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

22nd February 2024

Abstract

Deep learning-based models are achieving increasingly superior accuracy in classifying the readability of code. Recent research focuses mostly on different model architectures to further improve code readability classification. The models mostly use (parts of) the same labeled dataset, consisting of 421 code snippets. However, deep learning-based approaches are known to require a large amount of data in order to be efficient. Consequently, a large labeled dataset could greatly advance the research field of code readability classification. In this work, we investigate the use of a new dataset consisting of 36077 code snippets together with its novel generation approach. The generation approach involves the extraction and modification of code snippets from public GitHub repositories. The generated dataset is evaluated using a survey with 200 participants and by training a state of the art code readability classification model both with and without the new dataset. In the future, our dataset might increase the accuracy of all readability classification models.

Contents

1 INTRODUCTION

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 [30, 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 [8, 11, 32, 6]. 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% [8, 30, 12, 33] . In recent years, deep learning based models are able to achieve an accuracy of up to 88.0% [23, 24, 37, 26, 20, 21] .

However, a major limitation of these models is not their architecture, but the size of the available dataset for the code readability classification, which comprises 421 code snippets [8, 12, 33] .

Deep Learning based models perform better the more training data they get [14]. Therefore, one approach in order to further improve existing models is to gather more training data. This requires, as it was done previously, a lot of effort and persons willing to rate code based on their readability.

The main idea of this work is to investigate whether it is possible to achieve higher accuracy in code readability classification for Java using automated generated 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. [3] 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. [17]. After both steps, we have a new, automatically generated dataset for 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.

We will conduct a survey to confirm these assumptions (Assumption 1 and 2). There we will ask Java programmers to rate 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[1]. Particular attention is paid to a high proportion of people from industry. The readability rating is based on a five-point Likert scale [16] ranging from one (i.e., very unreadable) to five (i.e., very readable). We apply the same rating as done previously [8, 12, 33] , 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.

Besides the user study we will evaluate our approach by comparing performance of the towards model of Mi et al. [26] when trained with the new and the old dataset. The comparisons are based on common metrics such as accuracy, F1-score and MCC [9].

Our contributions are as follows:

- We combine and unify existing datasets [8, 12, 33] and make the result accessible
- We propose a approach to mine well readable code snippets
- We create a tool to decrease the readability of Java class files
- We propose a new dataset generation approach
- We introduce a new readability classification dataset
- We publish a re-implementation of the towards model of Mi et al. [26]
- We evaluate parts of our new dataset and our generation approach with a user study
- We evaluate our generation approach by comparing the model performance of the towards model of Mi et al. [26] trained with and without the new dataset

The survey confirms both, the well-readable-assumption (assumption 1) and the poorly-readable-assumption (assumption 2). Although our approach for creating a dataset works in principlewe were not able to address enough aspects of code readability with the proposed heuristics. Thus, our dataset probably only addresses a partial problem of code readability.

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

2 BACKGROUND AND RELATED WORK

In the following subsections you find an overview of the background and related work on code readability and our approach to dataset creation.

2.1 CODE READABILITY

In the domain of software development, the importance of code readability cannot be emphasized enough. Alongside understandability, it forms the basis for effective collaboration, comprehension, and maintenance of software systems [30, 1].

It is a critical aspect of software quality, significantly influencing the maintainability, reusability, portability, and reliability of the source code [2, 35]. Poorly readable code increases the risk of introducing bugs [18, 33] and can lead to higher costs during subsequent software maintenance and development [15]. On the other hand, readable code allows developers to identify and rectify bugs more easily [21].

Recent studies indicate that developers spend nearly 58% of their time reading and comprehending source code [8, 11, 32, 6, 38, 35, 41]. Therefore, it is important to ensure a high readability of code. In order to archive this, we need to define and measure code readability.

Buse and Weimer provides one of the first definitions: "We define readability as a human judgment of how easy a text is to understand."

Tashtoush et al. combines numerous other aspects from various definitions. According to them code readability can be measured by looking at the following aspects [38]:

- Ratio between lines of code and number of commented lines
- Writing to people not to computers
- Making a code locally understandable without searching for declarations and definitions
- Average number of right answers to a series of questions about a program in a given length of time

Recent definitions of code readability are shorter, trying to focus on the key aspects. Oliveira et al. defines readability as "what makes a program easier or harder to read and apprehend by developers" [28].

Also Mi et al. summarizes code readability as "a human judgment of how easy a piece of source code is to understand" [25]. This comes close to the definition of Buse and Weimer [8].

There are various related terms to readability: Understandability, usability, reusability, complexity, and maintainability [38] [38]. Among those especially complexity and understandability are closely related to readability.

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 [8, 7].

Readability is neither the same as understandability, as the key aspects of understandability are [33, 19, 40, 5]:

- Complexity
- Usage of design concepts
- Formatting
- Source code lexicon
- Visual aspects (e.g., syntax highlighting)

Posnett et al. states that readability is the syntactic aspect of processing code, while understandability is the semantic aspect [30].

Based on Posnett et al., Scalabrino et al. writes about readability: "Readability measures the effort of the developer to access the information contained in the code, while understandability measures the complexity of such information" [33, 30].

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 [34].

Comparing the definitions of code readability in literature we can see, that there are some common aspects in most definitions. These are:

- Ease/complexity of understanding/comprehension/apprehension
- Human judgment/assessment
- Effort of the process of reading (differentiation to understandability)

Based on this, we come up with the following definition:

Definition 2.1 (Code readability): *Code readability is the human assessment of the effort required to read and understand code.*

```java
/**
* Logs the output of the specified process.
*
* @param p the process
* @throws IOException if an I/O problem occurs
*/
private static void logProcessOutput(Process p) throws IOException
{
        try (BufferedReader input = new BufferedReader(new
        ↪  InputStreamReader(p.getInputStream())))
        {
                StrBuilder builder = new StrBuilder();
                String line;
                while ((line = input.readLine()) != null)
                {
                        builder.appendln(line);
                }
                logger.info(builder.toString());
        }
}
```

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

In the last years, researchers have proposed several metrics and models for assessing code readability with an accuracy of up to 81.8% [8, 30, 12]. In recent years, deep learning based models are able to achieve an accuracy of to 85.3% [24, 26] on available datasets. By using data augmentation the score can be improved even further to 87.3% [25]. Examining these works more closely in the following, we delve into their intricacies.

2.2 CONVENTIONAL CALCULATION APPROACHES

A first estimation for source code readability was the percentage of comment lines over total code lines [1]. Then researchers proposed several more complex metrics and models for assessing code readability [8, 30, 12, 33]. 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 [33].

2.3 DEEP LEARNING BASED APPROACHES

In recent years code readability classification is dominated by machine learning, especially deep learning approaches. As the quality of the models increased, so did their accuracy (see Figure 1).

```
private
        static
void
debug( Process
v1
)       throws IOException
{
        // Doo debug
        try (BufferedReader   b
        = new
        BufferedReader(
        new InputStreamReader(
        v1.getInputStream()
        )
        )
        )
        {
                StrBuilder b2=new StrBuilder();String v2;while
                ↪  (null!=(v2=input.readLine())){b2.appendln(v2);}
                ↪  // Doo stuff
                m.info(  builder.toString()
                );
        }
}
```

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

Mi et al. introduced a deep learning model called IncepCRM for code readability classification. IncepCRM automatically learns multi-scale features from source code with minimal manual intervention [23].

Mi et al. use Convolutional Neural Networks (ConvNets). Their proposed model, DeepCRM, employs three ConvNets with identical architectures, trained on differently preprocessed data [24].

Another study addresses concerns regarding the classification of code script readability by proposing an approach using Generative Adversarial Networks (GANs). The proposed method involves encoding source codes into integer matrices with multiple granularities and utilizing an EGAN (Enhanced GAN) [37].

Mi et al. address the limitation of the previous deep learning-based code readability models, which primarily focus on structural features. Their proposed method aims to enhance code readability classification by extracting features from visual, semantic, and structural aspects of source code. Using a hybrid neural network composed of BERT, CNN, and BiLSTM, the model processes RGB

Table 1: Accuracy scores of two-class readability classification models.

Model	Accuracy
Buse [8]	76.5 %
Possnet [30]	71.7 %
Dorn [12]	78.6 %
Scallabrino [33]	81.8 %
Mi_IncepCRM [23]	84.2 %
Mi_DeepCRM [24]	83.8 %
Sharma [37]	84.8 %
Mi_Towards [26]	85.3 %
Mi_Ranking [20]	83.5 %
Mi_Graph [21]	88.0 %

matrices, token sequences, and character matrices to capture various features [26].

Mi introduced a novel approach to code readability assessment by framing it as a learning-to-rank task. The proposed model employs siamese neural networks to rank code pairs based on their readability [20].

Mi et al. address the importance of code readability in software development and introduced a graph-based representation method for code readability classification. The proposed method involves parsing source code into a graph with abstract syntax tree (AST), combining control and data flow edges, and converting node information into vectors. The model, comprising Graph Convolutional Network (GCN), DMoNPooling, and K-dimensional Graph Neural Networks (k-GNNs) layers, extracts syntactic and semantic features [21].

You can find an overview over the accuracy scores for the models mentioned in Table 1.

The main contribution of this work is not a model that outperforms a state of the art model but rather a new dataset (generation approach). For evaluation we opted for the Mi_Towards model (hereinafter referred to as towards model) from Mi et al. [26]. We did not choose the best performing one, Mi_Graph, as its main contribution is to use the AST representation of the code, while our dataset generation approach includes features that are not represented in the AST [21].

2.4 DATA AUGMENTATION

All the mentioned models were trained with (a part of) the data from Buse, Dorn and Scalabrino consisting of a total of 421 java code snippets. The data was generated with surveys. They therefore asked developers several questions, including how well readable the proposed source code is [8, 12, 33]. We will refer to this dataset as old dataset.

The problem that there is little data in the area of code readability classification for machine learning models has been recognized.

Mi et al. address the challenge of acquiring a sufficient amount of labeled data for training deep learning models. Due to the time-consuming and expensive nature of obtaining manual labels, the researchers propose the use of data augmentation techniques to artificially expand the training set. They employ domain-specific transformations, such as manipulating comments, indentations, and names of classes/methods/variables, and explore the use of Auxiliary Classifier GANs to generate synthetic data [25].

Mi et al. address the challenge of multi-class code readability classification. Due to a scarcity of labeled data, most prior research focused on binary classification. The authors propose an enhanced data augmentation approach, incorporating domain-specific data transformation and Generative Adversarial Networks (GANs) [22].

Vitale et al. introduce a novel approach to automatically identify and suggest readability-improving actions for code snippets. The authors develop a methodology to identify readability-improving commits, creating a dataset of 122k commits from GitHub's revision history. They train the T5 model to emulate developers' actions in improving code readability, achieving a prediction accuracy between 21% and 28%. The empirical evaluation shows that 82-90.8% of the dataset commits aim to improve readability, and the model successfully mimics developers in 21% of cases [39].

2.5 DIVERSE PERSPECTIVES

There is also other important research in the field of readability classification that does not directly affect this work, but could have implications for future work.

Fakhoury et al. showed based on readability improving commit analysis that previous models do not capture what developers think of readability improvements. They therefore analyzed 548 GitHubhttps://github.com/2023-07-25 commits manually. They suggest considering other metrics such as incoming method calls or method name fitting [13].

Oliveira et al. conducted a systematic literature review of 54 relevant studies on code readability and legibility, examining how different factors impact comprehension. The authors analyze tasks and response variables used in studies comparing programming constructs, coding idioms, naming conventions, and formatting guidelines [28].

In a recent study participants demonstrated a consistent perception that Python code with more lines was deemed more comprehensible, irrespective of their level of experience. However, when it came to readability, variations were observed based on code size, with less experienced participants expressing a preference for longer code, while those with more experience favored shorter code. Both novices and experts agreed that long and complete-word identifiers enhanced readability and comprehensibility. Additionally, the inclusion of comments was found to positively impact comprehension, and a consensus emerged in favor of four indentation spaces [31].

Choi, Park et al. introduced an enhanced source code readability metric aimed at quantitatively measuring code readability in the software maintenance phase. The proposed metric achieves a substantial explanatory power of 75.74%. Additionally, the authors developed a tool named Instant R. Gauge, integrated with Eclipse IDE, to provide real-time readability feedback and track readability history, allowing developers to gradually improve their coding habits [10].

Mi et al. aim to understand the causal relationship between code features and readability. To overcome potential spurious correlations, the authors propose a causal theory-based approach, utilizing the PC algorithm and additive noise models to construct a causal graph. Experimental results using human-annotated readability data reveal that the average number of comments positively impacts code readability, while the average number of assignments, identifiers, and periods has a negative impact [27].

Segedinac et al. introduces a novel approach for code readability classification using eye-tracking data from 90 undergraduate students assessing Python code snippets [36].

2.6 DATA GENERATION

In addition to related work on models and datasets, there is also related work that uses some of the ideas that we will employ in our proposed approach to data generation.

Loriot et al. created a model that is able to fix Checkstyle[2] 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 [17]. A similar idea has been explored by Yasunaga and Liang [42]. 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[3] 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 [3].

3 MINED AND MODIFIED CODE FOR DATASET GENERATION

In the following subsections we will describe our approach.

3.1 WORK ON EXISTING DATASETS

Most of the related work (see Section 2) uses a combination of Buse and Weimer, Dorn and Scalabrino et al. data. The raw data from their surveys can be downloaded [4], but their data is not uniformly formatted, including ratings that are not Java code snippets, as well as the individual ratings rather than the mean of the ratings used for training machine learning models.

We converted and combined the three datasets into one: code-readability-merged. In recent years, Huggingface [5] established as the pioneer in making models and datasets available. Therefore we decided to publish the merged dataset on Huggingface [6].

3.2 CLASSIFICATION CONSIDERATIONS

In code readability classification it is still state of the art to make a binary classification into readable and unreadable code [23, 24, 37, 26, 20].

[2] `https://checkstyle.org/`, accessed: 2023-07-25
[3] `https://github.com/`, accessed: 2023-07-25
[4] `https://dibt.unimol.it/report/readability/`, accessed: 2024-02-18
[5] `https://huggingface.co/`, accessed: 2024-02-18
[6] `https://huggingface.co/datasets/se2p/code-readability-merged`, accessed: 2024-02-18

However, code readability classification is not a binary classification task per se. Mi et al. introduced a third, neutral class to address this problem [21].

When rating code snippets, a Likert scale [16] from 1 to 5 was mostly used. So one could argue, that there are actually 5 classes rather than two or three. However, the actual readability score of a code snippet lies somewhere in between. It can have any value from 1.0 to 5.0 and thus the range is continuous, not discrete. In particular, if you normalize by subtracting 1 and divide then by 4, we get a continuous value between 0.0 to 1.0. At this point, it is trivial to say that it is basically a regression problem.

The question arises as to why research in the field of code readability always views the problem as a (binary) classification problem. The answer to this question is probably worthy of its own research and therefore not covered in this work.

Our evaluation model is the towards model of Mi et al. [26]. Therefore, we want to show how they transformed the rating scores into a binary classification problem. First, the mean values of all scores are calculated. In a second step, the snippets are ranked according to their mean score. Then, the top 25% of the data is labeled as well-readable and the bottom 25% is labeled as badly-readable. The 50% of the data in between is not used at all [26].

While this transformation is fine in principle, especially with the argument that the data in the middle is neither readable or unreadable, it has drawbacks that only 50% of the available data is used for model training and evaluation:

First, the available data is further reduced from 421 evaluated Java code fragments to 210 code fragments for training. Note that a bottleneck in readability classification is the small amount of available data. So this is a significant loss.

Secondly, evaluation is performed with only those 210 snippets as well. This means, that the model was only evaluated on 50% of the available data. We suspect that this might be a thread to validity. It could be that the performance of the model is remarkably lower when the evaluation is performed with random, unseen data that also contains moderately readable code snippets.

However, we will both stick to the binary classification approach as well as to the towards model [26] to make our results comparable to theirs.

3.3 DATASET GENERATION APPROACH

Other than previous datasets for readability classification, our dataset is generated using an automated approach. The aim is to mine modify code from GitHub

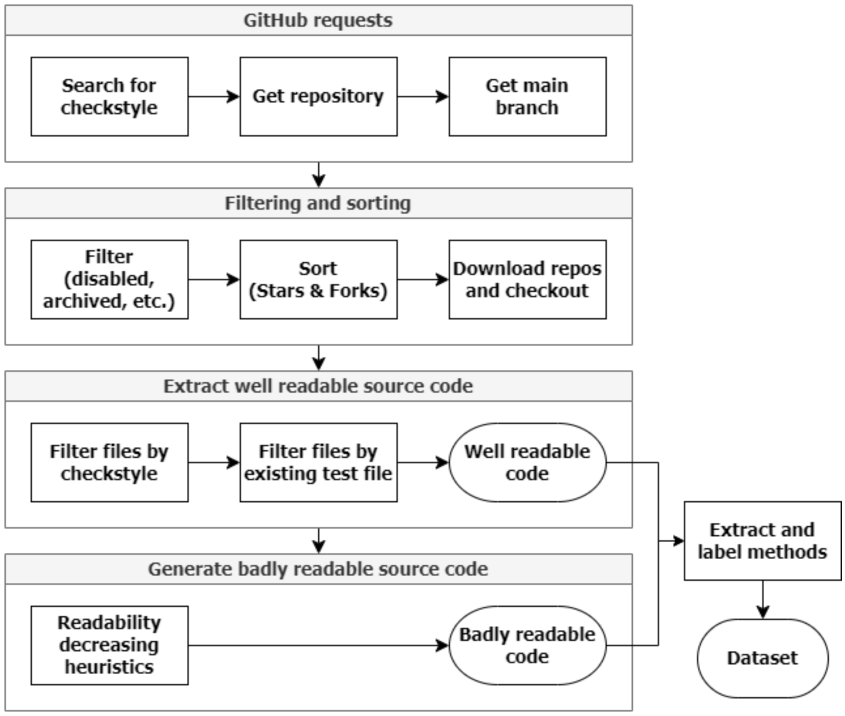


Figure 1: The used dataset generation approach.

to obtain both readable and unreadable code. This approach is novel to the best of our knowledge. You can find a visualization in Figure 1.

The approach is divided into four parts. The first three steps are used to mine well readable java code. In a final step, we will modify the well readable code to achieve our second goal, namely poorly readable source code.

We start by querying the GitHub REST API[7] for repositories that use checkstyle (query string: "checkstyle filename:pom.xml"). The repository informations (including the URL) are stored together with the main branches. We remove all repositories that do not fulfill these criteria:

- The repository is not a fork of another repository
- The repository is not archived
- The repository is not disabled
- The repository language is Java
- The repository has at least 20 stars

[7] `https://docs.github.com/en/rest`, accessed: 2024-02-15

- The repository has at least 20 forks

The remaining ones are sorted by their star and fork count (equally weighted). The 100 best are cloned and their main branch is checked out.

In a third step we run checkstyle ([8]) against the project own checkstyle configuration to get all java class files, that pass the own checkstyle test. From the java classes that passed this filter we extract all methods that have a comment of any kind at the beginning of the method. This results in 36077 code snippets which we assume to be well readable.

The fourth and final step is to generate badly readable code from the well readable one. Therefore we use the proposed Readability Decreasing Heuristics (RDH, see section 3.4). Afterwards we again extract all methods with a comment at the beginning of the method. Initially we planned to not require comments for the badly readable dataset part. However, it turns out that in this case all well-readable methods have a comment while most of the badly-readble do not have one. This lead to shortcut learning, whether a method has a comment or not instead of learning to distinguishing the methods by all other criteria as well.

3.4 READABILITY DECREASING HEURISTICS

We already mentioned, that Readability Decreasing Heuristics (RDH) are used to generate badly readable code from well readable code. In this section we will explain how we achieved this and what is meant by RDH. The RDH are a set of code manipulation heuristics that are applied to Java files. One part is performed on the abstract syntax tree (AST) representation of the Java files using the spoon library ([9]) [29]. Another part is performed when converting the AST back into Java files. From now on, we will use the abbreviation RDH or RDH tool when referring to the entire program. If a specific manipulation or a subset of all manipulations is meant, we will use [name] rdh instead.

The RDH tool converts the Java code of each well readable Java class file into an AST. In the end the AST is parsed back to Java code using an pretty printer. If nothing else is done, this results in the "none" rdh. Note that the code produced by the tool in this way will be slightly different from the original input code, as the styling and formatting of the original code will be overwritten by the default formatting of the Java Pretty Printer of the spoon library.

Various code changes can be made between the two steps and during printing (see Figure 2). The renaming is done while the code is in its AST representation

[8] `https://checkstyle.sourceforge.io/`, accessed: 2024-02-15
[9] `https://spoon.gforge.inria.fr/`, accessed: 2024-15-02

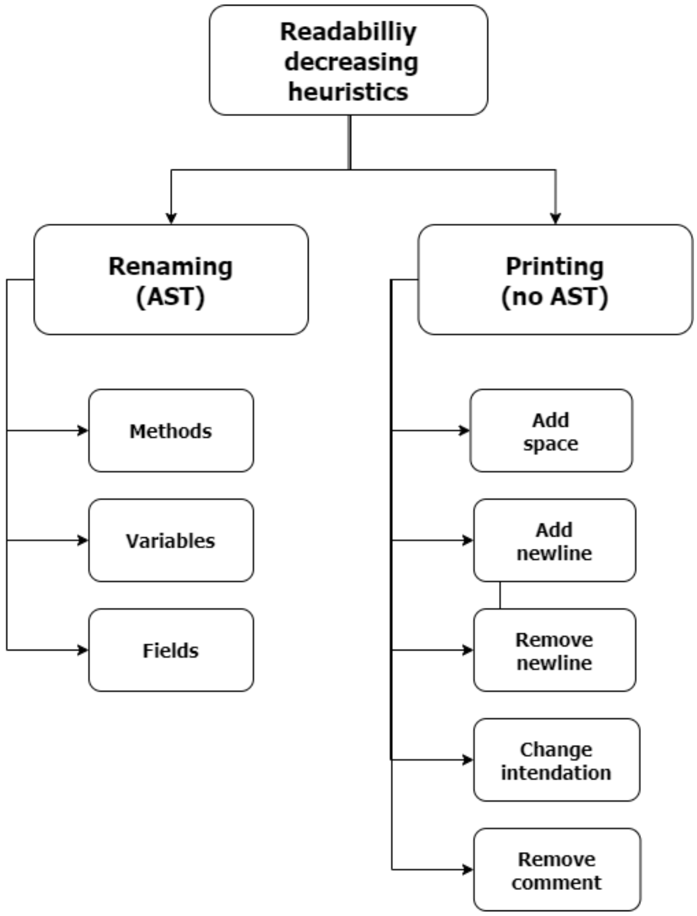


Figure 2: The available Readability Decreasing Heuristics (RDH).

to ensure that the declarations and usages of variables, fields and methods are all renamed to the same new name. The other transformations are performed when the AST is converted back to source code. This includes adding additional spaces, adding or removing newlines and changing the indentation of code in term of tabs.

The individual methods are then extracted from the class files. As already mentioned, we need a method comment for all methods. We therefore use the RDH "remove-comment" after completing the method extraction.

The RDH works with a configuration file in which one can specify a probability for each heuristic that can be applied. We have chosen the probabilities so that the generated code snippets are still realistic in the sense that they could also

Table 2: Readability Decreasing Heuristics that are not included in the final version of the new dataset and why.

Heuristic	Reason
`inlineMethod`	Makes methods too long
`add0`	Limited survey capacity
`insertBraces`	Limited survey capacity
`starImport`	No effect after comment extraction
`inlineField`	Limited survey capacity
`partiallyEvaluate`	Might increase readability

be written by humans. You can find the configuration file for the none rdh in Appendix **??**.

Some of the heuristics were not included in the final version of the new dataset. You can find them in table 2 together with the reason why they were not included.

We have also added Code2Vec [4] to the tool. This makes it possible to rename methods not only to iterating or arbitrary strings or numbers, but also to other realistic method names. The idea was to use worse method names predicted by Code2Vec and rename the methods to these. However, due to time and resource constraints regarding the survey, we did not pursue this approach. However, there is a corresponding mode supported by the tool. This can be used for further research.

3.5 CONSTRUCTION OF QUESTIONNAIRES

We evaluated the data set generated and the new approach with a survey. To do this, we had to carefully select suitable code snippets from the dataset. An overview of the approach can be found in Figure 3.

The first step was to find realistic configurations for the readability reduction tool. After an initial data set with the heuristics was created, a pilot study was conducted. Subsequently, the heuristics that proved to be too strong were checked and, if necessary, adjusted to be weaker according to the results of the pilot survey. The result was 9 different rdhs, which can be found in table 3. Together with the original methods this resulted in 10 different configurations.

The configurations are based on probabilities for different heuristics. A heuristic is applied with the specified probability to each occurrence of the object to which it is to be applied. For example, comment_remove is applied with a probability of 10% to each comment that occurs within the code snippet. The exact scope of

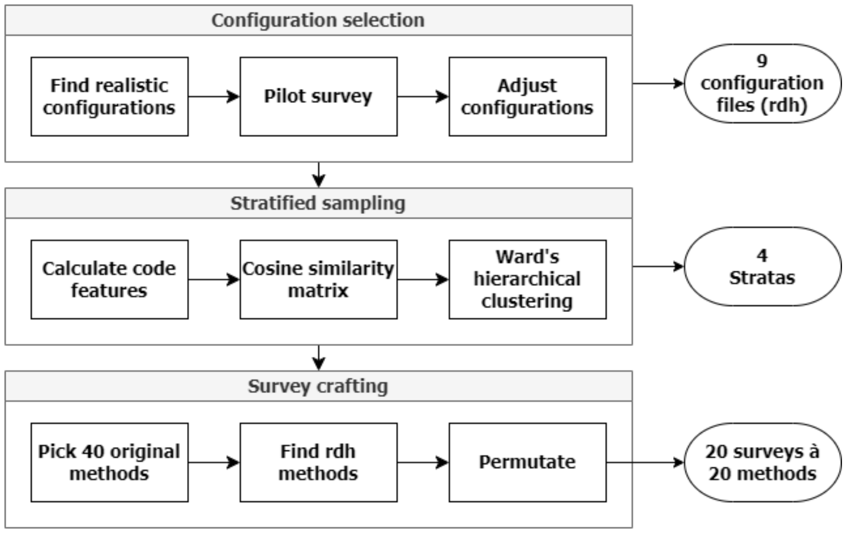


Figure 3: Steps performed to craft survey sheets from the dataset

changes for a method is therefore uncertain. It can even happen (especially with short methods such as getters and setters) that a method is not changed at all. For example, if a method only has a single comment and we use comment_remove, it is very likely that the method will not be changed at all.

In a second step, we applied a stratified sampling [**thompson2012sampling**] to distinguish between very simple methods such as getter and setter and more complex methods. In order to be able to compare the original methods with their modified variants, we only carried out the random sampling for the original methods and compared the rdh methods with these in a later step.

Thus, we first had to calculate features for the original code snippets. This was done using Scalabrino et al.s tool [33]. Therefore, a 110-dimensional features vector was calculated for each original code snippet. Next we compute the cosine similarity metrix between all feature vectors using scikit[10]. Finally, using the fastcluster implementation [**mullner2013fastcluster**] of Ward's hierarchical clustering we were able to cluster the methods into an arbitrary amount of clusters.

By comparing the merge distances in each step (see figure 4), we found that a cluster size of 4 makes the most sense: the merge distance of 5 to 4 is small, so we should still perform this merge, but the merge distance of 4 to 3 is large, so it is better not to perform this merge. Also, 4 is the size with the last possibility

[10]`https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html`, accessed: 2024-02-20

Table 3: Chosen configurations and their probabilities for the Readability Decreasing Heuristics.

Configuration	Probabilities
none	-
comments_remove	removeComment: 1.0
newline_instead_of_space	newLineInsteadOfSpace: 0.15
newlines_few	removeNewline: 0.3 spaceInsteadOfNewline: 0.05
newlines_many	add1Newline: 0.15 add2Newlines: 0.05
rename	renameVariable: 0.3 renameField: 0.3 renameMethod: 0.3
spaces_many	Add1Space: 0.2 Add2Spaces: 0.1 spaceInsteadOfNewline: 0.05
tabs	remove1IncTab: 0.2 add1IncTab: 0.1 remove1DecTab: 0.1 add1DecTab: 0.1 incTabInsteadOfDecTab: 0.05 decTabInsteadOfIncTab: 0.05
all7	all probabilites/7

for a small merge distance. We have manually assigned a name to each of the 4 cluster/strata, which can be seen in Table 4.

In the third step, we crafted the surveys from the layers. We decided to provide all 10 previously mentioned configurations for each original method, as we want to compare the original methods with their rdh variants. Since we have a survey capacity of 400 code snippets, we need to select 40 original code snippets (and then add all their rdh).

We chose to sample randomly from within the stratas. However, we distributed the 40 snippets among the stratas as can be seen in table 5.

We opted for a random sample within the strata. However, we distributed the 40 snippets across the strata as shown in Table 5.

This decision was made due to the relatively high frequency of methods that do not differ from their original methods (see Figure 5). Another reason for

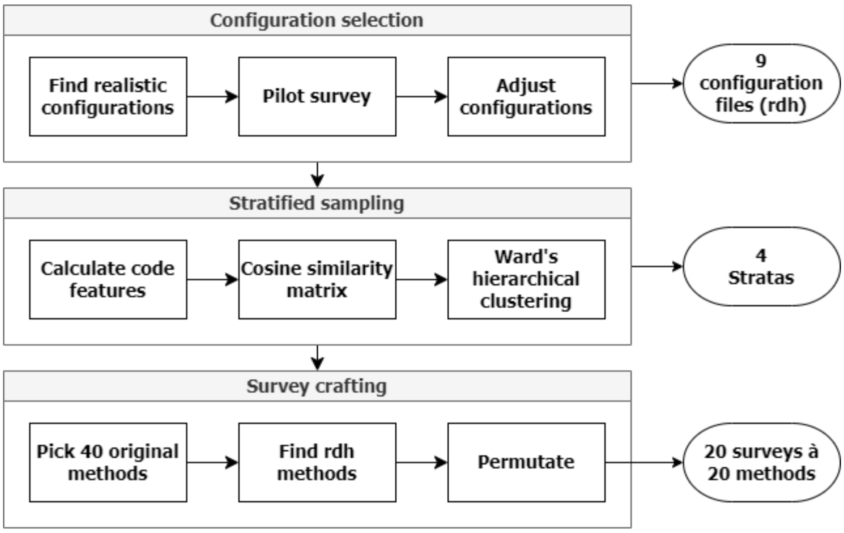


Figure 4: TODO: Replace. Merge difference and local derivative for amount of clusters/strata between 2 and 10.

Stratum	Method Type
Stratum 0	Simple methods
Stratum 1	Complex methods
Stratum 2	Magic number methods
Stratum 3	Medium complex methods

Table 4: Computed stratas and manually assigned name based on the methods within.

this decision is that particularly simple methods are rather uninteresting for the classification of readability, as they are often generated (e.g. by IDEs) and usually follow a straightforward pattern.

After selecting the 40 original methods, we next selected all 9*40 rdh variants that belong to the original methods. This was mostly done automatically based on the names of the original methods and the names of the rdh variant methods. However, if the method was renamed at an earlier stage due to the method renaming heuristic, the new method did no longer match the original method, so we had to match them manually.

Once we had collected all 400 methods, we had to distribute them across the 20 survey forms, each with 20 methods. In order not to manipulate the raters, we decided that a variant of each method could only appear once in each survey sheet.

Table 5: Strata distribution of sampled methods.

Stratum	Percentage	Count
Stratum 0	10 %	4
Stratum 1	40 %	16
Stratum 2	10 %	4
Stratum 3	40 %	16
Total	**100 %**	**40**

For example, if the original method is in questionnaire 1, the comment_remove variant (or another variant of the same method) must not be included in the same questionnaire.

For this purpose, we created four permutation matrices with 10 snippets each. The number 10 was chosen because it is possible to distribute 10 snippets, each with 10 variants, to at least 10 survey arcs without violating our condition. By combining two 10-permutation matrices, we were able to create 10 survey arcs with 20 code snippets each. An implication of this approach is that each survey sheet contains each variant exactly twice. By doing this twice, we obtain the desired distribution of 20 survey questionnaires with 20 methods each. Our condition also applies: There is only one variant of the same method in each questionnaire.

Finally, the methods for each survey questionnaire were randomly shuffled within each survey questionnaire. This was done to minimize the impact of the position of a snippet/variant within a survey on the rating.

Once the survey is complete, we will aggregate the ratings across all strats and all methods and group the results into the 10 rdhs. In this way, we want to assess whether the rdhs work, which rdhs work best and how strong the impact on readability of each rdh is. In addition, we will use the results of the survey to label our new dataset, which consists of the original and all7 rdh methods.

Now that we have generated a dataset and evaluated the impact of the different rdhs we use the survey results to label each snippet of the well readable code (original) with the mean over all ratings of the original variant over all survey results. Similary we label the all7 code snippets with its mean score. Note that for binary classification with the towards model this results in the same as labelling the original methods with the well readable class and the all7 rdh with the badly readable class.

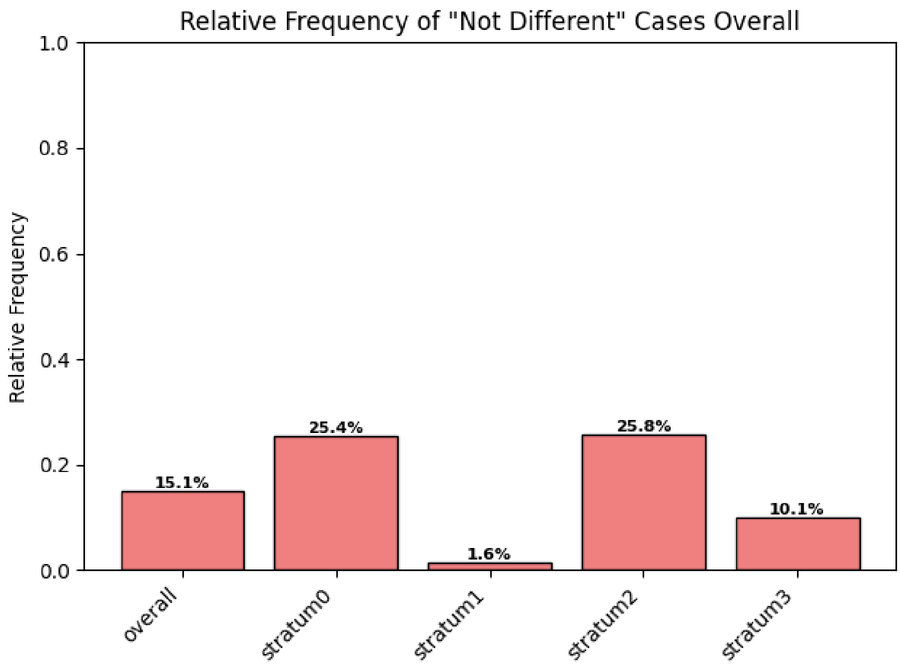


Figure 5: Relative frequency of the case that a rdh-method is not different from it's original method.

Each snippet of the well readable code (original) is labeled with the mean value of all ratings of the original variant across all survey results. Similarly, we label all7 code snippets with their mean value. Note that in the binary classification with the underlying towards model, this results in the original methods being labeled with the class "well readable" and the all7 rdh with the class "badly readable".

3.6 READABILITY CLASSIFICATION MODEL

Next we investigate whether it is possible to score a higher accuracy as the towards model in classifying code readability with our new dataset.

We have therefore created our own implementation of the model with Keras[11]. In contrast to the publicly available code of Mi et al.[12], our model also includes (batch) encoders required for the model to be trained on new data and to perform the prediction task for new code snippets. In addition, our model supports fine-

[11] `https://keras.io/about/`, accessed: 2024-02-20
[12] `https://github.com/swy0601/Readability-Features`, accessed: 2024-02-20

tuning by freezing certain layers as well as storing intermediate results, such as the encoded dataset. During evaluation, the model returns the evaluation statistics in the form of a json file.

During implementation, we also encountered the following potential problem with the model: The token length for the bert encoding (bert-base-cased[13]) used in the model is fixed at 100. What is a token in a piece of code? In addition to special tokens that mark the beginning [CLS] and the end [SEP] of the input, each word is represented by a token. However, each special character (such as /(),;= and many more) is also represented by its own token. Java identifiers are split according to the convention of upper and lower case. Long words are in turn divided into several tokens.

Consider the method from Listing 1. With a token limit of 100, the last encoded token is the last closing parenthesis in line 9. Everything from line 10 onwards is no longer encoded, which means that the information is lost after the semantic encoder or for the model. To put it in other words: The model of Mi et al. only considers the first few lines of code snippets in its semantic component.

The visual and structural encoders have similar limitations, but to a much lesser extent. The structural encoder encodes the first 50 lines of each code snippet and the visual encoder encodes the first 43 lines. While the constraints for these two coders seem to be long enough to fully capture most code snippets, the semantic coder seems to be too limited to do so.

Although we want to note these limitations, we will keep them in order to later allow a fair comparison of the datasets with the model of Mi et al. trained on the combined dataset.

Our code is publicly available on GitHub. TODO: Refs

You can find an overview over all programs used to create the merged dataset, the new dataset the model and all our evaluation results in Figure 6.

4 EVALUATION

To determine the quality of the new dataset, we are completing a user study to evaluate the newly generated data for readability classification models. The aim 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?

[13] `https://huggingface.co/google-bert/bert-base-cased`, accessed: 2024-02-20

Figure 6: Overview over the used scripts and their output.

Therefore, 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 **??**).

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 **??**).

Research Question 3: ***(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 [14]. 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. [26]. Therefore we will train their proposed model with different datasets and then evaluate the models against each other (section **??**).

We will present our results in the following subsections.

4.1 SURVEY

The results of our survey are divided into two parts: The results of the pilot survey, which were used to improve the main survey pre-launch, and the results of the main survey, which were used to answer our research questions and to craft our dataset.

PILOT SURVEY

The pilot survey was used to adjust the rdh probability settings and the survey itself before the start.

1. Experimental setup: We therefore manually sampled 20 code snippets across all stratas but mainly from statum 1, due to reasons mentioned in section 3.5. Between the 6th and the 14th of January 10 participants, mostly students, participated in the survey. Additionally to rating 20 code snippets the participants were also asked to complete an additional survey form where they could provide feedback about the actual survey.

Here they were asked the following questions:

1. How long did it take you to complete the survey? (short answer)
2. How clear was your task? (1 (very unclear) to 5 (very clear))
3. What problems were with the task? If there were none, leave blank. (long answer)

4. What problems were there with the survey tool? If there were none, leave blank. (long answer)
5. What improvements would you make to the survey? If none, leave blank. (long answer)
6. Do you have any other feedback? If none, leave blank. (long answer)

The participants answers can be found TODO.

The feedback of the pilot survey was used in the following ways to prepare the prolific study:

- To adjust the survey texts and questions
- To estimate how long completion of one survey sheet will take
- To adjust the readability-decreasing heuristic settings
- To discover problems with the survey tool
- To discover fundamental problems with the dataset

2. Threats: The results do not generalize. We did not sample a the data in a specific, (semi-) automated way, so there is a selection bias. The survey participants were not selected among all Java programmers randomly. The final formulations for the prolific survey were adjusted afterwards and therefore were sub-optimal. However, we did not use the results from this survey to evaluate our dataset or the generation approach, as the intention of the survey was rather to prepare for the main survey.

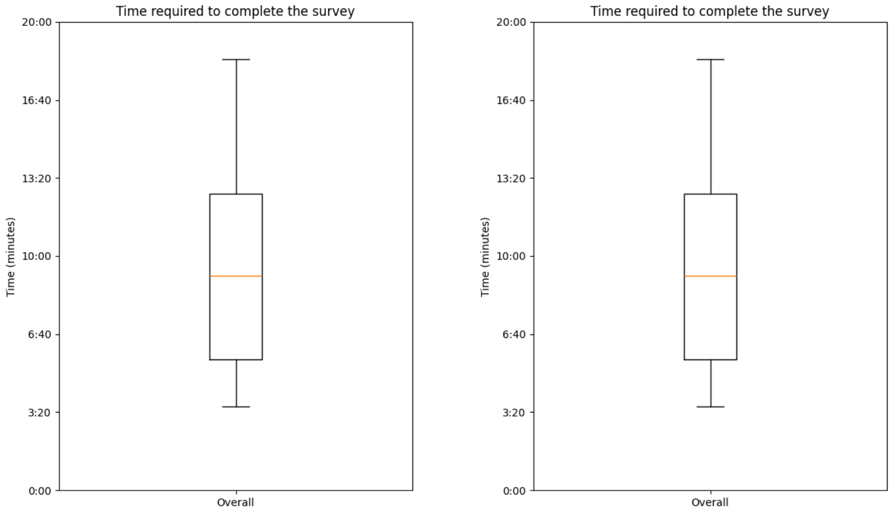
3. Results: The pilot survey provided information on how much time it would take to evaluate 20 code snippets in terms of their readability. An overview of the times required can be found in Figure 7b. On this basis, we estimated the time required for our survey at 10 minutes.

Most of the problems that occurred were due to the survey tool (e.g.: "I also felt that I should use the drop-down menu at the top left."). You can find all feedback in the appendix TODO.

In addition, a manual evaluation of the various rdhs, especially for stratum 1, revealed some clues:

First, the original methods were rated comparably well, which suggests that the assumption of good readability is correct.

Secondly, the rdh tool always specified each imported method or class completely with its fully specified classifier. For example, instead of "InputStream", "java.io.InputStream" was written. This gave the participants the feeling that



(a) Time required by the participants to complete the prolific survey.

(b) Time required by the participants to complete the pilot survey.

Figure 7: Time required to complete the surveys.

the code was not written by a human and drastically reduced readability. We then adapted the rdh tool to print the shorter name. However, this has the disadvantage that it can no longer be guaranteed that the generated code will compile and behave exactly like the original. However, this property was lost anyway when extracting the individual methods from the class files.

Thirdly, some rdhs were configured too strongly, so that for some methods it was no longer assumed that they were written by a human. These rdhs were adjusted and their probabilities reduced accordingly (for example newlines_few). All results can be found in table 6.

Once the aforementioned adjustments had been made and the feedback on the survey instrument had been implemented, the actual study was carried out.

PROLIFIC SURVEY

In this section we summarize the results of the main study conducted via prolific.

Table 6: Mean score ratings for the pilot survey.

Stratum	RDH	Score
Stratum 3	methods	4,6
Stratum 0	tabs_few	4,3
Stratum 2	tabs_few	3,8
Stratum 1	methods	3,7
Stratum 2	methods	3,7
Stratum 3	newlines_many	3,3
Stratum 1	comments_remove	3,1
Stratum 0	spaces_few	3,0
Stratum 1	all_weak_3	3,0
Stratum 1	newlines_many	2,9
Stratum 1	spaces_few	2,6
Stratum 1	misc	2,4
Stratum 2	newlines_few	2,4
Stratum 1	tabs_few	2,2
Stratum 1	tabs_many	2,2
Stratum 1	spaces_many	2,1
Stratum 1	newlines_few	1,7
Stratum 3	tabs_many	1,7
Stratum 1	all_weak	1,3
Stratum 1	all	1,2

1. Experimental setup: The survey was conducted using Tien Duc Nguyen's Code Annotation Tool (see Figure 8) along with the platform prolific[14] for the recruitment and payment of participants. The survey was conducted between January 31, 2024 and February 7, 2024. A total of 221 participants took part. Each of the 20 questionnaires was answered by 11 participants (similar to the survey of Scalabrino et al. [33]). In one survey, one more participant was assigned by mistake. We end up with a margin of error of 29.55% at a confidence of 95% for an individual snippet. However, we will aggregate over stratas and multiple snippets later anyway. Each questionnaire consists of 20 code snippets. Consequently, 400 different code snippets are rated in total. The surveys were configured in such a way that each participant could only take part in one of the questionnaires. You can find the texts of the survey in appendix **??**. The questionnaires were crafted as described in section 3.5.

[14] `https://app.prolific.com/`, accessed: 2024-02-21

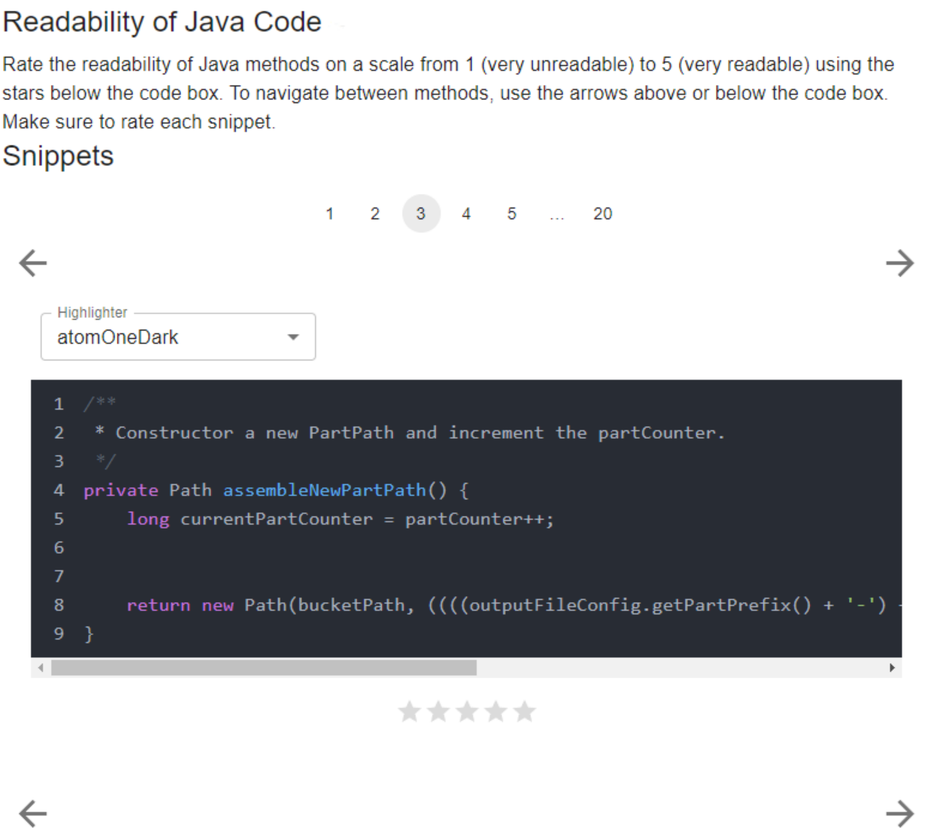


Figure 8: Tool for rating a code snippet from the perspective of a survey participant.

The target population consists of Java programmers selected by prolific. They may be students or work in industry. They can come from any country. Overall, there were no requirements other than familiarity with Java (see also table 7).

The internal research questions are as follows:

- Does the well-readable-assumption (Assumption 1) hold?
- Does the poorly-readable-assumption (Assumption 2) hold?

The results for these questions are equally important, and thus none of them is prioritized over the other. To answer them, the assumptions are considered as hypotheses along with the following associated null hypotheses:

- For Assumption 1: The mined code exhibits a normal distribution of readability scores.

Table 7: Target population for the prolific survey.

Type	Target Population
Target Audience	Java programmers
Unit of Observation	Java programmers
Unit of Analysis	Java programmers
Search Unit	Selected by Prolific (Programming Languages: Java)
Source of Sampling	Prolific

- For Assumption 2: The readability of code does not significantly deteriorate compared to the original code snippet.

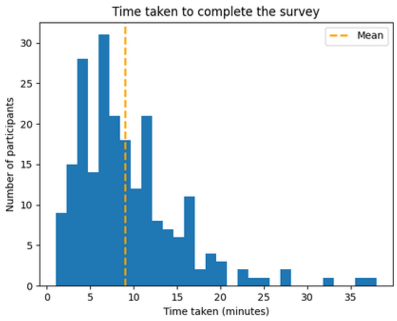
The survey neither contained demographic questions nor filter questions. Besides the readability questions, each user was asked the following dependent question: "How would you describe your familiarity with Java?". The user could answer within a five point Likert scale: expert (5), advanced (4), intermediate (3), beginner (2), novice (1).

The expenditure for this survey was about €500.

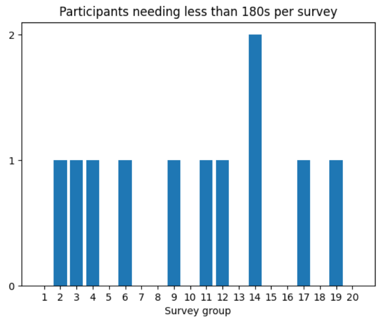
2. Threats: We identified the following threats:

- **Ill-defined Target Population:** Ensuring a well-defined target population is critical to the survey's quality. To mitigate this threat, we define our target population. Additionally, we conduct a pre-survey evaluation (see Section 4.1) to ensure the adequacy of our target population definition. Thereby, we enhance content and construct validity.
- **Sampling Method (Stratified Sampling):** Our chosen sampling method is well-defined and proven in practice. This approach ensures that our sample represents all parts of the population under investigation. This is improving the survey's external validity.
- **Insufficient Responses for Drawing Conclusions:** To prevent drawing conclusions from an insufficient number of responses, we scale our survey to an appropriate size. This guarantees that we collect a substantial volume of responses, allowing for robust statistical analysis.

3. Results: You can find an overview over the time required by the participants in the Figures 7a and **??**. Additionally you can find the amount of participants that required less than 3 minutes for one survey sheet in Figure **??**. The results of those participants might be a thread to validity to which we will come back later.



(a) Time required by participants to complete the survey.



(b) Participants per questionnaire requiring less than 3 minutes.

Figure 9: Time analysis of participants completing the prolific survey.

An overview of the time required by the participants can be found in the figures 7a and **??**. In addition, figure **??** shows the number of participants who took less than 3 minutes to complete a survey questionnaire. The results of these participants could have a negative impact on validity, which we will come back to later.

The participants' familiarity with Java is shown in Figure 10.

The ratings for each RDH for all strata combined can be found in the figures 11a and 11b. Figure 11a shows that the mean value of our original methods is 3.68, while for all7 it is 3.26. We label each method in both groups with the corresponding mean score.

Summary (RQ1 - select-well):

The readability ratings of code snippets mined from Github are not very accurate as we take the mean of all ratings for all methods and assign it to each snippet. However, the score of 3.68 is 0.XX TODO larger than the mean score for all ratings in the old dataset. Therefore we conclude that well readable assumption (Assumption 1) holds.

We evaluated whether the deviation in the ratings between the various RDHs has statistical significance. We therefore used the Mann-Whitney-U-Test comparing the ratings for all snippets for a RDH against the corresponding none snippets. You can find the results of this in Table 8.

We analysed whether the difference in ratings between the different rdhs is statistically significant. To do this, we used the Mann-Whitney U test to compare

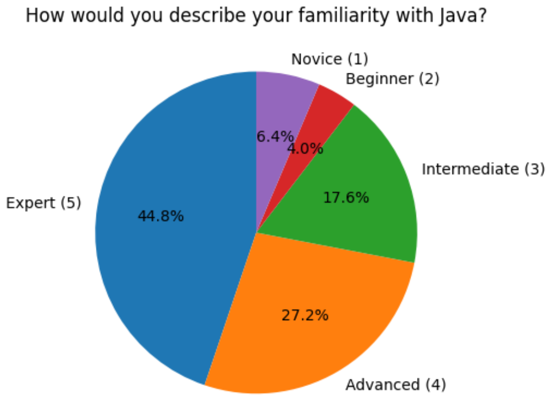


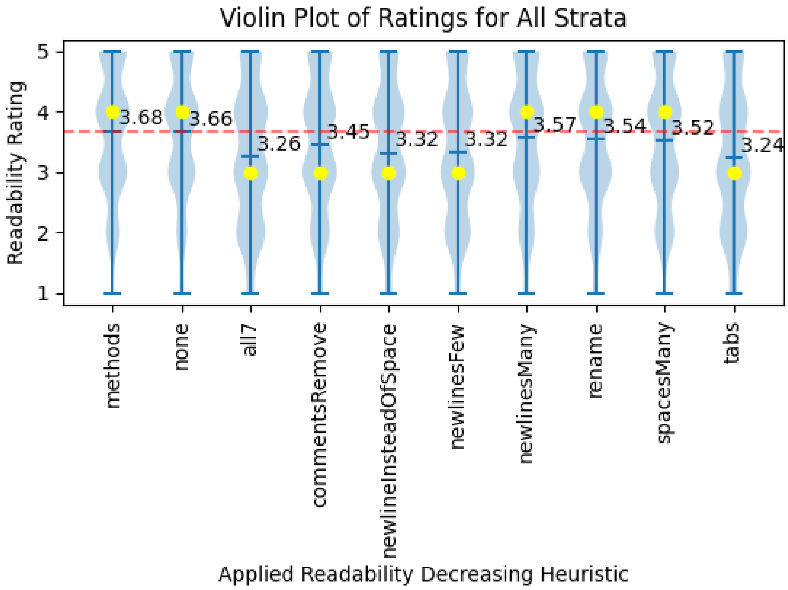
Figure 10: Familiarity of prolific survey participants with Java.

the ratings for all snippets for an rdh with the corresponding none rdh snippets. The results can be found in table 8.

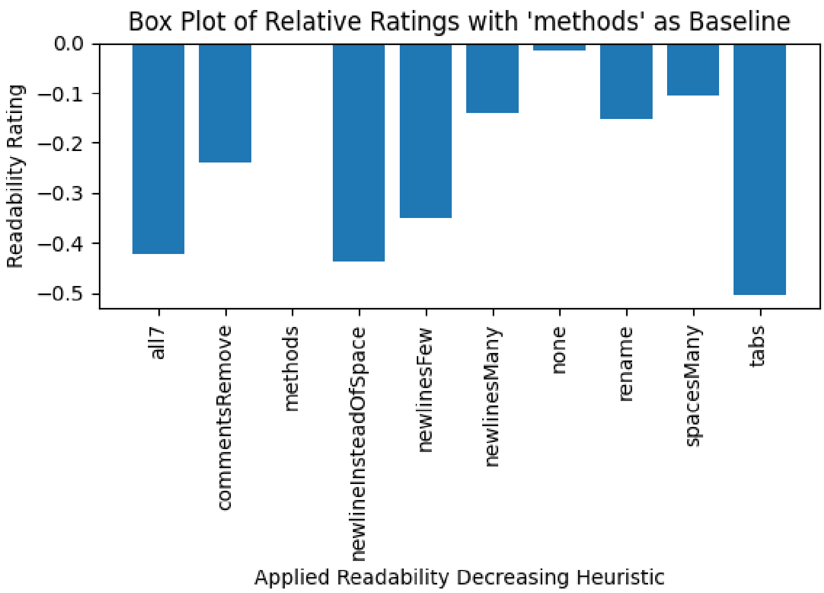
Our results suggest that no modification (none rdh) besides converting to the AST and back makes no difference to the original methods. The difference could just as well be due to random variation with a probability of 92%. If we compare the RDHs with the none methods, we can be sure that the scores of all methods except newlines many and rename are indeed statistically different from the scores of none. If we consider binary readability classification and split the data for none and Newlines Many into into two classes (not-readable: 1,2; readable: 3-5) we also get significance that the ratings for newlines many are statistically significantly different from none (TODO: Add p Values). This leaves only rename where we can not confirm statistical significance. Overall, this confirms that the heuristics actually reduce the readability of the given code.

Summary (RQ2 - generate-poor):

All of the 7 heuristics but rename decrease readability by a significant extend compared to none. We estimate the readability decrease for a certain probability oaf a certain type as can be seen in Figure 11b. The poorly readable assumption (Assumption 2) holds.



(a) Absolute survey ratings for each rdh and all stratas



(b) Relative survey ratings for each rdh and all stratas compared to original

Figure 11: Survey ratings for each rdh and all stratas.

Table 8: Mann-Whitney U test results of none against each RDH.

Comparison	p
None - Methods	9.22×10^{-1}
None - Newlines Few	5.23×10^{-6}
None - Spaces Many	4.07×10^{-2}
None - Newlines Many	3.00×10^{-1}
None - Comments Remove	3.64×10^{-3}
None - Rename	9.90×10^{-2}
None - Newline Instead Of Space	4.57×10^{-6}
None - Tabs	3.06×10^{-8}
None - All7	1.80×10^{-7}

Table 9: Performance of different dataset configurations for the same model. New-Old is training on the new dataset and fine tuning on the old one.

Train	Eval	Acc	Prec	Rec	AUC	F1	MCC
New	New	91.8 %	92.3 %	91.3 %	91.8 %	91.7 %	83.6 %
New	Old	61.9 %	63.6 %	63.6 %	63.6 %	63.6 %	23.6 %
Old	Old	53.8 %	52.6 %	77.8 %	65.2 %	62.8 %	08.7 %
Old	Old	84.7 %	87.7 %	82.3 %	85.0 %	83.7 %	70.4 %
New-Old	Old	80.4 %	84.0 %	73.8 %	78.9 %	77.2 %	60.0 %
New210	New210	80.9 %	82.7 %	77.6 %	80.2 %	78.9 %	60.9 %

4.2 MODEL TRAINING RESULTS

The results of the training, evaluation and fine-tuning can be found in the table 9.

When we train the model on the new dataset and evaluate it with 10-fold cross-validation, we obtain an average accuracy of 91.8%. However, if we evaluate the trained model on the old dataset, we get an accuracy of only 61.9%. From this we can draw some conclusions:

The towards model works well for our new dataset. However, the readability determined with the all dataset differs from the readability with our approach. Otherwise, the values for all-krod and krod-all would be similar to the value for all-all. This indicates that our dataset is not suitable for a general classification of readability, but we may have found a subproblem. However, adding more features to reduce readability and well-designed data augmentation could overcome this limitation.

While the model trained on the new dataset is able to classify readability to a certain degree, the reverse is not the case, as 53.8% is almost a random classifier. This suggests that fine-tuning a model trained on the new dataset using the entire dataset could lead to better results than the original towards model.

To check whether our implementation of the model works correctly, we also included the old-old case in the comparison. Here we achieve a very similar accuracy to Mi et al. (84.7% vs 85.3%), which indicates that our implementation of the model works correctly.

We tried to fine-tune with the old dataset by freezing different layers of the model trained with the new dataset. The best we could achieve was to freeze the input layers as well as the first convolution and pooling layer of all encoders. However, the performance of this fine-tuned model was still worse than the baseline model. This is in direct contrast to our earlier assumption that such fine-tuning could lead to better results. One explanation for this could be that the model is too small to be effective with the larger amount of data. Introducing more or bigger layers so that the model can store more features internally could solve this problem. However, this is not part of this work, in which we only deal with a new dataset (generation approach).

Summary (RQ3 - new-data):

An accuracy within the new dataset of 91.8% surpasses all previous scores. However, the accuracy of 61.9% when evaluating on the old dataset suggests, that the new training data is less valuable than the old one. We could neither improve this accuracy by applying fine-tuning. While it is possible that we successfully addressed a sub problem of readability, we are not able to improve existing readability models in general.

We also trained the model with a random sample of 210 datapoints to get a initial insight of what a change in the training size might accomplish. As we can see, the model has similar metrics as the all-all model. When we now compare the stats, for example the accuracy of the krod-krod against the all-all model, we can see, that an improvement of about 8% might be possible with a larger dataset. This suggests the importance of finding new ways of data generation for readability classification.

We also trained the model with a random sample of 210 data points to gain insight into what a change in training size might do. As we can see, the model has similar metrics to the all-all model. If we now compare the statistics, e.g.

the accuracy of the new-new compared to the old-old model, we see that with a larger dataset an improvement of about 8% is possible. Similar results are suggested by previous research on data augmentation, where an accuracy of 87.3% was achieved Mi et al. This emphasizes the importance of finding new ways of generating data for readability classification.

5 DISCUSSION

The main drawback of our approach is that we rely on estimations to create the new dataset. The score labels of our code snippets are rough estimates and not exact values. Accurate ratings would require manual review of 63460 code snippets by human annotators and is therefore not feasible.

A thread related to the study could be the sampling approach used. While we argue that we avoided spending resources on labeling rdh data that is likely not different from the original methods or rather uninteresting, this might also introduce statistical errors to our survey results.

When comparing the model performance trained on the new and old data set, it should be noted that the old data set is small. Consequently, comparisons between classifiers trained on the old dataset may also be unreliable [26].

6 CONCLUSIONS

Recent research in the field of code readability classification has mainly focused on various deep learning model architectures to further improve accuracy. Little attention is paid to the fact that only 421 labeled code snippets are available to train these models. We introduced a novel approach to generate data, with which we created a dataset of 36077 code snippets. Although our results show that the dataset does not have the same quality as previous data, it still captures code readability and could accordingly contribute to improve code readability classification accuracy in future research.

The new approach to generating data sets has an advantage that is not yet used in this work: For the first time, it is possible to generate a dataset with one well readable and a second, less readable and functionally equivalent code snippet. This could be used to train various models, including transformers. Such a transformer could take the code as input and improve its readability. We suspect that such a tool could be of great benefit to programmers.

A current limitation of the new dataset is that it only works for Java code. A suggestion for future work is to overcome this limitation by extending the tool for other programming languages. This is not trivial, as one has to adapt the readability reduction heuristics to work with another language. Furthermore, a general tool that works for all languages will be difficult if not impossible.

As Mi et al. suggested, another useful representation for code readability studies could be the syntax tree representation of code [21]. One could try to improve the performance of the towards model by adding another representation encoding extractor for Java code that automatically extracts the abstract syntax tree of the code.

A crucial aspect of code readability is naming. For the scope of methods, the most crucial part are methods names. Therefore one could improve this tool by adding a component that explicitly considers how well a method name fits its body. An important aspect of the readability of code is the naming. For the scope of methods, the method names are the most important part. Therefore, the towards model could be improved by adding a component that explicitly takes into account how well a method name matches its body. This component might be similar to Code2vec [4].

Further research could also consist of finding and evaluating other encodings that represents the code in a different way.

Another way to improve existing code readability classifiers, such as the towards model, could be to develop a different structure for some layers of the model. We suggest increasing the size and depth of the layers so that the new dataset can be made useful. Alternatively, a completely different model architecture could be developed.

The heuristics described in this work are only part of the possible heuristics that could be developed. Additional heuristics could further improve the diversity of poorly readable code. This could increase the number of internal features that a model can learn, which in turn could increase the accuracy of the model.

In summary, there are many opportunities to further investigate and thus most likely improve the classification accuracy of code readability. Our new data set and the generation approach could be useful here.

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] Duaa Alawad et al. 'An empirical study of the relationships between code readability and software complexity'. In: *ArXiv preprint* abs/1909.01760 (2019). URL: `https://arxiv.org/abs/1909.01760`.

[3] Miltiadis Allamanis, Hao Peng and Charles Sutton. 'A Convolutional Attention Network for Extreme Summarization of Source Code'. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2091–2100. URL: `http://proceedings.mlr.press/v48/allamanis16.html`.

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

[5] Kent Beck. *Implementation patterns*. Pearson Education, 2007.

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

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

[8] 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.

[9] 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.

[10] Sangchul Choi, Sooyong Park et al. 'Metric and tool support for instant feedback of source code readability'. In: *Tehnički vjesnik* 27.1 (2020), pp. 221–228.

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

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

[13] 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.

[14] Joel Hestness et al. 'Deep learning scaling is predictable, empirically'. In: *ArXiv preprint* abs/1712.00409 (2017). URL: `https://arxiv.org/abs/1712.00409`.

[15] John Johnson et al. 'An empirical study assessing source code readability in comprehension'. In: *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE. 2019, pp. 513–523.

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

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

[18] Umme Ayda Mannan, Iftekhar Ahmed and Anita Sarma. 'Towards understanding code readability and its impact on design quality'. In: *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*. 2018, pp. 18–21.

[19] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[20] Qing Mi. 'Rank Learning-Based Code Readability Assessment with Siamese Neural Networks'. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–2.

[21] Qing Mi et al. 'A graph-based code representation method to improve code readability classification'. In: *Empirical Software Engineering* 28.4 (2023), p. 87.

[22] Qing Mi et al. 'An enhanced data augmentation approach to support multi-class code readability classification'. In: *International conference on software engineering and knowledge engineering*. 2022.

[23] Qing Mi et al. 'An inception architecture-based model for improving code readability classification'. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. 2018, pp. 139–144.

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

[25] Qing Mi et al. 'The effectiveness of data augmentation in code readability classification'. In: *Information and Software Technology* 129 (2021), p. 106378.

[26] 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.

[27] Qing Mi et al. 'What makes a readable code? A causal analysis method'. In: *Software: Practice and Experience* 53.6 (2023), pp. 1391–1409.

[28] Delano Oliveira et al. 'Evaluating code readability and legibility: An examination of human-centric studies'. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 348–359.

[29] Renaud Pawlak et al. 'Spoon: A library for implementing analyses and transformations of java source code'. In: *Software: Practice and Experience* 46.9 (2016), pp. 1155–1179.

[30] 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.

[31] Talita Vieira Ribeiro and Guilherme Horta Travassos. 'Attributes influencing the reading and comprehension of source code–discussing contradictory evidence'. In: *CLEI Electronic Journal* 21.1 (2018), pp. 5–1.

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

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

[34] 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.

[35] Todd Sedano. 'Code readability testing, an empirical study'. In: *2016 IEEE 29th International conference on software engineering education and training (CSEET)*. IEEE. 2016, pp. 111–117.

[36] Milan Segedinac et al. 'Assessing code readability in Python programming courses using eye-tracking'. In: *Computer Applications in Engineering Education* 32.1 (2024), e22685.

[37] Shashank Sharma and Sumit Srivastava. 'EGAN: An Effective Code Readability Classification using Ensemble Generative Adversarial Networks'. In: *2020 International Conference on Computation, Automation and Knowledge Management (ICCAKM)*. IEEE. 2020, pp. 312–316.

[38] Yahya Tashtoush et al. 'Impact of programming features on code readability'. In: (2013).

[39] Antonio Vitale et al. 'Using Deep Learning to Automatically Improve Code Readability'. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 573–584.

[40] Greg Wilson and Andy Oram. *Beautiful code: Leading programmers explain how they think.* " O'Reilly Media, Inc.", 2007.

[41] Xin Xia et al. 'Measuring program comprehension: A large-scale field study with professionals'. In: *IEEE Transactions on Software Engineering* 44.10 (2017), pp. 951–976.

[42] Michihiro Yasunaga and Percy Liang. 'Graph-based, Self-Supervised Program Repair from Diagnostic Feedback'. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 10799–10808. URL: `http://proceedings.mlr.press/v119/yasunaga20a.html`.