CHALMERS
UNIVERSITY OF TECHNOLOGY

The Effects of Identifier Characteristics on Software Product Quality

Master's thesis in Information Technology

HENRIK BOSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

The Effects of Identifier Characteristics on Software Product Quality

HENRIK BOSTRÖM

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

The Effects of Identifier Characteristics Software Product Quality
HENRIK BOSTRÖM

Supervisor: Miroslaw Staron, Dept. of Computer Science and Engineering
Examiner: Regina Hebig, Dept. of Computer Science and Engineering

Master's Thesis 2020
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

The Effects of Identifier Characteristics on Software Product Quality
HENRIK BOSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Background: Software projects consist of many identifiers, and how those identifiers should be named to improve software product quality is an opinionated subject. Some studies have tried to find the relation between identifiers characteristics and software product quality, but these studies have been conducted on small sets of software projects ($N \leq 12$). Consequently, a need exists of large scale studies that investigate how identifier characteristics effect and can be used as predictors of software product quality.
Aim: This study evaluates if source file averages of the identifier characteristics length, number of containing words, number of containing digits, and casing consistency can predict the software product quality of source files.
Method: With identifier and software product quality data found in 60,315 source files from 1,000 open-source Java software repositories, linear regression models are fitted on the data. The models can then be evaluated in terms of accuracy and coefficient importance to find if identifier characteristics could be used as predictors of software product quality.
Results: Bayesian linear regression models with identifier characteristics and the size of source files as independent variables had R-squared accuracies ranging from 0.008-0.545 of predicting the software product quality of source files. But, none of the models' identifier characteristics were accepted as important for the change of the prediction based on the ROPE + HDI decision rule with a [-0.1,0.1] ROPE interval.
Conclusions: The identifier characteristics investigated in the study could not be used to predict the study's measurement of software product quality. Further work is needed in investigating the same identifier characteristics relation to other measures of software product quality and with source files written in other programming languages than Java.

Keywords: Identifier characteristics, software product quality, empirical software engineering.

Contents

List of Figures

List of Tables

1
Introduction

An identifier is a sequence of characters that identifies source code entities such as classes or local variables in a program [1]. A majority of developers think that code comprehension is a serious problem [2] and since about 70% of all code consists of identifiers [3] it shows that identifier naming is an important research area. Previous studies have shown that breaking naming conventions rules, e.g., that an identifier should not be longer than 25 characters, negatively correlates with different readability, maintainability, and complexity quality measurements [4]. It has also been shown that identifier characteristics, such as the length of an identifier, can be used to predict fault-prone modules [5], [6] and readability measurements [6].

In current research [4]–[6] on identifiers relation to software product quality, only a small number (N$\leq$12) of software projects have been analyzed, which makes the generalizability of those studies' results questionable. Consequently, the relation between identifier characteristics and software product quality needs to be examined further by using large samples of software projects. In this study, we sample a large and diverse set of software projects to provide generalizable results of how identifier characteristics can be used to predict software product quality.

The following is the research question of this study: **R1:** *Can identifier characteristics be used as predictors of software product quality?* Software product quality of different characteristics is measured by counting the number of code rule violations found by a static code analyzer on a source file level. The analyzed identifier characteristic measurements are source file averages of the identifier characteristics length, number of containing words, number of containing digits, and casing type consistency. These identifier characteristics are analyzed because of their prevalence in previous studies [4]–[6]. Source files are sampled from 1,000 Java open-source repositories to find if identifier characteristics can be population-level predictors of software product quality.

The thesis contributes with (if found) identifier characteristic predictors that can be used by **practitioners** to identify the software product quality of software projects. During the study, two open-source tools for repository sampling and software project identifier extraction are developed that can be used by **researchers** in future empirical studies about source code identifiers.

The following is the outline of the thesis: **Chapter 2:** Background of identifier characteristics, internal measures of software product quality, related work. **Chapter 3:** Case study design including the objective, subjects, data collection procedures, and data analysis procedures. **Chapter 4:** Descriptive statistics of found subjects and variables, correlations among independent variables, diagnosis of models, and evaluation of models. **Chapter 5:** Discussions of the results and its validity. **Chapter**

6: Conclusion of results and future work proposals.

2

Background

This chapter presents what an identifier characteristic is, how software product quality can be measured, PMD as internal measurement of software quality, and finally related work.

2.1 Identifier Characteristics

An identifier is a string that identifies a source code entity [1]. In the Java language, the available entity types are package, class, interface, member, parameter, and local variable [7]. In this study, we define an identifier characteristic as a measurable identifier property that can be seen as a binary, categorical, or numerical variable. Examples: the length of an identifier, if an identifier breaks a naming convention or the entity an identifier identifies.

2.2 Measurements of Software Product Quality

Quality measurements quantify software properties that can indicate different quality characteristics of a software product [8]. An internal measurement address the internal and typically static properties of a software product [8]. For example, an internal measurement could be the number of lines of code or the number of code rule violations found in a source file. An external measurement address the external and often dynamic properties of a software product [8]. For example, an external measurement could be the number of exceptions thrown by a program over a period of time or the ratio of functional requirements that a software product satisfies.

Analyzing the external measurements of a software product often requires building and executing the product, which can be time-consuming if done on a large scale. To analyze a large set of projects within the time frame of this study, we only look at internal measurements of software product quality as they can be found in automated ways without running and monitoring any software projects.

2.3 PMD as an Internal Measurement of Software Product Quality

In this study, we use a static code analyzer to find an internal measurement of software product quality. FindBugs[1] is a static code analyzer that has been used in other related studies [4], [6] but the issue with FindBugs is that all files have to be compiled before they can be analyzed, which is not suited for a large scale study. An alternative to FindBugs is PMD[2] that is a static code analyzer that can analyze Java source files without compiling them, which suits this study. PMD has been used extensively in other studies related to software maintenance [9] and should be reliable enough for this study. PMD has many rules divided into rulesets[3] of different categories. To make sure that PMD has code rules that can be related to a variety of software product quality characteristics, we list all of PMD's rulesets, example rules, and what software product quality characteristic from [8] that they can be linked to:

- The ruleset "Code Style" enforces the PMD coding style. The ruleset detects broken naming conventions, casing conventions, and other conventions related to code style. If a development team conforms to the PMD coding style, it could become easier to reuse, for example, methods in multiple classes as every maintainer is familiar with the code style. Ease of reusing software modules can be linked to the **reusability** quality subcharacteristic from [8].
- The ruleset "Best Practices" enforces miscellaneous coding practices approved by the PMD authors. Example rules:
  - Abstract classes should have at least one abstract method.
  - Parameters should not be reassigned.
  - For each loops should be used whenever they can replace a normal for loop.

  In this study, these rules are interpreted as stylistic and only helpful for coding consistency purposes. The ruleset can be linked to the quality subcharacteristic **reusability** from [8] for the same reasons as the "Code Style" ruleset.
- The ruleset "Documentation" has rules that check for offensive comments, comments that are considered too long, and missing comments for classes, fields, public/protected methods, and enums. These rules can be linked to the **learnability** quality subcharacteristic from [8] as more and clear comments can make it easier to learn a codebase.
- The ruleset "Design" has max limits of the following complexity, coupling and size metrics.
  - Cyclomatic complexity [10]: a metric of complexity based on the number of paths found in a method [10].
  - NPath [11]: an extension of the McCabe's cyclomatic complexity metric that also takes the path depth and the control flow structures into consideration when evaluating the complexity of a function [11].

[1]FindBugs http://findbugs.sourceforge.net/
[2]PMD `https://pmd.github.io/`
[3]PMD rulesets `https://pmd.github.io/latest/pmd_rules_java.html`

  - God Class [12]: a metric based on a combination of complexity, size, and coupling metrics that are used to find god classes [12]. A god class is a class that does too many things and that have too many external references [12].
  - Size metrics: number of methods, fields, and imports for one class.

  If a software module is complex, large, and is coupled to many other software modules, it could be an indication that a software module should be split into several smaller, less complicated, and more maintainable modules. Since these rules encourage modular methods/classes, they can be linked to the **modularity** quality subcharacteristic from [8].
  This ruleset also have rules for other code design related practices. Two examples:
  - Do not use exceptions for control flow. Exceptions are computationally expensive and avoiding them can improve the execution time performance of an application.
  - Simplify boolean expressions; for example, `return isFoo == true` could be changed to `return isFoo` to avoid the comparison computation `isFoo == true`, before returning.

  These type of rules from this ruleset can be linked to the **time behaviour** quality subcharacteristic from [8] since they can improve the execution time performance of a program.
- The ruleset "Multithreading" enforces coding practices that could be preferred in projects that utilize multiple threads. Example rules:
  - Use the concurrent hashmap instead of the sequential hashmap whenever possible, as it could improve the performance in some cases.
  - Do not use the `volatile` (atomic access) keyword because it can be misunderstood and requires an understanding of parallel programming.
  - Use synchronized blocks (`synchronized(this){...}`) instead of synchronized methods (`synchronized void foo() {...}`) because a developer who add new statements to a method might miss that the statements will be executed sequentially.

  These rules could improve the execution time or help developers to avoid unwanted behavior in a multithreaded environment, and therefore, these rules can be linked to the **performance efficiency** characteristic and the **fault tolerance** subcharacteristic from [8].
- The ruleset "Performance" is related to resource allocation and execution time optimization. Example rules:
  - The `StringBuilder` class should be used instead of string concatenation.
  - Do not initialize primitive types with their default value.
  - Presize new `StringBuffer`s e.g. `new StringBuffer(20);` to avoid the buffer from being resized at runtime.

  These rules can be linked to the subcharacteristics **resource utilization** and **time behaviour** from [8] as the rules could reduce the allocated space on the stack/heap and decrease the execution time of a program.
- The ruleset "Error Prone" has rules against code prone to compiler or runtime errors. Some of the rules are:

  - Do not use reserved words as identifier names, such as, `int enum = 0;` or `double assert = 1.0;`. These variables cause compiler errors in newer version of Java [7].
  - Do not use `instanceof` in an exception catch clause because it indicates that the catch clause catches a general exception, e.g., `Throwable` and that the derived exceptions are differentiated and handled with the help of the `instanceof` keyword. The preferred approach is to have multiple exception catch clauses for every specific exception type since the compiler makes sure that all exception types have a responding catch clause.
  - Do not check if an object is null followed by using the object in any way. For example, do not write `if (str != null || !str.equals(""))` because if `str` is null, then the second statement will cause an null pointer exception.

  These rules can be related to the **fault tolerance** subcharacteristic from [8] since exceptions or compiler errors can be avoided if the rules are followed.
- The ruleset "Security" has two rules that ensure that cryptographic keys are not hardcoded. An alternative, and in certain situations, a safer way of storing keys, is to load the keys by a configuration file. These rules can be linked to the **confidentiality**[8] subcharacteristic as the keys might be confidential and would be accessible to anyone who has access to the code if they are hardcoded.

As seen in the list above, PMD is related to many software quality characteristics and should be suitable as an internal measurement of software product quality.

2.4 Related Work

Naming flaws and software product code quality. Butler, Wermelinger, Yu, *et al.* found multiple statistically significant correlations between identifier naming flaws and measurement of internal code quality in eight different open-source software projects [4]. Some of the naming flaws were:

- identifiers shorter than eight characters or longer than 25 characters
- identifiers composed of numeric words or numbers
- incorrectly capitalized identifiers

Several significant correlations were found between naming flaws and the internal software product quality measurements, such as, cyclomatic complexity [10], a readability index [6], and a maintainability index [13]. The correlations between naming flaws and the warnings from the static code analyzer Findbugs[4] were project dependant and less significant.

Readability metric and software product code quality. Buse and Weimer trained a readability classifier on human annotations that, with a precision of 75-80%, could classify code snippets as less or more readable [6]. The code snippet features included average identifiers length that had predictive weight around zero and max length that had a predictive weight around 0.4. The readability classifier was also used to try and predict FindBugs warnings with an average F1-score of 0.63 on 12 projects [6].

[4]FindBugs `http://findbugs.sourceforge.net/`

Identifiers length and software product code quality. Kawamoto and Mizuno used identifiers length as a features in a machine learning classifier to predict if classes were faulty or not based on the SZZ algorithm that can find version history changes that are prone to needing a fix [14]. The classifier was tested on two software projects, the NetBeans IDE that resulted in an accuracy of 0.895, and the Eclipse IDE that resulted in an accuracy of 0.855 [5].

Identifiers and comprehension. Lawrie, Morrell, Feild, *et al.* found through an experiment that comprehension is improved when identifiers are spelled out and consists of full words [15]. It was also found that comprehension of abbreviations was almost as high as normal identifiers [15]. Sharif and Maletic experimented with eye trackers and found that camel-cased identifiers take about 20% longer time to comprehend compared to underscored variables.

Naming conventions conformity. Naturalize is a framework that can classify a project's coding conventions and suggest changes to code in a project that does not conform to those conventions [17]. The framework was evaluated against open source projects by submitting changes proposed by Naturalize to see if the owners of the projects would accept those changes [17]. Out of 18 proposed changes to five open source projects, 14 changes were approved [17]. The study shows how critical coding conventions, which includes identifier naming conventions, are for developers when they perceive the quality of a software module.

Small software project sample sizes. [4]–[6] all analyzed the relationship between identifier characteristics and internal code quality but on a small set of software projects ($N \leq 12$). No studies were found that analyzed identifiers' effect on software quality on a larger scale.

Similar software projects. [4]–[6] all analyzed well known open-source software projects. [4] and [6] even analyzed four of the same projects, but the version might have differed slightly.

In relation to this study. There is a clear research gap of studies related to identifiers and software product quality that analyze software projects on a large scale. In this study we will look at many more software projects in comparison to previous studies in order to find more generalizable results. When we decide on identifier characteristics to analyze, we want them to be comparable to previous studies. For that sake, we will analyze some identifier characteristics related to identifier's length as seen in [4]–[6], identifier's number of words/digits as seen in [4], [15] and identifier's casing types as in [4], [16].

3
Method

This chapter presents the case study's design based on the guidelines described by Runeson and Höst in [18]. The design includes the study's objective, research question, data collection procedures, and data analysis procedures.

3.1 Objective and Research Question

The objective is to analyze if identifier characteristics can be used to predict the software product quality of source files from the viewpoint of developers in the context of open-sourced Java software projects. The goal is achieved by answering the following research question. **R1:** *Can identifier characteristics be used as predictors of software product quality?*

3.2 Subjects

Software repositories are sampled from the software development platform GitHub[1] that hosts over 745,448[2] public Java repositories. When repositories are sampled, it could be advantageous for a more representative result to divide the repository population into subgroups based on a repository characteristic and then draw random samples from each subgroup. This type of sampling is called stratified sampling [19]. The idea behind stratified sampling in the context of this study is that how identifiers are named in repositories, and the software product quality of repositories can vary depending on different repository characteristics, such as the popularity of a repository. If only random samples are picked from the population, and the sample size is relatively small, then there is an inherent risk that repositories with the exact same characteristics are drawn, which could lead to unvaried observations and ungeneralizable results.
In this study, we utilize stratified sampling by dividing the population of GitHub repositories into subgroups based on the repositories' number of stars/watchers[3]. The stars/watchers characteristic has been shown in [21] to identify both repository popularity and if individuals or organizations own the repositories. The repository population is divided into four subgroups of different ranges of repository

[1]GitHub `https://github.com/`

[2]Based on GHTorrent's GitHub data dump from 2019-06-01. Find data dump at `https://ghtorrent.org/downloads.html`

[3]The stars and watchers ratings are interchangeable because GHTorrent changed its database schema during its development [20]. On GitHub, stars and watchers are two different ratings.

Range	Subgroup Size	Sample Size
$[1, 1]$	465,905	625
$[1, 2]$	93,181	125
$[2, 5]$	93,181	125
$[5, 81817]$	93,181	125
Total	745,448	1000

Table 3.1: The table shows the stars/watchers ranges, subgroup sizes, and the number of samples drawn from each subgroup, for the sampling of GitHub repositories.

stars/watchers. From each subgroup, samples are drawn randomly, and in total, N=1,000 samples are drawn from all subgroups. The total sample size could be higher, but downloading and working with vast amounts of repositories takes time, and the sample size's upper limit was set so the study could be completed within its time frame. To decide the number of samples drawn from each subgroup we use the proportional allocation to strata (subgroups) procedure from [22]. Proportional allocation means that the number of samples drawn from a subgroup should be in proportion of the subgroup's size divided by the total size of all subgroups [22]. For example, the star/watcher subgroup with the range $[2, 5]$ stars/watchers has a subgroup size of $93,181$, and since the population size is $745,448$, the sample size for that subgroup becomes $\frac{93,181}{745,448} \cdot 1000 = 125$. All subgroup star/watcher ranges, their subgroup sizes, and sample sizes can be seen in Table 3.1.

Source files are found in each sampled repository and are used for the analysis of identifier characteristics ability to predict software product quality. In this study, we analyze source files with Bayesian linear regression models, which are computationally heavy, and thus limiting the number of source files that we use to fit those models. Thus, we reduce the number of source files by sampling them from the sampled repositories. Once again, we use the randomized stratified sampling procedure from [19], where the subgroups, in this case, are source files that belong to the same software repository. In addition, we also want to save some source files to be used for testing the accuracy of the models. We define the sampled source files that are used to fit the models as the fit dataset and the sampled source files that are used to test the accuracy of the models as the test dataset. For each repository, 45% of all source files are randomly sampled. Then 70% of those source files are added to the fit dataset and 30% to the test dataset. We use percentages instead of numeric sample sizes since we do not know about the number of source files in each repository before we sample the repositories. In the results, we present the actual number of source files in the fit- and test datasets that will most likely not have a perfect 30/70 ratio as rounding of sample sizes will frequently occur for each repository.

3.3 Collection of Subjects

Software repositories. During the study, a Java CLI tool named ghtorrent-sampler[4] was created to help automate the randomized stratified sampling of software repositories from GitHub[5].
ghtorrent-sampler is built around a database of GitHub metadata provided by the GHTorrent project [20]. The GHTorrent database was created because the GitHub API[6] is rate limited (5,000 HTTP requests/hour) and HTTP requests of resource lists (users, repositories, pull-requests, issues, etc.) are size limited [20]. The limitations make fetching of any large list of resources from the GitHub API, for example, a list of all Java software repository names, a time-consuming task. GHTorrent works around the limitations by listening to GitHub's event streams that provide information about newly updated resources in real-time, and stores all updated resources in an SQL database [20].
GHTorrent enables ghtorrent-sampler to find all GitHub Java repositories, their number of stars/watchers, and their GitHub API URLs, in one single query to the metadata database. ghtorrent-sampler then use all information about the queried repositories to perform the previously explained sampling procedure and outputs all sampled repositories clone URLs to a text file. The text file can then be used in combination with a Git[7] CLI to download every sampled Java repository.
Source files. java-name-and-quality-miner[8] (JNQM) is another Java CLI tool that was developed during the study. JNQM's purpose is to analyze source files (explained in Section 3.4.2) and sample source files from sampled repositories. For the sampling of source files, JNQM counts the number of analyzed source files found in one software repository's folder, and then use that information to sample source files by the previously described sampling procedure. The resulting samples of analyzed source files are outputted into a CSV-file where values in the column "Usage" differentiates fit versus test samples (see Table 3.2).

3.3.1 Replication

No datasets or repositories will be open for the public to avoid privacy complaints and future work. If someone wants to find the exact same data, then ghtorrent-sampler has a seed argument that can be set to 1257 to find the exact same repositories as in this study.

3.4 Data Analysis

This section begins by defining the metrics relevant for the analysis and how they are evaluated, followed by descriptions of the study's variables, Bayesian linear models, and model evaluation procedures that are used to reach a conclusion **R1**.

[4]ghtorrent-sampler `https://github.com/hb-p/ghtorrent-sampler`
[5]GitHub `https://github.com/`
[6]GitHub API `https://developer.github.com/v3/`
[7]Git `https://git-scm.com/`
[8]java-name-and-quality-miner `https://github.com/hb-p/java-name-and-quality-miner`

Repo.	File	NV	SFLOC	AL	AW	AD	ACC	Usage
0	{...}	59	248	11.179	1.692	0.0	0.956	FIT
0	{...}	13	58	10.833	1.916	0.0	1.0	FIT
0	{...}	23	93	14.111	2.333	0.0	1.0	TEST

Table 3.2: Example records of source file observations that include information about the source files measured identifier characteristics and software product quality. Repo. = Repository id. File = Source file name (has been anonymized for privacy reasons). NV = Number of PMD code rule violations. SFLOC = Source file lines of code. AL = Average identifier length. AW = Average number of words found in identifiers. AD = Average number of digits in identifiers. ACC = Average casing consistency of identifiers. Usage = If the source file belongs to the fit- or test dataset.

3.4.1 Source File Metrics

In this section, we define the metrics that we collect from sampled source files. Each metric is described, has been given a formula (if needed), and is exemplified.

3.4.1.1 Average Length (AL)

Description. The average length (AL) metric is the sum of identifier lengths in one source file divided by the source file's number of identifiers.
Formula. $AL = \frac{A}{B}$, where A = sum of identifier lengths in one source file and B = the source file's number of identifiers.
Example. A source file with the identifiers $I = \{\text{'foo'}, \text{'fooBar'}, \text{'fooBarZar'}\}$ has an $AL = \frac{3+6+9}{3} = 6$.

3.4.1.2 Average Words (AW)

Description. The average words (AW) metric is the sum of the number of words found in all identifiers in one source file divided by the source file's number of identifiers. The number of words that are found in one identifier is calculated with an algorithm that can be seen in Figure 3.1. The following is a simplified description of the word counting algorithm. First, a word count variable is set to one. Then, the identifier characters $\{c_0, ..., c_i\}$ are iterated from c_1. When a character c_k is encountered where c_{k-1} has a different character type (uppercased letter, lowercased letter, or digit), then the word count is increased by one, and the iteration of characters continues at c_{k+1}. For example, the identifier $c_{0..14}$ = {f,o,o,B,a,r,Z,A,R,t,a,r,1,2,3} has character type changes at c_3, c_6, c_9, c_{12} which results in a word count equal to five. The algorithm also looks for the underscore ('_') character that is seen as a word splitting character, meaning an underscore has a preceding word and a proceeding word. Still, the underscore character itself is not a word. For example, the identifier $c_{0..6}$ = {FOO_BAR} has the same character types, but since c_3 is equal to an underscore, the word counter is increased by one, which results in a word count of two.
Formula. $AW = \frac{A}{B}$, where A = sum of the number of words found in all identifiers

Name	Regular Expression	Example
Camel	"\b(([a-z]+([A-Z][a-z]*)+)|[a-z]2,)\d*\b" [23]	fooBar
Pascal	"\b([A-Z][a-z]+)+\d*\b" [23]	FooBar
Underline	"\b(([a-z]+(\d*)+_)+([a-z]*\d*)+)\b" [23]	foo_bar
Hungarian	"\b([gmcs]_)?(p|fn|v|h|l|b|f|dw|sz|n|d|c|ch|i|by|w|r|u)(Max|Min|Init|T|Src|Dest)?([A-Z][a-z]+)+\d*\b" [23]	cFooBar
Capital	"\b([A-Z]*(\d*)+_)*([A-Z]*\d*)+\b" [23]	FOO_BAR

Table 3.3: Names and regular expressions of casing types from [23]. The examples are identifiers that conforms to the regular expression on the same row.

in one source file based the algorithm in Figure 3.1 and B = the source file's number of identifiers.

Example. A source file with the identifiers $I = \{$'foo', 'fooBar', 'fooBAR_ZAR'$\}$ has an $AW = \frac{1+2+3}{3} = 2$.

3.4.1.3 Average Digits (AD)

Description. The average digits (AD) metric is the sum of the number of digits found in identifiers in one source file divided by the source file's number of identifiers.

Formula. $AD = \frac{A}{B}$, where A = the sum of the number of digits found in all identifiers of one source file and B = the source file's number of identifiers.

Example. A source file with the identifiers $I = \{$'foo123', 'FOO_111111'$\}$ has an $AD = \frac{3+6}{3} = 2$.

3.4.1.4 Average Casing Consistency (ACC)

Description. The average casing consistency (ACC) is the sum of the casing consistencies (CCs) for all identifier types (class, interface, annotation, annotation member, enum, enum constant, parameter, and local variable) found in one source file divided by the source file's number of CC measurements. The CC is the maximum number of identifiers of the same casing type divided by the number of identifiers of the same type. The casing types used for the CC measurement are defined in [23], and their names, regular expressions and examples of conforming identifiers can be seen in Table 3.3.

Formulas. $CC = \frac{A}{B}$ where A = the maximum number of identifiers of one type that conforms to one casing type (Camel, Pascal, Underline, Hungarian, or Capital) in one source file and B = the source file's number of unique identifier types. $ACC = \frac{C}{D}$ where C = sum of CCs of one source file and D = the source file's number of CC measurements.

Example. Let a source file have the class identifiers $CI = \{\text{FooBar}, \text{BarFoo}\}$ and field identifiers $FI = \{\text{BAR\_FOO}, \text{barFoo}\}$. The CC of CI is equal to 1.0 as both identifiers conforms to the pascal casing type. The CC of FI is equal to 0.5 as one identifier conforms to the capital casing type and one identifier conforms to the camel casing type. The CCs of the source file results in an $ACC = \frac{0.5+1.0}{2} = 0.75$.

```
int numberOfWords(String name) {
    if (name.length() <= 0)
        return 0;
    if (name.length() <= 2)
        return 1;
    char[] chars = name.toCharArray();
    int numberOfWords = 0;
    int i = 1;
    while (i < chars.length - 1) {
        ++numberOfWords;
        while ((i + 1) < chars.length - 1 &&
            !hasCharTypeChanged(chars[i], chars[i + 1]) &&
            !isSplitter(chars[i + 1]))
            ++i;
        if (isSplitter(chars[i + 1]))
            i += 2;
        else
            ++i;
    }
    return numberOfWords;
}

boolean hasCharTypeChanged(char left, char right) {
    if (Character.isDigit(left) &&
        Character.isDigit(right))
        return false;
    if (Character.isLowerCase(left) &&
        Character.isLowerCase(right))
        return false;
    if (Character.isUpperCase(left) &&
        Character.isUpperCase(left))
        return false;
    return true;
}

boolean isSplitter(char c) {
    return c == '_';
}
```

Figure 3.1: Algorithm for finding the number of words in one identifier.

3.4.1.5 Source File Lines of Code (SFLOC)

Description. In this study, source file lines of code (SFLOC) refers to the number of code lines found in one source file. All types of lines, including empty, commented, and code lines, are included in the metric.
Example. A source file with one code line, two commented lines, and three empty lines have a $SFLOC = 6$.

3.4.1.6 Number of PMD Code Rule Violations (NV)

Description. NV is the number of code rule violations found by the static code analyzer PMD[9] in one source file. The ruleset[10] used for NV includes all rules from the PMD Java ruleset[11] except:

- Rules related to identifier names as they are directly related to the identifier characteristic measurements. For example, the PMD Java ruleset has rules for the maximum/minimum length of local variables that are directly related to the AL measurement. Directly related rules are unwanted as they already define desired values of the identifier characteristics measurements, thus creating measurement biases and risk of misleading results.
- The loose package coupling rule[12] as it requires individual project level configurations that could introduce measurement biases.
- The data flow anomaly analysis rule[13] that has a known implementation issue[14] that can cause false observations of code that does not violate the rule.

The PMD code rules are related to a diverse set of software product quality characteristics. Thus, NV should be a suitable metric to measure the overall software product quality of a source file.

3.4.2 Evaluation of Metrics

java-name-and-quality-miner[15] (JNQM) that samples source files (as described in Section 3.3) also evaluates all measurements of all source file metrics. JNQM evaluates measures for all source files in one software repository at a time and then outputs the resulting measurements to a CSV-file with rows formatted as in Table 3.2.
To find identifiers from all source files, JNQM uses the Java abstract syntax tree (AST) parser JavaParser[16]. In the AST, identifiers and their types (class, interface, annotation, annotation member, enum, enum constant, parameter, and local

[9] PMD `https://pmd.github.io/`
[10] PMD ruleset used for NV `https://github.com/hb-p/java-name-and-quality-miner/blob/master/pmd-non-naming-rules.xml`
[11] PMD Java rules `https://pmd.github.io/latest/pmd_rules_java.html`
[12] Loose package coupling PMD rule `https://pmd.github.io/latest/pmd_rules_java_design.html#loosepackagecoupling`
[13] Data flow anomaly PMD rule `https://pmd.github.io/latest/pmd_rules_java_errorprone.html#dataflowanomalyanalysis`
[14] GitHub issue of the data flow anomaly PMD rule. `https://github.com/pmd/pmd/issues/873`
[15] java-name-and-quality-miner `https://github.com/hb-p/java-name-and-quality-miner`
[16] JavaParser `https://github.com/javaparser/javaparser`

variable) can be found. The resulting identifiers from each source file are used to evaluate the measurements of the identifier characteristics metrics AL, AW, AD, and ACC.
To find PMD code rule violations JNQM uses the PMD API[17] that enables running the static code analyzer programmatically on any source file. JNQM runs the PMD static code analysis on one source file at a time and, based on the number of code rule violations found, evaluates NV.
JavaParser and PMD can both cause parsing errors in cases where the Java syntax of a source file is not recognized. When parsing errors occurs, the source file that could not be parsed is not included in the CSV-file of source file measurements. JNQM keeps track of the number of parsing errors that have occurred in the analysis of all source files, and the resulting parsing error frequency is reported in the results.
All sampled software repositories need to be analyzed no matter what Java language versions they are written in to create unbiased results. Both JavaParser and PMD support AST-parsing and static analysis of source files written in the Java language versions 2-14[18], which should cover almost all Java code that can be found on GitHub. The support of almost all Java language versions by PMD and JavaParser was the reason why only Java software projects were analyzed in this study. For any other programming language, we had difficulties finding AST-parsers that were open-sourced, and that supported all of the programming language's versions.

3.4.3 Variables

The dependent variable of the study is the software product quality metric NV that was selected because of the diverse set of software quality characteristics it measures. The independent variables of the study are the identifier characteristic metrics AL, AW, AD, ACC, and the size metric SFLOC. AL and AW measure length and verboseness and can be related to how much information that identifiers hold in a source file. ACC measurements capture aspects of how consistent the naming practices are on a source file level. AD was selected based on the idea that there could be a difference between how informative letters and digits are in order to describe an identifier. As a group, AL, AW, AD, and ACC covers a wide variety of identifier characteristics and their ability to predict the dependent variable of the study NV is what this study investigates. Source file size (SFLOC) was selected as a baseline predictor that the other identifier characteristic measurements predictive ability of NV can be compared against.
In some parts of the study, it becomes valuable to standardize the variables to make them more comparable to each other. The standardization is evaluated with the following formula where x is the variable, $mean(x)$ is the mean of x and $sd(x)$ is the

[17]PMD API Documentation `https://pmd.github.io/pmd/pmd_userdocs_tools_java_api.html`

[18]JavaParser states that it supports Java language versions 1-14 on their GitHub page (`https://github.com/javaparser/javaparser`). PMD has no official documentation of what language versions it supports, but the release notes (`https://pmd.github.io/latest/pmd_release_notes_old.html`) and the source code (`https://github.com/pmd/pmd/blob/master/pmd-java/etc/grammar/Java.jjt`) show that the Java language versions 2-14 have been supported at some point in time.

standard deviation of x:

$$z = \frac{x - mean(x)}{sd(x)}$$

In the parts of the study where the variables are standardized, their names are prefixed with a 'z', for example, zAL or zAW.
The independent and dependent variables are presented with descriptive statistics and distribution visualizations in the results.

3.4.4 Model Definitions

To answer **R1**, we analyze the independent variables' ability to predict the dependent variables with the help of three Bayesian generalized linear regression models. In this section, we define the three models and reason about their likelihood functions, linear models, and priors.
Likelihood function. In this study, we have an expected large sample size of source files used to fit the linear regression models. Because of the large sample size, we expect occasional outliers that are data points that do not follow the general pattern of the evaluated source file measurements. A likelihood function that is not as affected by outliers is the Student's-t distribution function [24] $StudentT(\nu, \mu_i, \sigma)$ that becomes our likelihood function of choice.
Linear models. Now we define the linear models used for each model $M_{0...2}$:

$$\begin{aligned}
M_0 &: \mu_i = \alpha + \beta_{zSFLOC} zSFLOC_i \\
M_1 &: \mu_i = \alpha + \beta_{zACC} zACC_i + \beta_{zAD} zAD_i + \beta_{zAL} zAL_i + \beta_{zAW} zAW_i \\
M_2 &: \mu_i = \alpha + \beta_{zSFLOC} zSFLOC_i + \beta_{zACC} zACC_i + \beta_{zAD} zAD_i + \beta_{zAL} zAL_i + \\
&\quad \beta_{zAW} zAW_i
\end{aligned}$$

Each linear model has a mean μ_i, which is the mean of the Student's-t function of the dependent variable for a vector i of values of the independent variables. Each coefficient β_j represents the fixed population-level effect on the mean μ_i of an independent variable j. The intercept of the linear models α represents the dependent variable's value when the independent variables are set to zero. By using linear models, we assume that the independent variables have a constant additive effect on the dependent variable [25]. This assumption is made as our knowledge of the relationship between the independent and dependent variables is limited. Polynomial models could have been used as an alternative to linear models, but that would have increased the risk of overfitting [25].
Linear models purpose. The linear model of M_0 represents the prediction of the dependent variable if only the size of a source file is known. The linear model of M_1 represents the dependent variable's prediction if only the identifier characteristics of a source file are known. The linear model of M_2 represents the prediction of the dependent variable if both the size and the identifier characteristics of a source file are known. By comparing $M_{0..2}$, we can analyze how well a source file's software

product quality can be predicted by its identifier characteristics in relation to or in combination with the size of the source file.

Priors. As we are using Bayesian linear models, we also need to define priors for all variables' posterior distributions. For the coefficients B_j we use weakly informative priors $Normal(0, 10)$ based on recommendations from Stan's GitHub page[19] for priors of standardize coefficients. The weakly informative priors guard the models' coefficients against unreasonably large values [25]. The R-package BRMS [26] defines all other priors we use for Bayesian analysis. For the linear models' intercepts α and the likelihood function's σ, BRMS [27] sets the prior dynamically to a Student's-t function with a degree of freedom and gamma function based on the standard deviation of the dependent variable. The generated priors are supposed to be weakly informative priors [27]. For the likelihood function's ν, BRMS [27] sets the prior to $Gamma(2, 0.1)$.

Full model definitions. The full definitions of the models with likelihood functions, linear models, and priors becomes:

$$
\begin{aligned}
M_k : NV &\sim StudentT(\nu, \mu_i, \sigma) \\
log(\mu_i) &= [\text{linear model of } M_k] \\
\beta_j &= Normal(0, 10) \\
\alpha &= [\text{dynamic}] \\
g(\nu) &= Gamma(2, 0.1) \\
logm1(\sigma) &= [\text{dynamic}]
\end{aligned}
$$

M_k is any of the the three models $M_{0..2}$. The functions log, $logm1$, and g are link functions that enables the use of likelihood functions other than the normal distribution [25]. The [dynamic] priors refers to BRMS [26] dynamically defined Student-t priors.

Multicollinearity issues. One problem with the defined models $M_{1..2}$ is that their independent variables might be correlated. If the independent variables are highly correlated, it can cause multicollinearity issues that make it difficult to interpret the models [25]. To avoid multicollinearity, we conduct a collinearity test between all independent variables based on the Pearson correlation coefficient (Pearson's R). McElreath recommends Pearson's R to be less than 0.9 [25], so if we find correlations close to the recommended value, we will remove independent variables from the models until all correlations are weak. The correlation test is presented in the results.

3.4.5 Diagnosing Models

To diagnose the models, we first fit them on the fit dataset with the help of BRMS [26] that use Markov chains Monte Carlo (MCMC) [28] to find the posterior distributions of the models' variables. In this study, we use four MCMC chains with

[19] Stan's prior choice recommendations `https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations`

1,500 warm-up iterations and 3,000 normal iterations to find the posterior distributions. The chains should mix and converge for the sampled posterior distributions to be trustworthy [25], which can be diagnosed with trace plots that visualize the chains [25], the R-hat convergence metric [29], and the bulk/tail effective sample size (ESS) metrics [30]. The trace plots should have chains that look fuzzy/mixed, which indicates that they have converged and mixed [25]. The R-hat convergence metric should be less than 1.1 on all posterior distributions to indicate that the chains have converged [30]. If the bulk/tail (ESS) metric is smaller than 400 for a posterior distribution, then the posterior distribution's R-hat measurement can not be trusted [30]. All of these metrics will be verified for all models in the results.

3.4.6 Evaluation of Models

To answer the research question **R1**, we begin by looking at how accurate the models can predict the software product quality of the source files in our test dataset. The accuracies are tested with Bayes R-squared [31], which on a scale of 0.0-1.0, gives us the ratio of variance in the dependent variable that could be explained by the independent variables [31]. After the accuracies have been measured, we find which identifier characteristics in the models $M_{1..2}$ that were important for the prediction of the dependent variable with the region of practical equivalence (ROPE) + highest density interval (HDI) decision rule from [32]. The rule states that if 95% of a coefficient's HDI can be found outside the proximity of a null value (ROPE), then the null value can be rejected [32]. In this study, the null values are the hypothesis that a model's identifier characteristics have no impact on the dependent variable's prediction. ROPE limits should be defined based on the decisions audience [32], e.g., a business that reads this thesis might be interested in a very small ROPE limit as predicting a single violation could save them a lot of money. Since this study is not related to the industry in any way, we decided to use the small effect ROPE limit $[-0.1, 0.1]$ that was defined in [32]. Finally, if identifier characteristics are found to be important predictors, we report those predictors and the out-of-sample accuracies of their models.

4
Results

4.1 Descriptive Statistics

In this section, we present descriptive statistics of the collected subjects and the datasets of evaluated measurements of identifier characteristics and software product quality. The purpose is to see if the collected data seems reasonable in relation to the sampling procedures and to form an understanding of the data before we get into the details of the identifier characteristic predictions of software product quality.

4.1.1 Collected Subjects

Software repositories. The descriptive statistics of the sampled software repositories' number of stars/watchers and size in lines of code (LOC) can be seen in Table 4.1. The median of 1.0 on the repositories number of stars/watchers is in accordance with the sampling procedure of software repositories where most samples were drawn from the 1-1 stars/watchers range subgroup.
The range of 1,080 stars/watchers and the standard deviation of 62.31 stars/watchers shows that a varied amount of repositories have been sampled in terms of popularity. In addition, even if we did not take the repositories LOC into account in the sampling procedure, the standard deviation of 1,958.5 LOC and the range of 6,178,852 LOC, suggests that our sampling method worked well in terms of finding repositories of diverse sizes as well.

	Mean	SD	Median	Range
Stars/Watcher	10.05	62.31	1.0	1,080
Repository LOC	29,260.1	233,590.3	1,958.5	6,178,852

Table 4.1: The table presents the mean, standard deviation, median, and the range of the number of stars/watchers and the lines of code of the sampled repositories.

Source files. From the sampled software repositories, 188,903 (99.446%) source files were successfully parsed, and 1,058 (0.554%) caused parsing errors. Some parsing errors were expected since there is no guarantee that the source files of all repositories do not have any syntax errors.
Now to the result of the source file sampling procedure. From the population of all successfully parsed source files, 85,475 (45.248%) source files were sampled and split into a fit- and test dataset with 60,315 and 25,160 source files, respectively. The

resulting fit dataset size to test dataset size ratio became 29.436%/70.564%, which is not far from the sought after 30%/70% split ratio.

4.1.2 Independent and Dependent Variables

The variables' descriptive statistics can be seen in Table 4.2, and the distributions can be found in Figure 4.1. The following are the key observations we make of the descriptive statistics and distributions.

- AD's distribution that has a high density when AD is zero suggests that digits in identifier names are uncommon.
- At least 95% of all sampled source files have the highest ACC measurement, which means that many source files have identifiers that have been named consistently in terms of their casing type.
- SFLOC's wide distribution and high standard deviation show that source files of diverse sizes have been sampled.
- Based on the descriptive statistics of NV, less than 5% of the source files have less than two code rules violations, which means that the number of source files with no violations is small.

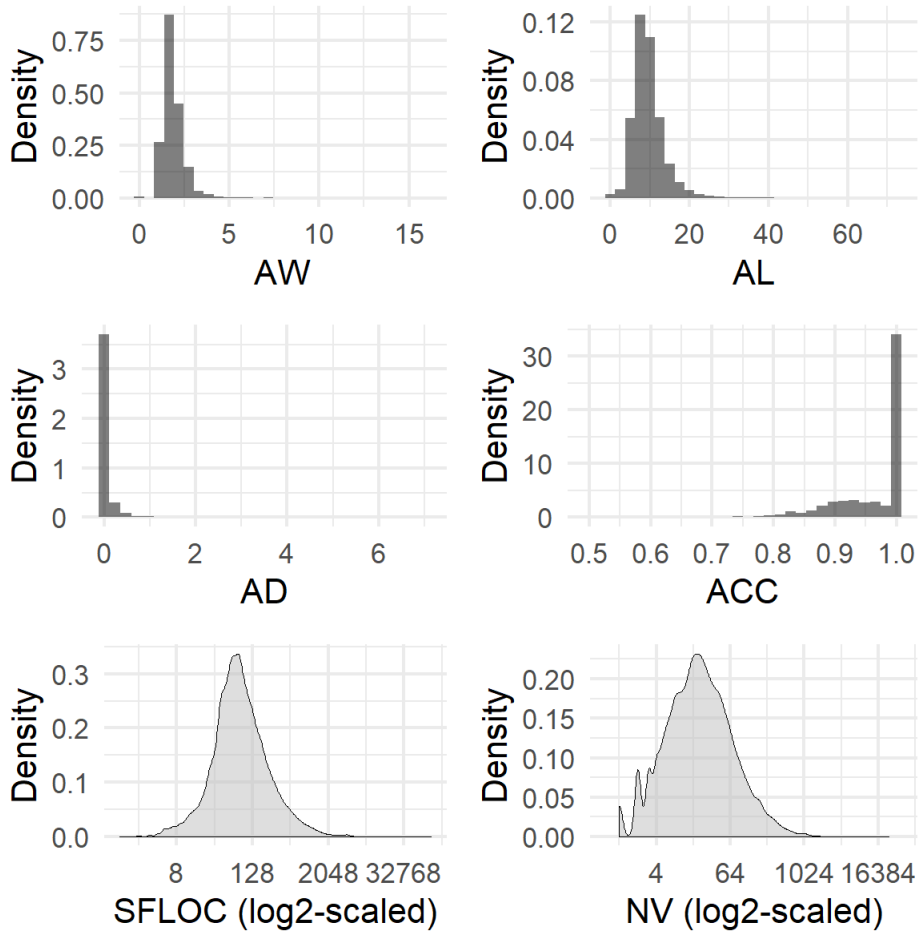

Figure 4.1: Distributions of variables AW, AD, AL, ACC, and SFLOC. The density plots of SFLOC and NV have log_2-scaled x-axes which.

Variable	Mean	SD	Median	Skewness	Kurtosis	Q5.0	Q95.0
AL	9.67	4.19	9.00	2.33	14.42	4.82	16.80
AW	1.88	0.65	1.75	3.03	25.59	1.18	3.00
AD	0.05	0.19	0.00	10.06	161.06	0.00	0.27
ACC	0.96	0.06	1.00	-1.72	3.05	0.84	1.00
SFLOC	154.07	486.44	77.00	80.78	13080.92	16	486.00
NV	46.65	144.72	19.00	66.43	9927.92	2.00	171.00

Table 4.2: The table shows descriptive statistics of collected measurements AL, AW, AD, ACC, SFLOC, and NV.

4.2 Correlations Among Independent Variables

To identify multicollinearity issues, we look at Table 4.3 that shows Pearson's R coefficients of all pairs of the study's independent variables. The correlation $corr(AL, AW) = 0.896$ is close to the recommended maximum Pearson's R coefficient of $corr(X, Y) = 0.9$ that according to [25] could cause multicollinearity problems. The problematic correlation is combated by removing AW from the linear models of M_1 and M_2:

$$M_1 : \mu_i = \alpha + \beta_{zACC} zACC_i + \beta_{zAD} zAD_i + \beta_{zAL} zAL_i$$
$$M_2 : \mu_i = \alpha + \beta_{zSFLOC} zSFLOC_i + \beta_{zACC} zACC_i + \beta_{zAD} zAD_i + \beta_{zAL} zAL_i$$

The decision to remove AW was motivated by AL being more directly related measurements from previous studies [4]–[6], [33].

Variable.	AL	AW	AD	ACC	SFLOC
AL	1.000				
AW	0.896	1.000			
AD	-0.041	0.046	1.000		
ACC	0.231	0.140	-0.065	1.000	
SFLOC	0.007	0.033	0.112	-0.138	1.000

Table 4.3: Correlation matrix of the study's independent variables.

4.3 Diagnosing Models

The sampled posteriors, the models they belong to, their R-hat diagnostic [29], and their bulk/tail effective sampling size (ESS) [30], can be seen in Table 4.4. We see that all R-hat values are less than 1.01 and that the effective sample sizes are more than 400 for all posterior distributions, which means that the chains have converged and that the posterior distributions can be trusted [30].

Model	Coefficient	Estimate	R-hat	Bulk ESS	Tail ESS
M_0	α	-0.03	1.00	5003	4413
M_0	β_{zSFLOC}	0.99	1.00	4554	3129
M_0	σ	0.05	1.00	1876	2717
M_0	ν	1.26	1.00	1875	2555
M_1	α	-0.20	1.00	5093	4707
M_1	β_{zAL}	-0.01	1.00	6997	4697
M_1	β_{zAD}	-0.00	1.00	7283	4492
M_1	β_{zACC}	-0.07	1.00	5670	4636
M_1	σ	0.07	1.00	1710	2241
M_1	ν	1.00	1.00	1552	1067
M_2	α	-0.03	1.00	2882	3424
M_2	β_{zAL}	-0.00	1.00	6826	5008
M_2	β_{zAD}	0.00	1.00	7022	4274
M_2	β_{zACC}	-0.01	1.00	5653	4483
M_2	β_{zSFLOC}	0.95	1.00	2780	3093
M_2	σ	0.05	1.00	2276	2796
M_2	ν	1.25	1.00	2083	2609

Table 4.4: The table presents the posterior distributions of the given coefficients and their estimate, R-hat, Bulk ESS and Tail ESS.

We also look at the trace plots of every sampled posterior distribution to visually verify that the chains have mixed and converged. One example trace plot can be seen in Figure 4.2 has the fuzzy/mixed pattern that is desired. All other trace plots were also fuzzy/mixed and can be found in Appendix A.

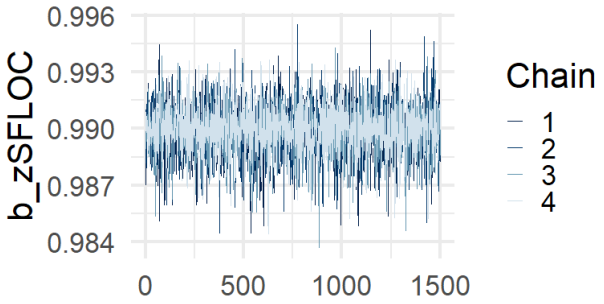

Figure 4.2: This figure shows the trace plot of the Markov chains from model M_0 for coefficient β_{zSFLOC}. The traces seems to be stable around the posterior center, and the chains look fuzzy/mixed, which indicates that the Markov chains are healthy [25].

4.4 Evaluation of Models

Now we look at how accurately the models predicted the software product quality of our test dataset of source files. The Bayes R-squared [31] measurements (see

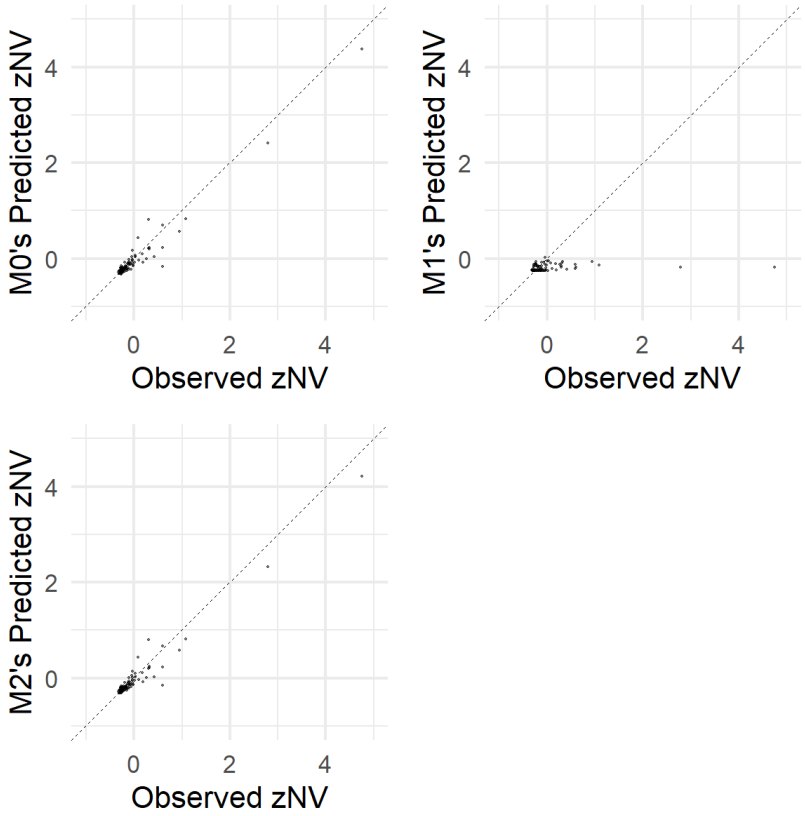

Figure 4.3: The plots show model-predicted zNVs compared to observed zNVs based on 100 sampled source files from the test dataset.

Table 4.5) show that M_0's accuracy was slightly better ($R^2_{M0} - R^2_{M2} = 0.001$) in comparison to M_2's accuracy, and that M_1's accuracy was about 68 times worse in comparison to M_0 and M_2. In other words, the models with the SFLOC independent variable had a much higher accuracy in comparison to the model with only identifier characteristics.

Model	Estimate	Q2.5	Q97.5
M_0	0.546	0.546	0.546
M_1	0.008	0.007	0.008
M_2	0.545	0.544	0.545

Table 4.5: Bayes R-square estimates and confidence interval for models $M_{0..2}$.

Before we present the ROPE + HDI decision rules results of M_1 and M_2 we will take a look at some plots where 100 random source files were sampled from the dataset and predicted by $M_{0..2}$. In the plots (see Figure 4.3) where the x-axis is the observed zNV, and the y-axis is the predicted zNV, it can be seen that M_0 and M_2 have quite similar predictions while M_1 makes worse predictions the larger zNV becomes.
Now we formalize if the identifier characteristics have any influence on the prediction of zNV with the ROPE + HDI decision rule with the ROPE interval $[-0.1, 0.1]$. The results of the decisions can be seen in Table 4.6. All identifier characteristics, both

Model	Coefficient	Decision	inside ROPE	89% HDI
M_2	β_{zAL}	Accepted	100.00 %	[0.00, 0.00]
M_2	β_{zAD}	Accepted	100.00 %	[0.00, 0.00]
M_2	β_{zACC}	Accepted	100.00 %	[-0.01,-0.01]
M_2	β_{zSFLOC}	Rejected	0.00 %	[0.95, 0.96]
M_1	β_{zAL}	Accepted	100.00 %	[-0.01, -0.00]
M_1	β_{zAD}	Accepted	100.00 %	[-0.00, -0.00]
M_1	β_{zACC}	Accepted	100.00 %	[-0.07, -0.06]

Table 4.6: The table presents the results of the HDI-ROPE decision rule [32] on the independent variables of M_1 and M_2. The decision column shows if we either reject or accept the null value range, i.e., if 95% of the HDI of an independent variable lies within our outside of the interval $[-0.1, 0.1]$

for model M_1 and M_2, were accepted not to be important predictors of zNV. The zSFLOC predictor was rejected as not an important predictor and explained why the accuracy of M_2 and M_0 was similar.

Answer to R1: The identifier characteristics AL, AD, and ACC were accepted to not be important predictors of software product quality of source files in all of their models based on the ROPE-HDI decision rule with a defined ROPE limit of [-0.1,0.1].

5
Discussion

This chapter presents inferences on the result and possible threats to the result's validity.

5.1 Inferences

Local-level or population-level predictions. The finding of this study was that the identifier characteristics AL, AD, and ACC were accepted not to be important predictors of source files' software product quality. In comparison to previous studies [4]–[6], we might have found the population-level predictions while other studies only found local-level predictions because of their small number (N$\leq$12) of analyzed software repositories.
PMD[1] analyze code on a source file level while FindBugs[2] that was used in studies [4], [6] analyze on a Java bytecode level. The type of code rule violations they find differs, and one or the other could be more affected by the study's identifier characteristics.
Linear assumption. By using linear models, we assume that the relationship between the independent variables and the dependent variable is linear. A polynomial model of some kind might have given a better fit.
Averaging. All identifier characteristics have been averaged on a source file level to be comparable by NV. When they are averaged, information about their source file level distributions are lost. Measuring software product quality on a source file level might have caused the identifier characteristics to not being able to predict NV.
Identifier type. Had the identifiers been grouped by their types, better predictions might have been found. For example, how classes are named in comparison to local variables that can be vastly different.
Bayesian models that use Monte Carlo sampling are slow by nature. A machine learning model might have been a better choice in terms of finding predictors and being able to utilize all sampled source files.

5.2 Threats to Validity

The following sections present discussions around possible threats to the validity and reliability of the study.

[1]PMD `https://pmd.github.io/`
[2]FindBugs `http://findbugs.sourceforge.net/`

5.2.1 Construct Validity

PMD[3] was used to measure software product quality. It became the tool of choice because of its wide use in other software maintenance studies [9], and because of the PMD code rules being related to many software product quality characteristics. The false-positive rate of PMD's code rule checks could not be found, which means that the result might be affected by unknown false positives, even if all code rules have been unit tested. An additional validity threat concerning PMD is that almost all available code rules were used when finding NV. If only a subset of rules had been used, some identifier characteristics might have been found to be important predictors of software product quality.
java-name-and-quality-miner's [4] accuracy of finding identifiers was evaluated manually, by writing mock files with known identifiers, and checking the found identifier were the same. The identifier characteristics measurements were also unit tested. But, there is always a risk of bugs, and faulty identifier data might have influenced the results.

5.2.2 Internal Validity

Stratified sampling of software repositories poses a selection bias threat as random samples are selected from subpopulations instead of the entire population of software repositories. This threat has not been mitigated since it increase the generalizability of the results by sampling a wide range of repositories in terms of popularity. It is a trade-off between internal- and external validity, that is well known in empirical software engineering [34].
Confounding was mitigated by comparing regression models with and without the source file size (LOC) independent variable. The purpose of the study is not to investigate the causality of software product quality but rather to find ways of predicting software product quality.

5.2.3 External Validity

The generalizability of the results was improved by investigating a large (N=1,000) and varied set of software repositories. One generalizability threat is that all software repositories are open-sourced. If industry software repositories had been analyzed as well, then the results would have been even more generalizable. Another threat is that only Java software repositories were analyzed while code in other programming languages might have given other results.

5.2.4 Reliability

As said earlier, the datasets and the collected GitHub repositories will not be made available for the public to protect personal data. But, all tools that can be used to find the sampled data have been made public, and have been configured with

[3]PMD `https://pmd.github.io/`
[4]java-name-and-quality-miner `https://github.com/hb-p/java-name-and-quality-miner`

seeding options, which makes it possible to replicate the entire study, even the random elements. If any of the GitHub repositories that have been sampled in this study are deleted from GitHub, the the study would not be 100% replicable. But, since we are using 1,000 repositories in the first place, if one or two repositories are not the same, the result would still be similar.

6
Conclusion

Software projects consist of many identifiers, and how those identifiers should be named to improve software product quality is an opinionated subject. Some studies have tried to find the relation between identifiers characteristics and software product quality, but these studies have been conducted on small sets of software projects (N$\leq$12). Consequently, a need exists of large scale studies that investigate how identifier characteristics effect and can be used as predictors of software product quality.

In this study we used identifier characteristics and software product quality data from 60,315 source files from 1,000 open-source Java software repositories to fit Bayesian linear regression models. The models were used to find if the identifier characteristics could be used as predictors of software product quality.

Bayesian linear regression models with identifier characteristics and the size of source files as independent variables had R-squared accuracies ranging from 0.008-0.545 of predicting software product quality. But, the models' identifier characteristics were accepted as not important for the change of the prediction based on the ROPE + HDI decision rule with a [-0.1,0.1] ROPE interval. So no identifier characteristics were found to be useful as predictors of software product quality.

More research is needed in investigating the same identifier characteristics relation to other measures of software product quality and with source files written in other programming languages than Java.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[2] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits", in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: Association for Computing Machinery, 2006, pp. 492–501, ISBN: 1595933751. DOI: 10.1145/1134285.1134355. [Online]. Available: https://doi.org/10.1145/1134285.1134355.

[3] F. Deissenbock and M. Pizka, "Concise and consistent naming [software system identifier naming]", St. Louis, MO, USA: IEEE, 2005, pp. 97–106, ISBN: 0-7695-2254-8. DOI: 10.1109/WPC.2005.14.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study", Madrid: IEEE, 2010, pp. 156–165, ISBN: 978-1-61284-369-8. DOI: 10.1109/CSMR.2010.27.

[5] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers", Osaka: IEEE, 2012, pp. 30–34, ISBN: 978-1-4673-4366-4. DOI: 10.1109/IWESEP.2012.15.

[6] R. P. Buse and W. R. Weimer, "A metric for software readability", in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 121–130, ISBN: 9781605580500. DOI: 10.1145/1390630.1390647. [Online]. Available: https://doi.org/10.1145/1390630.1390647.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*. Addison-Wesley Professional, 2000.

[8] ISO/IEC, "Iso/iec 25010 system and software quality models", Tech. Rep., 2010.

[9] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction", in *International Conference in Software Engineering for Defence Applications*, Springer, 2018, pp. 165–175.

[10] T. J. McCabe, "A complexity measure", *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[11] B. A. Nejmeh, "Npath: A measure of execution path complexity and its applications", *Communications of the ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[12] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[13] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and application of an automated source code maintainability index", *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997. DOI: 10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S.

[14] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?", *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005, ISSN: 0163-5948. DOI: 10.1145/1082983.1083147. [Online]. Available: https://doi.org/10.1145/1082983.1083147.

[15] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory", *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[16] B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and under_score identifier styles", in *2010 IEEE 18th International Conference on Program Comprehension*, IEEE, 2010, pp. 196–205.

[17] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions", Feb. 17, 2014. DOI: 10.1145/2635868.2635883. arXiv: 1402.4182v3 [cs.SE].

[18] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

[19] J. Neyman, "On the two different aspects of the representative method: The method of stratified sampling and the method of purposive selection", in *Breakthroughs in Statistics*, Springer, 1992, pp. 123–150.

[20] G. Gousios, "The ghtorrent dataset and tool suite", in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236, ISBN: 978-1-4673-2936-1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132.

[21] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform", *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[22] *Encyclopedia of survey research methods au - lavrakas, paul*, Thousand Oaks, California, Jun. 2008. DOI: 10.4135/9781412963947. [Online]. Available: https://doi.org/10.4135/9781412963947.

[23] Y. Wang, C. Wang, X. Li, S. Yun, and M. Song, "How are identifiers named in open source software? about popularity and consistency", *arXiv preprint arXiv:1401.5300*, 2014.

[24] A. Gelman and J. Hill, *Data analysis using regression and multilevel/hierarchical models*. New York: Cambridge University Press, 2007, vol. Analytical methods for social research, xxii, 625 p.

[25] R. McElreath, *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. CRC Press, 2016. [Online]. Available: http://xcelab.net/rm/statistical-rethinking/.

[26] P.-C. Bürkner, "brms: An R package for Bayesian multilevel models using Stan", *Journal of Statistical Software*, vol. 80, no. 1, pp. 1–28, 2017. DOI: 10.18637/jss.v080.i01.

[27] P.-C. Buerkner and M. P.-C. Buerkner, "Package 'brms'", 2016.

[28] W. K. Hastings, "Monte Carlo sampling methods using Markov chains and their applications", *Biometrika*, vol. 57, no. 1, pp. 97–109, Apr. 1970, ISSN: 0006-3444. DOI: 10.1093/biomet/57.1.97. eprint: https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf. [Online]. Available: https://doi.org/10.1093/biomet/57.1.97.

[29] A. Gelman, D. B. Rubin, *et al.*, "Inference from iterative simulation using multiple sequences", *Statistical science*, vol. 7, no. 4, pp. 457–472, 1992.

[30] A. Vehtari, A. Gelman, D. Simpson, B. Carpenter, and P.-C. Bürkner, *Rank-normalization, folding, and localization: An improved "0562R for assessing convergence of mcmc*, 2019. arXiv: 1903.08008 [stat.CO].

[31] A. Gelman, B. Goodrich, J. Gabry, and A. Vehtari, "R-squared for bayesian regression models", *The American Statistician*, vol. 73, no. 3, pp. 307–309, 2019. DOI: 10.1080/00031305.2018.1549100. eprint: https://doi.org/10.1080/00031305.2018.1549100. [Online]. Available: https://doi.org/10.1080/00031305.2018.1549100.

[32] J. K. Kruschke, "Rejecting or accepting parameter values in bayesian estimation", *Advances in Methods and Practices in Psychological Science*, vol. 1, no. 2, pp. 270–280, 2018. DOI: 10.1177/2515245918771304. eprint: https://doi.org/10.1177/2515245918771304. [Online]. Available: https://doi.org/10.1177/2515245918771304.

[33] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study", Lille: IEEE, 2009, pp. 31–35, ISBN: 978-0-7695-3867-9. DOI: 10.1109/WCRE.2009.50.

[34] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering", in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 9–19.

A

Trace Plots of Models

The following are the trace plots of the sampling of posterior distributions for models $M_{0..2}$.

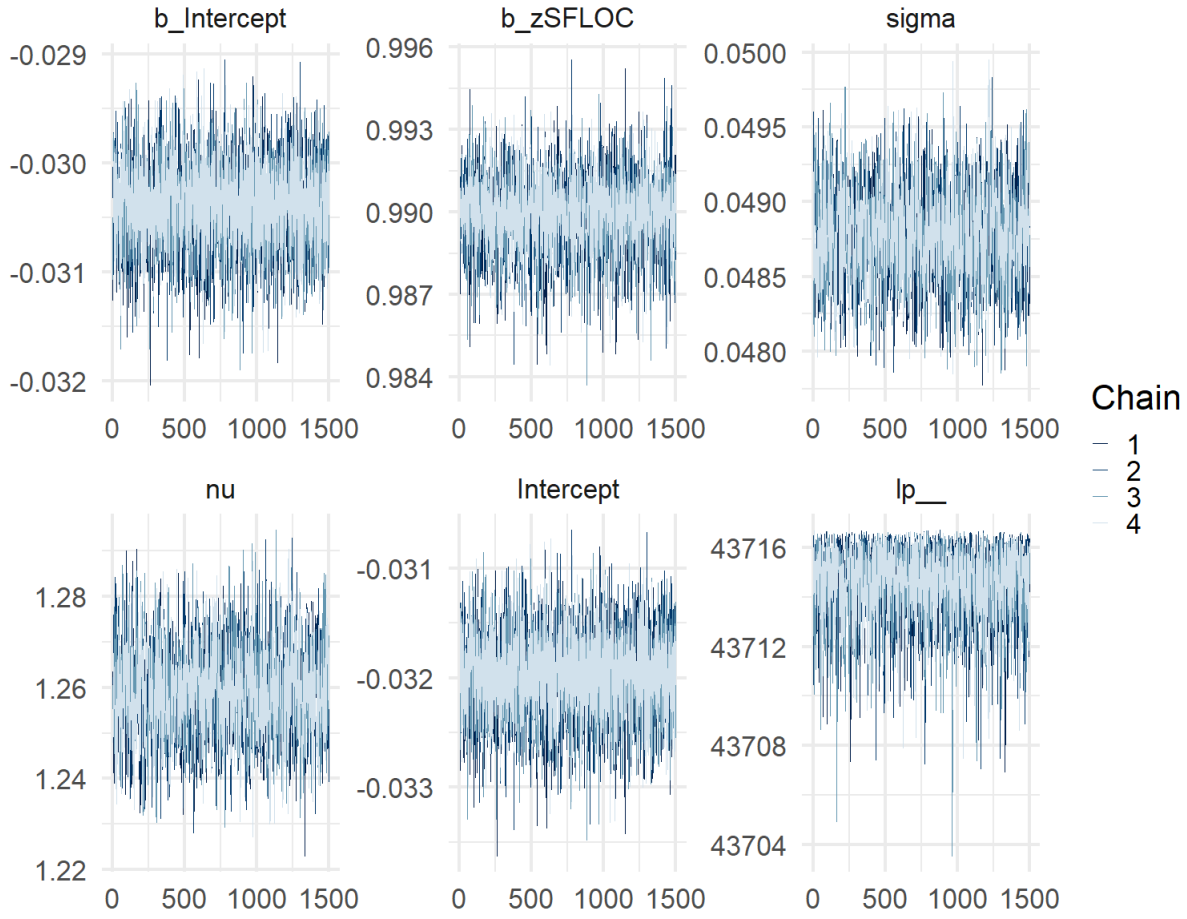

Figure A.1: This figure shows the trace plots of the Markov chains from model M_0. The traces seems to be stable around their posterior centers, and the chains look fuzzy/mixed, which indicates that the Markov chains are healthy [25].

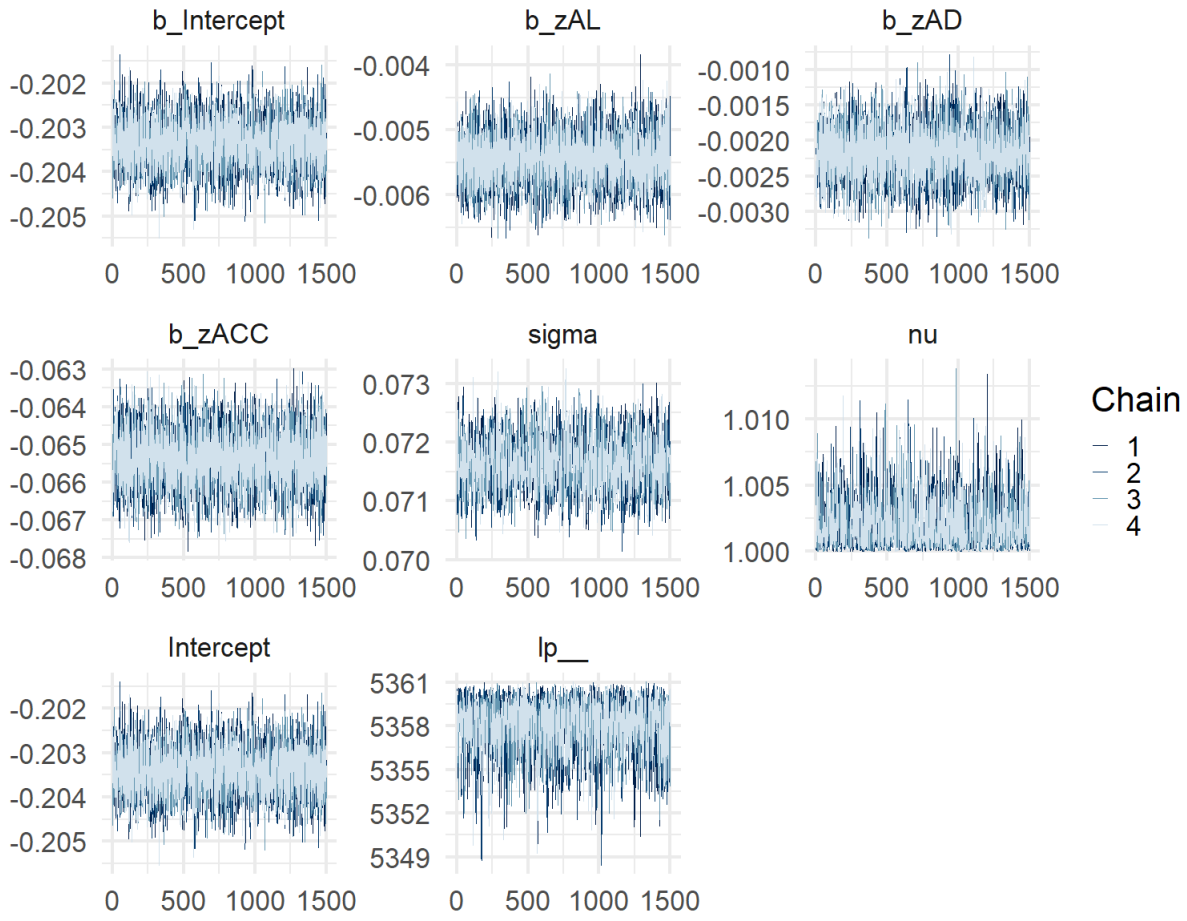

Figure A.2: This figure shows the trace plots of the Markov chains from model M_1. The traces seems to be stable around their posterior centers, and the chains look fuzzy/mixed, which indicates that the Markov chains are healthy [25].

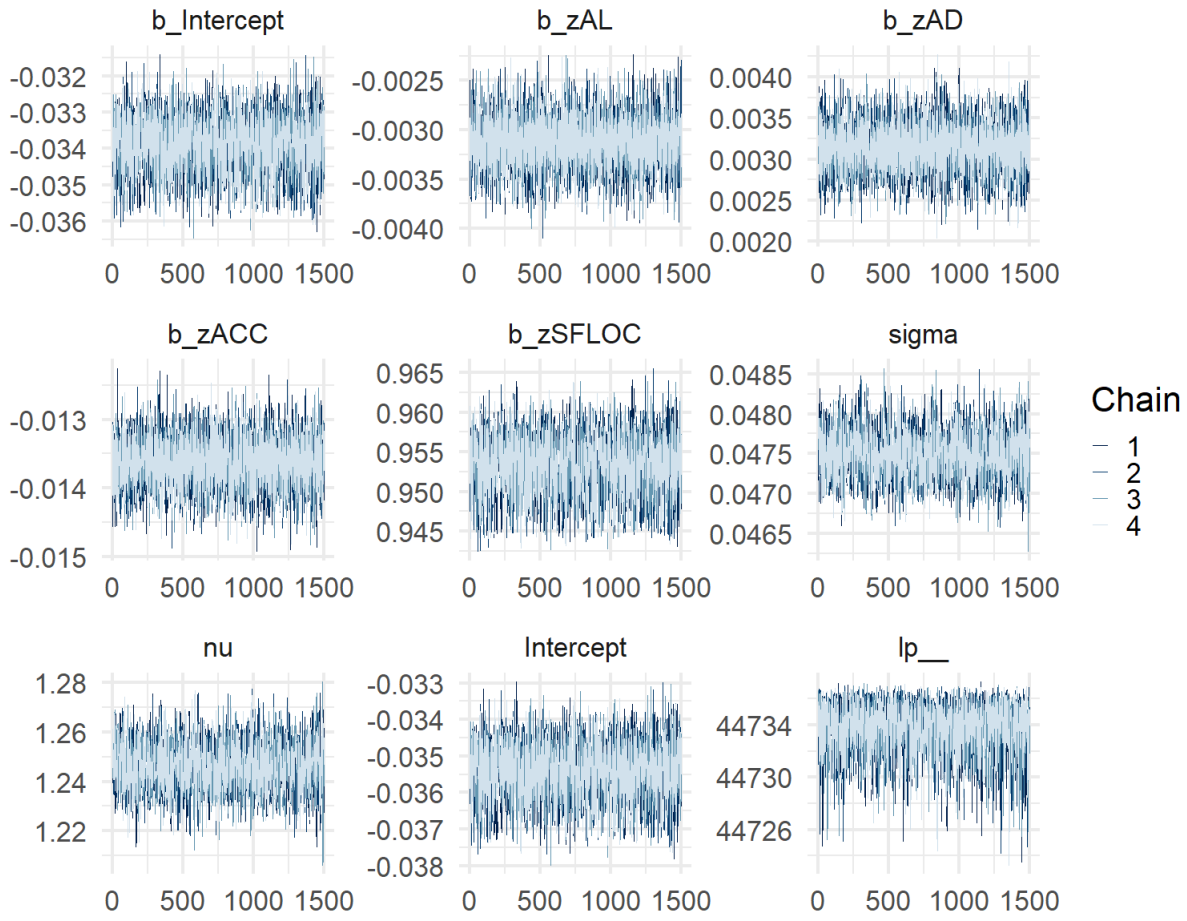

Figure A.3: This figure shows the trace plots of the Markov chains from model M_2. The traces seems to be stable around their posterior centers, and the chains look fuzzy/mixed, which indicates that the Markov chains are healthy [25].