12, pp. 997–1003, 1994.

[45] L. L. Harlow, S. A. Mulaik, and J. H. Steiger, *What if there were no significance tests?* Psychology Press, 1997.

[46] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *TSE*, vol. 34, no. 2, pp. 181–196, 2008.