Advancing Code Readability: Mined & Modified Code for Dataset Generation

Lukas Krodinger

Master Thesis in M.Sc. Computer Science
Faculty of Computer Science and Mathematics
Chair of Software Engineering II

Matriculation number	89801
Supervisor	Prof. Dr. Gordon Fraser
Advisor	Lisa Griebl

19th November 2023

Abstract

This work presents an innovative method for generating datasets intended for code readability classification in the context of a master's thesis. We offer a comprehensive overview of code readability, delving into existing classifiers that consider both manually crafted and automatically extracted code readability features. Furthermore, we summarize existing datasets of manually annotated Java code snippets.

The core contribution of this work lies in the introduction of an automatic data generation technique, alongside a dataset produced using this methodology. Our approach relies on the extraction and modification of code snippets sourced from public GitHub repositories. Notably, our dataset significantly surpasses the scale of any previously available dataset designed for readability classification.

To evaluate the newly generated dataset, we conducted a user survey and train a state of the art code readability classification model, both with and without the integration of the new dataset. This analysis aims to assess the quality of our new dataset and the effectiveness of the generation approach employed.

Contents

1 INTRODUCTION

George Klare (1963) defines readability as "the ease of understanding or comprehension due to the style of writing." This definition focuses on writing style as separate from issues such as content, coherence, and organization. In a similar manner, Gretchen Hargis and her colleagues at IBM (1998) state that readability, the "ease of reading words and sentences," is an attribute of clarity. The creator of the SMOG readability formula G. Harry McLaughlin (1969) defines readability as: "the degree to which a given class of people find certain reading matter compelling and comprehensible." This definition stresses the interaction between the text and a class of readers of known characteristics such as reading skill, prior knowledge, and motivation. Edgar Dale and Jeanne Chall's (1949) definition may be the most comprehensive: "The sum total (including all the interactions) of all those elements within a given piece of printed material that affect the success a group of readers have with it. The success is the extent to which they understand it, read it at an optimal speed, and find it interesting." [**dubay2004principles**]

1

WE define readability as a human judgment of how easy a text is to understand. The readability of a program is related to its maintainability, and is thus a key factor in overall software quality. Typically, maintenance will consume over 70a software product [4]. Aggarwal claims that source code readability and documentation readability are both critical to the maintainability of a project [1]. Other researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities [8], [32], [34]. Readability is so significant, in fact, that Elshoff and Marcotty, after recognizing that many commercial programs were much more difficult to read than necessary, proposed adding a development phase in which the program is made more readable [10]. Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability [21] to ensure maintainability, portability, and reusability of the code. Haneef proposed adding a dedicated readability and documentation group to the development team, observing that, "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [15]. [6]

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, 3, 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.[20]

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 (e.g., syntax highlighting) have been widely recognized as elements that impact program understanding [5, 6, 7]. [20]

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 [20]

Readability is one important quality attributes for software source codes. Readability has also significant relation or impact with other quality attributes such as: reusability, maintainability, reliability, complexity, and portability metrics.[**tashtoush2013impact**]

Code Readability can be defined as a human judgment on how easy it is to understand a program source code [1], the ratio between lines of code and number of commented lines [2], writing to people not to computers [3], making a code locally understandable without searching for declarations and definitions [4], and also, the average number of right answers to a series of questions about a program in a given length of time [5]. Definitions show clearly that such quality attribute is related to some other attributes such as: understandability, usability, reusability, complexity, or maintainability. Code readability is usually connected with comments and naming standards. While those are two major factors that impact readability, there are also some other aspects to consider. One of the new concepts that appear in relation with software quality and readability is the refactoring process. It means changing code structure externally without affecting internal behavior to improve readability, flexibility, and enable easier modifications [12-14]. Refactoring often aims to add new non functional objectives for code, without affecting its main purpose by testing the code frequently. In [15] the author defines "improving code readability" as the first advantage for code refactoring. The survey briefly explains code refactoring techniques such as: field encapsulation, type generalization, or methods renaming. Refactoring advantages include: improving readability, maintainability, less complexity, and reusability. [**tashtoush2013impact**]

A substantial portion of the cost of software during its life cycle is consumed not in its development, but in its ongoing maintenance. One of the factors that leads to improved code maintainability is its readability. When code is difficult to read, it is difficult for subsequent developers to understand its flow and its side effects, and they are likely to introduce new bugs while trying to fix old ones or while extending the code's original functionality. But how do software developers know they have written readable code? [**sedano2016code**]

code readability, i.e., what makes a program easier or harder to read and apprehend by developers [**oliveira2020evaluating**]

In software engineering, the terms readability, legibility, understandability, and comprehensibility have overlapping meanings. For example, Buse and Weimer

[22] define "readability as a human judgment of how easy a text is to understand". In a similar vein, Almeida et al. [23] affirm that "legibility is fundamental to code maintenance; if source code is written in a complex way, understanding it will require much more effort". In addition, Lin and Wu [24] state that ""Software understandability" determines whether a system can be understood by other individuals easily, or whether artifacts of one system can be easily understood by other individuals". Xia et al. [25] treat comprehension and understanding as synonyms, expressing that "Program comprehension (aka., program understanding, or source code comprehension) is a process where developers actively acquire knowledge about a software system by exploring and searching software artifacts, and reading relevant source code and/or documentation".[**oliveira2020evaluating**]

In the realm of software development, the significance of code readability cannot be overstated. Together with understandability, it serves as the foundation for efficient collaboration, comprehension, and maintenance of software systems [18, 1]. Maintenance alone will consume over 70% of the total lifecycle cost of a software product and for maintenance, the most time-consuming act is reading code [6, 9, 19, 4]. Therefore, it is important to ensure a high readability of code. In order to archive this, we need to measure readability.

In the last years, researchers have proposed several metrics and models for assessing code readability with an accuracy of up to 81.8% [6, 18, 10, 8]. In recent years, deep learning based models are able to achieve an accuracy of up to 85.3% [16, 17]. However, these models do not capture what developers think of readability improvements [11]. This suggests that there is room for improvement in readability classification of source code.

2 BACKGROUND AND RELATED WORK

2.1 READABILITY

Code readability is a human judgment of how easy a program source code is to understand [**buse2008evaluating**]. It is a subjective concept that is influenced by a number of factors, including the complexity of the code, the usage of design concepts, the formatting of the code, the source code lexicon, and the visual aspects of the code [20].

Code readability is important because it has a significant impact on other quality attributes of software, such as maintainability, reusability, reliability, and portability [**tashtoush2013impact**]. When code is readable, it is easier for developers to understand and make changes to, which can lead to reduced maintenance costs and improved software quality. Readable code is also more likely to be re-

used by other developers, which can save time and effort. Additionally, readable code is more likely to be reliable, as it is easier for developers to identify and fix bugs.

There are a number of different definitions of code readability in the literature. Some of these definitions are:

"Code readability is the ease of understanding a program source code." [**buse2008evaluating**] "Code readability is the ratio between lines of code and number of commented lines." [**sedano2016code**] "Code readability is writing to people, not to computers." [**oliveira2020evaluating**] "Code readability is making a code locally understandable without searching for declarations and definitions." [**oliveira2020evaluating**] "Code readability is the average number of right answers to a series of questions about a program in a given length of time." [**oliveira2020evaluating**] While these definitions vary, they all share the common theme of measuring how easy it is for humans to understand the code.

Differences between the terms readability and understandability in code

Readability is a human judgment of how easy a program source code is to read and comprehend [**buse2008evaluating**, **sedano2016code**]. It is concerned with the syntactic aspects of code, such as the use of meaningful variable names, consistent formatting, and clear commenting.

Understandability is the ability to grasp the meaning of a program source code and how it works [**oliveira2020evaluating**, 18]. It is concerned with the semantic aspects of code, such as the underlying logic and the use of design patterns.

In other words, readability is about how easy it is to read code, while understandability is about how easy it is to understand code.

Code readability is important for a number of reasons, including:

Maintainability: Readable code is easier to maintain, as it is easier for developers to understand and make changes to [citebuse2008evaluating, [20]]. This can lead to reduced maintenance costs and improved software quality. Reusability: Readable code is more likely to be reused by other developers, as it is easier to understand how to use it [citeoliveira2020evaluating]. This can save time and effort. Reliability: Readable code is more likely to be reliable, as it is easier for developers to identify and fix bugs [citebuse2008evaluating]. Productivity: Readable code can help developers to be more productive, as they spend less time trying to understand and debug the code [citescalabrino2018comprehensive]. In addition, code readability can also have a positive impact on other quality attributes of software, such as portability and testability.

To improve readability classification, we need to capture what readability is. We define readability as a subjective impression of the difficulty of code while trying to understand it [18, 6]. Readability of code is a perceived barrier that needs to be overcome before it is possible to work with the code. The more readable code is, the lower the barrier [18]. To give an example for high vs low readability, consider the code of listing 1 from GitHub[1] and compare it to the code with the same functionality of listing 2. You will notice that the first piece of code is more readable than the second one.

Readability is not the same as complexity. Complexity is an "essential" property of software that arises from system requirements, while readability is an "accidental" property that is not determined by the problem statement [6, 5].

There is another related term: understandability. Readability is the syntactic aspect of processing code, while understandability is the semantic aspect [18]. For example, a developer can find a piece of code readable but still difficult to understand. Recent research gives evidence that there is no correlation between understandability and readability [21].

```
1   /**
2    * Logs the output of the specified process.
3    *
4    * @param p the process
5    * @throws IOException if an I/O problem occurs
6    */
7   private static void logProcessOutput(Process p) throws IOException
8   {
9           try (BufferedReader input = new BufferedReader(new
        ↪  InputStreamReader(p.getInputStream()))))
10          {
11                  StrBuilder builder = new StrBuilder();
12                  String line;
13                  while ((line = input.readLine()) != null)
14                  {
15                          builder.appendln(line);
16                  }
17                  logger.info(builder.toString());
18          }
19  }
```

Listing 1: An example for well readable code of the highly rated Cassandra GitHub repository

```
1   private
2           static
3   void
4   debug( Process
5   v1
6   )       throws IOException
7   {
8           // Doo debug
9           try (BufferedReader    b
10          = new
11          BufferedReader(
12          new InputStreamReader(
13          v1.getInputStream()
14          )
15          )
16          )
17          {
18                  StrBuilder b2=new StrBuilder();String v2;while
            ↪   (null!=(v2=input.readLine())){b2.appendln(v2);}
            ↪   // Doo stuff
19                  m.info(  builder.toString()
20                  );
21          }
22  }
```

Listing 2: The same example as in listing 1 but modified to be poorly readable

2.2 CLASSICAL CALCULATION APPROACHES

A first estimation for source code readability was the percentage of comment lines over total code lines [1]. In the last years, researchers have proposed several more complex metrics and models for assessing code readability [6, 18, 10, 20]. Those approaches used handcrafted features to calculate how readable a piece of code is. They were able to achieve up to 81.8% accuracy in classification [20].

2.3 DEEP LEARNING BASED APPROACHES

More recent models use Deep Learning approaches in order to generate the features automatically. Those models have proven to be more accurate, achieving an accuracy of up to 85.3% [16, 17].

All the mentioned models were trained on the data of Buse, Dorn and Scalabrino consisting of in total 660 code snippets. The data was generated with surveys. They therefore asked developers several questions, including the question, how well readable the proposed source code is [6, 10, 20].

Fakhoury et al. showed based on readability improving commit analysis that these models do not capture what developers think of readability improvements. They therefore analyzed 548 GitHub[2] commits manually. They suggest considering other metrics such as incoming method calls or method name fitting [11].

2.4 RELATED WORK

Loriot et al. created a model that is able to fix Checkstyle[3] violations using Deep Learning. They inserted formatting violations based on a project specific format checker ruleset into code in a first step. They then used a LSTM neural network that learned how to undo those injections. Their approach is working on abstract token sequences. Their data is generated in a self-supervised manner [15]. A similar idea has been explored by Yasunaga and Liang [22]. We will use the idea of intentional degradation of code for data generation.

Another concept we will employ is from Allamanis et al. They cloned the top open source Java projects on GitHub[2] for training a Deep Learning model. Those top projects were selected by taking the sum of the z-scores of the number of watchers and forks of each project. As the projects have thousands of forks and stars and are widely used among software developers, they can be assumed to be of high quality [2].

3 DATASET GENERATION APPROACH

4 READABILITY CLASSIFICATION MODEL

We will investigate whether it is possible to score a higher accuracy as current models in classifying code readability for Java using Deep Learning. Therefore, we will train the model from Mi et al. [17] with more data. We will consider augmenting the model with a method name classifier and incorporating semantic encoding for tabs and spaces. The training data will be generated in a novel way for classification of readability, inspired by Loriot et al. [15]. The method name classifier is similar to Code2Vec [3]. The combination of all components is novel to the best of our knowledge. You can find a visualization of the planned modifications of Mi et al.'s model in figure 1. We will focus on generating training data, as the approach will be usable for further research in the field of source code readability.

Deep Learning based models perform better the more training data they get [12]. Therefore, one approach in order to further improve existing models is to gather

---

[2] https://github.com/, accessed: 2023-07-25
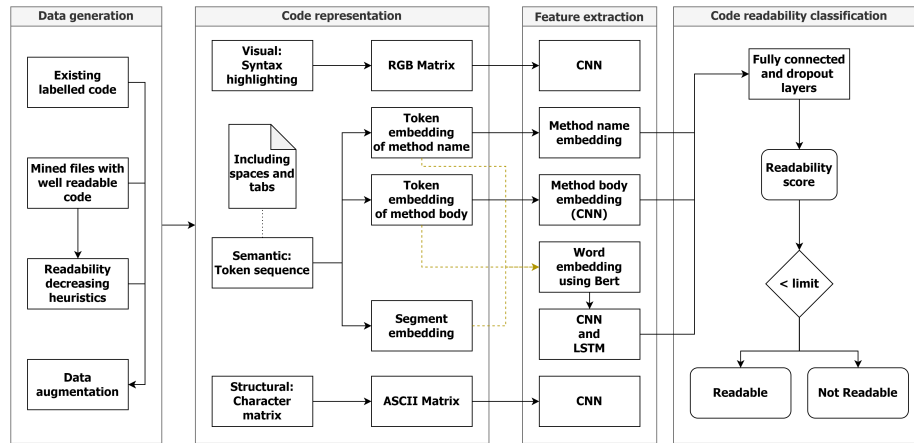[3] https://checkstyle.org/, accessed: 2023-07-25

Figure 1: Overview of the planned approach.

more training data. This requires, as it was done previously, a lot of effort and persons willing to rate code based on their readability. We present another approach for gathering training data.

In a first step, GitHub repositories with known high code quality are downloaded and labeled as highly readable. We select repositories using a similar approach as Allamanis et al. [2] and then assume that they contain only well readable code. In a second step, the code is manipulated so that it is subsequently less readable. This approach is similar to the approach of Loriot et al. [15]. After both steps, we have a new, automatically generated training dataset for source code readability classification.

This brings up the question, how to manipulate code so that it is less readable afterwards. We therefore introduce a tool called Readability Decreasing Heuristics. As the name suggests this is a collection of heuristics that, when applied to source code, lower the readability of it. For example such a heuristic is to replace spaces with newlines. Another example is to increase the indentation of a code block by a tab or multiple spaces. Moreover, with most changes it is also possible to do exactly the opposite (replacing newlines with spaces, decreasing indentation), which in most cases also decreases the readability of source code.

Code snippets in Java are syntactically the same, before and after applying Readability Decreasing Heuristics. Complexity did not change either. However, if various modifications are applied many times, those changes are capable of lowering the readability of source code, as the comparison of listing 1 and listing 2 suggests.

Note that we assume two things for the data generation approach:

Assumption 1 **(well-readable-assumption)** The selected repositories contain only well-readable code.

Assumption 2 **(poorly-readable-assumption)** After applying Readability Decreasing Heuristics, the code is poorly readable.

In recent years it was shown that Deep Learning models can be further improved by modifying the structure of the architecture or by introducing new components, parts or layers to existing architectures. We suggest two improvements for the model of Mi et al. [17]. Firstly, we want to embed spaces and tabs as semantic tokens. Secondly, adding a method name fitting classifier as a component of the overall model could be an improvement. If there is time left, we will try to surpass the performance of recent source code readability classifiers with those improvements to data generation and the model.

We will evaluate our suggestions with two methods. Firstly, we conclude a user study. Secondly, we compare code readability models with each other.

4.1 USER STUDY

The goal of the user study is to answer the following key questions:

1. Does the well-readable-assumption (assumption 1) hold?

2. Does the poorly-readable-assumption (assumption 2) hold?

We will achieve this by showing programmers code snippets that were generated with the presented approach. Therefore, human annotators give each code snippet a rating of its readability. The annotators are selected by prolific[4]. Particular attention is paid to a high proportion of people from industry. The readability rating is based on a five-point Likert scale [13] ranging from one (i.e., very unreadable) to five (i.e., very readable). We apply the same rating as done previously [6, 10, 20], but, other than before, we will not use the rating for labeling the training data. Instead, we will only use the ratings to validate a few randomly selected code snippets out of many that are automatically labeled.

4.2 COMPARING MODELS

Besides the user study we will evaluate our suggestions by comparing machine learning models against each other. The comparisons are based on common metrics such as accuracy, F1-score and MCC [7]. One can distinguish further between the following variants of comparing models:

In one variant we compare models that have the same architecture (same layers, same weight initialization, same components, etc.) while they differ in the data they are trained on. For example, we can train a model with the old and new data-sets, separately and combined. If done for multiple model architectures we can evaluate how the differences in training data influence the model performance.

Another variant would be to compare models with different architecture but the same training data. In this way, we can evaluate newly introduced components by measuring and comparing the performance of such models.

A third comparison variant is created by combining the first two. Both of them lead to many options in what to compare, especially if only small changes to training data or model architecture are done. To find out, if our suggestions lead to a better model overall, we will compare our newly created model with all changes at once to the state-of-the-art model of Mi et al. [17].

---

[4] `https://www.prolific.com/`, accessed: 2023-09-30

4.3 RESEARCH QUESTIONS

We come up with the following research questions:

Research Question 1: *(select-well)* *Can automatically selected code be assumed to be well readable?*

In our new approach for generating training data, we assume that the code from repositories is readable under certain conditions (assumption 1). We want to check whether that holds. To answer this question we will use the results of the user study (section 4.1).

Research Question 2: *(generate-poor)* *Can poorly readable code be generated from well readable code?*

It is not sufficient to have only well readable code for training a classifier. We also need poorly readable code. Therefore, we will try to generate such code from the well readable code. We will investigate whether this is possible in principle, and we will propose an automated approach for archiving this: Readability Decreasing Heuristics.

As the name already suggests, the applied transformations on the source code are only heuristics. To answer, whether the generated code is badly readable (assumption 2) we will utilize the results of the user study (section 4.1).

Research Question 3: *(best-heuristics)* *Which heuristics are best to generate poorly readable code from well readable code?*

We want to compare the modifications of the proposed heuristics for generating poorly readable code to each other. Therefore we will train the same classifier model with badly readable code generated by different Readability Decreasing Heuristics. We will then evaluate the model variations against each other (section 4.2) to answer the research question.

Research Question 4: *(new-data)* *To what extent can the new data improve existing readability models?*

It was shown that Deep Learning models get better the more training data is available [12]. This holds under the assumption that the quality of the data is the same or at least similar. We want to check if the quality of our new data is sufficient for improving the Deep Learning based readability classifier of Mi et al. [17]. Therefore we will train their proposed model with and without the new data and then evaluate the models against each other (section 4.2).

Research Question 5: *(embedding-spaces)* *Optional: To what extend does the embedding of spaces and tabs in semantic code representations improve readability classification?*

The state-of-the-art model of Mi et al. [17] does consider spaces and tabs only in its visual component. We want to investigate if it can improve the quality of a Deep Learning based model if spaces and tabs are encoded as semantic tokens. We also want to investigate if this makes the visual component superfluous. We will evaluate the proposed improvement as described earlier (section 4.2).

Research Question 6: *(name-classifier)* *Optional: To what extend does the usage of a method name classifier improve readability classification?*

Correct naming of identifiers is crucial for ensuring readability of software programs. It is of outstanding importance for readability of code that the name of methods fit the method bodies [14]. We want to introduce a new component to the model of Mi et al. [17] that is built similar to Code2Vec [3]. We want to investigate if the newly introduced component improves the quality of the resulting model. We will evaluate the proposed improvement as previously described (section 4.2).

5 EVALUATION

TODO

6 CONCLUSION

TODO

Bibliography

[1]   Krishan K Aggarwal, Yogesh Singh and Jitender Kumar Chhabra. 'An integrated measure of software maintainability'. In: *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*. IEEE. 2002, pp. 235–241.

[2]   Miltiadis Allamanis, Hao Peng and Charles Sutton. 'A convolutional attention network for extreme summarization of source code'. In: *International conference on machine learning*. PMLR. 2016, pp. 2091–2100.

[3]   Uri Alon et al. 'code2vec: Learning distributed representations of code'. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[4]   Barry Boehm and Victor R Basili. 'Defect reduction top 10 list'. In: *Computer* 34.1 (2001), pp. 135–137.

[5]   Frederick Brooks and H Kugler. *No silver bullet*. April, 1987.

[6]   Raymond PL Buse and Westley R Weimer. 'Learning a metric for code readability'. In: *IEEE Transactions on software engineering* 36.4 (2009), pp. 546–558.

[7]   Davide Chicco and Giuseppe Jurman. 'The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation'. In: *BMC genomics* 21.1 (2020), pp. 1–13.

[8]   Ermira Daka et al. 'Modeling readability to improve unit tests'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 107–118.

[9]   Lionel E Deimel Jr. 'The uses of program reading'. In: *ACM SIGCSE Bulletin* 17.2 (1985), pp. 5–14.

[10]   Jonathan Dorn. 'A General Software Readability Model'. In: 2012.

[11]   Sarah Fakhoury et al. 'Improving source code readability: Theory and practice'. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 2–12.

[12]   Joel Hestness et al. 'Deep learning scaling is predictable, empirically'. In: *arXiv preprint arXiv:1712.00409* (2017).

[13]   Rensis Likert. 'A technique for the measurement of attitudes.' In: *Archives of psychology* (1932).

[14]   Kui Liu et al. 'Learning to spot and refactor inconsistent method names'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1–12.

[15]   Benjamin Loriot, Fernanda Madeiral and Martin Monperrus. 'Styler: learning formatting conventions to repair Checkstyle violations'. In: *Empirical Software Engineering* 27.6 (2022), p. 149.

[16]   Qing Mi et al. 'Improving code readability classification using convolutional neural networks'. In: *Information and Software Technology* 104 (2018), pp. 60–71.

[17]   Qing Mi et al. 'Towards using visual, semantic and structural features to improve code readability classification'. In: *Journal of Systems and Software* 193 (2022), p. 111454.

[18]   Daryl Posnett, Abram Hindle and Premkumar Devanbu. 'A simpler model of software readability'. In: *Proceedings of the 8th working conference on mining software repositories*. 2011, pp. 73–82.

[19]   Spencer Rugaber. 'The use of domain knowledge in program understanding'. In: *Annals of Software Engineering* 9.1-4 (2000), pp. 143–192.

[20]   Simone Scalabrino et al. 'A comprehensive model for code readability'. In: *Journal of Software: Evolution and Process* 30.6 (2018), e1958.

[21]   Simone Scalabrino et al. 'Automatically assessing code understandability: How far are we?' In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 417–427.

[22]   Michihiro Yasunaga and Percy Liang. 'Graph-based, self-supervised program repair from diagnostic feedback'. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10799–10808.