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	Elisabeth Griebl

21st March 2024

Acknowledgements

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. All models use (parts of) the same labeled dataset, consisting of 421 code snippets. However, deep learning-based approaches improve with a large amount of data. Therefore, a larger 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 69k code snippets together with its novel generation approach. The generation approach involves the mining and modification of code snippets from public GitHub repositories. We validate the generated dataset using a survey with 200 participants and by training and evaluating 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

1. INTRODUCTION

Code readability is of utmost significance in the domain of software development. Together with understandability, it serves as the foundation for efficient collaboration, comprehension, and maintenance of software systems [32, 1]. Maintenance alone consumes over 70 % of the total lifecycle cost of a software product and for maintenance, the most time-consuming act is reading code [8, 10, 35, 6]. Therefore, it is important to ensure a high readability of code. To archive this, we need to measure readability. Within the last years, researchers have proposed several metrics and models for assessing code readability with an accuracy of up to 81.8 % [8, 32, 13, 36]. In more recent years, deep learning-based models are able to achieve an accuracy of up to 88.0 % [22, 23, 39, 25, 19, 20].

However, a major limitation of these models is not their architecture, but the amount of available data for Java code readability classification, which comprises 421 code snippets [8, 13, 36]. The current training data originates from questionnaires where humans manually labeled the code snippets. This has two drawbacks: Firstly, manual labeling requires a lot of effort. Secondly, the dataset is very small for deep learning, as this requires a large amount of data [15]. To address these drawbacks, we aim to automatically generate more training data.

The idea of this work is to investigate whether it is possible to achieve higher accuracy in code readability classification using automatically generated data.

As a first step we mine GitHub[1] repositories with high code quality. Our criteria for high code quality are an elevated number of stars and forks, the use of method comments and the compliance with a checkstyle[2] specification. For example, a consequence of using checkstyle is that the readability of the code is increased. Therefore it is reasonable to assume that repositories that use checkstyle are more readable than other repositories (Section 3.3). We select Java files from the repositories that meet our criteria, extract methods from the Java classes in these files and label them as well readable (Assumption 1).

As a second step, all selected Java files are modified so that the code is subsequently less readable. You can find an exemplary result of this in Listing 1. We extract methods from these Java files and label them as poorly readable (Assumption 2).

After both steps, the result is a new automatically generated dataset for code readability classification, referred to as the mined-and-modified dataset.

---

[1]https://github.com/, accessed: 2024-02-29
[2]https://checkstyle.sourceforge.io/, accessed: 2024-02-09

```
1   /**
2   * Calculates the factorial of a given number.
3   *
4   * @param number The number to calculate the factorial of.
5           * @return The factorial of the input number.
6   */
7   public int calculateFactorial(int number) {
8           int factorial = 1;
9           for (int i = 1; i <= number; i++) {
10                  factorial *= i;
11          }
12          return factorial;
13  }
```

(a) An example of a simple and well readable Java method.

```
1   public int foo(int x) {
2           int y = 1;
3           for (int z=1; z<=x; z++) {
4           y *= z;}
5           return y;
6   }
```

(b) The same example as in Listing 1a but modified for poor readability.

Listing 1.: Well readable (Listing 1a) vs. poorly readable (Listing 1b) code.

How can we modify code so that it is less readable afterwards? We introduce a tool called Readability Decreaser (REDEC). It uses a collection of heuristics that reduce the readability of Java files. Such heuristics are replacing spaces with newlines or increasing the indentation of a code block by a tab or multiple spaces. Most changes also decrease readability when applied in reverse (like replacing newlines with spaces or decreasing indentation).

The Java code is syntactically the same, before and after applying REDEC. Functionality does not change either. However, if various modifications are applied many times, those modifications are capable of lowering the readability of source code, as Listing 1 shows.

We conducted a user study to validate the assumptions that the mined methods from the selected Java class files are well readable (Assumption 1) and that REDEC can modify the code to be poorly readable (Assumption 2). We thereby ensure that the mined-and-modified dataset contains well and poorly readable code, both of which are labeled correctly. Additionally, we verify the dataset by

training and evaluating a state of the art deep learning model with it. We use the readability model of Mi et al. [25] for this.

Our contributions are as follows:

- Although existing datasets [8, 13, 36] have different structures we combine and unify them into one merged dataset (Section 3.1).

- We reason that Mi et al. used only part of the available data for training and evaluating their readability classification model [25] (Section 3.2).

- We develop an approach for mining well readable Java methods and thereby achieve automated dataset generation for code readability. This allows us to introduce the new mined-and-modified dataset (Section 3.3).

- We succeed in creating a tool that can automatically decrease the readability of Java class files (Section 3.4): REDEC.

- We show a representative and resource-effective sample approach, since there is an enormous amount of possible combinations to sample methods for a user study (Section 3.5).

- We demonstrate limitations of the model of Mi et al. [25] (Section 3.6).

- We show through a survey (Section 4.1) that the well readable assumption (Assumption 1) and the poorly readable assumption (Assumption 2) hold.

- We show that the mined-and-modified dataset can be used for code readability classification models by training and evaluating the model of Mi et al. [25] with and without the mined-and-modified dataset (Section 4.2).

Our automated approach for creating a readability classification dataset is effective. The mined-and-modified dataset contains well readable (Assumption 1) and the poorly readable (Assumption 2) methods. We infer from the model training results (Section 4.2) that the mined-and-modified dataset captures different aspects of code readability as previous datasets [8, 13, 36].

Our code is publicly available on GitHub:

https://github.com/LuKrO2011/master-thesis

2. BACKGROUND AND RELATED WORK

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

2.1. CODE READABILITY

When talking about *code readability classification* we need to clarify what this term means. Therefore, we start with an overview of definitions of code readability.

Buse and Weimer provide the earliest definition [8]: 'We define readability as a human judgment of how easy a text is to understand.'

Tashtoush et al. combine numerous other aspects from various definitions. According to them code readability can be measured by looking at the following aspects [40]:

- 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. define readability as 'what makes a program easier or harder to read and apprehend by developers' [29].

Mi et al. summarize code readability as 'a human judgment of how easy a piece of source code is to understand' [24]. This comes close to the definition of Buse and Weimer [8].

There are various related terms to readability: Understandability, usability, reusability, complexity, and maintainability [40]. 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, whereas 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 [36, 18, 43, 5]:

- Complexity

- Usage of design concepts

- Formatting

- Source code lexicon

- Visual aspects (e.g., syntax highlighting)

Posnett et al. state that readability is the syntactic aspect of processing code, while understandability is the semantic aspect [32].

Based on Posnett et al., Scalabrino et al. write [36]: 'Readability measures the effort of the developer to access the information contained in the code, while understandability measures the complexity of such information'.

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 [37].

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

- Ease/complexity of understanding/comprehension/apprehension

- Human judgment/assessment

- Effort of the process of reading (differentiation to understandability)

Considering these aspect, the definition of Oliveira et al. captures the meaning of code readability best. Therefore, we use their definition:

*Code readability is '*what makes a program easier or harder to read and apprehend by developers*'* [29].

Researchers proposed several metrics and conventional models for assessing code readability with an accuracy of up to 81.8 % [8, 32, 13, 36]. In recent years, deep learning-based models are able to achieve an accuracy of up to 88.0 % [22, 23, 39, 25, 19, 20] on available datasets. 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, 32, 13, 36]. Those approaches used handcrafted features to calculate how readable a piece of code is. Handcrafted features are for example the number of identifiers, parentheses, blank lines or comments. In general, they are calculated by counting the number of certain tokens or words or by measuring certain code properties [36]. In contrast to machine and deep learning-based approaches (Section 2.3), humans determine the features of conventional models based on domain knowledge.

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

Model	Type	Accuracy
Buse [8]	Conventional	76.5 %
Possnet [32]	Conventional	71.7 %
Dorn [13]	Conventional	78.6 %
Scallabrino [36]	Conventional	81.8 %
Mi_IncepCRM [22]	Deep Learning	84.2 %
Mi_DeepCRM [23]	Deep Learning	83.8 %
Sharma [39]	Deep Learning	84.8 %
Mi_Towards [25]	**Deep Learning**	**85.3 %**
Mi_Ranking [19]	Deep Learning	83.5 %
Mi_Graph [20]	Deep Learning	88.0 %

Scalabrino et al. were able to achieve up to 81.8 % accuracy in code readability classification using handcrafted features [36].

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 (Table 1).

IncepCRM was the first introduced deep learning model for code readability classification. It automatically learns multi-scale features from source code with minimal manual intervention [22].

In a follow up paper Convolutional Neural Networks (ConvNets) were introduced to code readability classification in a model called DeepCRM. Unlike previously, DeepCRM employs three ConvNets with identical architectures and was trained on differently preprocessed data [23].

Another study proposed 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) [39]. It was able to surpass the accuracy of previous readability classification models as shown in Table 1.

Up to this point, the limitation of deep learning-based code readability models was to focus on structural features primarily. Mi et al. addressed this by proposing a model, *Mi_Towards*, that extracts features from visual, semantic, and structural

aspects of source code. Using a hybrid neural network composed of BERT, CNN, and BiLSTM, the proposed model processes RGB matrices, token sequences, and character matrices to capture various features of source code [25].

Previously code readability classification was considered mainly as a task that is applied to a single code snippet at once. A new approach was introduced which frames the problem as a ranking task. The proposed model employs siamese neural networks to rank code pairs based on their readability [19].

All accuracy scores in two class classification (Table 1) were surpassed by the introduction of 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 [20].

Until now researchers introduced many deep learn architectures and components to various classification models with the goal to surpass previous accuracy scores. The common limitation of all models is a dataset consisting of 421 code snippets. Therefore, the main contribution of this work is not a model that outperforms a state of the art model but rather a new dataset. For evaluation we opted for the *Mi_Towards* model (hereinafter referred to as towards model) from Mi et al. [25]. We did not choose the best performing one, *Mi_Graph* [20], 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 (Section 3.4).

2.4. DATA AUGMENTATION

All the mentioned models were trained with (parts of) the data from Buse and Weimer [8], Dorn [13] and Scalabrino et al. [36], consisting of a total of 421 Java code snippets. The data was generated with surveys. They therefore asked developers several questions, including how readable the proposed source code is [8, 13, 36]. We refer to the combination of their datasets as *merged* dataset.

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

A recently published paper addresses the challenge of acquiring a larger amount of labeled data, using augmentation. The researchers proposed to artificially expand the training set instead of the time-consuming and expensive process of obtaining labels manually. 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. They advance to a classification accuracy of 87.3 % [24]. Lately, researchers successfully enhanced data augmentation by incorporating domain-specific data transformation and Generative Adversarial Networks [21]. The results of both show, that more data has a significant impact on the reached classification accuracy. However, they artificially augment using the 421 code snippets of the merged dataset. Therewith, their augmented data is based on these 421 code snippets. Our new mined-and-modified dataset does not have this limitation.

Recently researchers developed a methodology to identify readability-improving commits, creating a dataset of 122k commits from GitHub's revision history. They used this dataset to automatically identify and suggest readability-improving actions for code snippets. They trained a T5 model to emulate developers' actions in improving code readability, achieving a prediction accuracy between 21 % and 28 %. The empirical evaluation shows that 82-91 % of the dataset commits aim to improve readability, and the model successfully mimics developers in 21 % of cases [42]. This shows the potential of a large dataset, but the used dataset and model results are hardly comparable with previous studies due to the usage of commits instead of code snippets. We, however, continue to use code snippets.

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 works.

Fakhoury et al. showed that previous models do not capture what developers think of readability improvements. Therefore, they analyzed 548 GitHub[3] commits manually. They suggest considering other metrics, such as incoming method calls or method name fitting [14].

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 analyzed tasks and response variables used in studies comparing programming constructs, coding idioms, naming conventions, and formatting guidelines [29].

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

---

[3]`https://github.com/`, accessed: 2023-07-25

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 [34].

Choi, Park et al. introduced an enhanced source code readability metric for quantitatively measuring code readability in the software maintenance phase. The proposed metric achieves a substantial explanatory power of 75.7 %. Additionally, the authors developed a tool named 'Instant R. Gauge', integrated with Eclipse IDE. It provides real-time readability feedback and tracks readability history, allowing developers to gradually improve their coding habits [9].

Mi et al. aimed to understand the causal relationship between code features and readability in their paper. To overcome potential spurious correlations, the authors proposed 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, revealed that the average number of comments positively impacts code readability, while the average number of assignments, identifiers, and periods has a negative impact [26].

Segedinac et al. introduced a novel approach for code readability classification using eye-tracking data from 90 undergraduate students assessing Python code snippets [38].

Although the approaches mentioned are not directly related to our work, they are related to the area of code readability classification and could have an impact on future research in this area.

2.6. DATA GENERATION

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

One concept we employ is from Allamanis et al. They cloned the top open source Java projects on GitHub[4] for training a Deep Learning model. They selected the top projects by taking the sum of the z-scores of the number of watchers and forks of each project. The projects have thousands of forks and stars and are widely used among software developers, thus the authors assumed the code within to be of good quality [3]. We also use fork and star counts as criteria for well readable code (Assumption 1).

---

[4]`https://github.com/`, accessed: 2023-07-25

Another concept we employ is the intentionally degradation of source code to create a dataset.

In the field of vulnerability detection of source code, Dolan-Gavitt et al. introduced a technique to automate the insertion of bugs into program source code. The generated dataset was used for vulnerability discovery tool evaluation [12].

Pradel and Sen apply a similar approach for the purpose of bug detection through a deep learning classifier. They generate training data through insertion of bugs into existing code. This process involves simple program transformations that introduce likely incorrect code examples, enabling the training of a classifier to distinguish between correct and incorrect code [33]. Yasunaga and Liang did similarly corrupt working programs into broken ones. They also suggest to pretrain models effectively before fine-tuning them on a manually labeled, target task [44]. We will employ the idea of pre and fine-tuning as well. Allamanis et al. enhanced their approach by generating the dataset as a by product of co-training two models. One model is a detector that learns to detect and repair bugs in the code. The second model is a selector, that learns to create buggy code that is hard to detect [2].

Loriot et al. took the idea and shifted the are to formatting. This topic comes closest to our topic of code readability. They created a model that is able to fix Checkstyle[5] violations using Deep Learning. In a first step, they inserted formatting violations into the code based on a project specific format checker ruleset. They then used a LSTM neural network that learned how to undo those injections. Their approach is working on abstract token sequences. Again, they generated the data in a self-supervised manner [17].

While the approach of intentional denigrating of source code to create a dataset was applied to the fields of vulnerability detection, bug detection and code formatting, we are the first to apply it to the field of code readability.

2.7. ABSTRACT SYNTAX TREE

When decreasing the readability of code (Section 3.4), we make use of the Abstract Syntax Tree (AST). An AST is a hierarchical structure and represents the syntactic structure of source code. It is composed of nodes and edges. Each node represents a language construct, such as expressions, statements, or declarations, while edges denote the relationships between the nodes.

An AST abstracts away specific syntactic details, focusing on the relationships between the nodes. ASTs are widely used in compilers, static analysis tools, and

---

[5]`https://checkstyle.org/`, accessed: 2023-07-25

refactoring engines to perform tasks such as semantic analysis, optimization and code transformation [27]. The latter is our use case.

We transform source code into the AST, make certain modifications to the AST (Table 3) and then transform the AST back into the code representation. The conversion back is done with a so-called Pretty Printer. Compared to performing the modifications on the code, this has certain advantages, which are further detailed in Section 3.4.

2.8. STRATIFIED SAMPLING

In statistical analysis, we often encounter situations in which the population can be divided into subgroups. Each group represents a particular characteristic. Stratified sampling is a technique used to ensure that each of these subgroups is adequately represented in the sample [41].

In preparation for the user study (Section 4.1), we use stratified sampling when creating the questionnaires. For this purpose, all code snippets are divided into homogeneous groups, so called strata, which are based on similarities with regard to certain code properties. Such properties are method length, line length or average identifier length. Within each stratum, the samples are then randomly selected to ensure that the sample is representative of the population. This method allows us to evaluate our approach to data generation more accurately. We explain the usage of stratified sampling in more detail in Section 3.5.

3. MINED AND MODIFIED CODE FOR DATASET GENERATION

The following subsections are about our approach.

3.1. WORK ON EXISTING DATASETS

Most of the related work (Section 2) uses a combination of the data of Buse and Weimer [8], Dorn [13] and Scalabrino et al. [36]. The raw data from their surveys can be downloaded [6], but their data is not uniformly formatted. The data includes ratings that are not Java code snippets. It consists of differently formatted individual ratings rather than the mean of the ratings used for training machine learning models. In contrast to our mined-and-modified code snippets theirs do not all have the scope of a method, but instead consist of a few lines of code.

---

[6]`https://dibt.unimol.it/report/readability/`, accessed: 2024-02-18

We converted and combined these three datasets into one: *code-readability-merged* or for short *merged*. In recent years, Huggingface[7] established as the pioneer in making models and datasets available. Therefore, we decided to publish the merged dataset on Huggingface[8].

3.2. CLASSIFICATION CONSIDERATIONS

When classifying the readability of code, the state of the art is to perform a binary classification into well readable and poorly readable code [22, 23, 39, 25, 19].

However, code readability classification is not a binary classification task per se. Mi et al. introduced a third, neutral class to address this problem [20]. When rating code snippets, previous studies used a Likert scale [16] from 1 (very unreadable) to 5 (very readable) [8, 13, 36]. While the amount of classes varies, one can encode the data internally as a single-value representation between 0 and 1 where a higher value means higher readability. The output of the model is well readable if the value after the last layer is above 0.5 and poorly readable otherwise.

Our evaluation model is the towards model of Mi et al. [25]. They transformed the rating scores into binary classification using a single-value representation as follows: First, they calculated the mean values of all scores. In a second step, they ranked the snippets according to their mean score. After that, they labeled the top 25 % of the data as well readable (1.0) and the bottom 25 % as poorly readable (0.0). They did not use the 50 % of the data in between [25].

While this transformation is fine in principle, especially with the argument that the data in the middle is neither well readable nor poorly readable, it has drawbacks that Mi et al. use only 50 % of the available data for model training and evaluation:

Firstly, they further reduce the available data is from 421 to 210 Java code snippets. Note that a bottleneck in readability classification is the small amount of available data. So this is a significant loss.

Secondly, they performed evaluation with only those 210 snippets. Thus, they evaluated the model 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.

---

[7] `https://huggingface.co/`, accessed: 2024-02-18
[8] `https://huggingface.co/datasets/se2p/code-readability-merged`, accessed: 2024-02-18
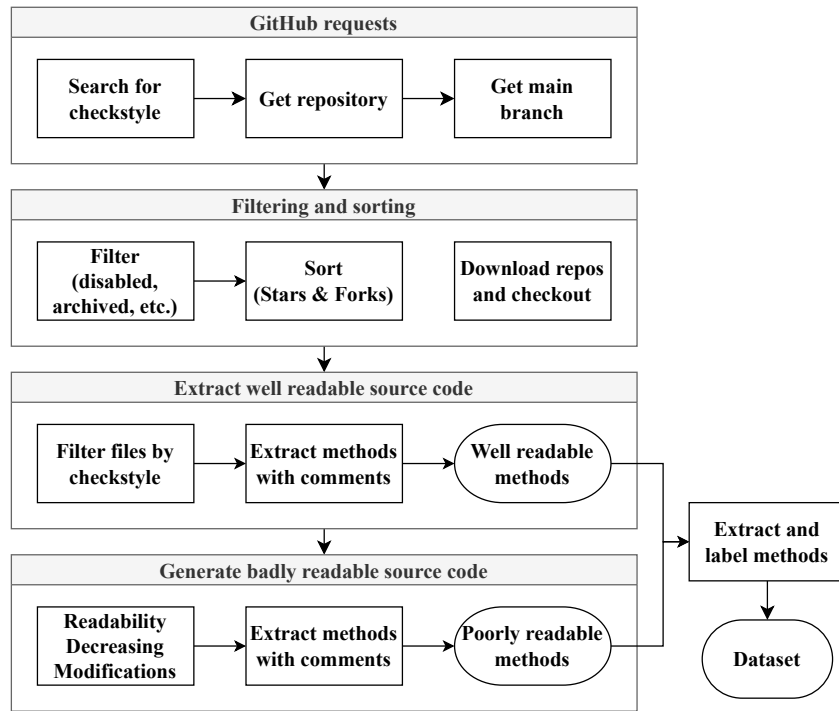
Figure 1.: The used dataset generation approach.

However, we continue to use the binary classification approach as well as evaluate on the 210 code snippets to make our results comparable with the results of Mi et al. [25].

3.3. DATASET GENERATION APPROACH

We refer to the dataset generated by our approach as the *mined-and-modified* dataset. Since we extract methods from code, *code snippets* and *methods* are synonyms for our mine-and-modify approach. This is not the case with the merged dataset, as there a code snippet is not necessarily an entire method.

In contrast to previous datasets for readability classification, we generate our dataset using an automated approach. The aim is to mine and modify code from GitHub to obtain both, well readable and poorly readable methods. This approach is novel to the best of our knowledge.

Our approach is divided into four parts (Figure 1). We use the first three steps to mine well readable Java code. In a final step, we modify the well readable code to achieve our second goal, namely poorly readable source code.

We start by querying the GitHub REST API[9] 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 are a fork of another repository, are archived or are disabled. Additionally, we delete repositories whose language is not Java and those lacking a minimum of 20 stars and forks. We sort the remaining repositories by their star and fork count (equally weighted). Then we clone the 100 best and check out their main branch.

In a third step, we run checkstyle[10] against the project's own checkstyle configuration to obtain all Java class files that pass the own checkstyle test. For this we use a tool from Maximilian Jungwirth[11] which is based on *Styler*[12] [17]. From the Java class files that passed checkstyle, we extract all methods that have a comment of any kind at the beginning of the method. This results in 39312 methods which we assume to be well readable.

The fourth step is to generate poorly readable code from the well readable one. To do this, we use the proposed REDEC tool (Section 3.4). We then extract all methods starting with a comment. Originally, we planned to not require comments for the poorly readable dataset part. However, it turns out that in this case, all well readable methods have a comment, while most of the poorly readable do not have one. This leads to shortcut learning, whether a method has a comment or not, instead of learning to distinguish the methods by all other criteria as well.

Finally, we take the mined well readable methods and the the modified poorly readable methods and combine them into the mined-and-modifed dataset. We remove code snippets that are identical for the original and the modified variant (Section 3.4) and balance the dataset using random sampling. We label each well readable method and each poorly readable method with the corresponding mean rating scores obtained through a later user study (Section 4.1). The created mined-and-modifed dataset consists of 69276 code snippets.

3.4. REDEC: READABILITY DECREASER

In this section we take a look at how we achieved to decrease the readability of code using the Readability Decreaser (REDEC). REDEC uses a set of code modification heuristics that are applied to Java files. We call these Readability Decreasing Modificatations.

---

[9]`https://docs.github.com/en/rest`, accessed: 2024-02-15
[10]`https://checkstyle.sourceforge.io/`, accessed: 2024-02-15
[11]`https://github.com/sphrilix/styler2.0`, accessed: 2024-03-21
[12]`https://github.com/ASSERT-KTH/styler`, accessed: 2024-03-11

Table 2.: All Readability Decreasing Modificatations with explanation and example.

#	Modification	Description	Example
1	`newline`	Replace a newline with none or multiple ones	Listing 2b, Lines 5-6
2	`incTab`	Replace a tab indentation with none or multiple ones	Listing 2b, Line 5
3	`decTab`	Replace a tab outdentation with none or more ones	Listing 2b, Line 7
4	`space`	Replace a single space with multiple ones	Listing 2b, Line 1
5	`newLine InsteadOf Space`	Replace a space with a newline	Listing 2b, Line 3-4
6	`spaceInsteadOf Newline`	Replace a newline with a space	Listing 2b, Line 2
7	`incTabInsteadOf DecTab`	Replace a tab outdentation with a tab indentation	Listing 2b, Line 9
8	`decTabInsteadOf IncTab`	Replace a tab indentation with a tab outdentation	Listing 2b, Line 8
9	`renameVariable`	Rename a variable declaration and its usages	Listing 2b, Line 1-3
10	`renameField`	Rename a field declaration and its usages	Listing 3b, Line 4
11	`renameMethod`	Rename a method declaration and its usages	Listing 2b, Line 1
12	`inlineMethod`	Replace a method call with the called code	Listing 3b, Line 7-8
13	`removeComment`	Remove a comment	Listing 2b, Line 1
14	`add0`	Add a zero to a number	Listing 2b, Line 2
15	`insertBraces`	Insert superfluous braces	Listing 2b, Lines 3-4
16	`starImport`	Replace a specific imports with a star-import	Listing 3b, Line 1
17	`inlineField`	Replace a static field with its value	Listing 3b, Line 7
18	`partially Evaluate`	Partially evaluate a constant	Listing 3b, Line 4

```
1   /**
2    * This method determines the sign of a given number and prints a
     ↪  corresponding message.
3    *
4    * @param number The input number to be checked.
5    */
6   public static void checkNumberSign(int number) {
7           if (number > 0) {
8                   System.out.println("Number is positive");
9           } else if (number < 0) {
10                  System.out.println("Number is negative");
11          } else {
12                  System.out.println("Number is zero");
13          }
14  }
```

(a) An example of a simple and well readable Java method.

```
1   public static void m0(int   v0) {
2           if (v0 > (0+0)) { System.out.println("Number is positive");
3           } else if ((v0 <
4           0)) {
5                   System.out.println("Number is negative");
6
7           } else {
8                   System.out.println("Number is zero");
9   } }
```

(b) The same example as in Listing 2a but modified for poorer readability.

Listing 2.: Well readable (Listing 2a) vs. poorly readable (Listing 2b) Java methods.

```
1   import java.util.Random;
2
3   public class TimeConverter {
4           public static final int MINUTES_PER_HOUR = 60;
5           public static final int HOURS_PER_DAY = 24;
6           public static final int MINUTES_PER_DAY = MINUTES_PER_HOUR *
        ↪  HOURS_PER_DAY;
7           public static final int SEED = 4242;
8
9           public int getRandomDays(int max) {
10                  Random random = new Random(SEED);
11                  return random.nextInt(max);
12          }
13
14          public int randomDaysInMinutes() {
15                  int days = getRandomDays(10);
16                  return days * MINUTES_PER_DAY;
17          }
18  }
```

(a) An example of a simple and well readable Java class file.

```
1   import java.util.*;
2
3   public class TimeConverter {
4           public static final int f0 = 1440;
5
6           public int randomDaysInMinutes() {
7                   Random random = new Random(4242);
8                   int days = random.nextInt(10);
9                   return days * f0;
10          }
11  }
```

(b) The same example as in Listing 3a but modified for poorer readability.

Listing 3.: Well readable (Listing 3a) vs. poorly readable (Listing 3b) Java class
files.

REDEC initially converts the Java code of a well readable Java class file into an Abstract Syntax Tree (AST, Section 2.7) using the spoon library[13] [30]. In the end REDEC parses the AST back to Java code using the Pretty Printer of the spoon library[14] [30]. If nothing else is done, this results in `just-pretty-print`. Note that the code of `just-pretty-print` is slightly different from the original code, as the Pretty Printer overwrites the styling and formatting of the original code by its default formatting.

Various modifications can be made between the two steps and during pretty-printing. You can find a description of each modification in Table 2 and examples of the modifications in Listing 2 and Listing 3.

In Table 2 tab *indentation* refers to the process of adding a tab ($\backslash t$) while *outdentation* refers to the opposite, namely removing a tab ($\backslash t$). For example:

1. The current tab count is 1 ($\backslash t \langle CODE \rangle$) (Listing 2a Line 7).

2. In the next line, we perform a tab indentation.

3. The current tab count is now 2 ($\backslash t \backslash t \langle CODE \rangle$) (Listing 2a, Line 8).

4. In the next line, we perform a tab outdentation.

5. The current tab count is now 1 ($\backslash t \langle CODE \rangle$) (Listing 2a, Line 9).

REDEC performs one part of the modifications on the Abstract Syntax Tree (AST) representation of the Java files. It executes another part when pretty-printing the AST back into Java files. The first part is executed on the AST to ensure that the functionality stays the same. For example, when REDEC renames a method, the declaration along with all references are renamed. The second part is not encoded in the AST and is therefore executed at source code level immediately after the reverse transformation. For example, the AST does not encode line breaks, so that changes to these line breaks must be applied to the source code rather the AST. You can see which modifications are applied when in Table 3.

REDEC applies a modification to each occurrence of the object it refers to with a specified probability. Due to the use of probabilities it can happen that no modification is applied. For example, we execute REDEC and set only `removeComment` to a probability of 10 %. Then the tool removes each comment of the given Java class files with a probability of 10 %. The exact amount of removed comments is uncertain. It can happen (especially for short methods within the class files) that a method is not changed at all. For example, if a method only has a single comment and we use `removeComment`, the probability that the method is not changed (besides the changes of `just-pretty-print`) is 90 %.

---

[13]`https://spoon.gforge.inria.fr/`, accessed: 2024-15-02
[14]`https://spoon.gforge.inria.fr/`, accessed: 2024-15-02

Table 3.: Available Readability Decreasing Modificatations along with their execution type (on AST or Code), their configuration type (a array of probabilities or a single one) and whether they are included in the final dataset. Appendix II for a concrete configuration example.

#	Modification	AST/Code	Config. Type	In Dataset
1	`newline`	Code	Array	✓
2	`incTab`	Code	Array	✓
3	`decTab`	Code	Array	✓
4	`space`	Code	Array	✓
5	`newLineInsteadOfSpace`	Code	Single	✓
6	`spaceInsteadOfNewline`	Code	Single	✓
7	`incTabInsteadOfDecTab`	Code	Single	✓
8	`decTabInsteadOfIncTab`	Code	Single	✓
9	`renameVariable`	AST	Single	✓
10	`renameField`	AST	Single	✓
11	`renameMethod`	AST	Single	✓
12	`inlineMethod`	AST	Single	
13	`removeComment`	Code	Single	✓
14	`add0`	AST	Single	
15	`insertBraces`	AST	Single	
16	`starImport`	AST	Single	
17	`inlineField`	AST	Single	
18	`partiallyEvaluate`	AST	Single	

By default, REDEC generates the new identifiers for the rename modifications (`renameVariable`, `renameField` and `renameMethod`) in an iterating manner. For each class file we start with *v0* for variables, *f0* for fields and *m0*. We increase the index of each (0 at the beginning) by 1 whenever a name is used. We also added a mode that uses Code2Vec [4] for the generation of identifiers for `renameMethod` instead. With that we can predict more realistic method names. Code2Vec generates multiple method name predictions at once. By picking not the best one but instead the one with the longest name we aim to decrease readability while choosing realistic method names.

REDEC does not support the removal of spaces, as this would cause keywords or identifiers to merge.

For the final dataset we exclude some of the modifications as we can see in Table 3. We exclude `inlineMethod` as it increased the length of methods drastically and

made the methods too long. While `starImport` might have an impact on the readability of class files, it has none on methods, since in Java the import statements are not within the methods. As we finally extract methods for our dataset, `starImport` has no impact. We chose not to include `add0`, `insertBraces`, `inlineField` and `partiallyEvaluate` for the reason of a limited survey capacity. For the same reason, we do not investigate the usage of Code2Vec for `renameMethod` either.

The REDEC tool works with a configuration file in which one can specify a probability for each available modificatation. For modificatations of configuration type *Array* (`newline`, `incTab`, `decTab` and `space`), an array of probabilities must be defined for the respective number of replacements. The probabilities of the array must sum up to 1. For modificatations of configuration type *Single* a single probability must be defined (Table 3). For example, `spaceInsteadOfNewline` can be configured with 0.05 meaning that each space is replaced with a newline ($\backslash n$) with a probability of 5 %. `space` can be configured with $[0.0, 0.7, 0.2, 0.1]$ meaning that each space is replaced with

- no space with a probability of 0 %

- a single space with a probability of 70 % (no change)

- two spaces with a probability of 20 %

- three spaces with a probability of 10 %

We select the probabilities for the generated code snippets so that they are still realistic, i.e. that they could also be written by humans. You can find the configurations in Table 4 and an exemplary file for `just-pretty-print` in Appendix II. `all7` is the average of the other 7 configurations: The probabilities of the other 7 configurations are added to one configuration and each probability is divided by 7. There is a special case regarding `removeComment`: `comments-remove` and `all7` use `removeComment` at 100 % for the user survey (Section 4.1). The reason for this is that a method without comments is realistic. For training and evaluation of the deep-learning model the probability was set to 10 % (for both affected configurations, `all7` and `removeComment`) to avoid shortcut learning (Section 4.2).

After applying REDEC to the Java files, we extract the methods. We require a method comment for all methods (Section 3.3). We therefore use `removeComment` after completing the method extraction.

Table 4.: Chosen configurations and their probabilities for the Readability Decreasing Modificatations. For better readability, we write *addX* and *removeX* instead of the array configurations. For example, we write *Add1Space: 20 %* and *Add2Spaces: 10 %*, but the configuration is *space: [0.0, 0.7, 0.2, 0.1]*.

Configuration	Probabilities
just-pretty-print	-
comments-remove	removeComment: 10 % or 100 %
newline-instead-of-space	newLineInsteadOfSpace: 15 %
newlines-few	removeNewline: 30 %
	spaceInsteadOfNewline: 5 %
newlines-many	add1Newline: 15 %
	add2Newlines: 5 %
rename	renameVariable: 30 %
	renameField: 30 %
	renameMethod: 30 %
spaces-many	Add1Space: 20 %
	Add2Spaces: 10 %
	spaceInsteadOfNewline: 5 %
tabs	remove1IncTab: 20 %
	add1IncTab: 10 %
	remove1DecTab: 10 %
	add1DecTab: 10 %
	incTabInsteadOfDecTab: 5 %
	decTabInsteadOfIncTab: 5 %
all7	all probabilites/7

3.5. CONSTRUCTION OF QUESTIONNAIRES

We evaluate the generated dataset and the new approach with a survey (Section 4.1). However, we cannot evaluate all mined and modified methods as the original methods consist of 39312 samples and there is an infinite number of possible configurations for REDEC that can be applied to each method. Therefore, we apply stratified sampling (Section 2.8) [41], create specific configurations and select the methods for our survey on this basis. You can find an overview of the approach in Figure 2.

Stratified sampling: We want to categorize the methods to make it easier to decide which methods to select. We also want to avoid over-representing very simple methods, such as Getters and Setters. Therefore, our first step is to apply
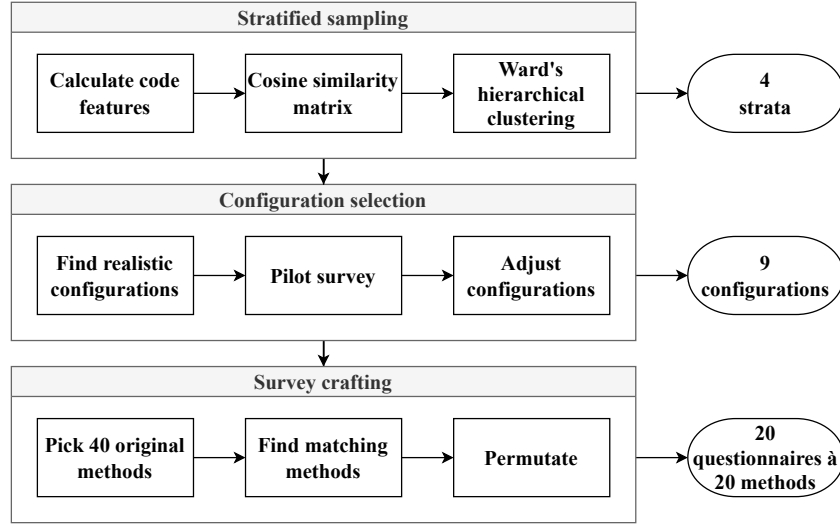
Figure 2.: Steps performed to craft questionnaires from the mined-and-modified dataset.

stratified sampling [41]. This allows us to divide the methods into different groups, so called strata, based on handcrafted features (Section 2.2). Since we want to compare the original methods with their modified variants later, we perform the sampling only for the original methods and add the REDEC methods in a later step.

We first calculate the handcrafted features for the original code snippets. We therefore use a tool of Scalabrino et al. [36]. We calculate a 110-dimensional feature vector for each original code snippet. Next we compute the cosine similarity matrix between all feature vectors using scikit[15]. Finally, using the fastcluster implementation [28] of Ward's hierarchical clustering we cluster the methods into an arbitrary amount of strata.

In each clustering step x, the two strata with the smallest cosine distance between their feature vectors are merged into one. This distance is the merged distance of the step, MD_x. For further investigation, we also calculate the difference of the merge distance to the merged distance of the previous step $x - 1$: Difference to previous $= MD_x - MD_{x-1}$. For each step, you can find the merge distance together with the difference to the previous step in Figure 3.

---

[15]`https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html`, accessed: 2024-02-20
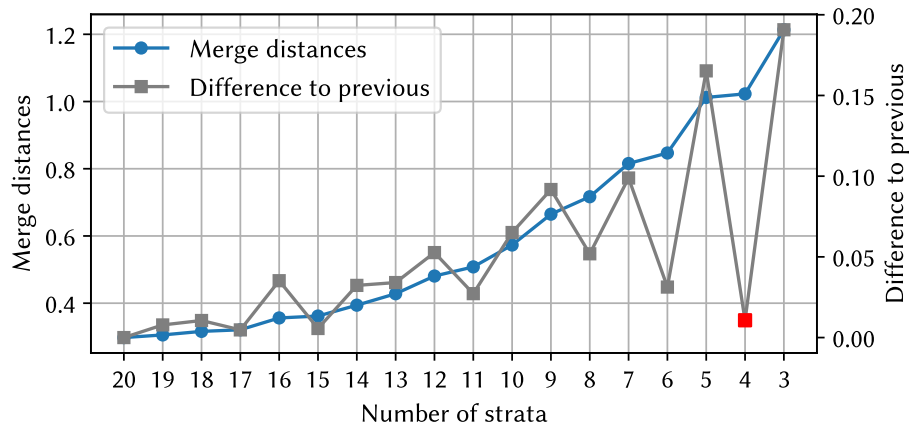
Figure 3.: Merge distances and difference to previous merge distance. The optimal difference to previous merge distance is highlighted in red.

The graph shows that a layer 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. Additionally, 4 is the size with the last possibility for a small merge distance.

We identify the type of methods within each of the 4 strata as described in Table 5. We also add the amount of methods within each stratum. Stratum 0 comprises simple methods, such as Getters and Setters. Stratum 1 consists of the most complex methods across all strata. Stratum 2 is characterized by methods containing numeric values with unexplained meaning, commonly referred to as magic numbers. Compared to the other strata, Stratum 2 has a smaller scope of 78 methods. Stratum 3 contains methods of medium complexity that exceed the simplicity of the Getters and Setters in Stratum 0, but do not come close to the complexity of the methods in Stratum 1. Overall, we split the methods according to their complexity, ranging from simple (Stratum 0) to medium (Stratum 3) to complex (Stratum 1), while Stratum 2 is a by-product.

Configuration selection: The next step is to find realistic configurations for the REDEC tool. We select the first configurations by manually checking individual outputs of REDEC. Then we conduct a pilot study and adjust the configurations based on the feedback. The final 9 configurations can be found in Table 4. Together with the original methods this results in 10 groups.

Survey crafting: Finally, we craft the questionnaires from the strata. We decide to provide all 10 configurations for each original method as we want to compare

Table 5.: The strata properties (name, manually assigned type of methods and the method count) and the amount of methods sampled for the survey (in percent and the total count).

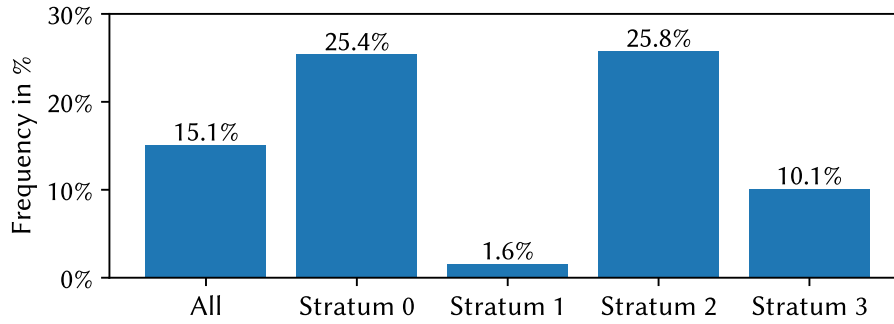| Properties			Sampling	
Name	Type of Methods	Count	Percentage	Count
Stratum 0	Simple methods	19016	10 %	4
Stratum 1	Complex methods	4280	40 %	16
Stratum 2	Magic number methods	78	10 %	4
Stratum 3	Medium complex methods	15938	40 %	16
Total		**39312**	**100 %**	**40**

Figure 4.: Frequency of the case that a REDEC variant is not different from its original method.

the original methods with their REDEC variants. We have a survey capacity of 400 code snippets (Section 4.1). Therefore, the capacity for each REDEC variant is $400/10 = 40$ code snippets. We start by selecting 40 original code snippets and then add all their REDEC variants. We opt for a random sample within the strata. However, we distribute the 40 snippets across the strata as shown in Table 5: We sample 4 methods each from Stratum 0 and Stratum 2 and 16 methods each from Stratum 1 and Stratum 3. The reason for choosing this approach is the relatively high frequency of methods that do not differ from their original methods in Stratum 0 and Stratum 2 (Figure 4). Additionally, 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 select all $9 * 40$ (#configurations $*$ #variant-capacity) REDEC variants that belong to the original methods next. We automatically do this based on the names of the original methods and the names of the REDEC variant methods. If REDEC renamed the method at an earlier stage due to the method renaming modificatation, the new method does no longer match the original method, in which case we match them manually.

Once we collected all 400 methods, we distribute them across the 20 question-naires, each with 20 methods. To not manipulate the raters, we decide that a variant of each method must only appear once in each questionnaire. For example, if the original method is in one questionnaire, the `removeComment` variant (or another variant of the same method) must not be included in the same questionnaire.

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

Finally, the methods of each questionnaire are randomly shuffled within itself. We do this to minimize the impact of the position of a snippet or variant within a survey on the rating.

3.6. READABILITY CLASSIFICATION MODEL

In this section we describe our approach for investigating whether it is possible to score a higher accuracy as the towards model of Mi et al. [25] in classifying code readability with the mined-and-modified dataset.

We created our own implementation of the model using Keras[16]. The model consists of three layers. A code representation layer, a feature extraction layer and a code readability classification layer. You can find an overview of the model architecture in Figure 5.

The input for the model is a labeled dataset consisting of code snippets and their readability classes (poorly or well readable). In the code representation layer,

---

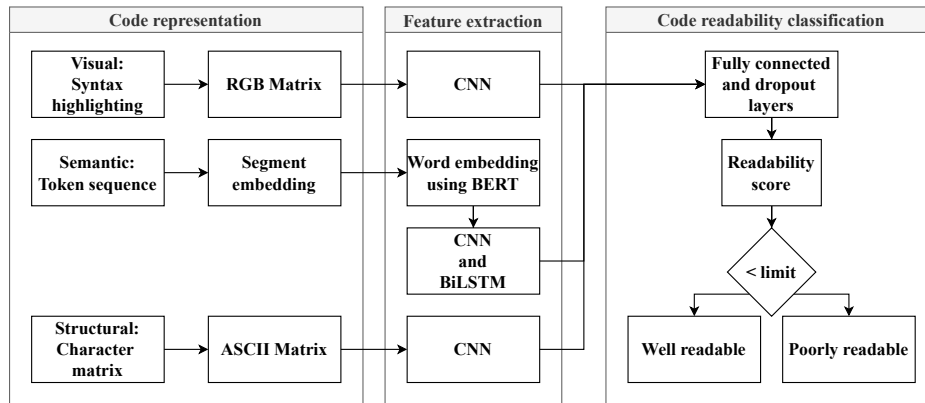[16] `https://keras.io/about/`, accessed: 2024-02-20

Figure 5.: The architecture of the towards model of Mi et al. [25].

Table 6.: The color encoding used by the visual component of the towards model [25].

Element	Color	Hex Code
Comment		#006200
Keyword		#fa0200
Identifier		#01ffff
Literal		#01ffff
Punctuation		#fefa01
Operator		#fefa01
Generics		#fefa01
Whitespace		#ffffff

three different code representations are generated from each code snippet: A visual, a semantic and a structural representation.

For the visual representation the syntax of the code is highlighted. Therefore Mi et al. assigned each type of syntactic element a color (Table 6). Instead of highlighting the words in the respective color, as done by an IDE, the words are replaced by color blocks instead (Figure 6b). Mi et al. used Eclipse[17] to highlight the code snippets and then took screenshots to obtain a RGB Matrix [25].

For the semantic representation we split the code into tokens (e.g., keywords and operators) and use BERT [11] to embed each token as a vector [25].

---

[17]https://www.eclipse.org/, accessed: 2024-03-02

```
1    // A method for counting
2    public void getNumber(){
3            int count = 0;
4            while(count < 10){
5                    count++;
6            }
7    }
```

(a) An exemplary Java code snippet.

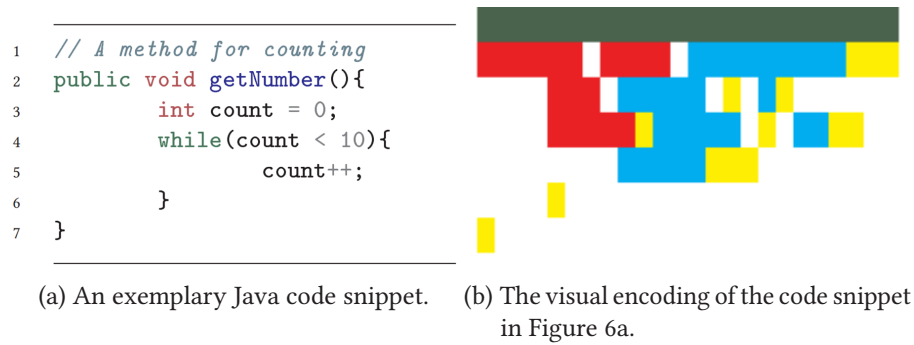(b) The visual encoding of the code snippet in Figure 6a.

Figure 6.: A code snippet and its visual encoding.

For the structural representation we split the code into characters and convert each into its corresponding ASCII value to obtain a ASCII matrix [25].

The model itself takes the three representations as input. We perform feature extraction on the RGB matrix and the ASCII matrix using a CNN for each. Each of the CNNs consists of multiple convolution and max pooling layers and a single flatten layer [25].

On the token embedding the model performs feature extraction using a BERT embedding layer, convolution layers, a max pooling layer and a BiLSTM [25].

After extracting the features from the three individual representations the output is merged and used as an input for the final step: code readability classification. In this step, the model consists of multiple fully-connected layers and a dropout layer. The output is a single value, namely the readability score. If the score is above a certain threshold, we classify the input as well readable, otherwise it is poorly readable [25].

We implemented this model as described by Mi et al. [25] with a few adjustments: In contrast to the publicly available code of Mi et al.[18], our model 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-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 form of a JSON file.

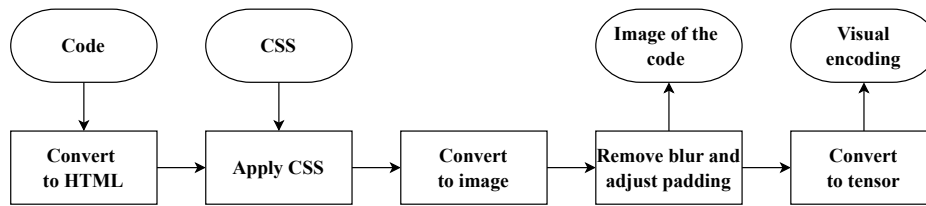[18]https://github.com/swy0601/Readability-Features, accessed: 2024-02-20

Figure 7.: The steps to automatically, visually encode code.

We made a further adjustment to the image encoding. To automate the generation of visual encodings we propose a different approach leading to a similar result. You can find an overview of our approach in Figure 7.

In a first step we use Imgkit[19] to convert the code to HTML. Thereby, a HTML class is assigned to each type of syntactic element. Next, we apply syntax highlighting using a CSS style sheet (Appendix IV). In a third step we use pygments[20] to convert the HTML with the applied CSS to an image. We use pillow[21] to remove blur and adjust the padding of the image. Finally, we load the image using opencv-python[22] which allows us to convert the image to an RGB tensor that is suitable as a model input. The advantages of our approach are that it is fully automated and that the used colors can be adjusted easily via the CSS style sheet (Appendix IV).

During implementation, we encountered the following potential problem with the model: The token length for the BERT encoding (BERT-base-cased[23]) used in the model is 100. What implications does this have? To answer this question we take a look at what a token comprises first. In addition to special tokens that mark the beginning *[CLS]* and the end *[SEP]* of the input, each word represents a token. Furthermore, each special character is also represented by its own token. Special characters are slashes (/), parentheses ((,), {, }), commas (,), semicolons (;), arithmetic signs (=, <, >) and many more. Java identifiers are split into several tokens according to the convention of upper and lower case. If an identifier is not present in the model's vocabulary, the tokenizer splits it further into sub-identifiers or characters that are in the vocabulary. For example in Listing 4, Line 6, the word 'int' is split into the tokens 'in' and 't' as 'int' is not part of the vocabulary of BERT-base-cased.

---

[19]https://pypi.org/project/imgkit/, accessed: 2024-03-02
[20]https://pygments.org/, accessed: 2024-03-02
[21]https://pypi.org/project/pillow/, accessed: 2024-03-02
[22]https://pypi.org/project/opencv-python/, accessed: 2024-03-02
[23]https://huggingface.co/google-bert/bert-base-cased, accessed: 2024-02-20

```
1   /**
2   * This method determines the sign of a given number and prints a
    ↪   corresponding message.
3   *
4   * @param number The input number to be checked.
5   */
6   public static void checkNumberSign(int number) {
7           if (number > 0) {
8                   System.out.println("Number is positive");
9           } else if (number < 0) {
10                  System.out.println("Number is negative");
11          } else {
12                  System.out.println("Number is zero");
13          }
14  }
```

(a) An example of a simple and well readable Java method.

```
1   [CLS] / * *
2   * This method determines the sign of a given number and prints a
    ↪   corresponding message .
3   *
4   * @ para m number The input number to be checked .
5   * /
6   public static void print S ign ( in t number ) {
7   if ( number > 0 ) {
8   System . out . print ln ( " Number is positive " ) ;
9   } else if ( number < 0 ) {
10  System . out . print ln ( " Number is negative " ) ;
11  } else {
12  System . out . print [SEP]
```

(b) The encoded-and-decoded variant of Listing 4a using BERT-base-cased with a limit
of 100 tokens. Space characters separate the tokens. Newlines are preserved for
readability.

Listing 4.: A Java method and its encoded-and-decoded variant.

Consider the method from Listing 4a. With a token limit of 100, the last encoded token is the last *print* in Line 12. Everything that comes after this is not encoded, which means that the information is lost for the semantic part of the model. Summarized, 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 smaller 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 encoders seem to be long enough to fully capture most code snippets, the semantic encoder seems to be too limited to do so in many cases.

Although we want to point out these limitations, we retain them in order to enable a fair comparison of the datasets.

Our code is publicly available on GitHub[24].

4. EVALUATION

We tested the mined-and-modified dataset in two ways. On the one hand we conducted a user study. On the other hand we evaluated the impact of using the dataset for the towards model of Mi et al. [25]. In detail we answer the following questions with both experiments:

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

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

Our assumptions are as follows:

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

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

Therefore, we come up with the following research questions:

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

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

---

[24]`https://github.com/LuKrO2011/master-thesis`, accessed: 2024-03-21

Research Question 2: *(modified-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 try to generate such code from the well readable code. We investigate whether this is possible in principle, and whether REDEC (Section 3.4) is able to achieve this.

The modifications REDEC applies on the source code are heuristics. To answer whether the generated code is actually poorly readable (Assumption 2) we utilize the results of the user study.

Research Question 3: *(new-data)* *To what extent can the new data improve existing code readability classification models?*

Previous research shows that Deep Learning models get better the more training data is available [15]. 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. [25]. Therefore we train their proposed model with combinations of the merged and the mined-and-modified dataset and compare the results.

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 (RQ 1 and RQ 2) and to craft our dataset.

PILOT SURVEY

1. Experimental setup: We manually sampled 20 code snippets across all strata but mainly from Stratum 1 and Stratum 3 due to reasons mentioned in Section 3.5. From January 6 to 14 in 2024, ten people took part in the survey. Eight of them were students and two of them worked in the industry. All of them had knowledge of computer science. They were not paid for participating in the survey. Additionally to rating 20 code snippets the participants were also asked to answer further questions to provide feedback about the survey:

1. *Short answer*: How long did it take you to complete the survey?

2. *Single choice (1 (very unclear) to 5 (very clear)):* How clear was your task?

3. *Long answer:* What problems were with the task? If there were none, leave blank.

4. *Long answer:* What problems were there with the survey tool? If there were none, leave blank.

5. *Long answer:* What improvements would you make to the survey? If none, leave blank.

6. *Long answer:* Do you have any other feedback? If none, leave blank.

2. Threats: The results do not generalize. We did not sample the Java snippets for rating in a specific or automated way, so there is a selection bias. The participants came from a private environment. This leads to a possible selection bias. We adjusted the survey instructions, explanations and questions afterwards, which might influence the ratings of the participants. However, we did not use the results from this survey to evaluate our dataset or the generation approach as the intention of the pilot survey was rather to prepare for the main survey.

3. Results: We analyze the results of the pilot survey with regard to three aspects: the time it took to complete the survey, the feedback from the participants and the ratings of the selected code snippets.

Completion Time: You can find the time it took the participants to complete the pilot survey and thus to rate 20 code snippets according to their readability in Figure 8. The fastest participant completed the survey in 7 minutes and 35 seconds, while the slowest participant took 18 minutes. Both, the average value and the mean value, are around 12 minutes. The boxplot (Figure 8) shows that the times are close together and there are no outliers in the time taken. We suspect that the participants in the pilot survey put more effort into completing the survey as we know them personally. Other participants may not make as much effort, so we set the time estimation for a questionnaire below average at 10 minutes.

Participant Feedback: All feedback of the participants regarding the pilot survey is listed in Appendix I. 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.'). Some of the feedback was regarding fully qualified class names, such as 'Java.io.InputStream'. We noticed that the Pretty Printer of the REDEC tool specified each imported method or class with its fully specified classifier. For example, instead of 'InputStream', 'Java.io.InputStream' was written in the REDEC code snippets. This gave the participants the feeling that the code was not written by a human and drastically reduced readability. Therefore we adapted the REDEC tool to print the shorter name.
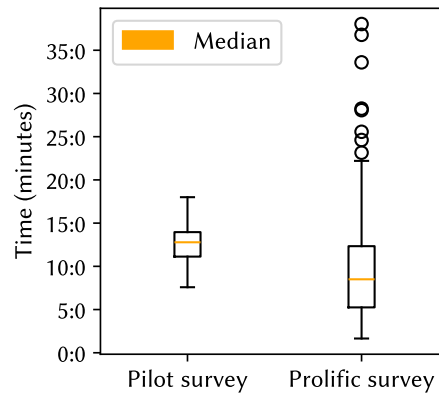
Figure 8.: Time required to complete a questionnaire.

Ratings: We adjusted the REDEC configurations. The rating for the last places (Table 7), such as *1.2* for *Stratum 1 - all*, suggest that these code snippets were particularly poorly readable. Thus, we re-examined the REDEC configurations and found that some of them are over-configured. This not only affects their readability, but also makes them look as if they were not written by human hands. Therefore, we reduced the probabilities for these configurations.

After adjusting the REDEC configurations and the survey tool according to the feedback, we launched the Prolific survey.

PROLIFIC SURVEY

In this section we summarize the results of the main survey conducted via Prolific[25].

1. Experimental setup: We conducted the survey using Tien Duc Nguyen's Code Annotation Tool (Figure 9) along with the platform Prolific[26] for the recruitment and payment of participants. The survey was conducted between January 31 and February 7 in 2024. A total of 221 participants took part. 11 participants answered each of the 20 questionnaires (similar to the survey of Scalabrino et al. [36]). In one survey, we assigned one more participant by mistake. We estimated the time to complete one questionnaire at 10 minutes (Section 4.1). Prolific set the maximum time allowed at 44 minutes. Participants who took longer received a time-out. This results in a margin of error of 29.55% at a confidence of 95% for an individual snippet. However, we aggregate over strata

---

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

Table 7.: Mean score ratings for the pilot survey.

Stratum	REDEC configuration	Score
Stratum 3	original	4.6
Stratum 0	tabs-few	4.3
Stratum 2	tabs-few	3.8
Stratum 1	original	3.7
Stratum 2	original	3.7
Stratum 3	newlines-many	3.3
Stratum 1	comments-remove	3.1
Stratum 0	spaces-few	3.0
Stratum 1	all4	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	all2	1.3
Stratum 1	all	1.2

and multiple snippets later to reduce the margin of error. Each questionnaire consists of 20 code snippets. Consequently, 400 different code snippets are rated in total. The questionnaires were configured in a way that each participant could only take part in one of the questionnaires. You can find the texts for the survey in Appendix III. We crafted the questionnaires as described in Section 3.5.

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.

The internal research questions are as follows:

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

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

Readability of Java Code

Rate the readability of Java methods on a scale from 1 (very unreadable) to 5 (very readable) using the stars below the code box. To navigate between methods, use the arrows above or below the code box. Make sure to rate each snippet.

Snippets

1   2   3   4   5   ...   20

←   →

Highlighter
atomOneDark   ▾

```
1  /**
2   * Constructor a new PartPath and increment the partCounter.
3   */
4  private Path assembleNewPartPath() {
5      long currentPartCounter = partCounter++;
6
7
8      return new Path(bucketPath, ((((outputFileConfig.getPartPrefix() + '-')
9  }
```

★★★★★

←   →

Figure 9.: Tien Duc Nguyen's tool for rating a code snippet from the perspective of a survey participant.

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 (original) is on average not better readable than the code from previous studies.

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

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

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.

Sampling Method: We have a larger proportion of code snippets from Stratum 1 and Stratum 3 (Section 3.5). While we argue that we avoided spending resources on labeling data that is likely not different from the original methods or rather uninteresting, this might also introduce statistical errors to our survey results. However, stratified sampling is well-defined and proven in practice. The approach ensures that our sample represents all parts of the population under investigation.

Insufficient Responses: 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.

Piece-work Effect: Survey participants are paid for taking part and completing a questionnaire. However, they receive the same amount of money regardless of their speed. Therefore, they receive more pay per minute if they hurry. This could have an impact on the accuracy with which they scored the code snippets. A comparison between the time required by a participant for a pilot questionnaire and a Prolific questionnaire (Figure 8) supports this suggestion. Especially the ratings of participants requiring less than 3 minutes (Figure 10b) to complete a questionnaire could have a negative impact on validity.

3. Results: We analyze the results of the Prolific survey with regard to three aspects: the time it took to complete the survey, the participants' familiarity with Java and the ratings with regard to the REDEC configurations.

Completion time: You can find an overview of the time that the participants required in Figure 8 and Figure 10a. The fastest participant completed the survey in 1 minute and 39 seconds, while the slowest participant needed about 38 minutes. The average time is 9 minutes and 45 seconds. The median time is 8 minutes and 30 seconds. The boxplot (Figure 8) shows that the completion times are not as close together as they are in the pilot survey. There are a couple of outliers.

We pay particular attention to the number of participants who took less than 3 minutes to complete a questionnaire (Figure 10b), as we assume that this is hardly possible without randomly selecting answers. In almost all questionnaires, the number of participants who took less than 3 minutes is either 0 or 1, while there is only one questionnaire where two participants took less than 3 minutes. Overall,
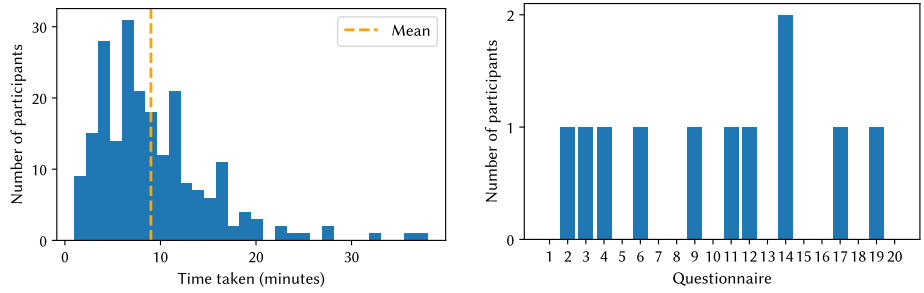
(a) Time required by participants to complete the survey.

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

Figure 10.: Time analysis of participants completing the Prolific survey.
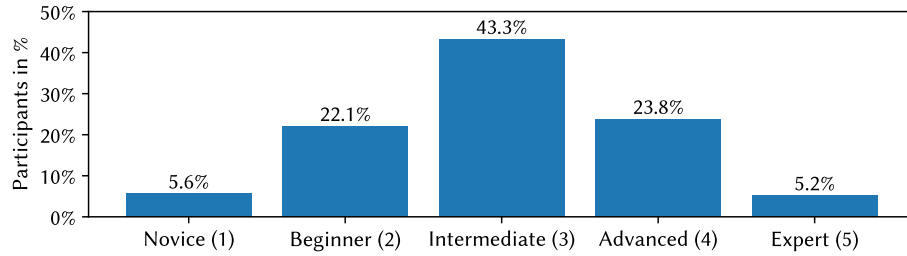

Figure 11.: Familiarity of Prolific survey participants with Java.

there are only a few participants who took less than 3 minutes. Therefore, we assume that most participants completed the survey with reasonable effort.

Familiarity with Java: Figure 11 shows the participants' familiarity with Java. According to their own estimation, most participants (43.3 %) have intermediate Java knowledge. 29.0 % of the participants are either advanced or experts with Java. This high level of familiarity with Java suggests that the quality of the evaluations received is high.

Ratings: You can find the ratings for each REDEC configuration for all strata in Figure 12. We added the ratings of the merged dataset for comparison. Figure 12a visualizes the distribution of ratings for each score from 1 to 5. It also shows the mean value and mode of the ratings for each variant. To facilitate comparison with the original methods, the mean value of the ratings of the original methods across all variants is marked with a red dashed line. The original method and

the just-pretty-print method are the most frequently rated with a score of 4 and their mean values of 3.69 and 3.67 are comparatively the highest.

In line with our expectation, we see that the ratings for `just-pretty-print` and `original` only differ by a very small amount of 0.02. A Mann-Whitney U test shows that the probability, that the difference between `just-pretty-print` and `original` is due to random variation, is 92 %. Therefore, we are sure that not the pretty-printer but the modifications caused the differences in readability.

The violins of Figure 12a illustrate the distribution of the ratings in detail. Note that the evaluators were only able to rate the discrete values 1 to 5. The deflections of the violins in between are for better visualization. The figure shows that for `original` and `just-pretty-print` most of the ratings are 3, 4 or 5. In addition, the violins of `merged` and `all7` look similar, with the latter having more 2 ratings. The violins of `newlines-many`, `rename` and `spaces-many` show more better (3, 4 and 5) and less worse (1 and 2) ratings than the ones of `comments-remove`, `newlines-instead-of-space`, `newlines-few` and `tabs`.

Figure 12a also shows that the mean value of the original methods is 3.69 and the mean value of the merged dataset is 3.34. The difference of 0.35 has statistical significance: We perform a Mann-Whitney U test for the rating values of the 40 original code snippets of the mined-and-modified dataset against all rating values of the merged dataset. The resulting p value is 1.11×10^{-9}, which is far below the $5\ \% = 5.00 \times 10^{-2}$ threshold and confirms statistical significance.

Summary (RQ 1 - mined-well)

The mean score for the `original` methods of the mined-and-modified dataset (3.69) is significantly larger than the mean score for all ratings in the merged dataset (3.34). Therefore we reject the null hypothesis and conclude that the well readable assumption (Assumption 1) holds.

Figure 12b shows the differences in mean values more precise. REDEC reduces the readability for the configurations `all7`, `comments-remove`, `newlines-instead-of-space`, `newlines-few` and `tabs` the most. The tool also reduces readability for `newlines-many`, `rename` and `spaces-many`, but not as much.

To find out whether the deviations are statistically significant we utilize the Mann-Whitney U test once more. We compare the ratings for all snippets for a REDEC configuration with the corresponding `just-pretty-print` snippets. Table 8 shows the results. We can be sure that the scores of all methods except `newlines-many` and `rename` are indeed statistically different from the scores of `just-pretty-print`.
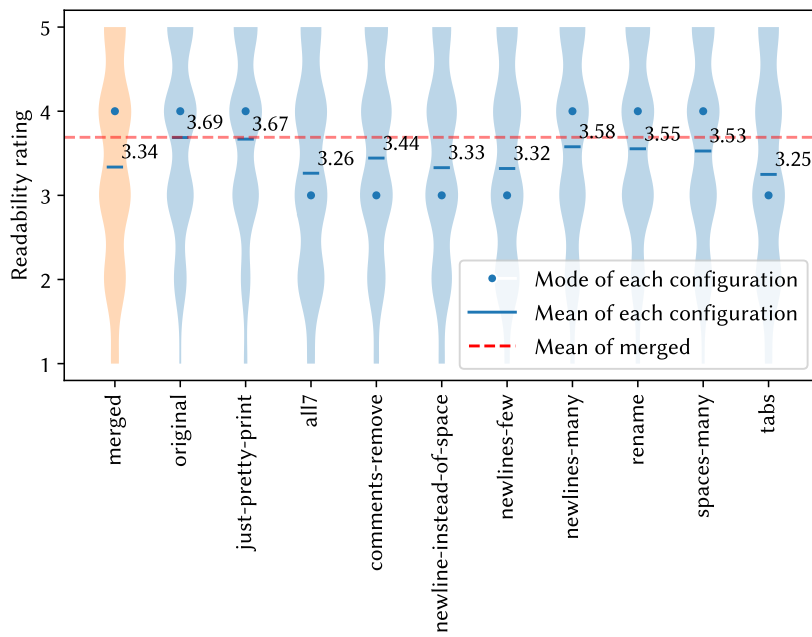
(a) Survey ratings for each REDEC configuration and all strata.
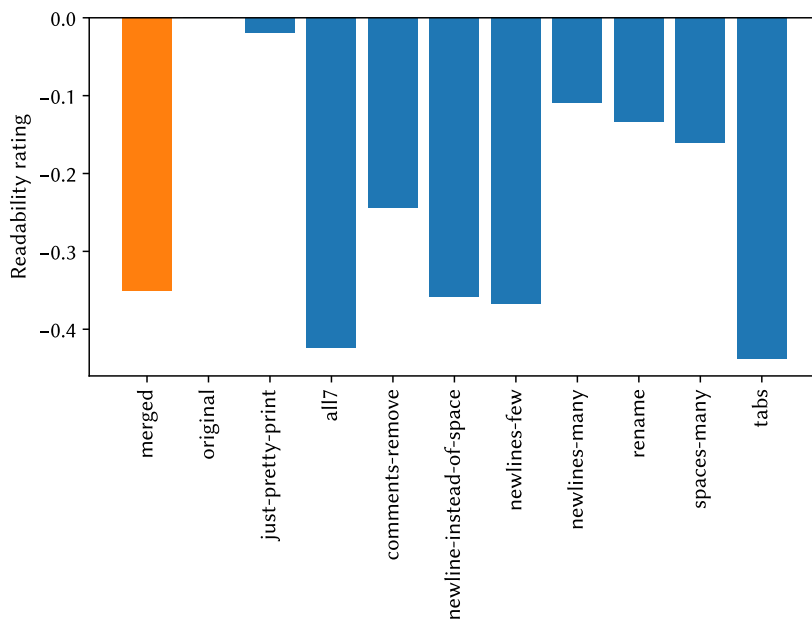
(b) Relative survey ratings for each REDEC configuration and all strata compared to all original methods.

Figure 12.: Survey ratings for each REDEC configuration and all strata. Additionally, merged is added.

Table 8.: Mann-Whitney U test results of each REDEC configuration against `just-pretty-print`. When p is smaller than $5\% = 5.00 \times 10^{-2}$ (**bold**) we conclude that the difference is significant.

Comparison against	p
methods	9.22×10^{-1}
newlines-few	5.23×10^{-6}
spaces-many	4.07×10^{-2}
newlines-many	3.00×10^{-1}
comments-remove	3.64×10^{-3}
rename	9.90×10^{-2}
newline-instead-of-space	4.57×10^{-6}
tabs	3.06×10^{-8}
all7	1.80×10^{-7}

If we consider binary readability classification and split the data into two classes (poorly readable: 1,2; well readable: 3-5) all but `rename` are statistically different from `just-pretty-print`. This includes `newlines-many` ($p = 0.035 = 3.5\% < 5\%$) for which we could not confirm statistical difference without binary classification.

Besides `just-pretty-print`, this leaves only `rename` where we can not confirm statistical significance. Overall, we showed that REDEC actually reduces the readability of source code.

> **Summary (RQ 2 - modified-poor)**
>
> All of the 7 configurations but `rename` decrease readability by a significant extend compared to `just-pretty-print`. We estimate the readability decrease for a certain probability of a certain type as can be seen in Figure 12b. We reject the null hypothesis and conclude that the poorly readable assumption (Assumption 2) holds.

4.2. MODEL TRAINING RESULTS

We use the notation *(train-evaluate)* to describe on which dataset the towards model of Mi et al. [25] is trained and evaluated. *mam* stands for our new mined-and-modified dataset. We use 10-fold cross-validation for evaluation. We aim to investigate the following things:

Table 9.: Performance of different dataset configurations for the same model. *mam* stands for the mined-and-modified dataset. *finetune* is training on the mined-and-modified dataset and fine tuning on the merged one.

Train	Eval	Acc	Prec	Rec	AUC	F1	MCC
merged	merged	84.7 %	87.7 %	82.3 %	85.0 %	83.7 %	70.4 %
mam	mam	92.2 %	92.3 %	92.0 %	92.2 %	92.2 %	84.4 %
mam	merged	61.9 %	66.7 %	54.5 %	60.6 %	60.0 %	24.8 %
merged	mam	56.8 %	54.7 %	78.7 %	66.7 %	64.6 %	15.2 %
finetune	merged	83.3 %	84.3 %	79.1 %	81.7 %	81.1 %	63.7 %

1. **Model evaluation:** To confirm that our implementation of the model scores similar accuracy as the original one of Mi et al. [25], we train and evaluate the model on the merged dataset (merged-merged).

2. **Internal evaluation:** To investigate how effective the model captures the readability aspects of the mined-and-modified dataset, we train and evaluate it on this dataset (mam-mam). By doing so, we also examine how effectively the model captures the differences between the `original` and `all7` methods which are modified with REDEC.

3. **Cross evaluation:** To assess how effective the mined-and-modified dataset is for predicting readability, we train the model on the mined-and-modified dataset and then evaluate its performance on the merged dataset (mam-merged). For advanced insights, we also train the model on the merged dataset and evaluate it on the merged dataset (merged-merged) and on the mined-and-modified dataset (merged-mam).

4. **Fine-tuning:** To assess what accuracy we can score in predicting readability we investigate training on the mined-and-modified dataset and fine-tuning and evaluating on the merged dataset (finetune-merged).

You can find the results in Table 9.

Model evaluation: When we train and evaluate the model on the merged dataset, we obtain an accuracy of 84.7 %. This is very similar to the results of Mi et al. (84.7 % vs 85.3 %). The deviation of 0.6 % accuracy might be due to randomness of the splits for 10-fold cross-validation. We can confirm the results of the paper [25].

Internal evaluation: When we train and evaluate the model on the mined-and-modified dataset, we obtain an average accuracy of 92.2 %. The towards model

architecture is well suited to learn the structure of the mined-and-modified dataset. In particular it learns the differences between `original` and `all7` methods and thus learns how to predict whether REDEC modified a code snippet.

Cross evaluation: When we train the model on the mined-and-modified dataset and evaluate it on the merged dataset (mam-merged), we get an accuracy of 61.9 %. This is 22.8 % worse than the accuracy we get when we train and evaluate the model on the merged dataset (merged-merged). When we train the model on the merged dataset and evaluate it on the mined-and-modified one (merged-mam), we get an accuracy of 56.8 %, which is close to the approximate accuracy of 50.0 % of a random classifier. If the scores for mam-merged, merged-merged and merged-mam would be similar, we would conclude that both datasets, the merged and the mined-and-modified one, address readability in general. Given that this is not the case and knowing for both datasets that they at least address some aspects of readability we conclude that we address different aspects of readability.

Fine-tuning: We tried to fine-tune the merged dataset by freezing different layers of the model trained with the mined-and-modified dataset. During evaluation we achieved the best results when freezing the input layers as well as the first convolution and pooling layer of all encoders. However, when evaluated on the merged dataset, the performance is still worse than the merged-merged variant. 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 lead to an improvement. However, this is not part of this work, in which we mainly focus on a new dataset.

> **Summary (RQ3 - new-data)**
>
> When trained with the mined-and-modified dataset and evaluated on the merged dataset, the model achieves an accuracy of 61.9 %. For comparison: When trained and evaluated using the merged dataset, the model achieves an accuracy of 84.7 %. We conclude that the mined-and-modified dataset does not improve code readability classification using the towards model.

5. DISCUSSION

Our survey (Section 4.1) shows that the mined-and-modified dataset captures readability. The model training results (Section 4.2) show that the mined-and-modified dataset captures different aspects of readability compared to the merged dataset. The question arises what the different aspects are and whether it is

possible to extend REDEC (Section 3.4) so that the same aspects are captured. By achieving this, we suggest that we could also achieve better evaluation results on the merged dataset than previous models.

We fine-tune the model with the merged dataset after training with the mined-and-modified dataset and evaluate it on the merged dataset (Section 4.2). We expect the classification accuracy of the resulting model to exceed that of the model trained on the merged dataset only. Our expectations are not met. This could be because the towards model structure is designed for a much smaller dataset and therefore cannot capture all the features of the mined-and-modified dataset while allowing for fine-tuning.

When merging existing datasets [8, 13, 36] into a single dataset (Section 3.1), we set readability score of a code snippet as the mean value of all its ratings. As only a limited number of people participated in each survey, this may introduce errors due to statistical deviations. Furthermore, the surveys were conducted under different conditions, e.g. different raters, different number of raters per snippet, different rater biases, different code scopes. When merging the datasets, we do not take these inequalities into account. This could lead to a bias in the merged dataset. However, previous approaches did this similarly.

The main advantage of our approach is the automation of data generation. This comes with a drawback: The score labels of the methods of the mined-and-modified dataset are estimations and not exact values (Section 3.3). However, accurate ratings would require human annotators to manually review 69k code snippets which is not feasible. In addition, with two-class classification, we only need to distinguish between well and poorly readable code, which we can do even without exact labels.

The probabilities for `removeComments` for the survey and the model training dataset differ (Section 3.3 and Section 3.4). When conducting the survey, we argued that realistic methods do not require comments. Therefore, we set the probability for `removeComments` to 100 %. However, when applying the model training, this led to shortcut learning of whether a method has a comment or not, instead of learning to distinguish based on all applied modifications. Therefore, we adopted REDEC so that `removeComments` is not applied to the Java files. Instead, we applied it separately after the method extraction with a probability of 10 %. Thus, the configurations `commentsRemove` and `all7` of the survey results (Section 4.1) do not exactly match the configurations of the model training (Section 4.2). However, we hardly rely on the survey results to train the model, and therefore we argue that this is a rather minor thread.

When comparing the model performance trained on the mined-and-modified and merged dataset, it should be noted that the merged dataset is small. Consequently, comparisons to classifiers trained on the merged dataset may be unreliable [25].

6. CONCLUSIONS

Recent research in the field of code readability classification 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 69k code snippets (Section 3). Although our results show that the mined-and-modified dataset does not capture the same aspects of readability as the merged one (Section 4), it still captures readability and could accordingly help to improve the classification of code in future research.

The new approach for generating data 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 mined-and-modified 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 a complex task, as one has to adapt the Readability Decreaser to work with another language. Furthermore, a general tool that works for all languages is difficult if not impossible.

As Mi et al. suggested, another useful representation for code readability studies could be the syntax tree representation of code [20]. One could try to improve the performance of the towards model [25] by adding another representation encoding extractor for Java code that automatically extracts the abstract syntax tree of the code.

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 consist of finding and evaluating other encodings that represent the code in a different way.

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

The modifications described in this work (Section 3.4) are only part of the possible modifications that can be developed. Additional modifications 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 of code readability. Our new dataset and the generation approach serve as a foundation for this.

Eigenständigkeitserklärung

Hiermit versichere ich, Lukas Krodinger,

1. dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe.

2. Außerdem erkläre ich, dass ich der Universität ein einfaches Nutzungsrecht zum Zwecke der Überprüfung mittels einer Plagiatssoftware in anonymisierter Form einräume.

Passau, 21. März 2024

Lukas Krodinger

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, Henry Jackson-Flux and Marc Brockschmidt. 'Self-supervised bug detection and repair'. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 27865–27876.

[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] 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.

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

[11] Jacob Devlin et al. 'Bert: Pre-training of deep bidirectional transformers for language understanding'. In: *arXiv preprint arXiv:1810.04805* (2018).

[12] Brendan Dolan-Gavitt et al. 'Lava: Large-scale automated vulnerability addition'. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 110–121.

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

[14] 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.

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] F.P. Miller, A.F. Vandome and M.B. John. *Abstract Syntax Tree*. VDM Publishing, 2010. ISBN: 978-613-3-77312-7. URL: https://books.google.de/books?id=5ns7YgEACAAJ.

[28] Daniel Müllner. 'fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python'. In: *Journal of Statistical Software* 53 (2013), pp. 1–18.

[29] 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.

[30] 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.

[31] Jannik Pewny and Thorsten Holz. 'EvilCoder: automated bug insertion'. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 2016, pp. 214–225.

[32] 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.

[33] Michael Pradel and Koushik Sen. 'Deepbugs: A learning approach to name-based bug detection'. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–25.

[34] 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.

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

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

[37] 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.

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

[39] 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.

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

[41] Steven K Thompson. *Sampling*. Vol. 755. John Wiley & Sons, 2012.

[42] 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.

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

[44] 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.

I. PILOT SURVEY FEEDBACK

How clear was your task? (1 = Very Unclear, 5 = Very Clear)
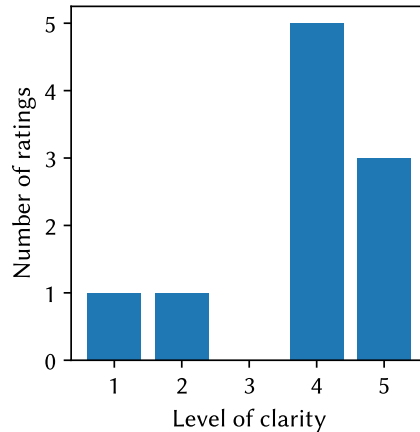
What problems were with the task? If there were none, leave blank.

- Did at first not know where to rate the code.

- I was confused about the textfield for the comments because I only re-membered that we should rate the code snippets, not that we have to make comments. Since I was not able to navigate back to the task description, I did not know what to do with them.

- Für einen Anfänger mit sehr wenig Java Erfahrung ist meiner Meinung nach der Code zu kompliziert.

- In the first place, I didn't really understand what readability meant. But after slide 3 or 4, I understood what this was about.

- I found it difficult to categorize the first examples because you don't know what's still to come. For example, what the least readable code is.

What problems were there with the survey tool? If there were none, leave blank.

- Mobile is not easy to use because of the scrolling needed to complete the survey.

- First, I needed to figure out how this tool works and that the rating is done with the stars below. I thought I should write my rating as a comment in the comment field below. After number 20, I didn't know whether I could close the survey or not.

- I also thought that I should use the drop-down menu on the upper left.

- It is sometimes necessary to swipe horizontally to see all of the code, which is a bit inconvenient.

- Für einen Anfänger ist das Tool meiner Meinung nach nicht geeignet. Der Code ist zu verschachtelt und teilweise unverständlich.

- After finishing the task, at least a message should be shown.

- I didn't understand what the button at the top left meant, where you could select the programming language. There were too many fonts to choose. I also wasn't sure whether to write a comment or not. It wasn't described at the beginning.

What improvements would you make to the survey? If none, leave blank.

- Maybe one sentence that one should use the stars for the rating, then it would be clear. Also, the submit note after the last question could contain that one can close the survey now.

- I suggest making the task description accessible during the rating.

- Maybe the option to leave the survey when clicking to submit.

- Mehr Hilfestellung zum Lesen des Codes. Mehr Beschreibung oder ein zusätzliches Cheat Sheet mit Bedeutungen von Befehlen.

- I think it's a good idea to ask the participant at the beginning to explain what readability means for him.

- I would leave out the buttons described above. I was missing a scrollbar at the bottom of the code-window. A conclusion page with a message like "Thank you for your participation", "You're Done!" or other further information was missing, too.

Do you have any other feedback? If none, leave blank.

- There were drop downs for the programming language, but choosing another language did not change anything. It was a bit confusing that (almost?) all code snippets had very long imports within the code, which made them poorly readable.

- I spent the most time understanding methods with complete Java import names. (org.foo.bar.ClassName).

- GOOD LUCK

II. READABILITY DECREASING MODIFICATATIONS CONFIGURATION FILE

```
1   newline:
2   - 0.0   # Probability for no newline
3   - 1.0   # Probability for one newline
4   incTab:
5   - 0.0   # Probability for no tab
6   - 1.0   # Probability for one tab
7   decTab:
8   - 0.0   # Probability for no tab
9   - 1.0   # Probability for one tab
10  space:
11  - 0.0   # Probability for no space; Must be 0.0
12  - 1.0   # Probability for one space
13  newLineInsteadOfSpace: 0
14  spaceInsteadOfNewline: 0
15  incTabInsteadOfDecTab: 0
16  decTabInsteadOfIncTab: 0
17  renameVariable: 0
18  renameField: 0
19  renameMethod: 0
20  inlineMethod: 0
21  removeComment: 0
22  add0: 0
23  insertBraces: 0
24  starImport: 0
25  inlineField: 0
26  partiallyEvaluate: 0
```

III. PROLIFIC SURVEY TEXTS

On Prolific:

Readability of Java Code

We study the readability of Java source code. Therefore, please read Java methods and rate their readability on a scale from 1 (very unreadable) to 5 (very readable).

At the top of the tool:

Readability of Java Code

Read the Java methods and rate their readability on a scale from 1 (very unreadable) to 5 (very readable) using the stars below the code box. To navigate between methods, use the arrows above or below the code box. Make sure to rate each snippet.

Introduction page 1:

This study aims to investigate the readability of Java source code. In this survey, we will show you 20 Java methods. Please read the methods thoroughly and rate how readable you think they are. Before we begin, please answer the following question:

How would you describe your familiarity with Java?

1. Expert

2. Advanced

3. Intermediate

4. Beginner

5. Novice

Introduction Page 2:

Below is an example of the interface for displaying and rating the code. Use the stars below the code box for your rating. Please rate the readability on a scale from 1 (very unreadable) to 5 (very readable). At the top left, you can adjust the syntax highlighting and theme (dark/light) according to your preferences (optional). Comments are not available during this survey.

[EXAMPLE]

Introduction Page 3:

This survey should take about 10 minutes to complete. Now you are ready to go!

IV. TOWARDS MODEL - VISUAL ENCODING COLORS

The following CSS was used to generate the background colors for the visual encoding. You can find an overview over all tokens on the pygments homepage[27].

```css
 1  /* Comment Styles */
 2  .c, .ch, .cm, .cp, .cpf, .c1, .cs {
 3          background-color: #006200;
 4          color: #006200;
 5  }
 6  /* Keyword Styles */
 7  .k, .kc, .kd, .kn, .kp, .kr, .kt {
 8          background-color: #fa0200;
 9          color: #fa0200;
10  }
11  /* Parentheses, Semicolon, Braces Styles */
12  .p, .o, .ow {
13          background-color: #fefa01;
14          color: #fefa01;
15  }
16  /* Whitespace Styles */
17  .w {
18          background-color: #fff;
19          color: #fff;
20  }
21  /* Names/Identifiers Styles */
22  .n, .na, .nb, .nc, .no, .nd, .ni, .ne, .nf, .nl, .nn, .nt, .nv {
23          background-color: #01ffff;
24          color: #01ffff;
25  }
26  /* Literals Styles */
27  .m, .mb, .mf, .mh, .mi, .mo, .s, .sa, .sb, .sc, .dl, .sd, .s2, .se,
  ↪    .sh, .si, .sx, .sr, .s1, .ss, .b, .bp, .f, .fm, .v, .vc, .vg,
  ↪    .vi, .vm, .i, .il {
28          background-color: #01ffff;
29          color: #01ffff;
30  }
31  /* Error Styles */
32  .err {
33          background-color: #fff;
34          color: #fff;
35  }
36  /* Generics Styles */
37  .g, .gd, .ge, .ges, .gr, .gh, .gi, .go, .gp, .gs, .gu, .gt {
38          background-color: #fefa01;
39          color: #fefa01;
40  }
```

---

[27] https://pygments.org/docs/tokens/, accessed: 2024-03-02