# Towards a Naming Quality Model

Sander Meester BSc.
University of Amsterdam /
Software Improvement Group
Sander.meester@student.uva.nl

Sanne Bouwmeester MSc.
Sanne.g.j.bouwmeester@gmail.com

Dr. Ana Maria Oprescu
University of Amsterdam
A.M.Oprescu@uva.nl

Dr. Magiel Bruntink
Software Improvement Group
m.bruntink@sig.eu



## Abstract

Having highly maintainable software decreases the time spent on development. Although various research efforts show that the names of identifiers play a large role in the readability and maintainability of code, code quality assessments often do not take these names into account. Although developers can usually quickly assess the quality of a name, the abstract nature of names makes a fully automated assessment difficult.

This research investigates the creation of a general naming quality model. Our proposed model assesses: a) the syntactic quality of Java method names, b) how well a method body matches its name semantically. We assess this using 1) a set of guidelines from literature, 2) a machine learning algorithm trained on AST representations of method bodies.

Initial results show that the combination of a rule-based approach and a deep learning model can correctly indicate what names need attention. By inspecting the names flagged as a violation by both approaches we found that the combination of syntactic and semantic information yields better results than either of them by themselves. Further validation experiments on a Github commit dataset show that the model can distinguish between good and bad names, but still has room for improvement.



*In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.org/*

## 1 Introduction

Software quality plays a crucial role in the field of software engineering [ISO10]. Higher quality software enables developers to implement new features and changes faster, lowering the development cost [HKV07]. Maintainability is one of the factors that defines the quality of the software, which is further divided into modularity, re-usability, analysability, modifiability, and testability [ISO10]. This research will be related to the analysability and modifiability factors of the source code. Although source code properties that define these factors have been described [HKV07], the names used in the code are not yet used to influence these properties, although they play a significant role in it.

Making code easily understandable by using proper names influences the analysability and modifiability of code, and therefore the maintainability of a codebase [Mar08; McC04]. Every variable, class, argument, method, file and directory is named by the developer. Well thought names are the first step in aiding the developer to gain an understanding of the code [Mar08; McC04]. If a developers chooses bad identifier names the functionality of the code is heavily obfuscated. For example, if a method is named `'b'`, no developer can guess the intention of it. When it is named `'calculateMedian'` however, its functionality is clear from just the name.

The importance of naming is supported by Lawrie et al. [Law+06], who state that next to the comments, the developer is almost exclusively dependent on the naming of identifiers to understand the code. According to Murphy-Hill et al. renaming is the

most commonly occurring refactoring done in codebases [MPB12]. Research has also shown that low quality identifiers can be related to lower quality code [But+10; But+09; Law+06; LFB06; LFB07; Rel04]

Despite the relevance of naming quality there is currently a lack of maintainability models and code quality analysis tools that take identifier naming into account. To overcome this we investigate the creation of a naming quality model that can improve maintainability models and that can be used in quality analysis software.

Names are harder to validate than other code metrics, since good names must be accurate abstraction of the source code it represents (its context). This abstraction of context reflects the meaning of the code, which we will refer to as the semantics of a name. To verify whether a name matches its context some understanding of the source code is necessary, which is easy to assess for a human but difficult in an automated assessment. In contrast to this are the syntactic qualities of a name, characteristics of the name itself, such as spelling and structure, which are more easily verified.

To do naming quality assessment we will employ a combination of a ruleset and machine learning model learned on code representations, which combines the syntactic and semantic aspects a name should adhere to.

Syntactic aspects of a name are well-researched and guidelines to which good names should adhere to can be found in literature [Rel04; But+09; Abe+09]. However, as stated before, to improve comprehensibility of code it is also essential that the method name correctly represents its method contents.

To achieve this, machine learning is employed to assess the semantics of the name. To train this model, it receives a large number of method names and their bodies as input. The names are represented as natural language, and method bodies as an Abstract Syntax Tree (AST). From these examples, the model learns code embeddings, which are numeric vector representations, for both the names and bodies. This results in similar method bodies having similar vector values, which also applies to method names.

We can then infer from this model what method body embeddings are often found with what method name embeddings, and base a prediction on what name would fit a body on this [APS16; Pen+15; Liu+19; Alo+19; ALY18]. Comparing this name to the actual name can inform us how well the real name fits the method. This semantic check can then be used in combination with a syntactic check, which will result in a more complete naming quality assessment.

### 1.1 Research goal and contributions

In this study we build upon existing research in naming quality to close the gap in software quality analysis by including a naming quality assessment. To achieve this goal we propose a model that uniquely combines syntactic and semantic information. Syntactic information is used since the guidelines and regulations code should adhere to are a tested and proven contributor to naming quality [But+09; But+10]. We improve on these known syntactic guidelines by introducing semantic information into the quality analysis model, which allows us to assess how well a method name fits the body. Semantic information has been extracted from code before but it has not been used to assess naming quality.

We validate this proposed naming quality model by building a proof-of-concept implementation that uses these concepts to generate a quality overview for a project.

Not only do we create a naming quality model and a proof of concept implementation, we also relate the assessment of a codebase using this model to other code metrics showing the relevance of the model and naming in general. Lastly, we contribute to a line of research combining expert knowledge with statistical machine learning approaches.

Initially this research will target method identifier quality, as this is the smallest abstraction unit with significant context, and is essential for understanding the code. Also, we build our tool to work for Java as a start since it one of the most widely used programming languages.

Ideally, the final model can be employed by developers to quickly assess the general naming quality of their codebase and compare it to other codebases. Additionally, it allows them to find occurrences of names that need attention to improve the maintainability of their code. The quality assessment performed by our proof-of-concept model will be validated by inspection, comparing it to other code metrics such as readability, and by running it on datasets with justified naming quality.

In this research we aim to answer the following research questions, focused towards investigating, validating, and improving the proposed model.

- **RQ1**: How well does the proposed model perform.
- **RQ2**: How does the syntactic correctness checker compare against the semantic correctness checker.
- **RQ3**: Can we improve the semantic model using the syntactic knowledge.

## 2 Background and Related Work

To create a naming quality model we have to be able to infer if the name itself is syntactically correct, and if it accurately reflects the context the identifier is used in (semantics), which for method names we define as the method body. We first analyse research conducted on assessing the syntactic quality of names in code, and then dive into research using a combination of natural language processing and machine learning to analyse source code, which is what we will use to assess the semantic quality of names. The section ends with an overview of how a naming quality model could be validated based on related approaches.

### 2.1 Syntactic quality assessment

A common approach to checking identifier syntax is to define rules that define what good quality identifiers need to adhere to [DP06; But+11; Esh+11; CT99; Abe+09]. These rules usually only focus on checking the structure of the name itself.

An example of using rules to define naming quality can be found in the studies by Butler et al. [But+10; But+09]. They used a list of twelve identifier guidelines aimed at the contents of the name itself, such as maximum words used in the identifier, or encodings in the name. They show that adherence to these guidelines can be related to less bugs. In contrast, Lawrie et al. [Law+06; LFB07] define good identifier quality based on the quality of just the words the name consists of. They collected a dictionary of English words, which had known abbreviations added, and argue that if the words in the name are not in this dictionary, the name does not have a clear meaning.

A different approach focuses on a set of consistency, conciseness, and composition rules [DP06]. Although useful in some cases, these rules are only applicable on 10% of all identifier names according to the results. To work well they also need developers to keep an identifier dictionary during development, containing all identifiers. Their rules are therefore hard to include in a fully automated and general assessment of naming quality and has a limited application area.

Other work has been done on investigating the identifiers used in the code. These attempts are however aimed on a very small syntactic aspect of naming (e.g. only that not both CamelCase and snake_case are being used, or only detecting homonyms), do not take semantic context into account, or do not do a quality assessment. [DP06; CT99; CT00; Abe+09; AT13; Arn+10; Esh+11; HØ09; All+14; All+15; Liu+19].

The most complete overview of what encompasses naming quality was defined by Abebe et al. [Abe+09] who defined so called lexicon bad smells, which, according to the authors, are indications of deeper code problems, just as normal code smells are. From this research, as well as the other studies, we collected a set of guidelines on the syntax of method names to use in our syntactic checker.

### 2.2 Semantic quality assessment

To compare a name and its context semantically in an automated assessment, both the name and the context need to be represented in the same space. These representations can then be compared against each other to calculate similarity and possibly indicate where names differ greatly from the context they represent.

#### 2.2.1 Creating similar representations

One method of matching an identifier and its context focuses on treating the words in both the identifier name and the context around it as English sentences [DDO11; Sri+10; AT10]. Representing both a method and a name as an English sentence effectively places the problem in the Natural Language space. This method assumes that: a) the vocabulary used in both the name and its context are similar; b) the structure of the code is not necessary to accurately assess the context of a name. The advantage of this language-based method is that we can lean on extensive research in comparing the semantics of English sentences. However, a lot of code specific information is lost when translating code to an English sentence, making this method of representing code and names unfit for our purposes.

Another method of representing information, which is widely used, is representing things as a numeric vector called an embedding. The idea is that semantically similar objects are mapped to similar vectors, which allows for comparison.

The first approaches in this area were on natural languages, not on source code [Mik+13]. These approaches use neural networks to create word embeddings based on the words that are around the word of interest, effectively representing a word as a vector of the words it co-occurs with. These embeddings are then used for a plethora of applications, such as machine translation.

New approaches then tried to create these embeddings from source code to try and leverage generalisable concepts from code. It is hypothesised that since software is made by humans, the language used has similar statistical properties as natural languages. We can use this approach to represent both the name and the method as a vector so they can be compared. Transforming input into a vector is usually done by training a machine learning model on a large amount of data, which then learns which input data corresponds to what vector. Research has shown that is is possible

to train probabilistic models from existing code that can reason about code not seen before [All+18].

Some approaches transform the source code into a token stream and use it as input for the machine learning model [All+15; APS16], where others attempts build an Abstract Syntax Tree (AST) out of the source code and use that as input [Pen+15; Liu+19; Alo+19; ALY18]. This input is then fed into a machine learning model to create numeric vector representations. These approaches were tested with a method name prediction task, where a model performs good when it predicts the exact name of a method body based on its vector. The second approach (AST based) achieves better results in terms of precision and recall on this task.

To evaluate whether method bodies and names match, we can use these vector representations as created by a machine learning model to compare them.

#### 2.2.2 Comparing a method name to its body

After transforming both the method name and body into a similar representation, there are multiple ways to use them to do a naming quality assessment. The simplest approach is calculating the similarity between a name and a body and using that as a measure.

A language-based approach to this is taken by Arnaoudova et al. [Arn+10] who take identifiers and comments from the body of a method and calculate the textual similarity between them. De Lucia et al. [DDO11] take it one step further and convert the extracted words in a vector representation so the vectors spaces of both methods and names be compared. Sridhara et al. [Sri+10] use a Software Word Usage Model (SWUM) to extract the action, theme and arguments of a given Java method. They are then able to generate a single English sentence that reflects what the method does. Such a method can be used to extract a sentence that should represent the method body, and compare it with the actual name. The biggest problem with these approaches in relation to our envisioned model is the fact that we only have a very limited amount of text when we look at method identifiers. The similarity between a large amount of text extracted from an entire method including comments is not that accurately comparable to just a name.

Other approaches make use of machine learning to train a model to recognise similar representations. Advantages of this are that more complex statistical relations can be modelled.

Liu et al. [Liu+19] use paragraph vectors and convolutional neural networks to extract embeddings of method names and bodies that can be used as input to train a deep learning model. For each method name and body name, their model selects all other names and bodies that are close in the vector space of embeddings. When the name and the body have no overlap in their respective space, they are considered inconsistent, and names of other methods that are close (i.e. that are very similar to the snippet of code) are suggested.

Although their goal is similar to ours, no quality assessment of any kind is being performed, which is the area we want to apply the models knowledge.

We can however perform a quality assessment by using a model that was introduced by Alon et al. [Alo+19]. They trained a neural machine learning model to predict the name of a method based on AST-based representations. Since the code2vec model by Alon et al. is completely open source, it can be employed by us to take it one step further and relate the output of the model to naming quality.

### 2.3 Model validation

To assess the performance of our model, we want to validate whether the names indicated by our model as low quality decrease the maintainability of the code, and that the names not flagged as violations do not decrease it. This is applicable to both the syntactic and semantic approach. On the one hand we need to relate our syntactic guidelines to improved code quality, and on the other hand we want to relate our semantic quality assessment to improved code quality.

To show the effect of low-quality identifiers Butler et al. have done two different empirical studies that show that syntactically flawed identifiers have a negative influence on code quality [But+10; But+09]. Although their guidelines are very basic and do not reflect the entirety of naming quality, they have shown that already neglecting these basic guidelines relates to more code quality issues reported by a static code analysis tool called FindBugs. To test whether guidelines help developers create better names, Relf [Rel04] identified a set of twenty-one guidelines and did a small study with developers. In this study developers were asked to accept or discard the guidelines provided. All guidelines except one were confirmed by the developers as helpful in creating more descriptive identifiers. These two findings combined show that guidelines do indeed lead to improved naming quality.

When looking at the semantic assessment, we need some other model or code metric to compare the output to. General code readability models only take structural aspects into account [Dor12; PHD11; BW10]. Structural aspects for example are number of comments or the length of a line, in other words syntactic aspects. Scalabrino et al. [Sca+16] show that adding textual aspect such as contents of comments and identifiers would improve those models. To do this, they add multiple textual aspects to the model.

These aspects are comments and identifiers consistency, the number of dictionary terms in identifiers, hypernym density, textual coherence, comments readability, and number of meanings of a snippet.

The metrics used by both Scalabrino et al. [Sca+16] and Buse et al. [BW10] to define code readability can be used by us to compare the assessed naming quality. They also have both created an annotated dataset of methods with readability scores. This readability score was obtained by letting computer science students and developers go over files and awarding them a score. Although this score is clearly not a perfect truth, significant similarity between the scores from the assessors and the scores from our model can indicate our model approaches comprehensibility in a correct way. The biggest problem is that the assessment is done on readability of the code snippets, and not on the quality of the names. The scores are given for readability of the code, not the names. Even though naming is a big part of readability the scores are not exclusively a naming quality metric.

Beside relating the model to readability scores one could also evaluate a model by comparing it to a dataset of known good and bad names. Unfortunately, there is no gold standard of high-maintainable and readable code to compare a model against. To evaluate their model, Liu et al. [Liu+19] collected a dataset of methods whose names were changed in a GitHub commit whereas the body remained the same. The names before the change are considered inconsistent with their body, and the names after the commit consistent. This dataset can be employed by us to test whether our model also flags the pre-commit names as being low quality more often.

# 3 Naming Quality Model

In this study we assess naming quality in two ways, syntactic correctness and semantic correctness. The model we porpose consists of two different parts, both having the objective to assess naming quality. The first part does a syntactic quality analysis based on literature guidelines, whereas the second part does a semantic analysis of the method name and body. The output of both parts, as well as the complete model, is mapped to a binary assessment ("violation"/"no violation") per name so that the results are comparable.

Our intuition is that the combined model will achieve better results in detecting low quality names than both approaches by themselves can. Next steps in combining the two approaches are described in more detail in Section 7.1.

An overview of the proposed model can be found in Figure 1. Each of the following sections highlights a part of the setup.

## 3.1 Source code parsing

To create a naming quality model we first need to be able to extract the relevant pieces of information from the source code. With the use of JavaParser[1] we create an AST representation of the source code which we can traverse to collect the info we needed. For the syntactic analysis, we parse the source code into a Java HashMap containing the file a method is in, the method name, and the return type.

For the semantic analysis, we need the output of the parser to match the input the code2vec neural models expects. Code2vec takes an AST as input, and generates all possible paths and their embeddings for it, see also Section 3.3. For this reason, we pass the entire AST to both approaches.

## 3.2 Syntactic quality assessment

We analyse the information retrieved from the parser using two different approaches. The first approach uses a rule-based model that checks the method name itself for violation of guidelines. As we concluded in our background section, research has been done that assesses the source code lexicon and indicates what high-quality identifiers should adhere to to increase readability and understandability. This research shows that a syntactic quality assessment is possible and beneficial. From the literature we collected a method naming ruleset that method identifiers should adhere to to be considered of high-quality, as shown in Table 1.

To check the source code for adherence to these guidelines, the output of the Java Parser is written to a file and read by a Python script. This script goes over all method names to check their adherence to the guidelines. The result is an overview of which method violates which guideline, and a total number of violations per guideline. Thereby laying the foundation of further experiments relating syntactic to semantic assessments.

Based on preliminary experiments some adjustments were made in the guidelines. We removed guideline two: "A method name should be composed of words found in the dictionary, and abbreviations and acronyms that are more commonly used than the unabbreviated form" since every new project introduced new abbreviations that were clear in their context, and as stated before, when an abbreviation is 'known' or more commonly used than its non-abbreviated form is hard to assess.

## 3.3 Semantic quality assessment

The second approach to assess naming quality revolves around learning a neural model on the extracted source

[1] `https://javaparser.org/`

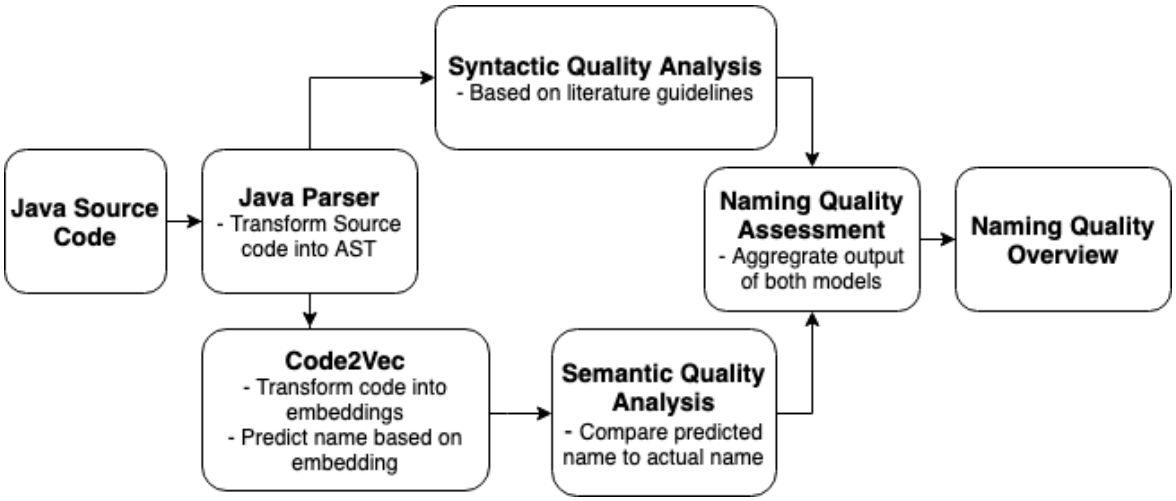


Figure 1: Architectural overview of proposed naming quality model

| ID | Description | Source |
|---|---|---|
| 0 | A method name should not be composed entirely of non-alphabetic characters. - e.g. `www` | [But+09; Rel04; Abe+09] |
| 1 | Identifiers should be composed of between two and four words. This prevents too general names. - e.g. `handle`, `calculate` | [But+09; Rel04] |
| 2 | Identifier names should be composed of words found in the dictionary and abbreviations, and acronyms, that are more commonly used than the unabbreviated form. - e.g. `createWTT`, `fooFlip` | [Abe+09; But+09; Rel04] |
| 3 | Identifier duplication, differentiated only by a digit - e.g. `create_tree`, `create_tree2` | [Rel04] |
| 4 | A method name that start with has/contains/is/check should return bool, and vice-versa A method that start with get/return/find should return non-bool and non-void | [Rel04; Abe+09] |
| 5 | Method identifiers should follow the following convention: `<verb> NotVerb*` | [Abe+09] |
| 6 | A method name should not employ both CamelCase and SnakeCase - e.g. `create_newHouse` | [Rel04] |
| 7 | Identifiers should not be composed entirely of numeric words or numbers. - e.g. `42`, `two` | [But+09; Rel04] |
| 8 | A method name should start with an alphabetic character - e.g. `1createHouse`, `2createHouse` | [Rel04; But+09; Abe+09] |
| 9 | A method name should end with an alphanumeric character - e.g. `createHouse!` | [Rel04; Abe+09] |
| 10 | A method name should not contain two underscores in succession - e.g. `create__new__house` | [But+09] |

Table 1: Literature identifier quality guidelines to improve code comprehension.

code data, and allowing it to assess what name would best fit a code snippet based on learned experience. This predicted name is then compared to the actual name.

Recent work by Alon et al. [Alo+19] resulted in a neural attention model that maps a method body onto a method name. This model takes an AST representation of a method body as input and returns an estimation of the name it predicts the body has, with different predictions and the confidence for each prediction. The attention mechanism of their model allows us to inspect how the model comes to its predictions and could teach us how to improve the model.

From the AST representation, each possible path through the AST is saved in a so called path-context containing the starting token, the terminal token, and nodes between these two tokens (e.g "x = 7" would become: $\langle x, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 7\rangle$ [Alo+19]). As seen in the example, the nodes are saved as a simplified representation, and arrows between the nodes denote if the next node is a parent or child node.

An example of the models output can be found in Figure 2 and 3, with the first example showing an example where the model correctly predicts the real name, and a second example where the model has more trouble predicting what the name should be.

```
Original name: setup|job
» (0.998678) predicted: ['setup', 'job']
» (0.000847) predicted: ['commit', 'job']
» (0.000291) predicted: ['write', 'file']
» (0.000032) predicted: ['metadata']
» (0.000031) predicted: ['set', 'output', 'path']
» (0.000030) predicted: ['cleanup', 'job']
» (0.000023) predicted: ['get', 'output', 'path']
» (0.000023) predicted: ['write', 'chunk']
» (0.000023) predicted: ['write', 'data']
» (0.000022) predicted: ['set', 'sorted']
```

Figure 2: Example of correct model prediction

```
Original name: create|remote|task
» (0.683358) predicted: ['find', 'task']
» (0.084563) predicted: ['create', 'task']
» (0.066416) predicted: ['adding', 'service']
» (0.039051) predicted: ['get', 'next', 'task']
» (0.038583) predicted: ['create', 'manager']
» (0.035006) predicted: ['create', 'directories']
» (0.019563) predicted: ['create', 'environment']
» (0.012980) predicted: ['get', 'sub']
» (0.011408) predicted: ['create', 'reference']
» (0.009071) predicted: ['new', 'task']
```

Figure 3: Example of less correct model prediction

We adapted the code2vec model to generate a large number of expected names for method bodies, and compared these expected names to the real names using natural language similarity methods.

To detect names that "violated" the ML model, we calculated the textual similarity between the guessed name and the real name using the Jaro-Winkler distance, putting more emphasis on the start of the name since this is defining for its functional intention. If the similarity was lower than .6 we considered the name to be too far off from the prediction, and counted it as a violation.

As described above we use this machine learning approach to validate the semantic integrity of a method name and the body it represents. In this study we compare this approach to the syntactic approach using guidelines from literature, as well as combine them to validate the structural and syntactic integrity of a name. Together they form the basis of our naming quality model.

## 4 Experimental Setup

In this Section we describe the experimental setup used to answer the research questions. Experiment 1.1 and 1.2 are aimed towards assessing the performance of the model using different approaches, and Experiment 2 aims to answer how the syntactic and semantic approach differ.

### 4.1 Experiment 1: Performance of the model

To assess the performance of the model and answer RQ1, we need an oracle that allows us to evaluate the output. This oracle should tell us how often the model is correct and how often it is not. Having this will also allow us to set a baseline to compare future improvements and iterations of the model to. To achieve this, we performed two experiments to try and validate the performance of the model.

#### 4.1.1 Experiment 1.1: Readability assessment

As a first validation attempt, we looked into the dataset made by Scalabrino et al. [Sca+18] and the dataset made by Buse et al. [BW10]. Both datasets contain a variety of Java code snippets with readability assessment scores of that snippet done by students and developers. For each snippet, we let both models check for violations and compared it to the average readability score given by the assessors.

If we can relate it to the readability assessments this could show that by our model highly-evaluated snippets are also considered highly readable by their model. Every new snippet can then be validated as either high or low quality considering their names based on the readability score. This could then represent our oracle that we base the models performance on.

Next to their datasets, we can use to readability score calculation tool as developed by Scalabrino et al. [Sca+18] that uses textual aspects to calculate a readability score for a snippet. If these score can be related to the amount of violations found by the model, we can use it as an oracle to define whether the model is correct.

#### 4.1.2 Experiment 1.2: Github commit dataset

As a second validation, we looked into the approach taken by Liu et al. [Liu+19] as explained in Section 2.3. They propose a dataset of method names that were changed in a commit where the body remained the same. They reason that since the name is changed while the body is not this indicates a change was needed to better reflect the body, meaning the old name did not reflect the content of the body. Therefore the names before the commit are seen as flawed names, and the names after the commit as not flawed.

From their dataset we removed duplicate method names which would skew the results, since the names all either do violate or do not violate the guideline, as the bodies are nearly equal. If the model happens to perform well in those occasions the overall performance seems better than it would on a more varied dataset. The resulting dataset contains 1597 methods.

For our experiment, we compare the assessment of our model on these names and their respective bodies to this qualification. Ideally, all pre-commit names should be indicated by our models as flawed, whereas all after-commit names should not be indicated as such. The underlying assumption made here is that no pre-commit names were already of high-quality and that every post-commit name is of high-quality.

### 4.2 Experiment 2: Comparing the semantic to the syntactic checker

Comparing the two different approaches tells us how they differ, what benefits and disadvantages each approach has, and how each approach performs. To answer RQ2, we investigated the output of both parts of the model both by looking at the number of flagged violations. We also performed an in-depth analysis of the models outputs to see what names they flag as violations and what names they do not.

A first experiment to compare the syntactic checker to the semantic checker was done by running the model over different Java projects and outputting the number of methods that each approach flags as a violation. This gives us insight in the difference between the semantic and the syntactic model, allowing a first answer to RQ2.

For this analysis the codebases Hadoop-MapReduce[2], Presto[3] and Cassandra[4] were used.

We report the number of methods that each model flags as a violation. The rule-based model outputs whether or not a name violates at least one of the guidelines. For the neural model we used the textual similarity between the actual name and the predicted name as explained in Section 3.3 to flag a name as a violation or not.

Besides plain numbers an in-depth qualitative analysis of the output will be reported, which further answers RQ2.

## 5 Results and Interpretation

The following section shows the results from the experiments explained in Section 4, and what we can learn from the resulting data. We also relate the results back to the research questions, evaluating whether a semantic analysis is possible and how the results differ from a syntactic analysis.

### 5.1 Experiment 1: Model validations

To evaluate the performance of the model, we performed two experiments with different datasets to see how well they are usable as a validation.

#### 5.1.1 Experiment 1.1: Readability assessments

The results of this experiment are shown in Table 2, where "at least one violation" indicates that there was at least one method in the snippet that violated both models. The snippets were overall small of size and mostly only contained a single method, so at least one

[2] https://github.com/apache/hadoop
[3] https://github.com/prestodb/presto
[4] https://github.com/apache/cassandra

method in most occasions is identical to all methods in the snippet.

| Dataset | Scalabrino | Buse |
|---|---|---|
| Snippets | 200 | 58 |
| Average readability score of snippets with at least one violation | 3.54 | 3.32 |
| Average readability score of snippets with zero violations | 3.6 | 3.22 |

Table 2: Overview of readability scores related to the naming quality model.

The results show that there is no clear relation between the readability scores given be the assessors and the violations indicated by the model. This can be explained by the assessors not taking the name itself too heavily into account when assessing the readability.

**Readability calculation tool**
The readability calculation tool from Scalabrino et al. [Sca+18] was also used to assess whether these scores could be related to the naming quality model. This tool uses linguistic metrics of the source to calculate a readability score for a code snippet. It was however found that the impact of names on the readability tool was not in line with standard coding guidelines of our own tool. As an example we take the code snippet as shown in Listing 1, but with different names. Changing the name from `'median'` to `'b'` changes the readability from 0.508 to 0.531, indicating an increase in the readability according to the readability model. Changing it again to `'calculateMedian'` gave a readability of 0.466, lowering the readability score even more. This shows that a change in the name of a method does not translate that well to the readability measurements.

These readability assessments therefore were deemed unfit to base our validation on. These results also again show that validating the quality of a name used for a method is a difficult thing to do.

```
public Integer median() {
    if(p1.size()==p2.size())
        return (p1.peek() + p2.peek())/2;
    return p1.size()>p2.size() ?
            p1.peek() : p2.peek();
}
```

Listing 1: Code snippet uses to assess the influence of the name on the readability score as calculated by the tool from Scalabrino et al.

#### 5.1.2 Experiment 1.2: Github commit dataset

The result of the analysis can be found in Table 3.

| | Number of methods | |
|---|---|---|
| | Pre-commit name | Post-commit name |
| Total | 1597 | 1597 |
| Violated both models<br>- similarity score <0.6<br>- violated at least one rule | 644 (40%) | 399 (25%) |
| Violated at least one rule | 937 (59%) | 647 (41%) |
| Similarity score <0.6 | 968 (61%) | 776 (49%) |

Table 3: Overview of commit change dataset related to the naming quality model.

From this data we can see that the number of old names that violated the model is significantly higher than the number of new names. If both the dataset and the model were perfect however, then the result should be 100% and 0%. Not every name in the dataset is actually improved after the change, or is improved but still flawed. Examples can be found in `‘‘read self"` being changed into `‘‘self read"` or `‘‘get current size"` into `‘‘size"`, and new names being `‘‘forcibly refresh all cluster state slow"` or `‘‘point num bytes"`. These entries in the dataset result in the model flagging some new names as flawed, which is not surprising as these names can be considered flawed. This does however not make up for the entire difference between ideal results and achieved results, as the model is also not perfect. These results set a baseline for this version of the model, which we can compare further iterations of our model to. An improvement on the assessment done on the dataset indicates that the model is better capable at distinguishing between good and flawed method names, which allows us to answer RQ3 in the future.

### 5.2 Experiment 2: Syntactic versus semantic approach

The results of experiment 2 are shown in Table 4.

| | Number of methods | | |
|---|---|---|---|
| | Hadoop | Presto | Cassandra |
| Total | 1017 | 14449 | 13684 |
| Violating both models | 149 | 2508 | 4177 |
| Violating semantic check (Similarity <0.6) | 389 | 6360 | 8078 |
| Violating syntactic check | 328 | 5250 | 7005 |

Table 4: Distribution of methods based on their violations

As seen in the table, both approaches find around the same number of methods violating the set restrictions. From all names detected as flawed by each approach, about half of each is also flagged by the other. The absence of a ground truth makes it difficult to assess the number of false positives and negatives. The low overlap is in part caused by the distinct nature of both approaches. Where the rule-based model assesses the syntactic integrity, the neural model focuses on semantic similarity.

These results indicate that improvements can be achieved when combining the semantic and syntactic information instead of using either. Almost half of the violations found by each checker are not flagged by the other. This shows that using only a guideline based approach achieves different results than a semantic analysis, giving an initial answer to RQ2, although the exact way they differ is not yet clear. Additionally, we see a significant difference in the percentage of methods found violating both checkers, showing that a difference in naming quality between projects could be indicated.

### 5.2.1 Qualitative analysis

To gain a feel for what kind of names each approach assessed as low quality, where they differ and whether semantic analysis is possible, we performed a manual qualitative inspection. This inspection gives us more insight into RQ2. In this analysis, we went over the output of the model consisting of method names and bodies, the predicted name and the similarity score, and the list of guideline violations.

When looking at the syntactic analysis, we found that guidelines 1, 4, 5 and 6 were often violated (>10% of names), where the other guidelines were only violated sporadically (<1%).

Among the names inspected that violated both models, no names were found that according to our opinion should not be improved. For example, the name `‘‘capacity"` was predicted by the ML model as `‘‘get starts"` (similarity: 0.3166) and violating both rule 1 and 5. This indicates that both the syntactic and semantic quality were low.

Names that violated only the guidelines were often single-word method name, which violate guideline 1, but were not found by the ML model. For example method name “handle”: was predicted to be “handle and thus had a similarity score of 1, even though the name is not very informative. This is likely caused by pollution of the training data of the code2vec model [Alo+19].

Names that violated only the ML prediction seemed to be a false positive in at least one in four inspected violations. This was caused by the predicted names being of low quality themselves. A correct example is method name `‘‘double who am i"` which was predicted to be named `‘‘test login"`. `‘‘double who am i"` is not a very descriptive name and even though `‘‘test login"` is not perfect either, the fact that they are very different is important for our assessment. Other violation indications were less correct, as for example method name `‘‘decode initial context token"` was predicted to be `‘‘put"`. Although their textual similarity is low and therefore should indicate a name not correctly describing the intention, the original name is actually quite descriptive and looks right, while the predicted name is uninformative and even violates the set guidelines. Investigating what causes this might lead us to being able to improve to model and prevent these mistakes from happening.

We found no names that violated neither of the models but could still be considered low quality names. Although this does not proof the model had no false negatives, it shows promising indications that the model can detect low-quality names using a combination of both approaches, without missing out on many low-quality names.

## 6 Discussion on current work

The results indicate that a naming quality model that uses both syntactic content and semantic context to assess the quality of identifiers is possible for the case of Java methods. We can positively answer RQ1 since from the output we see that the model seems to be able to detect what names need attention. The model also finds far more violations in pre-commit names compared to post-commit names, which also should be of higher quality.

When looking at the results of Experiment 2, we can see that the syntactic and semantic approach differ in what names they flag as violations. This provides an answer to RQ2, as well as indicate that there is benefit to be gained by combining both approaches.

The results also show that the model has room for improvement. As stated in Section 5.2.1, the machine

learning model sometimes predicts names that are in conflict with the guidelines itself, which shows that sometimes a disparity between the predicted name and the real name does not tell us much about the quality of the name. The model is also sometimes simply not capable of accurately estimating the name the method should have.

These flawed predictions can be due to the fact that machine learning model used to assess the similarity between a method's name and its contents is based on the assumption that the names in the training data are consistent with the body they represent. If the training data is flawed and contains a lot of names that do not reflect their body well, the output of the model on new data will be influenced by it. Since we have no large dataset of verified high-quality and low-quality names, this is unavoidable. A perfect assessment therefore seems not achievable, although our model does produce useful findings.

### 6.1 Threats to validity

Although the initial results are promising, there are some issues we would like to address.

It was mentioned earlier that the dataset based on commits as described in Section 5.1.2 is not a perfect truth. The degree to which a disparity between the expected ideal output of our model and the actual output can be addressed to either is hard to assess. Names could be changed in a commit due to other reasons than it not reflecting the body, such as adherence to naming conventions. To validate the correctness of the dataset, a manual evaluation by a group of professional developers could help, but a perfect yes/no golden dataset cannot be achieved since a 'good' name is not a binary metric. This also shows why a good naming quality model would be valuable.

The number of names indicated by the semantic model to be inconsistent with their body, which are not actually inconsistent, form a big threat to the models performance. This namely means that the machine learning model is not always predicting high-quality names itself, which is an assumption the quality assessment is based on. We address this issue Section 6.2.

Another point we want to address is that the qualitative analysis was done by the authors, instead of a group of volunteers, which might have led to bias in the interpretation of the models output. Another validation of the model done with a group of volunteers should help remedy this threat, although subjective opinion will always play a role in manual assessments.

The last issue is the fact that there is exists large number of naming guidelines, which only a subset of is employed by this research. Although these guidelines have been validated by other research, including other guidelines might influence the results. A complete list of all guidelines in existence is not feasible, but a more extensive list than currently employed could be beneficial in the future to achieve better results.

### 6.2 Future Work

Although the current implementation of the proposed model shows promising results, there are different possible changes that can be made that might lead to better performance, which are addressed in this subsection.

The way the similarity between the predicted name and the actual name is calculated can be done in many different ways, and investigating the impact of different approaches could lead to increases performance. At the moment, a Jaro-Wrinkler textual similarity is being used, whereas one could also use code or word embeddings to calculate how similar both vectors are in their embedding space.

There are other machine learning approaches that also generate a descriptive name or sentence based on the contents of a method body. These approaches for example use a token stream instead of an AST-based approach. Although the code2vec model by Alon et al. [Alo+19] achieved the best recall and precision, this was based on predicting the correct names, while for quality assessment other approaches might work better, making it worthy of investigation.

Even though the model achieves good results for assessing the quality of Java method names, one could also include other abstraction units such as classes in our assessment. This extra context knowledge could lead to better prediction of names and less false-positives.

It is also interesting to investigate how well the model applies to other programming languages. In theory, the model is generally applicable and can be extended to other languages, but it is worthy of investigation to see how the results differ.

So far, a golden dataset with annotated naming quality has not been found, making precision and recall no viable option for accuracy measurements. Finding one would make more extensive validation possible, so the search for one is still beneficial. Additionally, manual assessment of a the quality of a large part of methods by a group of developers could be done to create one ourselves.

## 7 Conclusion

To automatically assess the quality of identifier naming to enhance maintainability assessments, this research proposed a naming quality model that employs both a naive ruleset to check the syntactic quality of

method names, and a Neural Language Model to check the semantic quality.

To answer RQ1: *"How well does the proposed model perform."* we employed readability scores, which were inconclusive, and the Github commit dataset, which seemed to be useable for validation. This second experiment gave insight into the performance of the model, showing that the model finds less violations in the pre-commit names compared to the post-commit names, which is in line with the expectation that pre-commit names are of lower quality.

For RQ2: *"How does the syntactic correctness checker compare against the semantic correctness checker."* we compared the output of the semantic model to a syntactic one. From the names flagged as violations by the semantic model, about half are also flagged by the syntactic quality model, and vice versa, giving a first answer to RQ2. A more in depth analysis of the output has shown that a combination of both approaches seems to lead to the best results, as names flagged by both approaches were found to be of low-quality. By itself the semantic model suffers from a lot of false positives where the original name is a good one and the predicted name is not. This results in names being flagged as a non-descriptive while they actually are descriptive. By combining the semantic and syntactic checker the number of false positives seems to go down. This does lead to less low-quality names being found, but does indicate what names most need attention.

Initial validation attempts by relating it to other metrics show that a combination of both models seems to catch a lot of names that need attention, but the number of false positives and negatives is still significant. Validating the quality of a name is a difficult task, but our current approach is a step in a promising direction.

### 7.1 Next steps

As explained above the machine learning model is not always predicting high-quality names. This is an assumption the quality assessment is based on, and vital for the performance. To overcome this and to answer RQ3: *"Can we improve the semantic model using the syntactic knowledge."* we will try to infuse the neural model with rule-based knowledge.

Rules in combination with a deep learning model show results that exceed using only either one [Vil+11; GAS16]. We could use our ruleset to instead of just checking the name beforehand, allow it to influence the training of the neural model. The model would then adjusts its prediction based on the guidelines and optimise towards high-quality names as well; instead of just predicting the right name. This could help the predicting of low-quality names that itself violate the guidelines set by literature. This deeper integration of the guidelines would theoretically decrease the false positives from the neural model, and answer RQ3. This will be our first step towards attempting to improve the models capability of distinguishing between good and flawed method names. Uniquely this would combine literature-based approaches with deep learning approaches to assess the quality of names used in a project.

## References


[Abe+09] Surafel Lemma Abebe et al. "Lexicon bad smells in software". In: *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE. 2009, pp. 95–99.

[All+14] Miltiadis Allamanis et al. "Learning natural coding conventions". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 281–293.

[All+15] Miltiadis Allamanis et al. "Suggesting Accurate Method and Class Names". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 38–49. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786849. URL: http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/2786805.2786849.

[All+18] Miltiadis Allamanis et al. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 81.

[Alo+19] Uri Alon et al. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 40.

[ALY18] Uri Alon, Omer Levy, and Eran Yahav. "code2seq: Generating sequences from structured representations of code". In: *arXiv preprint arXiv:1808.01400* (2018).

[APS16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. "A convolutional attention network for extreme summarization of source code". In: *International Conference on Machine Learning*. 2016, pp. 2091–2100.

[Arn+10] Venera Arnaoudova et al. “Physical and conceptual identifier dispersion: Measures and relation to fault proneness”. In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE. 2010, pp. 1–5.

[AT10] Surafel Lemma Abebe and Paolo Tonella. “Natural language parsing of program element names for concept extraction”. In: *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE. 2010, pp. 156–159.

[AT13] Surafel Lemma Abebe and Paolo Tonella. “Automated identifier completion and replacement”. In: *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE. 2013, pp. 263–272.

[But+09] Simon Butler et al. “Relating identifier naming flaws and code quality: An empirical study”. In: *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE. 2009, pp. 31–35.

[But+10] Simon Butler et al. “Exploring the influence of identifier names on code quality: An empirical study”. In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE. 2010, pp. 156–165.

[But+11] Simon Butler et al. “Mining java class naming conventions”. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE. 2011, pp. 93–102.

[BW10] Raymond PL Buse and Westley R Weimer. “Learning a metric for code readability”. In: *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 546–558.

[CT00] Bruno Caprile and Paolo Tonella. “Restructuring Program Identifier Names.” In: *icsm*. 2000, pp. 97–107.

[CT99] C Caprile and Paolo Tonella. “Nomen est omen: Analyzing the language of function identifiers”. In: *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. IEEE. 1999, pp. 112–122.

[DDO11] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. “Improving source code lexicon via traceability and information retrieval”. In: *IEEE Transactions on Software Engineering* 37.2 (2011), pp. 205–227.

[Dor12] Jonathan Dorn. “A general software readability model”. In: *MCS Thesis available from (http://www. cs. virginia. edu/˜ weimer/students/dorn-mcs-paper. pdf)* (2012).

[DP06] Florian Deissenboeck and Markus Pizka. “Concise and consistent naming”. In: *Software Quality Journal* 14.3 (2006), pp. 261–282.

[Esh+11] Laleh M Eshkevari et al. “An exploratory study of identifier renamings”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 33–42.

[GAS16] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. “Towards deep symbolic reinforcement learning”. In: *arXiv preprint arXiv:1609.05518* (2016).

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. “A practical model for measuring maintainability”. In: *null*. IEEE. 2007, pp. 30–39.

[HØ09] Einar W Høst and Bjarte M Østvold. “Debugging method names”. In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 294–317.

[ISO10] ISO/IEC. *ISO/IEC 25010 System and software quality models*. Tech. rep. 2010.

[Law+06] D. Lawrie et al. “What's in a Name? A Study of Identifiers”. In: *14th IEEE International Conference on Program Comprehension (ICPC'06)*. June 2006, pp. 3–12. DOI: 10.1109/ICPC.2006.51.

[LFB06] Dawn Lawrie, Henry Feild, and David Binkley. “Syntactic identifier conciseness and consistency”. In: *null*. IEEE. 2006, pp. 139–148.

[LFB07] Dawn Lawrie, Henry Feild, and David Binkley. “Quantifying identifier quality: an analysis of trends”. In: *Empirical Software Engineering* 12.4 (2007), pp. 359–388.

[Liu+19] Kui Liu et al. “Learning to Sport and Refactor Inconsistent Method Names”. In: *41st ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE. 2019.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.

[Mik+13] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[MPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. "How we refactor, and how we know it". In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 5–18.

[Pen+15] Hao Peng et al. "Building program vector representations for deep learning". In: *International Conference on Knowledge Science, Engineering and Management*. Springer. 2015, pp. 547–553.

[PHD11] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. "A simpler model of software readability". In: *Proceedings of the 8th working conference on mining software repositories*. ACM. 2011, pp. 73–82.

[Rel04] Phillip Anthony Relf. "Achieving software quality through source code readability". In: *Quality Contract Manufacturing LLC* (2004).

[Sca+16] Simone Scalabrino et al. "Improving code readability models with textual features". In: *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE. 2016, pp. 1–10.

[Sca+18] Simone Scalabrino et al. "A comprehensive model for code readability". In: *Journal of Software: Evolution and Process* 30.6 (2018), e1958.

[Sri+10] Giriprasad Sridhara et al. "Towards automatically generating summary comments for java methods". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 43–52.

[Vil+11] Julio Villena-Román et al. "Hybrid approach combining machine learning and a rule-based expert system for text categorization". In: *Twenty-Fourth International FLAIRS Conference*. 2011.