UNIVERSITY OF MOLISE

DEPARTMENT OF BIOSCIENCES AND TERRITORY

BACHELOR'S THESIS IN COMPUTER SCIENCE

An Empirical Exploration of Deep Learning Techniques for Automated Enhancement of Code Readability in Large-Scale Software Repositories

---

Author:

Marco OMICINI

Supervisor:

Prof. Simone SCALABRINO

Dr. Antonio VITALE

Automated Software Delivery

Academic Year 2022/2023

Dedico questa tesi a tutti noi:

Cercatori di conoscenza, appassionati di computers.

Contents

Chapter 1

Introduction

1.1 Application Context

In the context of software systems evolution, it is universally recognized that adopting code maintenance practices is essential to mitigate the intrinsic technical debt in software development. This debt encompasses costs that accrue during the development process, such as writing unreadable code, lack of documentation, and the presence of duplicate, unused, or untested elements, all of which inevitably contribute to software degradation over time. Technical debt poses a challenge as it slows down the development process, leading to increased costs for companies. Over time, as Besker *et al.* [1] shows the release of new features experiences a slowdown concurrently with the increase in technical debt; As pointed out by Rios *et al.* [2] study on technical debt, this can lead to experience financial losses. To counteract this phenomenon and ensure greater software longevity, it is common practice to employ refactoring techniques. Refactoring involves modifying the source code to improve readability, maintainability, and comprehensibility. Unreadable code exhibits various characteristics, including excessively long procedures/classes, redundant or unclear comments, lack of documentation, and the use of so-called "magic numbers," along with the presence of unused code. Refactoring, therefore, plays a fundamental role in software systems evolution, and the automation of this activity represents an active area of research. The goal is to make these refactoring practices automatic since refactoring is a lengthy and meticulous activity, often difficult to execute correctly due to the need to deeply understand the source code. Readability thus emerges as a fundamental yet often overlooked aspect underlying proper refactoring and, consequently, the correct evolution of the software system.

1.2 Motivation and Objectives

Several studies have emphasized the critical role of code readability in software maintenance. While acknowledging the multifaceted and subjective nature of code readability, it is widely accepted that easily understandable code facilitates comprehension, modification, and maintenance. Some research has proposed metrics to evaluate code readability, while others have focused solely on specific aspects of automated readability improvement, such as variable naming. However, only recently a study has been proposed aiming to use machine learning models to holistically improve code readability. This cuttingedge study has spurred us to undertake this experimental thesis, as it is the first time that there is a proposal to concretely examine the potential of deep learning models in enhancing code readability within extensive projects. Our experimental thesis aims to evaluate the impact of using deep learning models on code readability and, consequently, on the overall software development experience. With this in mind, we have formulated the following research questions to guide our investigation:

RQ1: *What is the impact of applying automatic code readability enhancement considering snippets with varying levels of readability, namely low, medium, and high?*

With this question, we aim to understand how the model performs in different contexts (low, medium, high readability) and therefore determine if and especially in which context it excels in improving code readability.

RQ2: *Does readability enhancement through deep learning models change code behavior?*

Here, we are interested in understanding whether the model, by improving code readability, can alter the behavior of the code itself. This is a critical aspect because any modification to the source code could potentially introduce errors or undesired behaviors, thus mitigating the practice of refactoring inherent in the application of such a model.

1.3 Results

The results indicate that the model is capable of enhancing the readability of poorly readable and moderately readable code snippets, albeit to a limited extent, offering

improvements of up to 5% at most. Regarding highly readable methods, the model does not yield significant enhancements; in fact, in some cases, a decrease in readability is observed.

Except for a specific case study where there is no alteration of correctness, the results demonstrate that the model tends to primarily alter the behavior of code for automatic refactoring of poorly readable methods. In the case of moderately readable methods, it is observed that correctness is preserved in 33% of cases, while in the remaining 67%, such correctness is compromised. For highly readable methods, the correctness of procedures is usually preserved.

1.4 Thesis Structure and Organization

The study is structured into 6 phases (divided into chapters), each addressing different topics. In *Chapter 2*, we will explore the state-of-the-art literature, aiming to delineate the overall landscape of the problem and the currently available solutions. In *Chapter 3*, we will delve into the approach utilized to conduct this experimental study, while *Chapter 4* will discuss the case study and the various outcomes obtained. Subsequently, in *Chapter 5*, we will discuss the limitations and factors that may have influenced the validity of this study. Finally, in *Chapter 6*, we will present the conclusions and acknowledgments.

Chapter 2

Background and Related Work

2.1 Code readability

Code readability stands as a pivotal element in software maintenance, a fact supported by ample evidence in the literature. Tufano et al. [3] delineate various factors indicating when code quality deteriorates, commonly referred to as "code smells." The study shows that code smells are often introduced upon file creation or repeated modification of the same file. It is demonstrated that the file creator often introduces them, and this typically occurs months before a deadline or when the developer is under heavy workload. Sedano et al. [4] illustrate that developers are often unaware of the difficulty another developer faces in reading the code they produce, and how this awareness eventually drives developers to write more readable code, whether the initial code was readable or not. Scalabrino et al. [5] propose a model aimed at estimating code readability by improving upon previous state-of-the-art models. These improvements are based on previously unexplored textual features such as Comments and Identifiers consistency, which measures the overlap of terms used in comments with those used in the code body, and Identifier terms in dictionary, a metric aiming to identify how many of the words used belong to the English dictionary, as it has been shown that abbreviated naming is often correlated with lower readability.

Understanding these indicators is crucial, although their effectiveness is questioned by the study conducted by Fakhoury et al. [6], demonstrating that these metrics may be partially ineffective in capturing what a developer would define as an improvement in code readability. The study shows that the existing models are unable to

capture improvements in readability during software maintenance and proposes additional metrics to enhance their effectiveness. Additionally, Travares et al. [7] shed light on the intricacy of this issue, cautioning that attempts to enhance code readability through automated refactoring can inadvertently worsen the situation. Indeed, it has been shown that automated refactoring practices may introduce code smells. Another pivotal study conducted by Oliveira et al. [8] exposes some shortcomings in evaluation studies on readability conducted using human subjects, revealing that different approaches require different skills and that only some of these skills are utilized in these evaluations.

2.2 Machine learning to improve code readability

Enhancing code readability using machine learning models represents a dynamic area of research, with several studies showcasing their efficacy in automating refactoring tasks. For instance, Piantadosi [9] demonstrated how software tends to evolve and lose readability to a relevant extent as features are added and changes are made to it, and outlined guidelines such as small incremental commits and frequent refactoring practices to mitigate this issue. This observation underscores the importance of the study mentioned in the preceding paragraph, which investigates the efficacy of deep learning techniques in enhancing code readability. By addressing this issue, the study contributes to mitigating the potential loss of readability in evolving software systems.

Regarding the automation of refactoring activities, some studies have focused on automatically improving certain characteristics that describe more readable code. The study conducted by Mastropaolo et al. [10] demonstrates how deep learning models can be applied to automate variable renaming as a technique to enhance readability. Finally, Vitale et al. [11], the study that profoundly influenced our research direction, significantly advanced the field of automated refactoring. This study introduced a deep learning model capable of suggesting optimal actions (such as renaming, code extraction, styling operations, etc.) to enhance code readability. By training a neural network on a dataset comprising 122k commits, this approach effectively modified code without altering its behavior in 69% of cases and improved

readability in Java code snippets by 79.4%. These findings served as the foundation for our research, which aimed to apply this model in real-world application contexts.

Chapter 3

Empirical Study Design

In this empirical study, our aim was to evaluate the effectiveness of the-state-of-art deep learning model (referred to as the Code readability enhancement model) proposed by Vitale *et al.* [11] in improving the readability of Java code. To accomplish this, we decided to extract and evaluate the readability of code samples before and after applying the model, utilizing a tool aimed at assessing and measuring the readability of a given piece of code. The objective was to understand if and how this model could enhance readability. Therefore, we extracted a set of Java code snippets and evaluated their readability both before and after applying the model. Our analysis focused on assessing any improvements in readability brought about by the model's interventions.

Our study is steered by the following research questions:

- RQ_1: What is the impact of applying automatic code readability enhancement considering snippets with varying levels of readability, namely low, medium and high?

With this inquiry, our aim is to understand how the model performs across various scenarios presented to it. The objective is to assess in which area the model excels most prominently and whether there are significant differences between the different scenarios.

- RQ_2: Does readability enhancement through deep learning models change code behavior?

Here, we underscore the need to understand whether the model, by modifying the code, alters its behavior. This aspect is critical as it evaluates one of the essential aspects for proper refactoring practice, namely the preservation of code behavior.

3.1 Experimental Procedure

Our study is therefore based on an experimental approach aimed at answering two research questions, RQ1 and RQ2. To address these questions, we first need to obtain the data, which were collected from GitHub using its APIs. Once the data were obtained, we created the procedure to carry out the extraction phase of relevant parts (the methods of each class) and other information to generate the dataset that we will use as input for the model. This phase also serves as a foundation for the subsequent stages because it allows us to extract information about the position of the methods, the lines they are located on, and their class membership. Additionally, we use this phase to compute an initial assessment of the readability of the methods before they are modified by the model. To achieve this, we utilize the tool proposed by Scalabrino et al. [5]. During this phase, we also ensure to prepare and manipulate the methods to follow the input guidelines outlined by Vitale et al. [11], and therefore, we apply a lexer to replace some characters with their respective tokens. Subsequently, we proceed with modifying the methods using the deep learning model offered by Vitale et al. We then move on to the manual evaluation and testing phase, where we manually filter the results (discarding obvious syntactic and/or semantic errors) and recalculate the readability score. By doing this, we can answer research question RQ1.
To address RQ2, we will execute a patching phase, replacing the modified methods with the original ones, and then run the test suite of the projects under examination to understand if and how these changes have impacted the functionality of the procedure. This will allow us to determine if the behavior of the code has been altered.

As delineated above, our approach can be summarized as follows:

- **Data collection**: we collected a dataset of Java projects from GitHub;
- **Dataset creation**: we process the data in order to extract and tokenize the methods detected for our analysis and precompute the readability of those methods;
- **Model application**: we subjected the extracted methods to Vitale model in order to obtain the readability changes;

- **Manual evaluation and testing**: we manually evaluated the changes and tested the model's effectiveness in both readability and code alteration.

The pipeline of our approach is shown in Figure 3.1.

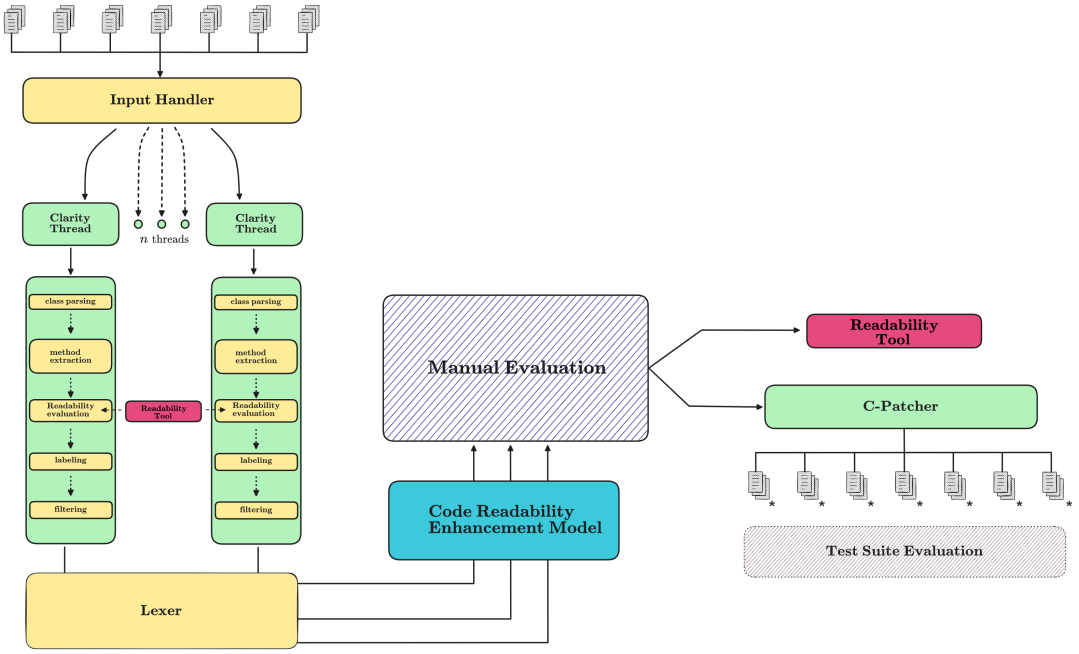

FIGURE 3.1

3.1.1 Data Collection

In this phase, we utilized software repository mining techniques to extract the data essential for our research. We carefully selected three sizable Java projects, each actively maintained and equipped with a comprehensive test case suite. The chosen projects are:

- **Guava**: an open source library from Google that provides a set of libraries useful for programming in Java;
- **Javaparser**: an open source library that allows you to analyze and manipulate Java source code;
- **Jenkins**: An open source automation system that provides support for building, testing, and deploying software.

3.1.2 Data Preprocessing and Dataset Creation

In this phase, our focus was on extracting methods from the selected projects. This approach was necessary because the insights gleaned from the Vitale et al. [11] study pertained specifically to Java code snippets, rendering reasoning at the class level impractical. To address this, we developed the "Clarity" tools, designed to extract methods from one or more Java projects, categorize them based on readability, and maintain a subset according to user-defined parameters. This tool comprises multiple components. Firstly, the Input Handler facilitates the analysis of multiple projects concurrently using multithreading techniques, primarily for optimization purposes. The Handler creates a set of threads corresponding to the number of input projects, with the user-defined numerical limit (defaulting to 4 threads) to prevent potential application crashes. Each thread executes a series of operations:

- 1. *Extract methods from java classes*
- 2. *Evaluate methods readability*
- 3. *Remove irrelevant results*
- 4. *Labeling*

Which we will describe in detail in the next paragraphs.

Methods Extraction

In this phase, we utilized the open-source tool *Javaparser* to parse the Java classes, extracting the Abstract Syntax Tree (AST) a hierarchical representation of the structure of code that abstracts away details like spacing and formatting, which facilitated the identification of methods. From the extracted methods, we preserved crucial components including the declaration, body, annotations, and comments. Subsequently, as shown in Figure 3.1 this data was encapsulated within a dummy class and saved to file for further analysis.

LISTING 3.1: Dummy Class

```
public class DummyClass {
    //comments
    @Notation
    public void method() {
    }
}
```

To circumvent the extraction of "superfluous" methods, we implemented certain exclusions during this phase. Specifically, we chose to disregard getters and setters, as well as constructors. For the former two, we employed a heuristic based on the observation that getters and setters typically consist of no more than three lines. As for constructors, we leveraged the Javaparser API to identify and exclude them from the extraction process. Although it is true that this heuristic may remove methods that are neither getters nor setters, we have still decided to use it. This decision is motivated by the fact that the likelihood of such methods containing readability errors is rather low, given the limited amount of code. Therefore, this explanation confirms the validity of the choice to exclude these methods.

Readability Evaluation

To evaluate the readability of the Java methods, we utilized an API from a readability analysis tool developed as a result of the study conducted by Scalabrino et al. [5]. This tool offers an interface for analyzing the dummy classes generated during the extraction process. Through the analysis phase, by analyzing the code and taking into account different metrics such as line length, number of blank lines, periods etc. this tool produces a readability score s such that $s \in [0, 1]$, where:

$$0 \implies \text{unreadable}$$

$$1 \implies \text{readable}$$

Once we obtained the score, we proceeded to save the necessary information for the tokenization and processing phase by the model. This data was saved in JSON

format, retaining additional information required for later stages. Figure 3.2is an example of the JSON structure:

LISTING 3.2: Method's Information


```
{
  "name": "orNull",
  "method": "\t\t@Override\n\t\t@CheckForNull\n\t\tpublic T
     orNull(){\n\t\t    return null;\n\t\t}",
  "startLine": 64,
  "endLine": 68,
  "classPath": "guava/android/guava/src/com/google/common/
     base/Absent.java",
  "readabilityScore": 0.8923128247261047,
  "label": "NONE"
}
```


Data Filtering and Labeling

In the final phase of data extraction, considering the potentially vast number of methods generated by a large project, we opted to retain only a subset. Specifically, we selected the 20% of the least readable methods, the 20% of the average readable methods, and the 20% of the most readable ones. To achieve this, we devised an algorithm that sorts the results based on the readability score and then filters out only the files meeting these percentage criteria. An example of the algorithm can be seen in Figure 3.2. As depicted, our decision was to retain not only the worst data, i.e., the least readable methods, but also the most readable and the averagely readable ones. This approach allows us to observe the model's performance across these three distinct scenarios.

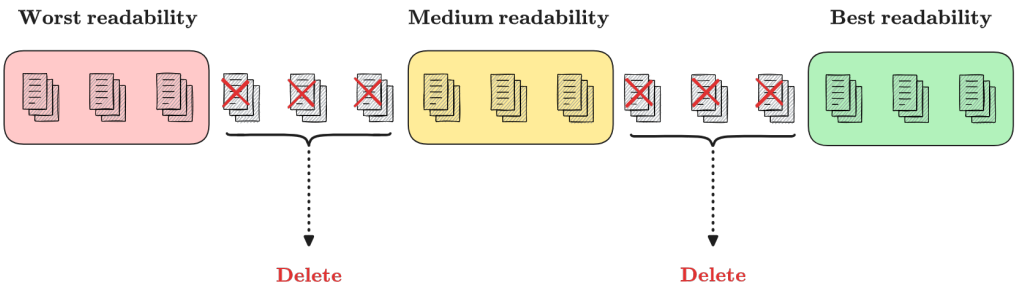

FIGURE 3.2

Tokenization

As mentioned, the model will take a series of methods as input and return an enhanced, more readable version. However, before proceeding with this, we need to pass our results through a lexer (lexical analyzer), which will enable us to divide the input text into meaningful sections known as tokens, representing the atomic units of the programming language.

An example of tokenization is illustrated in the Table shown below:

Original Method	**Tokenized Method**

`public void generate() {`
`  final int size = 0;`
`  final String alphabet = "";`
`  final String digits = "";`
`}`
	`public $whitespace$ void`
	`$whitespace$ generate ( )`
	`$whitespace$ { $newline$`
	`$indentation$ final`
	`$whitespace$ int $whitespace$`
	`size $whitespace$ =`
	`$whitespace$ $number$ ;`
	`$newline$ $indentation$ final`
	`$whitespace$ string`
	`$whitespace$ alphabet`
	`$whitespace$ = $whitespace$`
	`$string$ ; $newline$`
	`$indentation$ final`
	`$whitespace$ string`
	`$whitespace$ digits`
	`$whitespace$ = $whitespace$`
	`$string$ ; $whitespace$`

TABLE 3.1: Comparison between the original method and the tokenized method.

As evident, we eliminated extraneous characters such as spaces, newlines, and tabs, which convey text positioning but hold no significance for the language. Instead,

we replaced them with specific tokens representing their behavior. Subsequently, we apply the theory described above to the set of JSON files obtained from the preceding steps to generate the input file for the model. The Ruby program responsible for the tokenization phase will generate, at the end, the Dataset in the form of a ***TSV*** file.

3.1.3 Model Application

In this phase, we processed the data obtained in previous phases using the deep learning model provided by Vitale et al. [11]. The model was accessible via a collaborative environment called *Colab*, enabling the execution of Python instructions to configure the model and extract relevant data from the *.tsv* file. Specifically, we focused on the *tokenized_method* field, containing the tokenized method, which served as input for the model. The model processed this data and returned a new tokenized method with readability changes, which we subsequently inserted back into the *.tsv* file under a field named *model_prediction*

3.1.4 Data analysis

From this point onward, we were almost equipped to address research questions *RQ1* and *RQ2*. We initiated an analysis of the data obtained to address these questions, starting with a manual evaluation of the model's responses to validate the results and ascertain if the model had indeed altered the instances. During this manual review phase, for the methods modified by the model, we manually replaced tokens that couldn't be automatically replaced, such as **$string$** and **$number$**. This manual replacement was necessary as the output of the model is not deterministic, thus making it impossible to automatically predict which string or number corresponds to which token. Additionally, during this manual phase, we also adjusted strings that were previously in uppercase, as the model, as currently defined, only provides predictions in lowercase.

During this manual analysis, certain observations were made, and consequently, noticing a similar pattern in each modification, we decided to preserve a relatively

small subset of the analyzed methods. Indeed, the total number (again 20% of the total methods) amounted to approximately 5,000 methods per project. This number would have resulted in the need to manually analyze over 15,000 methods. To make the task more manageable, we decided to retain a sample of 100 methods per project. Additionally, to facilitate the modification of these methods, we decided to create a series of methodologies.

- In series model result detokenization.
- Automatic camelCase conversion.

To address RQ_1, we needed to analyze the code provided by the model using Scalabrino *et al.* [5] tool. To accomplish this, we applied the detokenization phase, which was divided into two parts. The first part, automatic, dealt with replacing all tokens that could be replaced without manual intervention, such as **$newline$** or **$indentation$**. The second part was manual because we had to remove all tokens that could not be removed automatically. To facilitate this procedure, we introduced the first methodology which helped automate a component of this manual phase, namely the dataset manipulation. The second methodology was employed to expedite the process and reduce the manual effort required. By implementing this procedure, we decreased the potential for manual error introduction. This methodology, shown in Figure 3.3, involved converting the methods into CamelCase format. We developed a parser that generated a binary tree from the method under analysis. The algorithm precisely divides the input string into a list of strings using space as a delimiter and performs the following operations for each string:

- 1. Set the string as the root of the tree.
- 2. Iterate through the string character by character, from left to right.
- 3. If we encounter a punctuation mark, set that mark as the root of the tree and the string preceding the mark as the left leaf. Then set the right node with the remaining unanalyzed string and recursively call the procedure on that node.

As we can observe, this algorithm defines an invariant that we can exploit to isolate words devoid of punctuation marks. Indeed, it will always be true that words

lacking such marks will be found, for each subtree, either in a left leaf or at the root position. With this established, it was straightforward to reconstruct the original string by isolating the words that needed to be converted into CamelCase. For the conversion, we used a Python library called *ninjaword*, which, employing a **N**atural **L**anguage **P**rocessing approach, successfully converted the strings into CamelCase with a reasonable success rate. It is worth noting that this library still has limitations. For example, the word "lightinglobe" can be correctly decomposed into multiple words, all equally valid: ["lighting", "globe"] or ["light", "in", "lobe"]. This highlights that, although this approach was simplifying in some respects, it certainly did not eliminate the need for manual analysis. Additionally, it's worth noting that this approach did not handle a certain type of keywords such as non-primitive data types since those types in Java start with an uppercase letter.

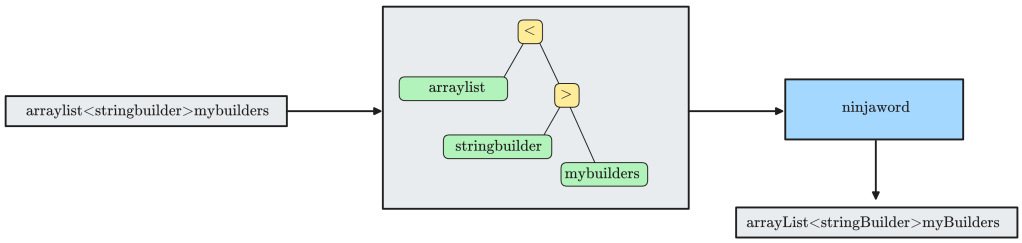

FIGURE 3.3

The manual analysis removed a significant portion of the methods.

- **Guava**: 67/100 methods were considered faulty
- **Jenkins**: 58/100 methods were considered faulty
- **Javaparser**: 58/100 methods were considered faulty

During this manual analysis, we corrected the syntax of certain methods by replacing tokens and rewriting procedures using the camelCase convention. Throughout this phase, we disregarded methods that were not modified by the model and discarded those deemed faulty. With "faulty," we mean that the method was not correctly modified by the model, thus resulting in syntax errors or even semantic errors, such as using an undeclared variable or omitting to return a value.
During this analysis phase, we also manually indented the results. This was necessary because the model was trained in such a way that it could not distinguish

4-space indentations rather than 8-space indentations for the token **$indentation$**. However, this manual indentation did not significantly alter the model's output; rather, it ensured that each statement was properly aligned. This precaution was taken to prevent misinterpretations by the Scalabrino et al. tool [5], and to ensure a more comfortable visualization.

We then executed Scalabrino *et al.* [5] tool to evaluate model prediction readability, and subsequently conduct a manual readability analysis, the objective was twofold: to identify minimal differences that the tool might have overlooked and to provide a secondary assessment of readability based on insights drawn from the work of Fakhoury et al. [6]. However, this secondary analysis does not rely on specific metrics but rather on perceived improvements as evaluated by the developer, namely myself. To address question RQ_2, we conducted a patching phase. Leveraging information about the class membership and the location of a given method within that class, obtained during the method extraction phase, it was straightforward to replace the original methods with the modified ones. Subsequently, we executed the test suite of the projects to understand if and how these changes impacted the functionality of the procedure. The execution of the test cases was atomic for each method, which means that, to prevent the modification of one method from impacting the results of another, we executed the test suite at every iteration of the patching procedure. This allowed us to determine whether the code behavior had been altered. The results of these analyses, along with other considerations, are reported in the next chapter.

Chapter 4

Results

In this section, the results obtained from the analysis conducted on the various systems, namely ***Javaparser***, ***Jenkins***, and ***Guava***, are presented. The results were obtained through the analysis of two main aspects: the readability of the code and the correctness of the test cases. Specifically, regarding the readability of the code, the average differences calculated by the Scalabrino et al. [5] tool before and after the application of the model were considered. Regarding the correctness of the test cases, the results obtained from the analysis conducted, individually for each method, on the respective systems' test suites injected with the methods modified by the model were considered.

Let's consider the results produced by the analysis on the systems, as shown in the Table 4.1.

System	Label	N. of Methods	Avg. Readability Score Diff.	N. of Test Passed	Avg. Manual Readability Score
Jenkins	LOW	14	-0.0049 ▼	%21.42 (3/14)	0
	MID	8	0.0187 ▲	%37.50 (3/8)	-0.0375 ▼
	HIGH	18	-0.0008 ▼	%77.77 (14/18)	0
Javaparser	LOW	26	0.0308 ▲	%3.84 (1/26)	0.0884 ▲
	MID	13	-0.0096 ▼	%38.46 (5/13)	0.0076 ▲
	HIGH	3	-0.0132 ▼	%0.0 (0/3)	0
Guava	LOW	18	0.0173 ▲	%100.0 (18/18)	0.0111 ▲
	MID	13	0.0369 ▲	%100.0 (13/13)	0.0923 ▲
	HIGH	2	-0.0625 ▼	%100.0 (2/2)	-0.1 ▼

TABLE 4.1: Analysis results

4.1 RQ1: Code Readability Improvement

The readability results indicate that the model by Vitale et al. [11] performs well with poorly and moderately readable methods, offering a slight improvement in readability. It's important to note that the model was trained on diff lines rather than at the method level. Therefore, when conducting this type of analysis, results may deviate from the training set, leading to outcomes like these. The results presented in Table 4.1 are particularly interesting as they demonstrate the best outcomes regarding moderately readable methods. For poorly readable methods, we can observe that the average readability improvement is higher for the Javaparser system (see Table 4.1). The only tangible negative results (an average decrease in readability) are observed for methods classified as highly readable.

An example of this could be as follows:

LISTING 4.1: Original Method

```
@Override
public int hashCode(){
    int result = kind;
    result = 31 * result + text.hashCode();
    return result;
}
```

LISTING 4.2: Modified Method

```
@Override
public int hashCode(){
   int result = kind.toString();
   result = 31 * result + text.hashCode();
   return result;
}
```

As observed, the decrease in readability (-0.06) can be attributed to the addition of the `toString()` method to the variable `kind`. Regarding readability, we conducted further manual analysis on instances, assessing the improvement or deterioration in readability of the changes made by the model using a score ranging from -1 to 1. We then reported the median score in Table 4.1. The average of the manual scores and

the average difference in readability are two indicators that can indeed be compared, as they were calculated using the same scale. However, they should be interpreted with caution as the manual evaluation was not conducted following specific metrics but rather based on personal assessment. That being said, we can observe that this average aligns with the average difference in readability suggested by the tool developed Scalabrino *et al.* [5] deviating only slightly for one of the cases. Although not apparent from the average, there are instances where the model significantly improved the readability of the methods, and such improvement is also reflected in the manual assessment. Take the following example:

LISTING 4.3: Original Method

```
14 private void generateVisitMethodForNode(BaseNodeMetaModel node,
       ClassOrInterfaceDeclaration visitorClass, CompilationUnit
       compilationUnit){
15     final Optional<MethodDeclaration> existingVisitMethod =
           visitorClass.getMethods().stream().filter(m -> "visit".
           equals(m.getNameAsString())).filter(m -> m.getParameter(0).
           getType().toString().equals(node.getTypeName())).findFirst()
           ;
16     if (existingVisitMethod.isPresent()) {
17         generateVisitMethodBody(node, existingVisitMethod.get(),
               compilationUnit);
18     } else if (createMissingVisitMethods) {
19         MethodDeclaration newVisitMethod = visitorClass.addMethod("
               visit").addParameter(node.getTypeNameGenerified(), "n").
               addParameter(argumentType, "arg").setType(returnType);
20         if (!visitorClass.isInterface()) {
21             newVisitMethod.addAnnotation(new MarkerAnnotationExpr(
                   new Name("Override"))).addModifier(PUBLIC);
22         }
23         generateVisitMethodBody(node, newVisitMethod,
               compilationUnit);
24     }
25 }
```

LISTING 4.4: Modified Method

```
26 private void generateVisitMethodForNode(BaseNodeMetaModel node,
       ClassOrInterfaceDeclaration visitorClass CompilationUnit
       compilationUnit) {
27     final Optional<MethodDeclaration>existingVisitMethod =
           visitorClass getMethods().stream()
28         .filter(m-> "visit" equals(m.getNameAsString()))
29         .filter(m-> m.getParameter(0).getType().toString().equals(
               node.getTypename()))
30         .findFirst();
31     if (existingVisitMethod isPresent()) {
32         onVisitMethodBody(node, existingVisitMethod get(),
               compilationUnit);
33     } else if (createMissingVisitMethods) {
34         onVisitMethodBody(node, newVisitMethod compilationUnit);
35     }
36 }
```

Although the model removed the nested `if(...)` statement within the `else if(){...}` construct, it intriguingly split the various `.filter()` and `.findFirst()` statements across multiple lines, significantly enhancing readability. In fact, in this instance, the readability improvement amounts to 0.5 compared to the original code, a result that mirrors the increase of 1 assigned in my manual assessment. Furthermore, this enhancement was observed for an instance initially considered to have low readability. Another noteworthy improvement, albeit minor, is evident in the `castValue` method of ***Javaparser***:

LISTING 4.5: Original Method

```
37 public static String castValue(String value, Type requiredType,
       String valueType){
38     String requiredTypeName = requiredType.asString();
39     if (requiredTypeName.equals(valueType))
40         return value;
41     return String.format("(%s) %s", requiredTypeName, value);
42 }
```

LISTING 4.6: Modified Method

```
43 public static String castValue(String value, Type requiredType,
       String valueType) {
44     String requiredTypeName = requiredType.asString();
45     if (requiredTypeName.equals(valueType)) {
46         return value;
47     }
48     return string.format("(%s) %s", requiredTypeName, value);
49 }
```

As we can observe, the only modification made was the addition of curly braces within the `if()` statement, effectively enhancing the clarity of the statement's block separation. An example of a decrease in readability is evident in this ***Guava*** method, named `tryDrainReferenceQueues()`

LISTING 4.7: Original Method

```
50 void tryDrainReferenceQueues(){
51   if (tryLock()) {
52       try {
53           drainReferenceQueues();
54       } finally {
55           unlock();
56       }
57   }
58 }
```

LISTING 4.8: Modified Method

```
59 void tryDrainReferenceQueues() {
60     drainReferenceQueues();
61     if (tryLock()) {
62         drainReferenceQueues();
63     }
64 }
```

In this example, we notice a reduction in the number of lines of code, at the expense of procedural clarity, thus making the operation `drainReferenceQueues()` more

ambiguous. As we see, the modifications to readability have been made in a subtle manner, and in some cases, they have not been made at all. However, the model has demonstrated the ability to enhance readability and, in some instances, provide a noticeable improvement. Nevertheless, such enhancements are, all in all, occasional, and on average, the fluctuations in improvement are relatively limited.

4.2 RQ2: Behavior Changes

The results demonstrate how the model tends to negatively alter the behavior of poorly readable methods. For moderately readable methods, it is observed that in 62% of cases, correctness is compromised. Regarding highly readable methods, the model, on average, does not compromise their functionality. In a specific case, namely the ***Guava*** system (see Table 4.1), it is observed that the model did not compromise the correctness of the methods, however, this result contradicts some instances observed for this system, casting doubt on the patching operation and the execution of the respective test cases for this particular instance. Another particularly important aspect to consider is the presence of a relatively low subset of methods compared to the originally considered 300. In fact, more than half of these (193/300) were deemed faulty during the manual analysis due to obvious syntactic and/or semantic errors. These values suggest that this model tends to compromise the majority of the analyzed methods. The reasons for such compromise are of various natures. For example, incorrect repositioning of parentheses clearly creates a syntactic error that prevents the method from being executed. Other times, however, the model exhibits anomalous behaviors in modification, completely altering the structure of the original method:

LISTING 4.9: Original Method

```
65        private static void explode(File archive, File destDir)
              throws IOException{
66            Util.createDirectories(Util.fileToPath(destDir));
67            // timestamp check
68            File explodeTime = new File(destDir, ".timestamp2");
```

```
69              if (explodeTime.exists() && explodeTime.lastModified()
                    == archive.lastModified())
70                  // no need to expand
71                  return;
72              // delete the contents so that old files won't
                    interfere with new files
73              Util.deleteRecursive(destDir);
74              try {
75                  Project prj = new Project();
76                  unzipExceptClasses(archive, destDir, prj);
77                  createClassJarFromWebInfClasses(archive, destDir,
                        prj);
78              } catch (BuildException x) {
79                  throw new IOException("Failed to expand " + archive
                        , x);
80              }
81              try {
82                  new FilePath(explodeTime).touch(archive.
                        lastModified());
83              } catch (InterruptedException e) {
84                  // impossible
85                  throw new AssertionError(e);
86              }
87          }
```

LISTING 4.10: Modified Method

```
88      private static void explode(file archive, file destdir) throws
            ioexception {
89        util.createdirectories (util.filetopath (destdir));
90        midnight = new file (destdir, $string$);
91        if (reportedtime.exists () && explodetime.lastmodified () ==
              archive.lastmodified ())
92            //////.get
                  ------------------------------------------------------------------
```

We can observe another example of this behavior even in this instance:

LISTING 4.11: Original Method

```
93
94  private int nextOutputChar() throws IOException{
95      int next = nextInputChar();
96      switch(next) {
97          case EOF:
98              return EOF;
99          case BACKSLASH:
100             {
101                 if (_backslashSeen) {
102                     return clearBackSlashSeen(next);
103                 }
104                 return backSlashSeen();
105             }
106         default:
107             {
108                 // An arbitrary character.
109                 return clearBackSlashSeen(next);
110             }
111     }
112 }
```

LISTING 4.12: Modified Method

```
113 private int nextoutputchar () throws ioexception {
114      int next = nextinputchar () ;
115      switch (next) {
116          case eof :
117          return eof ;
118          case backslash :
119      {
120      if (_backslashseen) {
121          return clearbackslashseen (next) ;
122      }
123      return backslashseen () ;
124      }
125      default :
126      {
127      $// g $ a n 1 4 3
128      return clearbackslashseen (next) ;
129      }
130 }
```

And in some cases, it has removed instructions necessary for the functioning of the method, as in this example where the *return* statement is removed:

LISTING 4.13: Original Method

```
131 @Deprecated
132 private static T handleResult(ParseResult<T> result){
133     if (result.isSuccessful()) {
134         return result.getResult().get();
135     }
136     throw new ParseProblemException(result.getProblems());
137 }
```

LISTING 4.14: Modified Method

```
@deprecated
 private static t handleresult(parseresult <t> result) {
     if (! result.issuccessful()) {
         throw new parseproblemexception(result.getproblems());
     }
```

In these examples, we notice that the strings are not in CamelCase, and the tokens **$string$** and **$number$** have not been replaced. This is because this phase has never been applied given the presumably flawed nature of the model's modification. As we can see, some procedures have been invalidated by purely syntactic changes. However, there have also been logical changes that went unnoticed during the initial analysis but were later discovered during the examination of the modifications made by executing the test suite of the respective projects. We can observe this behavior, for example, in the method `lookupFirst()` of ***Jenkins***:

LISTING 4.15: Original Method

```
@NonNull
public static U lookupFirst(Class<U> type){
  var all = lookup(type);
  if (!all.isEmpty()) {
      return all.get(0);
  } else {
      if (Main.isUnitTest) {
          throw new IllegalStateException("Found no instances of "
              + type.getName() + " registered (possible annotation
              processor issue); try using `mvn clean test -Dtest=
              ` rather than an IDE test runner");
      } else {
          throw new IllegalStateException("Found no instances of "
              + type.getName() + " registered");
      }
  }
}
```

LISTING 4.16: Modified Method

```
156 @NonNull
157 public static U lookupFirst(Class<U> type) {
158     if (!all.isEmpty()) {
159         throw new IllegalStateException("Found no instances of " +
              type.getName()+ " registered(possible annotation
              processor issue); try using `mvn clean test -Dtest=   `
               rather than an IDE test runner");
160     }
161     if (Main.isUnitTest) {
162         throw new illegalStateException("Found no instances of " +
              type.getName()+ " registered");
163     }
164 }
```

Moreover, we also observe an increase in readability of 0.4 compared to the original method. We can see that the method has been streamlined by the improvement operation (which explains the increase in readability performed by the model); however, this has compromised correctness, thus undermining the refactoring operation.

In general, the majority of instances are represented by results of this kind, although in some cases, this does not occur. It is clear that although enticing, the model is not yet able to offer an acceptable accuracy to be considered a valid approach to automated refactoring. Nevertheless, it remains extremely promising.

Chapter 5

Threats to Validity

It is crucial to acknowledge potential threats to the validity of an experiment. Firstly, it's important to clarify that the evaluation of readability results, both before and after modification, relies on a heuristic-based tool. While this evaluation may not be mathematically precise, it holds significant statistical value. This aspect is pivotal as much of the experiment hinges on these heuristics; hence, any inaccuracies in them could significantly impact the results. Another potential threat to validity arises from the experiment's reliance on a relatively small sample of projects. Consequently, we lack sufficient data to deem the results statistically representative. However, this serves as a solid starting point, as it underscores the genuine potential of such models, albeit in a limited capacity.

A potential threat to the validity of the study lies in the fact that a significant portion of the entire analysis relies on manual validation. This manual evaluation played a crucial and substantive role, given that the model, at the time of publication, only has the capability of outputting in lowercase. Therefore, this phase has played an important part in the study, and it is possible that this validation may have introduced errors that could have influenced the results.
Another threat to the validity of the study arises from the fact that the results regarding the correctness of the methods were obtained through the analysis of the test suite. This may not be sufficient to guarantee the correctness of the code, as such a test suite may not cover all possible scenarios in which a bug can be introduced.

Chapter 6

Conclusion and Future Work

As observed during this experimental study, the potential for applying machine learning techniques is undoubtedly high, but the results are still nascent. Indeed, although these techniques could significantly impact the software development industry, it is clear that such an impact may require time to mature. Nevertheless, this experimental thesis has provided a pipeline of operations and a dataset that could be valuable for future studies. Future work will focus on applying the pipeline defined in this study on the next iterations of the model, which will not have the same limitations as the current model. In addition, future work could focus on various aspects, such as applying these techniques to other programming languages and analyzing the behavior of future versions of the model considered in the study. In addition to this, future work could extend this research by using a larger amount of data compared to what we utilized in the analysis phases.

Bibliography

[1] Terese Besker, Antonio Martini and Jan Bosch. 'Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work'. In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 105–114. ISBN: 9781450357135. DOI: 10.1145/3194164.3194178. URL: https://doi.org/10.1145/3194164.3194178.

[2] Nicolli Rios et al. 'The most common causes and effects of technical debt: first results from a global family of industrial surveys'. In: ESEM '18 (2018). DOI: 10.1145/3239235.3268917. URL: https://doi.org/10.1145/3239235.3268917.

[3] Michele Tufano et al. 'When and Why Your Code Starts to Smell Bad'. In: 1.2 (2015), pp. 403–414. DOI: 10.1109/ICSE.2015.59.

[4] Todd Sedano. 'Code Readability Testing, an Empirical Study'. In: (Apr. 2016), pp. 111–117. ISSN: 2377-570X. DOI: 10.1109/CSEET.2016.36.

[5] Simone Scalabrino et al. 'A comprehensive model for code readability'. In: *Journal of Software: Evolution and Process* 30.6 (2018). e1958 smr.1958, e1958. DOI: https://doi.org/10.1002/smr.1958. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1958. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958.

[6] Sarah Fakhoury et al. 'Improving Source Code Readability: Theory and Practice'. In: 1 (2019), pp. 2–12. DOI: 10.1109/ICPC.2019.00014.

[7] Cleiton Tavares, Mariza Bigonha and Eduardo Figueiredo. 'Analyzing the Impact of Refactoring on Bad Smells'. In: SBES '20 3 (2020), pp. 97–101. DOI: 10.

1145/3422392.3422408. URL: https://doi.org/10.1145/3422392.3422408.

[8] Delano Oliveira et al. 'Evaluating Code Readability and Legibility: An Examination of Human-centric Studies'. In: (Sept. 2020), pp. 348–359. ISSN: 2576-3148. DOI: 10.1109/ICSME46990.2020.00041.

[9] Valentina Piantadosi et al. 'How does code readability change during software evolution?' In: *Empirical Software Engineering* 25.6 (Nov. 2020), pp. 5374–5412. ISSN: 1573-7616. DOI: 10.1007/s10664-020-09886-9. URL: https://doi.org/10.1007/s10664-020-09886-9.

[10] Antonio Mastropaolo et al. 'Automated variable renaming: are we there yet?' In: *Empirical Software Engineering* 28.2 (Feb. 2023), p. 45. ISSN: 1573-7616. DOI: 10.1007/s10664-022-10274-8. URL: https://doi.org/10.1007/s10664-022-10274-8.

[11] Antonio Vitale et al. 'Using Deep Learning to Automatically Improve Code Readability'. In: 5 (2023), pp. 573–584. DOI: 10.1109/ASE56229.2023.00112.

Ringraziamenti

Vorrei dedicare le prossime righe a tutte le persone che mi hanno sostenuto, direttamente o indirettamente, durante il mio percorso di vita.

Voglio ringraziare la mia famiglia, con tutto il cuore. Quando decisi di intraprendere questo percorso, di virare dalla rotta da me precedentemente considerata, voi ci siete stati e mi avete sostenuto, in tutti i sensi. Non potete sentire ciò che sento perché l'empatia è l'illusione di provare ciò che un'altra persona prova, ma sappiate che, nel momento in cui sto scrivendo questa dedica, e in tantissimi dei momenti che hanno caratterizzato questo viaggio, ho e ho avuto il cuore ricolmo di gratitudine. Grazie.

Ti ringrazio Cox per avermi spalleggiato in questo lungo context switch e per avermi spronato, in via indiretta, a dare il meglio di me. Ti ringrazio per avermi offerto un infanzia degna di essere vissuta e soprattutto, di esserti preso la parte più noiosa dello sviluppo software! <Kiss><Kiss> Ringo

(M)artina, che dire, ti dico grazie perché mi sei sempre stata accanto, con il tuo fare giocoso hai ispirato il meglio in me. Ora sono come una primogem che cade dal cielo... con che colore mi vedi? Nah, non importa, sappi che quello che penso di te è e sarà sempre questo: Sei una persona speciale e ti voglio e vorrò sempre bene, qualunque cosa tu faccio, dovunque tu sia.

Mamma, voglio ringraziarti per quello che hai fatto e per quello che fai tutti i giorni per me. La vita ci ha offerto alti e bassi, e nonostante tutto tu sei stata in grado di risorgere da ogni caduta. Ti ammiro per la tua tenacia, perché non è una qualità da poco. Da te ho ricevuto tanto, l'affetto di una madre, la comprensione di una madre e l'insegnamento che tutto, in fin dei conti, può essere messo in dubbio!. Ti voglio un mondo di bene.

Papà, voglio ringraziarti con tutto il cuore, non sono un grande scrittore, ma spero che le mie poche parole sgrammaticate possano farti capire quanto io sia fiero di te. Sei la persona che ha e continua ad ispirare la mia vita e penso che questo sia tutto ciò che un padre possa mai volere. Ti ringrazio per il tuo sostegno costante, per la tua capacità di capirmi e per la tua immancabile grinta che ha fatto di me ciò che sono ora. Ti voglio un mondo di bene.

Voglio ringraziare tutti i miei zii, zie, cugini e cugini. Siete parte di ciò che ha contribuito a rendere me ciò che sono ora e grazie al vostro sostegno e affetto, ho vissuto grandi momenti. Siete una parte importante, una parte che associo sempre alla festa e ai momenti felici. Vi ringrazio per essere rimasti coesi gli uni con gli altri perché, grazie a voi, ho potuto apprezzare a pieno il concetto di famiglia.

Voglio ringraziare anche te Sonia! Si si, ora ti faccio piangere un po'! Già fatto? È stato facile. Sul serio, magari pensi di non esserlo stato ma lascia che ti contraddica, sei stata davvero importante per me. Grazie per avermi offerto cosi tanto, senza che io potessi fare altrettanto per te. Sappi che mi ricordo di tutto il bene che ricevo. Sei una persona vera, una Donna vera (altro che rose villein!) e sono contento di poter apprezzare la tua grazia e fermezza da cosi vicino. Ti voglio tanto bene.

Ai miei nonni, vi voglio bene e vi ringrazio per aver creduto in me. La distanza non vi ha impedito di dimostrarmi il vostro affetto e appoggio. Nonna ti ringrazio per tutto quello che mi hai dato: un letto dove dormire, tanto amore e pranzi e cene buonissime. Sei un'artista in quello che fai, oltre che una nonna esemplare, e sappi che io ti voglio e vorrò sempre un mondo di bene. Nonno, i tuoi racconti sono l'esperienza di una vita vissuta a pieno e io sono grato di aver avuto modo di imparare dal tuo vissuto. Mi hai insegnato ad avere grinta (e fede) e a non mollare mai (e ad anticipare il pericolo quando guido!). Tu, prima di mio padre, hai reso il nostro cognome degno di rispetto, e io mi impegnerò a fare altrettanto.

Ringrazio tutti i miei colleghi di università: Mike, Fra, Leo, Mario, GC, Romanella, CruCru e Giuseppe. Senza di voi starei ancora facendo ricerca operativa all'ombra

di qualche banco! Sul serio siete stati una spalla importante e voglio esprimervi tutta la mia gratitudine per quello che avete rappresentato per me. Voi avete reso questo viaggio veramente speciale.

Voglio ringraziare i miei patini, Roberto e Milena, e ovviamente la zia Angelina! Come avrei fatto senza di voi? Grazie per avermi sostenuto, davvero. Seppure non ci vediamo sempre so di poter contare su di voi, e sul vostro affetto. Spero un giorno di poter ricambiare quanto mi avete, e state dando. Vi voglio bene.

Voglio ringraziare i miei (per sempre) coinquilini!

Jonathan, meglio usare il nome piuttosto che il soprannome. Sai bene, meglio di molti, quanto sia solitario e schivo, ma nonostante questo e le nostre stranezze abbiamo legato davvero tanto. Sei stato un tassello chiave nella mia crescita e ho piacere di pensare che anche io lo sia stato per te. Ti voglio bene, sul serio, perché ci siamo dimostrati quanto a fondo siamo simili e quanto profondamente siamo diversi. Sei l'unica persona con cui condivido il peso e la leggerezza del verbo. Come conoscitori della vera verità, possiamo camminare tranquilli sulla terra, sempre pronti a dare una spiegazione, consapevoli che non saremo mai in grado di darne una. Nel frattempo, magari, ci inventeremo qualcosa che sia meglio del Buddismo.

Pink, perché mi odi? :P Voglio prendere questo spicchio di carta per ringraziarti, per la tua presenza, per le chiacchierate che ci siamo fatti, per i momenti in bici e le insalate di riso che mangiavamo con gusto. Sei stato prezioso nel mio percorso, un mentore che ho mancato di cerimoniare come si deve e perciò lo voglio fare ora. Il valore di una persona è definito da quello che fa per gli altri, e tu per me hai fatto tanto. Ah, grazie a te posso dire "I use arch btw" in giro.

Lelio, come ci sei finito qui? Non serve che te lo chieda, credimi, anche il poco tempo che abbiamo passato assieme mi ha fatto affezionare. Sei una persona estremamente piacevole da avere attorno. Perciò voglio ringraziare anche te, perché anche se per poco abbiamo avuto modi di legare. Abbiamo fatto squadra in più di un'occasione

e grazie a questo abbiamo sconfitto alcuni mostri sacri **cough* *cough** fisica... che molti si sognano di superare con un così gradevole supporto. Sei stata una presenza rinfrescante, e nei miei ultimi giorni ad Isernia, ho rimpianto di non aver potuto passare più tempo con te

Mi caro amico Mike, siamo stati l'uno a fianco dell'altro per tanto tempo e abbiamo condiviso molte giornate assieme. Abbiamo visto cose interessanti (come sono scomparsi quei bicchieri sulla tettoia del terminal?) e condiviso risate, grigliate e ragionamenti tutt'altro che scontati. Ti ringrazio per avermi regalato tutto questo, e per avermi sostenuto comunque anche quando non potevamo vederci. Sei ciò più desidero in un amico.

Lucchetto, ti voglio ringraziare per tutto ciò che ha fatto per me nel corso del tempo. Grazie per avermi tenuto compagnia, quando potevi e quando non potevi, per aver condiviso con me parte della tua vita e per avermi fatto capire il valore di un amico. Sei una persona speciale e ti voglio bene.

Francesco tu e io abbiamo condiviso tante cose, il turbinio delle superiori, e la calma dei prati su a Kingswood eight. Ci siamo tenuti l'un l'altro sui binari e per questo non posso che dirti grazie. Sei stato la mia spalla e la persona che mi ha aiutato a superare i momenti più difficili. Compagno di strada e ora compagno di università. Credo proprio di doverti dire grazie perché è anche grazie a te che sono qui, tutto intero, a smanettare con il software.

Al mio amico Marco. Sei stato prezioso nella mia infanzia, mi sei stato vicino nei momenti più bui della vita, senza chiedere nulla in cambio. Sei una persona di grande importanza e ti voglio bene. Grazie per tutti i momenti che abbiamo passato a bighellonare assieme, per avermi spalleggiato anche quando chi consideravo amico si preoccupava bene di fare il contrario di ciò che un amico fa. Ti voglio bene, e spero quanto prima di spaccarmi con te a black ops zombie.

Pascarella, voglio ringraziarti di vero cuore per essere stato parte della piccola cerchia di persone che ha reso la mia adolescenza memorabile. Con te ho condiviso molto, dai momenti di riflessione, alle pazzie, alle passeggiate di notte a Largo Zullo. E' anche grazie a te che sono ciò che sono. Sei stato cruciale nel mio percorso e sono convinto che lo sarai ancora. La nostra amicizia è davvero speciale perché io e te siamo, in fin dei conti, l'uno l'opposto dell'altro e nell'anomalia che ci lega trovo si nasconda l'affascinante natura dei rapporti umani.

Ringrazio il mio amico Albio, sei una delle persone che ricordo con più gioia. Quando ti penso non posso fare a meno di ripensare al momento in cui, alla fine dell'ora di cucina, il professore Serafino ti ha trovato intento a fustigare Vittorio con una cinta. Questa immagine per me sintetizza i nostri anni alle superiori, eccentrici eppure bellissimi. Sei esempio di stile ed eleganza, così caratteristico da non poter essere dimenticato.

Ringrazio anche tutti i miei amici delle superiori. Voi siete stati una delle parti più importanti della mia vita perché avete cambiato in meglio ciò che pensavo delle persone. Grazie per avermi accettato, per avermi permesso di esprimere ciò che ero e in parte ciò che sono ancora. Voi avete cambiato la mia vita e spero anche io di aver cambiato la vostra.

Ringrazio tutte le persone che mi sono state vicine nella mia vita in Irlanda. Leanda per avermi accolto nella sua casa, Dave, Antonella, Irina, Alessandro, Ricardo, e Martina, voi siete i protagonisti di uno dei momenti più duri e più belli della mia vita e vi ringrazio per averlo reso indimenticabile. Siamo stati un gruppo, una squadra e poi una famiglia vera e propria. Grazie Mary per aver mosso il mio cuore. Grazie anche te Mike, il nostro periodo li a è stato unico anche grazie a te. Grazie Riky e Leo, senza di voi non conoscerei munchkin, e non avrei dei cosi bei ricordi del mio ultimo periodo li in Irlanda.

Ringrazio tutti i miei professori, in particolar modo il professore Scalabrino che è stato fonte di ispirazione e il correlatore Antonio Vitale che mi ha seguito con dedita

pazienza. Un grazie va anche al professor Oliveto, perché grazie a lei, ho avuto la fortuna e l'onore di appassionarmi al mondo della programmazione e dell'informatica. Ringrazio anche lei, professore Capobianco, per avermi mostrato come può essere complessa e allo stesso tempo affascinante la matematica, e per avermi insegnato la forma e l'eleganza del rigore matematico. Ringrazio anche, in particolar modo, il professore Gennaro Parlato, perché grazie a lei ho ricevuto la capacità di astrarmi da ogni tecnologia, e funzionare solo a logica. Ringrazio la professoressa Troncaceli perché mi ha sempre supportato nell'elettrizzante, seppur breve, periodo di rappresentante del corpo studentesco. Ringrazio tutti gli altri professori, i membri dell'UGO e tutto il corpo docenti. Di meriti ne siete pieni. Sono grato di avervi permesso di essere miei maestri. Ringrazio tutto lo staff dell'università, che nell'ombra ha permesso che tutto ciò potesse esistere. Grazie per il vostro lavoro, perché senza di voi non sarei qui a scrivere queste righe.

Dunque grazie a tutti voi.
Che possiate raggiungere tutti i vostri obiettivi.
Ad ogni costo, con ogni mezzo.

"You keep moving forward. Even if you die. Even after you die."