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

Acknowledgements

I would like to express my gratitude to the following individuals whose support and contributions were invaluable throughout the completion of this thesis:

Professor Dr. Gordon Fraser, my supervisor, for his insightful feedback, academic support, and provision of resources from the chair. His expertise and guidance were invaluable in navigating the complexities of this research.

Elisabeth Griebl, my advisor, for her invaluable guidance, support, and encouragement throughout the entire process of this thesis. Her expertise, feedback, and supervision were instrumental in shaping this thesis.

Benedikt Fein, for introducing and supporting the use of podman with slurm. The provision of scripts and his technical expertise and support were crucial in overcoming the challenges encountered during this research.

Tien Duc Nguyen, for his development and support of the survey instrument. His contributions were essential for the data collection in this work.

Christina Praml, for her unwavering support, invaluable advice, and constant encouragement. Not only her personal support but also her professional comments and criticism were of extraordinary usefulness.

Maximilian Jungwirth, for his contributions to the main ideas of REDEC, provision of tools for applying Checkstyle to Java files, and his careful proofreading. His ideas, tools, and comments made this work possible in the first place.

Katrin Schmelz, Julia Krodinger and Severin Primbs for their careful proofreading and valuable feedback. Their attention to detail and constructive criticism contributed significantly to the clarity and coherence of this work.

To all those mentioned above and to countless others who have helped me along the way, thank you for your support and encouragement.

Sincerely,

Lukas Krodinger

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.

We investigate the use of a new dataset consisting of 69k code snippets 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

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, 11, 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, 14, 36]. In more recent years, deep learning-based models were able to achieve an accuracy of up to 88.0 % [23, 24, 39, 26, 20, 21].

A major limitation of these models is not their architecture, but the amount of data for Java code readability classification, which comprises 421 code snippets [8, 14, 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 [16]. 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 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

---

[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 1.1a but modified for poor readability.

Listing 1.1: Well readable (Listing 1.1a) vs. poorly readable (Listing 1.1b) code.

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, we modify all selected Java files so that the code is subsequently less readable. For an exemplary result, see Listing 1.1. 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.

How can we modify code so that it is less readable afterward? 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 1.1 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. [26] for this.

Our contributions are as follows:

- Although existing datasets [8, 14, 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 [26] (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 succeeded 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. [26] (Section 3.6).

- We show through a survey (??) that the well readable assumption (Assumption 1) and 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. [26] with and without the mined-and-modified dataset (Section 4.4).

Our automated approach for creating a readability classification dataset is effective. The mined-and-modified dataset, consisting of 69k samples, contains well readable (Assumption 1) and the poorly readable (Assumption 2) methods. We infer from the model training results (Section 4.4) that the mined-and-modified dataset captures different aspects of code readability as previ-

ous datasets [8, 14, 36]. Our code is publicly available on GitHub: `https://github.com/LuKrO2011/master-thesis`.

2

Background and Related Work

In the following sections, we describe the background and related work on code readability and our approach for dataset generation.

2.1 CODE READABILITY

When talking about *code readability classification* we need to clarify what this term means. 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' [30].

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

There are various related terms to readability: Understandability, usability, reusability, complexity, and maintainability [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, 19, 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 aspects, 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' [30].

Researchers proposed several metrics and conventional models for assessing code readability with an accuracy of up to 81.8 % [8, 32, 14, 36]. In recent years, deep learning-based models achieved an accuracy of up to 88.0 % [23, 24, 39,

Table 2.1: Accuracy scores of two-class readability classification models.

Model	Type	Accuracy
Buse and Weimer [8]	Conventional	76.5 %
Posnett et al. [32]	Conventional	71.7 %
Dorn [14]	Conventional	78.6 %
Scalabrino et al. [36]	Conventional	81.8 %
Mi_IncepCRM [23]	Deep Learning	84.2 %
Mi_DeepCRM [24]	Deep Learning	83.8 %
Sharma and Srivastava [39]	Deep Learning	84.8 %
Mi_Towards [26]	**Deep Learning**	**85.3 %**
Mi_Ranking [20]	Deep Learning	83.5 %
Mi_Graph [21]	Deep Learning	88.0 %

26, 20, 21] 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, 14, 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. 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 machine learning, especially deep learning, dominates code readability classification. As the quality of the models increased, so did their accuracy (Table 2.1).

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 [23].

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 [24].

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 2.1.

The limitation of deep learning-based code readability models was to focus on structural features. 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 Bidirectional LSTM, the proposed model processes RGB matrices, token sequences, and character matrices to capture various features of source code [26].

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 [20].

All accuracy scores in two-class classification (Table 2.1) 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 an Abstract Syntax Tree (AST), combining control and data flow edges, and converting node information into vectors. The model, comprising Graph Convolutional Network (GCN), DMoNPooling, and K-dimensional Graph Neural Networks (k-GNNs) layers, extracts syntactic and semantic features [21].

Until now researchers introduced many deep learning architectures and components to various classification models 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 a new dataset. For evaluation, we opted for the *Mi_Towards* model (hereinafter referred to as Towards model) from Mi et al. [26]. We did not choose the best performing one, *Mi_Graph* [21], 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 [14] 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, 14, 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 % [25]. Lately, researchers successfully enhanced data augmentation by incorporating domain-specific data transformation and GANs [22]. 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.

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 continue to use code snippets.

2.5 DIVERSE PERSPECTIVES

Other important research in the field of readability classification does not directly affect this work but could have implications for future works.

Fakhoury et al. showed that models do not capture what developers think of readability improvements. Therefore, they analyzed 548 GitHub commits

manually. They suggest considering other metrics, such as incoming method calls or method name fitting [15].

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 [30].

Ribeiro and Travassos demonstrated a consistent perception that Python code with more lines was deemed more comprehensible, irrespective of their level of experience. However, when it came to readability, variations were observed based on code size, with less experienced participants expressing a preference for longer code, while those with more experience favored shorter code. 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 [10].

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 [27].

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 that 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 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 use the fork and star counts as criteria for well readable code (Assumption 1).

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

In the field of vulnerability detection in source code, Dolan-Gavitt et al. introduced a technique for automatically inserting bugs into source code. The generated dataset was used for the evaluation of vulnerability detection tools [13].

Pradel and Sen applied a similar approach to general bug detection using a deep learning classifier. They generate training data by inserting bugs into existing code. This process involves simple program transformations that are likely to introduce incorrect code. They train a classifier to distinguish between correct and incorrect code [33]. Yasunaga and Liang did convert working programs into broken ones. They pre-trained models on the automatically generated dataset before fine-tuning them using a manually labeled one [44]. We also use the idea of pre-training and fine-tuning. Allamanis et al. has further improved this 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. used the idea and shifted the area to code formatting. This topic comes closest to our topic of code readability. They created a model that is able to fix checkstyle violations using deep learning. In the first step, they inserted formatting violations into the code based on a project-specific format-checking ruleset. They then used an LSTM neural network that learned how to undo these insertions.

All of the mentioned approaches have in common that they generate a dataset in an automated way by using the idea of deliberate denigration of source code. This approach has already been successfully used in the areas of vulnerability

detection, error detection, and code formatting. We are the first to apply it to the area of code readability classification.

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 is an abstraction that removes specific syntactic details, focusing on the relationships between nodes. ASTs are widely used in compilers, static analysis tools, and refactoring engines to perform tasks such as semantic analysis, optimization, and code transformation [28]. The latter is our use case.

We transform source code into the AST, make certain modifications to the AST (Table 3.2), 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 encounter situations in which the population can be divided into subgroups. Each group represents a 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 (??), we used 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 sections describe our approach.

3.1 WORK ON EXISTING DATASETS

Most of the related work (Chapter 2) uses a combination of the data of Buse and Weimer [8], Dorn [14] and Scalabrino et al. [36]. The raw data from their surveys can be downloaded[1], but their data is not uniformly formatted. In addition to Java, the dataset of Dorn also includes Python and Cuda[2] code snippets. All datasets consist 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.

We converted and combined these three datasets into one: *code-readability-merged* or for short *merged*. In recent years, Huggingface[3] established as the pioneer in making models and datasets available. Therefore, we decided to publish the merged dataset on Huggingface[4].

3.2 CLASSIFICATION CONSIDERATIONS

It is state of the art to perform a binary classification into well readable and poorly readable code [23, 24, 39, 26, 20].

---

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

[2] `https://developer.nvidia.com/cuda-toolkit`, accessed: 2024-03-23

[3] `https://huggingface.co/`, accessed: 2024-02-18

[4] `https://huggingface.co/datasets/se2p/code-readability-merged`, accessed: 2024-02-18

However, code readability classification is not a binary classification task per se. Mi et al. introduced a third, neutral class to address this problem [21]. When rating code snippets, previous studies used a Likert scale [17] from 1 (very unreadable) to 5 (very readable) [8, 14, 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.

The model used for evaluation is the Towards model of Mi et al. [26]. 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 the 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 [26].

While this transformation is legitimate, 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:

They reduce the available data 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 considerable loss.

They evaluated their model using those 210 snippets which corresponds to 50 % of the available data. We suspect that this might be a threat to validity. It could be that the performance of the model is remarkably lower when the evaluation is performed with random, unseen data that also contains moderately readable code snippets.

However, we continue to use the binary classification approach as well as evaluate the 210 code snippets to make our results comparable with the results of Mi et al. [26].

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 does not apply to the merged dataset, as 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
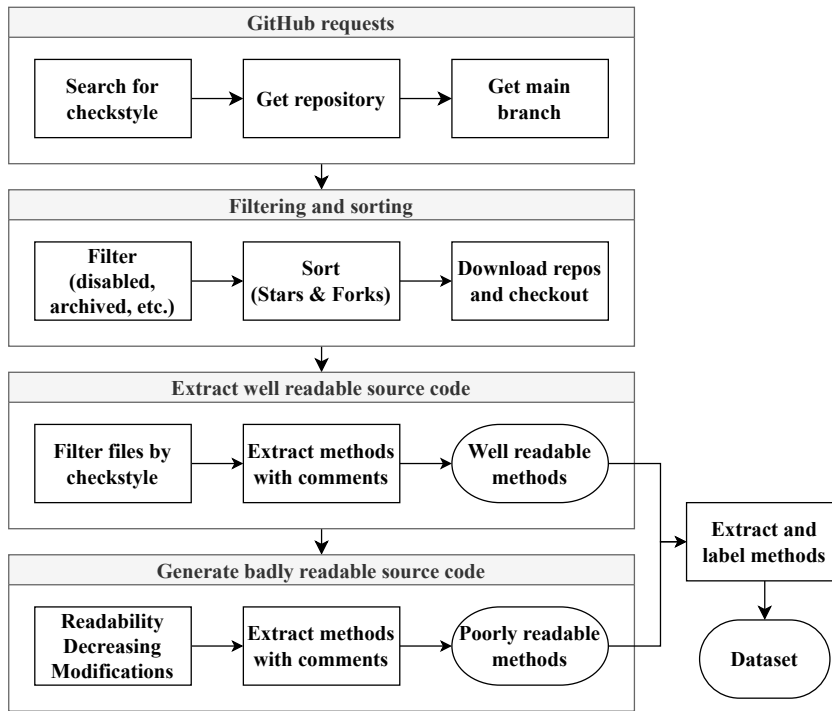
Figure 3.1: The used dataset generation approach.

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 3.1). We use the first three steps to mine well readable Java code. In the 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[5] for repositories that use checkstyle (query string: 'checkstyle filename:pom.xml'). The repository information (including the URL) is 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.

We run checkstyle 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

---

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

Maximilian Jungwirth[6] which is based on *Styler*[7] [18]. 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.

We generate poorly readable code from the well readable one. To do this, we use the proposed REDEC tool (Section 3.4). We extract all methods starting with a comment. Originally, we planned not to require comments for the poorly readable dataset part. However, 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.

We take the mined well readable methods and the modified poorly readable methods and combine them into the mined-and-modified dataset. We remove code snippets that are identical for the original and the modified variant (Section 3.4). We 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 4.3). The created mined-and-modified 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 Modifications.

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[8] [31]. In the end REDEC parses the AST back to Java code using the Pretty Printer of the spoon library [31]. 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 3.1 and examples of the modifications in Listing 3.1 and Listing 3.2.

---

[6]`https://github.com/sphrilix/styler2.0`, accessed: 2024-03-21
[7]`https://github.com/ASSERT-KTH/styler`, accessed: 2024-03-11
[8]`https://spoon.gforge.inria.fr/`, accessed: 2024-15-02

Table 3.1: All Readability Decreasing Modifications with explanation and example.

#	Modification	Description	Example
1	`newline`	Replace a newline with none or multiple ones	Listing 3.1b, Lines 5-6
2	`incTab`	Replace a tab indentation with none or multiple ones	Listing 3.1b, Line 5
3	`decTab`	Replace a tab outdentation with none or more ones	Listing 3.1b, Line 7
4	`space`	Replace a single space with multiple ones	Listing 3.1b, Line 1
5	`newLine InsteadOf Space`	Replace a space with a newline	Listing 3.1b, Line 3-4
6	`spaceInsteadOf Newline`	Replace a newline with a space	Listing 3.1b, Line 2
7	`incTabInsteadOf DecTab`	Replace a tab outdentation with a tab indentation	Listing 3.1b, Line 9
8	`decTabInsteadOf IncTab`	Replace a tab indentation with a tab outdentation	Listing 3.1b, Line 8
9	`renameVariable`	Rename a variable declaration and its usages	Listing 3.1b, Line 1-3
10	`renameField`	Rename a field declaration and its usages	Listing 3.2b, Line 4
11	`renameMethod`	Rename a method declaration and its usages	Listing 3.1b, Line 1
12	`inlineMethod`	Replace a method call with the called code	Listing 3.2b, Line 7-8
13	`removeComment`	Remove a comment	Listing 3.1b, Line 1
14	`add0`	Add a zero to a number	Listing 3.1b, Line 2
15	`insertBraces`	Insert superfluous braces	Listing 3.1b, Lines 3-4
16	`starImport`	Replace a specific imports with a star-import	Listing 3.2b, Line 1
17	`inlineField`	Replace a static field with its value	Listing 3.2b, Line 7
18	`partially Evaluate`	Partially evaluate a constant	Listing 3.2b, 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 3.1a but modified for poorer readability.

Listing 3.1: Well readable (Listing 3.1a) vs. poorly readable (Listing 3.1b) 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 3.2a but modified for poorer readability.

Listing 3.2: Well readable (Listing 3.2a) vs. poorly readable (Listing 3.2b) Java
class files.

In Table 3.1 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 3.1a 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 3.1a, 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 3.1a, 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 the source code level after the reverse transformation. For example, the AST does not encode line breaks. Changes to these line breaks must be applied to the source code rather than the AST. You can see which modifications are applied when in Table 3.2.

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 %.

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 can cause keywords and/or identifiers to merge, which can result in the code no longer compiling. For example, consider the space between *int* and *number* in Line 6 in Listing 1.1.

Table 3.2: Available Readability Decreasing Modifications along with their execution type (on AST or Code), their configuration type (an array of probabilities or a single one), and whether they are included in the final dataset. Appendix B 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	

If we remove the space, the result is *intnumber*. Since the syntax is violated, the code no longer compiles.

For the final dataset, we exclude some of the modifications as we can see in Table 3.2. 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 modification. For modifications 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 modifications of configuration type *Single* a single probability must be defined (Table 3.2). 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 be written by humans. This is done empirically by examining exemplary outputs of REDEC with different configurations. You can find the resulting configurations in Table 3.3 and an exemplary file for `just-pretty-print` in Appendix B. `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 (**??**). 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 4.4).

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.

3.5 CONSTRUCTION OF QUESTIONNAIRES

We evaluate the generated dataset and the new approach with a survey (**??**). 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 based on the resulting strata. You can find an overview of the approach in Figure 3.2.

Table 3.3: Chosen configurations and their probabilities for the Readability Decreasing Modifications. 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

Stratified sampling: We want to categorize the Java methods to make it easier to decide which ones to select. We also want to avoid over-representing very simple methods, such as Getters and Setters. Therefore, our first step is to apply 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. Such features are, for example, the average line length and the code snippet length. Next, we compute the cosine
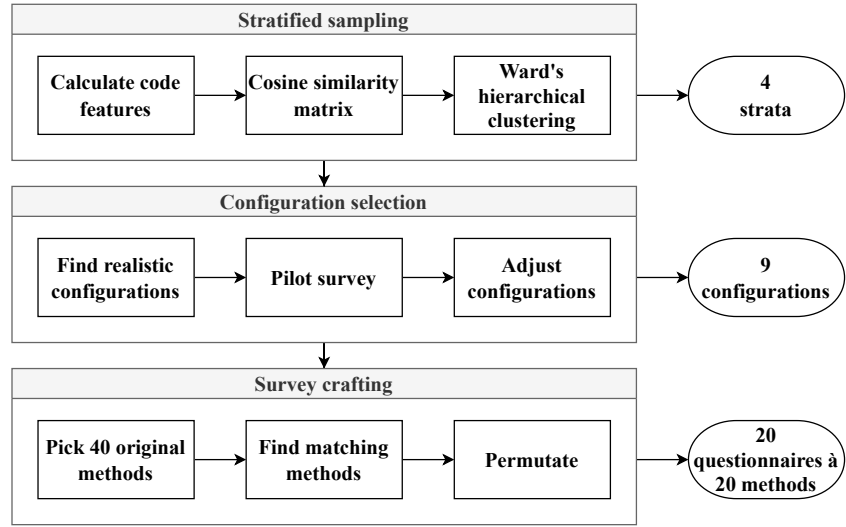
Figure 3.2: Steps performed to craft questionnaires from the mined-and-modified
dataset.

similarity matrix between all feature vectors using scikit[9]. The matrix contains
the similarity between all calculated vector pairs, based on the cosine of the
angle between the vectors. The distance between multiple vectors is called the
cosine distance. Finally, using the fastcluster implementation [29] 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
between the merge distance and the merged distance of the previous step $x - 1$:

$$\text{Difference to previous} = MD_x - MD_{x-1}$$

For each step, you can find the merge distance together with the difference to
the previous step in Figure 3.3.

The graph shows that a strata 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. In general, other layer
sizes are also suitable where the merge distance to the respective step is small
and to the next step is large. The *difference to previous* graph (Figure 3.3) depicts
those points as minima. Therefore, for example, 6 or 8 are also suitable. Of these

---

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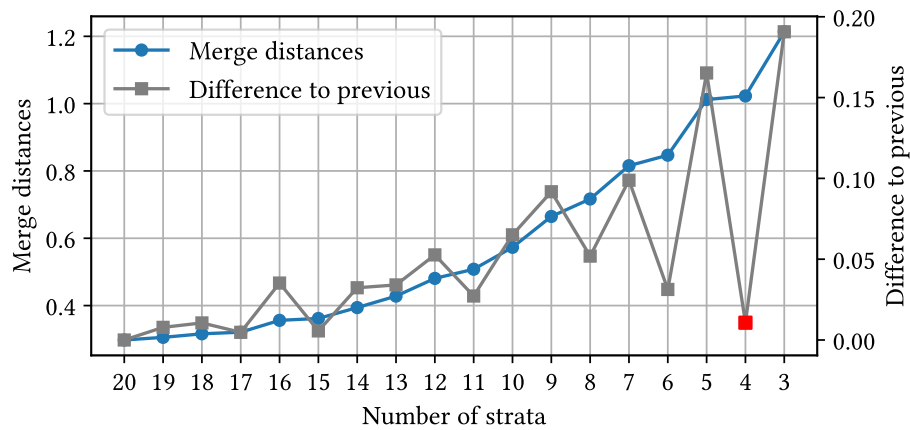`pairwise.cosine_similarity.html`, accessed: 2024-02-20

Figure 3.3: Merge distances and difference to previous merge distance. The optimal difference to the previous merge distance is highlighted in red.

suitable sizes, a strata size of 4 is the last option with a small merge distance before the number of strata becomes too small. We therefore opted for 4 strata.

We identify the type of methods within each of the 4 strata as described in Table 3.4. We also add the number 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 meanings, 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 3.3. Together with the original methods this results in 10 groups.

Survey crafting: Finally, we craft the questionnaires from the strata. We decided to provide all 10 configurations for each original method as we want to compare the original methods with their REDEC variants. We have a survey

Table 3.4: The strata properties (name, manually assigned type of methods, and the method count) and the number of methods sampled for the survey (in percent and the total count).

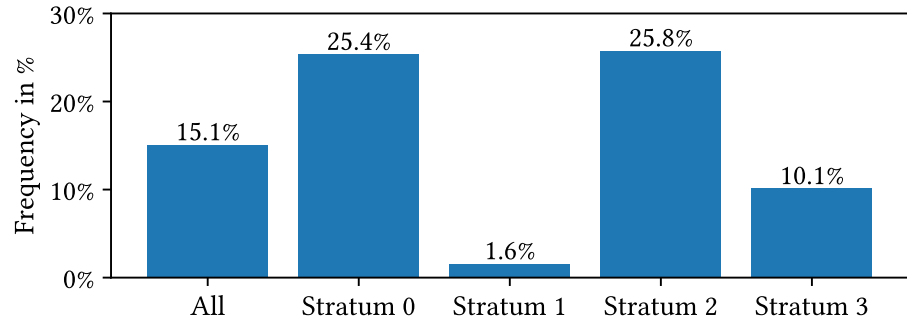| Properties			Sampling	
Name	Type of Methods	Count	Percentage	Count
Stratum 0	Simple methods	19016	10 %	4
Stratum 1	Complex methods	4280	40 %	16
Stratum 2	Magic number methods	78	10 %	4
Stratum 3	Medium complex methods	15938	40 %	16
Total		**39312**	**100 %**	**40**

Figure 3.4: Frequency of the case that a REDEC variant is not different from its original method.

capacity of 400 code snippets (Section 4.3). Therefore, the capacity for each REDEC variant is $400/10 = 40$ code snippets. We start by selecting 40 original code snippets and then we 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 3.4: 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 3.4). Additionally, simple methods are rather uninteresting for the classification of readability, as they can be 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 do this automatically 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 modification, the new method no longer matches the original method, in which case we match them manually.

Once we collected all 400 methods, we distributed them across the 20 question-naires, each with 20 methods. To not manipulate the raters, we decided 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 chose 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. This approach implies 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. [26] in classifying code readability with the mined-and-modified dataset.

We created our own implementation of the model using Keras[10]. The model uses three encodings. A code representation encoding, a feature extraction encoding, and a code readability classification encoding. You can find an overview of the model architecture in Figure 3.5.

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, the model generates three different code representations from each code snippet: A visual, a semantic, and a structural representation.
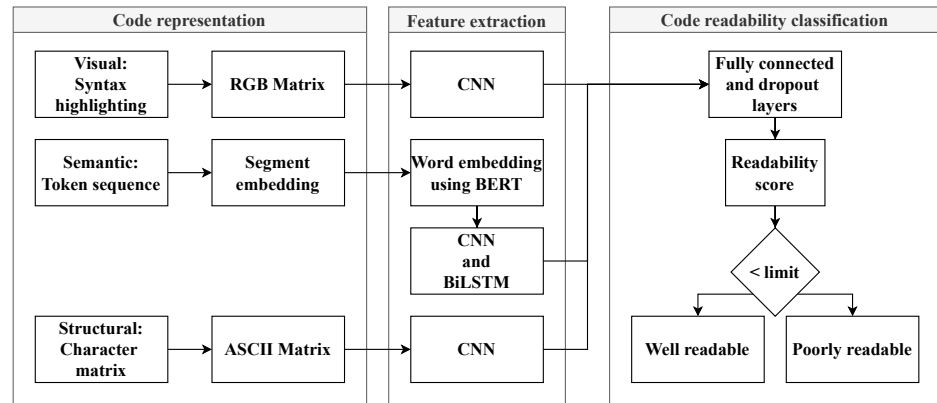
---

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

Figure 3.5: The architecture of the Towards model of Mi et al. [26].

Table 3.5: The color encoding used by the visual component of the Towards model [26].

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

For the visual representation, the syntax of the code is highlighted. Therefore Mi et al. assigned each type of syntactic element a color (Table 3.5). Instead of highlighting the words in the respective color, as done by an IDE, the words are replaced by color blocks instead (Figure 3.6b). Mi et al. used Eclipse[11] to highlight the code snippets and then took screenshots to obtain a RGB Matrix [26].

For the semantic representation, we split the code into tokens (e.g., keywords and operators) and use BERT [12] to embed each token as a vector [26].

For the structural representation, we split the code into characters and convert each into its ASCII value to obtain an ASCII matrix [26].

---

[11] `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.

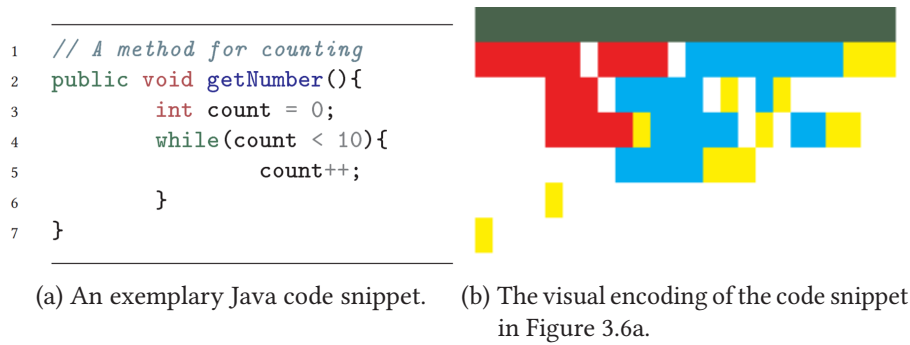(b) The visual encoding of the code snippet in Figure 3.6a.

Figure 3.6: A code snippet and its visual encoding.

The model 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 [26].

On the token embedding the model performs feature extraction using a BERT embedding layer, convolution layers, a max pooling layer, and a Bidirectional LSTM (BiLSTM) [26].

After extracting the features from the three individual representations the output is merged and used as 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 [26].

We implemented this model as described by Mi et al. [26] with a few adjustments: In contrast to the publicly available code of Mi et al.[12], 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 the evaluation, the model returns the evaluation statistics as a JSON file.

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 3.7.

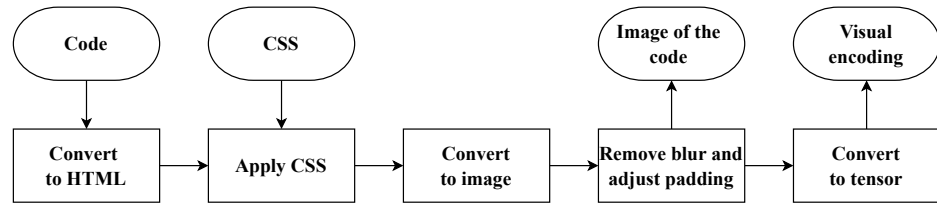[12]https://github.com/swy0601/Readability-Features, accessed: 2024-02-20

Figure 3.7: The steps to automatically, visually encode code.

In the first step, we use Imgkit[13] to convert the code to HTML. Thereby, an HTML class is assigned to each type of syntactic element. Next, we apply syntax highlighting using a CSS style sheet (Appendix D). In the third step, we use pygments[14] to convert the HTML with the applied CSS to an image. We use pillow[15] to remove blur and adjust the padding of the image. Finally, we load the image using opencv-python[16] 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 D).

During implementation, we encountered the following potential problem with the model: The token length for the BERT encoding (BERT-base-cased[17]) used in the model is 100. We take a look at what implications this has, and therefore we first need to look at what a token comprises. 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 3.3, 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.

Consider the method from Listing 3.3a. 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

---

[13]https://pypi.org/project/imgkit/, accessed: 2024-03-02
[14]https://pygments.org/, accessed: 2024-03-02
[15]https://pypi.org/project/pillow/, accessed: 2024-03-02
[16]https://pypi.org/project/opencv-python/, accessed: 2024-03-02
[17]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 3.3a using BERT-base-cased with a limit of 100 tokens. Space characters separate the tokens. Newlines are preserved for readability.

Listing 3.3: A Java method and its encoded-and-decoded variant.

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 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 make our results comparable with those of Mi et al. and thus enable a fair comparison of the datasets.

4

Evaluation

4.1 RESEARCH QUESTIONS

We tested the mined-and-modified dataset in two ways. We conducted a user study and evaluated the impact of using the dataset for the Towards model of Mi et al. [26]. 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 REDEC, 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.

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) can achieve this.

The modifications REDEC applies to 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 [16]. This applies under the assumption that the quality of the data is the same or at least similar. We want to check if the quality of our new data is sufficient for improving the deep learning-based readability classifier of Mi et al. [26]. We train their proposed model with combinations of the merged and the mined-and-modified dataset and compare the results.

4.2 PILOT SURVEY

In the following, we describe our setup, possible threats to validity and the results of the pilot survey.

4.2.1 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, 2024 ten people took part in the survey. Eight of them were students and two of them worked in the industry. All of them have computer science knowledge. 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.

4.2.2 THREATS TO VALIDITY

The threats regarding the pilot survey are as follows:

External Validity: We did not sample the Java snippets for rating in a specific or automated way, so there is a selection bias. Participants coming from a private environment further exacerbate this bias. Due to both, the results do not generalize.

Internal Validity: Ten people took part in the pilot survey. Due to the small number of participants, it is not possible to draw reliable conclusions about the strata or REDEC configurations. However, the results are sufficient to provide an indication.

Construct Validity: The accuracy of the participants' ratings for the code snippets is uncertain. We see no incentive for participants to intentionally provide incorrect ratings. We come to the conclusion to measure readability.

None of the mentioned threats has any impact as we do not use the results of this survey for the evaluation of our dataset generation approach. The intention of the pilot survey was rather to prepare for the main survey.

4.2.3 RESULTS

We analyze the results of the pilot survey regarding 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: Figure 4.1 shows the time it took the participants to complete the pilot survey and thus to rate 20 code snippets according to their readability. 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 4.1) 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 from the participants regarding the pilot survey is listed in Appendix A. 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
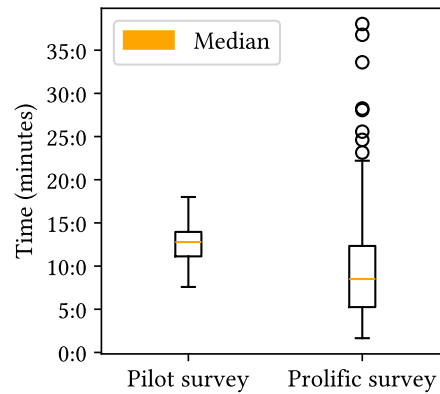
Figure 4.1: Time required to complete a questionnaire.

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.

Ratings: We adjusted the REDEC configurations. The rating for the last places (Table 4.1), 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.

4.3 PROLIFIC SURVEY

In this section, we summarize the results of the main survey conducted via Prolific[1].

---

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

Table 4.1: 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

4.3.1 EXPERIMENTAL SETUP

We conducted the survey using Tien Duc Nguyen's Code Annotation Tool (Figure 4.2) along with the platform Prolific for the recruitment and payment of participants. The survey was conducted between January 31 and February 7, 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 include his results in the evaluation. We estimated the time to complete one questionnaire at 10 minutes (Section 4.2). 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. A margin of error of 29.55% means that the actual readability value of a code snippet varies by up to 29.55% in both directions from the evaluation result. However, we aggregate over strata and multiple snippets later to reduce the margin of error. Each questionnaire consists of 20 code snippets. Consequently, 400 different code snippets are rated

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

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 C. 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?

The results of 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).

4.3.2 THREATS TO VALIDITY

We identified the following threats:

External Validity: Due to our questionnaire construction approach (Section 3.5), 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. Ensuring a well-defined target population is critical to the survey's quality. To mitigate the threat of an inadequately defined target population, we define it explicitly.

Internal Validity: To prevent concluding 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. 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 4.1) supports this argument. Especially the ratings of participants requiring less than 3 minutes (Figure 4.3b) to complete a questionnaire could have a negative impact on validity.
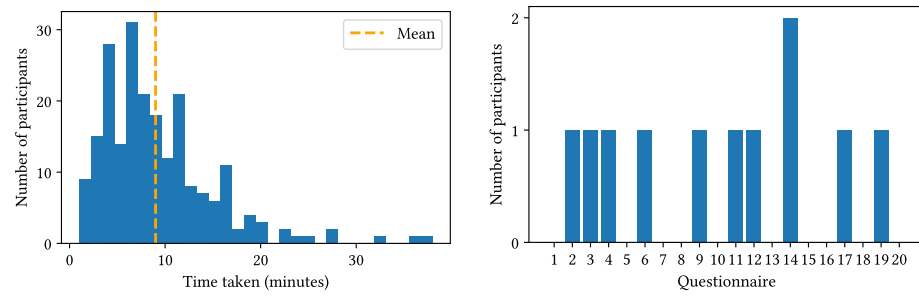
(a) Time required by participants to complete the survey.

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

Figure 4.3: Time analysis of participants completing the Prolific survey.

Construct Validity: The accuracy of the participants' ratings for the code snippets is uncertain. Apart from the already mentioned aspect that participants might hurry, we see no incentive for participants to deliberately give false ratings. We come to the conclusion to measure readability.

4.3.3 RESULTS

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

Completion time: You can find an overview of the time that the participants required in Figure 4.1 and Figure 4.3a. 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 4.1) 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 attention to the number of participants who took less than 3 minutes to complete a questionnaire (Figure 4.3b), 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, there are only a few participants who took less than 3 minutes. We assume that most participants completed the survey with reasonable effort.
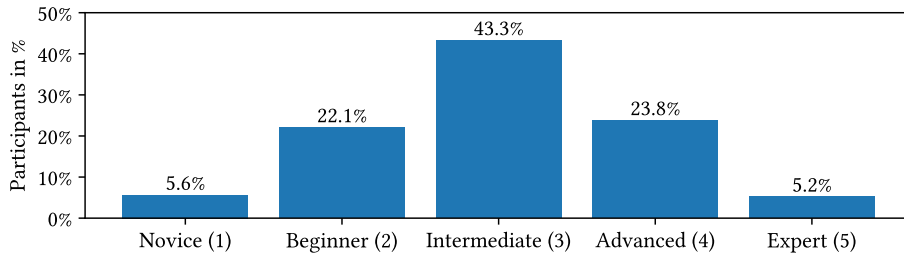
Figure 4.4: Familiarity of Prolific survey participants with Java.

Familiarity with Java: Figure 4.4 shows the participants' familiarity with Java. According to their estimation, most participants (43.3 %) have intermediate Java knowledge. 29.0 % of the participants stated that they are either advanced or experts in 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 4.5. We added the ratings of the merged dataset for comparison. Figure 4.5a 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 negligible 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 4.5a illustrate the distribution of the ratings in detail. Note that the evaluators were only able to rate the discrete values from 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 4.5a 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 4.5b shows the differences in mean values more precisely. RE-DEC 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 determine 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 4.2 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`.

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 `rename` where we can not confirm statistical significance. Overall, we showed that REDEC reduces the readability of source code.
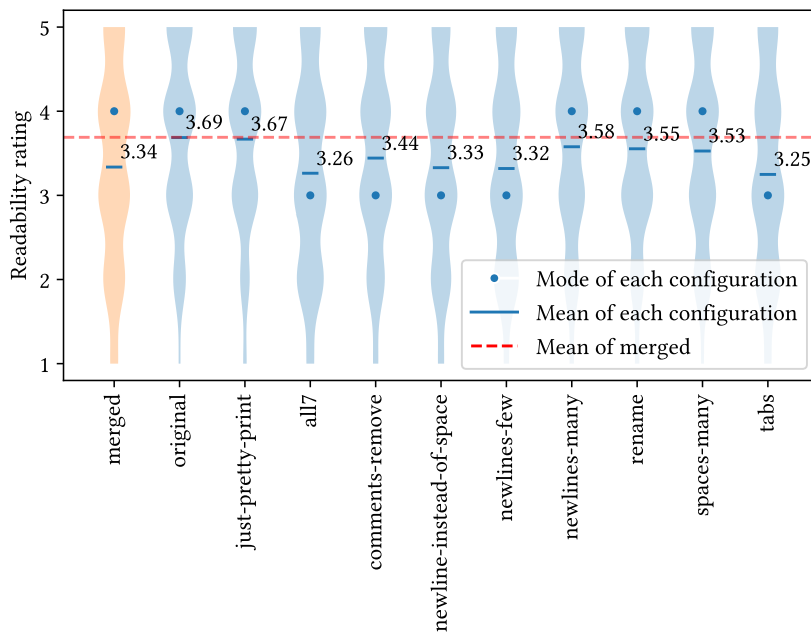
(a) Survey ratings for each REDEC configuration and all strata.
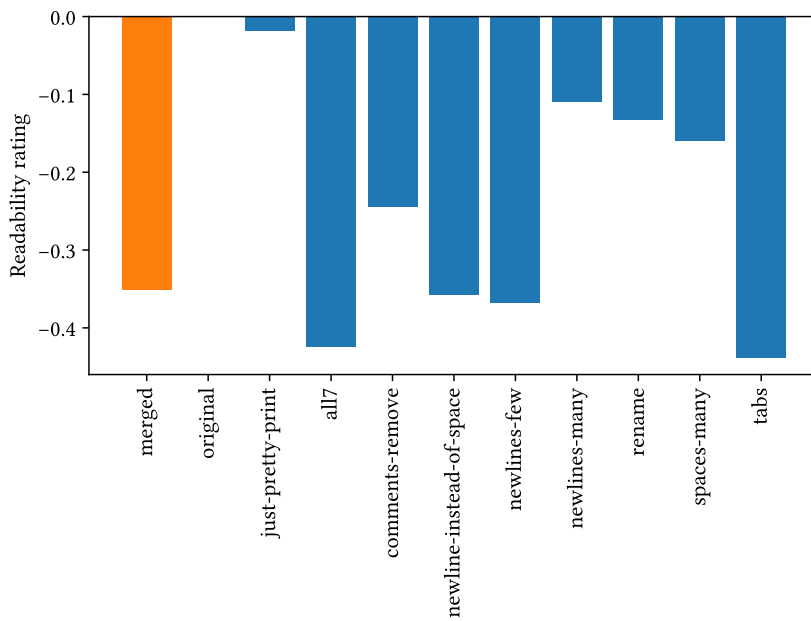
(b) Relative survey ratings for each REDEC configuration and all strata compared to all original methods.

Figure 4.5: Survey ratings for each REDEC configuration and all strata. Additionally, merged is added.

Table 4.2: 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}

Summary (RQ 2 - modified-poor)

All of the 7 configurations but `rename` decrease readability by a significant extent compared to `just-pretty-print`. We estimate the readability decrease for a certain probability of a certain type as can be seen in Figure 4.5b. We reject the null hypothesis and conclude that the poorly readable assumption (Assumption 2) holds.

4.4 MODEL TRAINING

In the following, we describe our setup and the results of the model training.

4.4.1 EXPERIMENTAL SETUP

We use the notation (`train-evaluate`) to describe on which dataset the To-wards model [26] is trained and evaluated. We aim to investigate the following things:

1. Model evaluation: To confirm that our implementation of the model scores similar accuracy as the original one of Mi et al. [26], we train and evaluate the model on the merged dataset (`merged-merged`).

2. Internal evaluation: To investigate how the model captures the readability aspects of the mined-and-modified (short: `mam`) dataset, we train and evaluate it

Table 4.3: Performance of different dataset configurations for the same model. `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 %

on this dataset (`mam-mam`). We 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`). We 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 the 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`).

4.4.2 RESULTS

Table 4.3 specifies on which dataset we train (`Train`) and on which dataset we evaluate (`Eval`) the Towards model. The table shows the results for each combination using the following evaluation metrics: Accuracy (`Acc`), Precision (`Prec`), Recall (`Rec`), Area under the Curve (`AUC`), F1-Score (`F1`) and the Matthews Correlation Coefficient (`MCC`) [9].

1. Model evaluation: We train and evaluate the model on the merged dataset (`merged-merged`) and obtain an accuracy of 84.7 % . This is similar to the results of Mi et al. (84.7 % vs 85.3 %). The deviation of 0.6 % accuracy might be due to the randomness of the splits for 10-fold cross-validation. We can confirm the results of the paper [26].

2. Internal evaluation: We train and evaluate the model on the mined-and-modified dataset (`mam-mam`) and obtain an average accuracy of 92.2 %. The

Towards model architecture is well suited for learning the structure of the mined-and-modified dataset. It learns the differences between `original` and `all7` methods and learns how to predict whether REDEC modified a code snippet.

3. Cross evaluation: We train the model on the mined-and-modified dataset and evaluate it on the merged dataset (`mam-merged`) and obtain 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. Since this is not the case and we know for both datasets that they address aspects of readability we conclude that we address different aspects of readability.

4. Fine-tuning: We tried to fine-tune the merged dataset by freezing different layers of the model trained with the mined-and-modified dataset (`finetune-merged`). 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. Our explanation for this is 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 4.2) shows that the mined-and-modified dataset captures readability. The model training results (Section 4.4) show that the mined-and-modified dataset captures different aspects of readability compared to the merged dataset. The question arises as to what the different aspects are and whether it is possible to extend REDEC (Section 3.4) so that the same aspects are captured. We assume that we could 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 4.4). 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, 14, 36] into a single dataset (Section 3.1), we set the 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 numbers of raters per snippet, different rater biases, and 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 need to distinguish between well and poorly readable code, which we can do without exact labels.

We determined the configuration for the REDEC tool empirically (Section 3.4). This was necessary because we first have to define configurations in order to obtain results. We proved significance for most configurations, but for the `rename` configuration we suggest to intensify the configuration.

We tried to choose the configurations for REDEC so that the modified code snippets are realistic, but whether they actually are has not been investigated (Section 3.4). One goal was to exceed the classification accuracy with our dataset through fine-tuning, and for this goal it is a secondary aspect that the snippets are realistic. We decided not to ask for realism in our study due to resource limitations. If we use the dataset beyond training of a model, we must investigate to what extent our modifications change the realism of code snippets.

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 (**??**) do not exactly match the configurations of the model training (Section 4.4). We hardly rely on the survey results to train the model. We argue that this is a rather minor threat.

By modifying code, REDEC introduces certain patterns for poorly readability (Section 3.4). For example, `comments-remove` removes a comment. This allows the classifier to decide that a method without a comment is poorly readable. `rename` introduces the pattern *v/m/f + number*, which the classifier can use to infer poor readability. Patterns from changes that alter line breaks, spaces and tabs are more complex to infer, but we suggest that they are present. One could argue that the individual patterns are shortcuts that the classifier can learn to determine readability. However, combining these patterns with a certain probability requires the model to learn all patterns. We suggest that the model overcomes shortcuts and learns code features instead. These features determine the readability of code, as our survey results showed (Section 4.3).

When comparing the model performance trained on the mined-and-modified and merged dataset, it should be noted that the merged dataset is small (Section 3.1). Consequently, comparisons to classifiers trained on the merged dataset may be unreliable.

We use a state-of-the-art model to evaluate the mined-and-modified dataset (Section 4.4). We chose the Towards model due to its high accuracy and as it uses different encodings to represent the code snippets (Section 3.6). Evaluation on just this single model does not allow generalization to all readability classification models. For further and more general evaluation of our dataset it is necessary to consider other state-of-the-art models or other encodings as well.

6

Conclusions and Future Work

Recent research in the field of code readability classification focused on various deep learning model architectures to further improve accuracy. Researchers paid little attention 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 (Chapter 3). Although our results show that the mined-and-modified dataset does not capture the same aspects of readability as the merged one (Chapter 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. We suggest to overcome this limitation by extending the tool to 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 [21]. One could try to improve the performance of the Towards model [26] 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 readability or developing 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.

The modifications described in this work (Section 3.4) are 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.

REDEC supports two ways of renaming identifiers: an iterative method and the use of Code2Vec for method names (Section 3.4). In the iterative approach, the names are shortened in many cases (e.g. *v0* or *m0*). We remove the meaning encoded in the name of the identifier, which should make it less readable. However, short method names tend to increase readability which contradicts our objective. The Code2Vec approach only supports method names and is therefore very limited. Since both approaches have drawbacks, this suggests that another way of determining the target identifier must be developed.

With our new dataset, we generated a large amount of data to train readability deep learning classifiers. When integrated into an IDE, this gives developers feedback on the readability of their code [10], allowing them to measure and improve the readability of their code. The code becomes more readable, which enables efficient collaboration, comprehension and maintenance [32, 1]. In addition, readability is the most time-consuming act in software maintenance and consumes over 70 % of the total lifecycle cost of a software product [8, 11, 35, 6]. Our dataset can therefore contribute to reduce software costs in the future.

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, 24. 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: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato et al. 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.

[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 Jr. 'No Silver Bullet Essence and Accidents of Software Engineering'. In: *IEEE Computer* 20 (Apr. 1987), pp. 10–19.

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

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

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

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

[12] Jacob Devlin et al. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[13] 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.

[14] Jonathan Dorn. 'A General Software Readability Model'. In: *University of Virginia* (2012).

[15] 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.

[16] Joel Hestness et al. *Deep Learning Scaling is Predictable, Empirically*. Dec. 2017.

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

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

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

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

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

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

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

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

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

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

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

[28] F.P. Miller, A.F. Vandome and M.B. John. *Abstract Syntax Tree*. VDM Publishing, 2010. ISBN: 978-613-3-77312-7.

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

[30] 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.

[31] 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.

[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: *International Journal of Software Engineering and Its Applications* 7 (2013), pp. 441–458.

[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.

A

Pilot Survey Feedback

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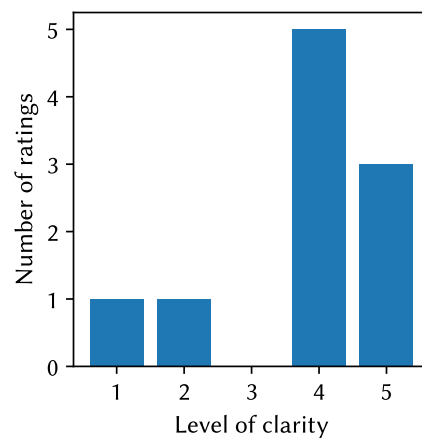
What problems were with the task? If there were none, leave blank.

- 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.

- Translated from German: In my opinion, the code is too complicated for a beginner with very little Java experience.

- 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.

- Translated from German: In my opinion, the tool is not suitable for beginners. The code is too convoluted and sometimes incomprehensible.

- 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

B

Readability Decreasing Modifications 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
```

C

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!

D

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[1].

```css
1   /* Comment Styles */
2   .c, .ch, .cm, .cp, .cpf, .c1, .cs {
3           background-color: #006200;
4           color: #006200;
5   }
6
7   /* Keyword Styles */
8   .k, .kc, .kd, .kn, .kp, .kr, .kt {
9           background-color: #fa0200;
10          color: #fa0200;
11  }
12
13  /* Parentheses, Semicolon, Braces Styles */
14  .p, .o, .ow {
15          background-color: #fefa01;
16          color: #fefa01;
17  }
18
19  /* Whitespace Styles */
20  .w {
21          background-color: #fff;
22          color: #fff;
23  }
24
25
26  /* Names/Identifiers Styles */
27  .n, .na, .nb, .nc, .no, .nd, .ni, .ne, .nf, .nl, .nn, .nt, .nv {
28          background-color: #01ffff;
29          color: #01ffff;
30  }
31
```

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

```
32   /* Literals Styles */
33   .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 {
34           background-color: #01ffff;
35           color: #01ffff;
36   }
37
38   /* Error Styles */
39   .err {
40           background-color: #fff;
41           color: #fff;
42   }
43
44   /* Generics Styles */
45   .g, .gd, .ge, .ges, .gr, .gh, .gi, .go, .gp, .gs, .gu, .gt {
46           background-color: #fefa01;
47           color: #fefa01;
48   }
```