# The model of code readability features: visual, structural and textual

Ivana Zeljković, Jelena Slivka, Goran Savić, Milan Segedinac
Faculty of Technical Sciences/Department of computing and control engineering, Novi Sad, Serbia
ivana.zeljkovic@uns.ac.rs, slivkaje@uns.ac.rs, savicg@uns.ac.rs, milansegedinac@uns.ac.rs

***Abstract* — This paper describes the model that captures a variety of features important for code readability: visual, structural and textual. Code readability can be defined as a measure of how easy it is to understand the logical context of the source code, work on it collaboratively and maintain the same. Successful classification of code as readable or unreadable is a prevalent problem in today AI's world. By the automatic discovery of unreadable code, we could significantly reduce the time needed for software development and maintenance, enforce best practices and potentially discover bugs in the code. Current solutions for the measurement of code readability are still unsatisfactory from the aspect of accuracy. To build an accurate code readability model, an appropriate dataset is needed. All existing solutions use a set of structural and/or textual features, but none of them use a visual component. Accordingly, this work represents an improvement in the described problem, introducing visual component. The visual features in our model serve to express the visual focus of the person while reading a piece of code with the hypothesis that a person will pay more attention to the more complex and penitentially less readable parts of the programming code. On the other side, the structural features are used to express the key programming concepts of the target programming language and describe the impact of code's general structure on its readability, as well as the impact of some rarely used concepts in contrast to often used ones. Finally, the textual features, extracted from comments and identifiers, describe the semantics of software's logic and in that way contribute to a higher degree of code comprehension and readability.**

## I. INTRODUCTION

Keeping code readable is of crucial importance for the success of every software project. Readability is one of the key preconditions for easy and quick understanding of the written code, which is necessary for every phase of the software's life cycle. The degree in which the code satisfies readability criteria determines the speed of development, as well as maintenance of the software [1].

The main goal of the model introduced trough this paper is a formal description of aspects that have the most significant impact on code readability. As such, this model has a dual purpose, in software development, as well as in the educational domain. Regarding software development, this model could be an appropriate base for the development of a tool that measures code readability. The tool like this one with its assessment of code readability would make it easier and faster to write code. Also, its use would enable software engineers the insight into good and bad practices and teach them to organize their code in a better, clearer way. This would also improve the quality of the software by reducing the number of potential bugs and decrease the time it takes to maintain the final software. Another benefit of the model of code readability features would be an improvement in software engineering education. By analyzing the features that turned out to be critical for code readability from the aspect of students, we can gain insight into concepts that require the highest level of student attention, as well as identify concepts that are obstacles to overcome individual problems in understanding code.

The main goal of this work is the formal specification of system for creating the dataset that is used for the development of the code readability classifier. This system relies on three main components: (1) visual data extractor, used for extracting visual features by filtering the data that the eye tracker device collects while a person reads a programming code; (2) static analyzer, used for extracting structural features; (3) textual analyzer, used for extracting textual features.

The first section, *Structural component*, describes a set of structural features considered in our model of code readability features, as well as how they are extracted from code snippets. In the second section, *Textual component*, is given a description of code readability features, analyzed from the textual aspect. The third section, *Visual component*, represents the novelty introduced through this work, which gives a detailed description of visual features. The last section gives a short review of existing models of code readability features and also describes future work.

## II. STRUCTURAL COMPONENT

The structural features are used to describe how the source code is constructed, i.e., which particular concepts from a set of key concepts of the target programming language are used. Most of the programming languages have a lot of concepts in common, but there are also structural and semantic peculiarities characteristic to every particular programming language.

Structural features are important when assessing the readability of the source code. For example, the *try-catch* block is a prevalent concept in languages such as *Java* or *C#*, but not in *Python*, where *catch* keyword is replaced with *except* keyword. Also, one of the concepts in *Python*, that, based on our previous educational experience, is relatively rarely used and a potential reason for the confusion is *for* loop with *else* clause. Programmers new to *Python* think that this structure is wrong from syntax aspect, which is why they tend to assess examples with this structure as unreadable. Another example is the importance of the code layout. The example that illustrates this in the best way is a well known *if-else* structure, which can be written using an inline or block layout (Fig. 1 and Fig. 2). Block layout is used in 90% of the time, while inline layout (ternary operator) is used only in 10% of the time. The reason for this is that the block layout shows in

```python
def func(words):
    words_count = {}
    for word in words:
        words_count[word] = 1 if word not in words_count else words_count[word]+1
```

Figure 1: Inline layout (ternary operator) of *if-else* structure in *Python*

```python
def func(l1):
    all = {}
    for el_1 in l1:
        if not(el_1 in all):
            all[el_1] = 1
        else:
            all[el_1] = all[el_1] + 1
```

Figure 2: Block layout of *if-else* structure in *Python*

a very clear way the condition of branching and operations that should be executed in every single branch. In contrast, a ternary operator takes programmers more time for recognizing condition and actions per every branch.

The set of structural features that our model captures represents a modified set of structural features given in [2]. In [2] target programming language was *Java*, which is the main reason why some of these features are not considered in our model since the model described through this paper is developed for snippets written in *Python* programming language. From the structural model in [2], several irrelevant features are removed: the average number of commas and spaces per code line, the maximal number of occurrences of any character and feature that describes indentation (preceding whitespace). The main reason why these features are not considered in the described model is their low impact on code readability assessment, proven in [2].

On the other side, some new features, that complement a set of concepts characteristic for *Python*, are added. Table 1 shows all features that structural component includes. The first column in the table describes features, as well as a way how they are determined (number of characters; observing each line of code or scope of the entire snippet). All new structural features, introduced through this paper, are marked with * in front of their name. Other features (without * in the name) are references from [2]. For each feature is marked which value is calculated, average and/or maximal. For features which values are determined based on the entire snippet (e.g., number of if constructions, number of loops…), only maximal value is calculated.

The module used for static code analysis performs an analysis of the *Abstract Syntax Tree* (AST) of the input code, considering the set of *Python*'s concepts needed to get information about the values of defined structural features. A code snippet that represents the input in this module can be any syntactically correct *Python* code. The output of this module is a static report that includes all structural features in our model with concrete values that uniquely describe the analyzed snippet. Furthermore, this report can be used for generating an observation that represents corresponding *Python* snippet in dataset instance that is based on this feature model.

## III. TEXTUAL COMPONENT

The textual features capture the domain semantics of the software and add a new layer of semantic information to the source code, in addition to the programming language semantics [2]. Lines of code with comments that describe their purpose using natural language, which is more understandable to human (Fig. 3 and Fig 4), as well as carefully chosen identifiers (Fig. 5 and Fig. 6) could improve code understanding, even though it is one of the less readable examples, from the structural point of view.

In contrast, poorly defined identifiers (e.g., meaningless abbreviations or identifiers that contain only one letter) or inconsistent identifiers can cause wrong understanding of code and thus result in a change of assessment of its readability, from a human perspective. Another perspective for analyzing the impact of natural language on code readability is the degree of how specific terms (words that are used in identifiers) are. For example, if the term has multiple meanings, this can be the reason of misunderstanding of code's purpose, but, in case a term has a very specific meaning, the difference between the programmers intended meaning and cognitive model of someone who reads the code is minimal.

```python
def func(number):
    """Make list of squares of integer values from specified range"""
    result_list = []

    number <<= 2 >> 1
    for num in range(1, number):
        result_list.append(num*num)

    return result_list[:3]

result = func(4)
```

Figure 3: Snippet example written with a comment that describes the purpose of the code

```python
def func(limit):
    result_list = []

    limit *= 4 / 2
    numbers = range(1, limit)
    for num in numbers:
        result_list.append(num**2)

    return result_list[0:3]

result_list = func(4)
```

Figure 4: Snippet example written without a comment that describes the purpose of the code

```python
def func(l1):
    all = {}
    for el_1 in l1:
        if not(el_1 in all):
            all[el_1] = 1
        else:
            all[el_1] = all[el_1] + 1

    res = list()
    l = all.items()
    while len(res) <= 3:
        for k, v in l:
            res.append(k*v)

    return res[::-1]

l = func(['a', 'b', 'ab', 'c', 'ab', 'b', 'ab'])
```

Figure 5: Snippet example written with meaningless identifier

```python
def func(words):
    words_count = {}
    for word in words:
        words_count[word] = 1 if word not in words_count else words_count[word]+1

    multiplied_words = list()
    all_words = words_count.items()
    counter = 0
    while counter <= 3:
        for key, value in all_words:
            multiplied_words.append(key*value)
            counter += 1
    else:
        return multiplied_words[::-1]

words = ['b', 'a', 'b', 'ab', 'c', 'ab', 'ab']
result = func(words)
```

Figure 6: Snippet example written with meaningful identifiers

The set of textual features considered by this model is the same as in [2], and it contains the following features:

1. *Comments and identifiers consistency* - represents the overlap between unique terms extracted from comments and unique terms extracted from identifiers. Before calculation, set of terms extracted from comments is expanded with a set of synonyms, for each particular comment term. Synonyms are determined using WordNet [3] corpus.
2. *Identifier terms in the dictionary* - represents the percentage of terms that are full-words, i.e., exist in the appropriate dictionary, e.g., WordNet corpus. Also, the percentage of nonexistent terms is calculated, but here we didn't consider one-letter identifiers.
3. *Narrow meaning identifiers* - represents how specific are terms used in identifiers. The particularity of each term is determined through observing hypernym tree of that term, considering the number of steps necessary to come from the current node (a node that contains concrete term) to the root node in the tree structure. For this purpose, as well as for the previous two features, WordNet corpus is used, since inside it word relations such as hypernyms and hyponyms are defined.
4. *Comments readability* - determines the readability of comments using *Flesch-Kincaid* [4] index, commonly used to assess the readability of natural language texts.
5. *The number of meanings* - determines the polysemy of a snippet, analyzing the number of meanings of terms used in identifiers. This value, the number of meanings, is derived from WordNet corpus since inside it exists at least one definition of each word. From the aspect of this feature, maximal and average values are considered as relevant.
6. *Textual coherence* - represents the number of "concepts" implemented by a source code snippet [2]. To estimate the number of concepts, snippet's AST is used to detect syntactic blocks. Each syntactic block is individually processed to extract a set of unique terms used in identifiers. After that, vocabulary overlap is computed for each pair of blocks. Relevant values from the aspect of this feature are minimal, maximal, as well as average overlap.

The module used for extraction of textual features performs a syntactical analysis of the input code, considering comments and identifiers. Before starting with textual analysis of the code, the input must be preprocessed to extract the textual content of the input code and get single terms prepared for textual analysis. Preprocessing steps are:

- Remove non-textual tokens: programming language keywords, special symbols, and operators;
- Split the remaining tokens into separate words (according to one of two naming conventions - camel case notation and underscore notation);
- Remove stop-words from comments (e.g. articles, adverbs);
- Extract stems from comments words using *Snowball* algorithm [5].

Similar to the output of the first described module, the static analyzer, the output of this module is also in the form of a report that includes all specified textual features with concrete values that uniquely describe the analyzed snippet, from the textual aspect. This report can be used to complement the existing observation that represents the analyzed code in dataset instance with the values that describe snippet from a textual perspective.

## IV. Visual Component

The visual features are important from the aspect of analysis how people read the code: what people consider to be more or less important parts, whether they read keywords, whether they read code from top to bottom or trace actions backward, starting from the end of the code, i.e., function call. During code analysis, the human focus is more often on complex or unreadable parts than on keywords, readable and familiar concepts or structures. An example that explains this claim is the *if-else* construction. In the case of the inline layout (e.g., the code example shown in Fig. 1), keywords are built-in in one line, so that we can't skip them. On the contrary, we read them to determine the boundaries of *if* and *else* branches and their respective actions.

The set of visual features, considered in this model, includes features that describe the path of evaluation of a piece of code by a human. These features express the average and maximal time (in seconds) of observation for all of the considered *Python*'s concepts (identifier, function call, arithmetic/comparison/bitwise/logical operator, assignment/augmented assignment statement, if construction, loop…), that have been already mentioned and explained above, in section about structural features.

Table 1: Structural features

| *Feature* | *Average* | *Maximal* |
|---|---|---|
| * Snippet length (number of lines) | + | + |
| Code line length (number of characters) | + | + |
| Identifier length (number of characters) | + | + |
| Number of identifiers (per code line) | + | + |
| * Number of occurrences of the most frequent identifier (on snippet level) | | + |
| Number of keywords (per code line) | + | + |
| Number of blank lines (on snippet level) | | + |
| Number of comment lines (on snippet level) | | + |
| * Comment length (number of characters) | + | + |
| Number of numerical constants (per code line) | + | + |
| * Number of textual constants (per code line) | + | + |
| Number of parentheses (per code line) | + | + |
| Number of assignments (on snippet level) | | + |
| * Number of augmented assignments (on snippet level) | | + |
| * Number of function calls (per code line) | + | + |
| * Number of occurrences of the most frequent function call (on snippet level) | | + |
| Number of branches (if, elif) (on snippet level) | | + |
| Number of loops (for, while) (on snippet level) | | + |
| * Number of break points (on snippet level) | | + |
| * Number of continue points (on snippet level) | | + |
| * Number of try blocks (on snippet level) | | + |
| * Number of except blocks (on snippet level) | | + |
| * Number of raise expressions (on snippet level) | | + |
| * Number of finally blocks (on snippet level) | | + |
| * Number of return statements (on snippet level) | | + |
| Number of arithmetic operations (per code line) | + | + |
| * Number of logical operations (per code line) | + | + |
| Number of comparison operations (per code line) | + | + |
| * Number of bitwise operations (per code line) | + | + |
| * Number of list comprehensions (on snippet level) | | + |
| * Number of dictionary comprehensions (on snippet level) | | + |
| * Number of set comprehensions (on snippet level) | | + |
| * Number of lambda functions (on snippet level) | | + |
| * Number of identity operators (is, is not) (per code line) | + | + |
| * Number of membership operators (in, not in) (per code line) | + | + |

The visual data extractor is third and the most important component of this model. During the process of the code evaluation by a human, the persons gaze is followed by an auxiliary device, eye tracker. During the recording, this device forms a database with 30 properties that describe the human's gaze and also one video record that shows the entire observed content. We consider 6 out of these 30 properties to be relevant and sufficient to describe the captured gaze, while the rest of them represent more technical information that is irrelevant from the perspective of our model, such as position of mouse cursor, URL of webpage which is observed, diameter of left/right eye pupil in the camera image…

Properties that are used to describe the whole process of eye tracking in this module are:

- FPOGV - flag that indicates if fixation (gaze) is valid or not (fixation is invalid in the case of blinking or fast transition of the view);
- FPOGID - fixation ID number;
- FPOGX - the x coordinate of the fixation, as a percentage of the screen width (0 to 1);
- FPOGY - the y coordinate of the fixation, as a percentage of the screen height (0 to 1);
- FPOGD - the duration of the fixation expressed in seconds;
- TIME - timestamp of the fixation expressed in seconds.

Firstly, the module filters valid fixations according to appropriate flag (FPOGV) from the resulting database. Next step is extraction of the period of observation (from the properties *timestamp from* and *timestamp to*) for each concept that has been observed. This report is created by the visual data extractor by using two inputs, that represent products of recording with the eye tracker device:

- filtered database (only valid fixations with a timestamp in a given range are kept)
- video record.

The video record is used to extract the frames from the defined time period so that fixations from the database can be mapped on particular content shown on the screen. The video frame is processed using computer vision techniques to get the map that describes relative coordinates of boundaries (minimum and maximum x coordinate, minimum and maximum y coordinate) of every code line in the code shown on the video frame. Further, every fixation is mapped on the appropriate concept in the code, using its FPOGX and FPOGY values, as well as the previously mentioned map, that describes the boundaries of every line in the code.

The final output of visual extractor is the structure that for every considered *Python*'s concept stores the information about average and maximal observation time, expressed in seconds.

## V. Conclusion

The most significant results of code readability classification/measurement have been achieved and described in [2] and [6]. In [6] authors created a training dataset, which includes only the structural component, i.e., features obtained by static analysis of the observed codes. In [2], authors have significantly improved on [6] by creating a new dataset that, in addition to structural features, introduces new, textual component, i.e., features that are related to the textual aspect of the code. This extension of training dataset added a new dimension to the analysis of readability for different codes and significantly improved classification performance.

This paper extends the previously mentioned models by adding a visual component to already defined structural and textual components, that are used in datasets on which these models are trained. Accordingly, the topic of this work is creating a model that besides already mentioned components also includes a visual one, i.e., features that describe patterns of human attention when evaluating codes. This could be an improvement that can contribute to a more in-depth understanding of how the code readability is assessed and potentially in higher accuracy of code readability classifiers.

Apart from the introduction of a new aspect in the code readability analysis process, this work also represents an extension and improvement regarding the number of concepts in the target programming language, that are taken into consideration during static code analysis to define the structural features.

The future work is the development of the automatic code readability classifier, that is trained on the dataset based on the described model, as well as evaluating the significance of the visual component. Also, in case that our classification model shows better performance than current state-of-the-art models, next step in the development would be the extension of the model of code readability features so that it supports other commonly used programming languages because its current version is limited to only one programming language, *Python.*

## REFERENCES

1. Collar Jr, E. and Valerdi, R., “Role of software readability on software development cost.”, 2006.
2. Scalabrino, S., Linares-Vasquez, M., Oliveto, R. and Poshyvanyk, D., “A comprehensive model for code readability”, *Journal of Software: Evolution and Process*, 30(6), p.e1958., 2018.
3. Miller, G.A., “WordNet: a lexical database for English”, *Communications of the ACM*, 38(11), pp.39-41., 1995.
4. Flesch, R., “A new readability yardstick*”, Journal of applied psychology*, 32(3), p.221., 1948.
5. Snowball stemming algorithm documentation, [Online]. Available: http://snowball.tartarus.org/
6. Buse, R.P. and Weimer, W.R., “Learning a metric for code readability”, *IEEE Transactions on Software Engineering*, 36(4), pp.546-558., 2010.