SCHOOL OF ADVANCED TECHNOLOGY
SAT301 FINAL YEAR PROJECT

Automatic Software for Judging Source Code Quality

Final Thesis

In Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Engineering

Student Name	:	Guanyuming He
Student ID	:	2035573
Supervisor	:	Thomas Selig
Assessor	:	Paul Craig

Abstract

In an increasingly digitalised world, automatic grading software (auto-graders) has been used by teachers of programming modules across the globe to evaluate the pieces of source code students submitted for assignments. Yet, the existing evaluation methods typically are concerned with the correctness of code, neglecting the quality of it. In this final year project, the author aims to fill the gap by **first** devising a plan of judging source code quality and **then** applying the plan to develop a real software system that works on Java source code. The plan and the system offer a number of features useful in programming education: **(a)** the plan is transparent and deterministic, thus arguably fair; **(b)** the plan judges quality from multiple aspects, indentation, spacing, naming, style, and more; **(c)** the system can be easily extended and customised for different standards of code quality.

Keywords: source code quality, automatic grading, auto-grader, programming, programming education

Acknowledgements

I, Guanyuming He, would like to express my deepest gratitude to my supervisor, Dr. Thomas Selig, who not only willingly provides extremely useful and professional guidance and feedback on my project, but also gives suggestions about problems elsewhere in my academic life.

My thanks also go to other staff of the university who together have been building an undergraduate environment that I have enjoyed for four years. I probably could not have had this experience at any other university in China. Special thanks to Dr. Pengfei Song, who provides important information and report templates for this Final Year Project SAT301.

Lastly, I must acknowledge my parents, who have financially supported me during my undergraduate study, despite various hardness like COVID in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation, Aims and Objective

1.1.1 Motivation

Digitalisation is taking place in the world. You may notice the rapid growth of digital devices and computers in our own lives, which agrees with what the Oxford English Dictionary [1] defines "digitalisation" as: "*The adoption or increase in use of digital or computer technology by an organization, industry, country, etc.*" It is evident that a major part of modern life has already been digitalised. M. Castells argued in his famous book [2] that one of the deciding qualities of the modern age is the digitalisation of "*the new economy, society, and culture*." Moreover, several studies revealed the huge impact of digitalisation in various countries, such as those of the European Union, and Russia [3, 4]. Besides, M. Muktiarni et al. recently analysed the digitalisation trend in education [5].

Similar to other prominent trends in society, digitalisation has resulted in massive changes to individuals' lives. If you happen to be a teacher/professor who teaches a few computer science or programming courses, then you probably have noticed a striking growth in the number of students enrolled, a phenomenon that has also been observed in some surveys [6, 7].

As a consequence, it is unsurprising that the teaching staff of these courses have started using automatic grading software (also called *auto-graders*) to assist them in grading student coursework and assignments. In fact, one of the earliest instances of utilising auto-graders in programming courses is recorded in a 1965 paper [8], where the authors discussed two grader programs used in Stanford University since 1961. The two programs could decide the correctness and performance (running time, number of certain operations done, etc.) of a student's work.

It has been more than 50 years since the first auto-grader was used in programming courses, and currently, there exists a large variety of such tools offering different features. J. Caiza and J. Alamo [9] reviewed a large set of mature or innovating auto-graders in 2013, and discovered notable features, including scalability, plagiarism detection, security (in the sense that the grades and evaluation process are protected), course platform integration, and GUI grading. In the last ten years, a few other novel auto-grading approaches have been invented: S. Parihar et al. [10] created an auto-grading system that can additionally repair some syntax errors in student submissions; X. Liu et al. [11] proposed an auto-grader that decides the correctness of a student's submission by comparing the semantic execution path of the submitted program with a correct reference implementation.

Despite the novel features that these contemporary auto-grader systems possess, nearly none of them can evaluate the source code quality of a student's submission. In principle, the term "code quality" refers to properties concerning how well humans can understand/modify/distribute a piece of source code, thus covering a broad range of aspects, such as readability, maintainability, and usability. Because software engineers usually work in teams at present, these aspects of source code are of significant importance. It is expected that students in introductory programming courses are very likely to make mistakes in such aspects, since they usually are satisfied as soon as their programs produce the right result. H. Keuning et al. [12] examined "*two million Java programs of novice programmers recorded in four weeks of one academic year*" to have found not only an enormous amount of code quality problems, but also the fact that the problems were rarely repaired. A more alarming discovery identified numerous code quality problems from the year-4 undergraduate students who participated in a "*highly competitive Software Engineering programme*" [13], which heavily implies the necessity of dealing with code quality problems in programming education.

Therefore, in my Final Year Project, I want to contribute to this situation by developing an independent software system that can automatically judge the source code quality of submitted source code in coursework and assignments for students. As it is independent, integrating it into existing auto-graders that lack this function is also possible.

1.1.2 Aims and Objective

In this subsection, I define the project aims and objectives of this project. First, I will present the two major aims of my project. Then, I will define what objectives I must satisfy to achieve each aim.

The two major aims

AIM1 Create a method of judging source code quality that is powerful and suitable enough for being used in programming education.

AIM2 Apply the method for **AIM1** in developing a real software system that can at least judge Java source code quality.

Objectives for each aim

- Objectives for **AIM1**. Because code quality is a broad subject, I must ensure that it can be evaluated from different angles using my method. In addition, I need to consider the factors in programming education to apply the method there.

  A1O1 The method is powerful enough. That is, it can judge the code quality from multiple aspects: spacing, indentation, naming, comments, etc.

  A1O2 The method is suitable for programming education. That is, it is *transparent* and *fair*.

  A counterexample includes the machine learning approaches that are based on *neural networks*[1]. A neural network is a network of connected *neurons*. A neuron takes some inputs that are weighted, adds them together using a special threshold function, and finally produces a output that may be an input of another neuron in the network.

  One or more neurons' output are used as the output of the network, and to improve the output when training the network, a process called *back propagation* is usually used. In this process, each weight w in the network will be moved according to the partial derivative of the loss function, usually defined as $-\frac{1}{2}(\mathbf{d}-\mathbf{z})^2$, with respect to w, where $\mathbf{d}$ is the desired output and $\mathbf{z}$ is the actual output of the network.

  When the loss function is acceptably close to zero, the training is complete and the weights then become fixed. A neuron is simple, and the key of a neural network is the connections between them and the weights of the connections. However, it is extremely difficult to make sense of these connections and weights (i.e. usually one does not know why a network outputs some value), a fact

[1] It might not be obvious to some that those currently prosperous generative AI systems like ChatGPT are essentially applications of (deep) neural networks.

that makes neural networks function like black-boxes in various applications, which are not transparent and explainable, thus not fair.

- Objectives for **AIM2**. Source code quality is quite subjective by nature, and varies from programming language to language. Therefore, my software system must support heavy configuration and extension by design.

  A2O1 The system I develop must be able to work on Java source code, and can be easily extended to work on other programming languages.

  A2O2 The system I develop must provide a framework in which it is easy to configure and extend the system for different source code quality standards.

1.2 Literature Review

A literature review was conducted as early as when I was writing the specifications of this project. At that time, one important goal of it was to identify the gaps in the current literature about code-quality judging auto-graders.

After the *Interim Report* of my project was finished (about half in the whole process), I realised that the literature review could use more papers to investigate the existing auto-graders and existing methods to judge code quality. Consequently, I reviewed a number of other papers and added the results to my previous literature review. Nevertheless, they all followed the same review process.

Hence, in this section, I will present an **improved** version of the literature review based on the version I have presented in my *Interim Report*. I will start by describing how the papers are reviewed in detail, including how they are selected and filtered. At length, I will discuss the results of the improved literature review and raise a few research questions based on the gaps identified.

1.2.1 Review Questions

The literature review was first conducted to answer these critical questions, while emphasising the first and second ones, since I was specifying the project and deciding what to be done then:

1. What gaps exist between the state-of-the-art research on auto-graders and that on measuring code quality?
2. What are the applications of auto-graders?
3. What are the most popular/noteworthy standards of source code quality?
4. What algorithms have been used to measure code quality using those standards?

After the Interim Report, the sole goal became the completion of the software development, so I reviewed a number of papers additionally to answer the last two questions in extra details.

For clarity and conciseness, I will demonstrate the two individual review processes as a whole in the rest of this section.

1.2.2 Gather and Filter Papers

First and foremost, I want to point out that because I am the sole researcher in the project, the paper selection was inevitably biased, and probably could not cover every corner of the current literature. Nevertheless, I followed systematic approaches in the process of gathering and filtering papers to make the results as neutral as possible. In this subsection, I will illustrate the process.

Gathering papers. In order to answer the four review questions, I need to cross different subjects and search for different combinations of their keywords. Table 1.1 lists all the keyword combinations and the resulting phrases I used to search for papers on *Google Scholar*.

At first, I collected $n_1 = 40$ papers. After my Interim Report, I collected an additional $n_2 = 13$, resulting in a total of $N = n_1 + n + 2 = 53$ papers. Because many of them were filtered out, I do not list all of them in this report. Instead, I will only list those that passed the filtering later in this section.

Filtering Papers N is certainly very large for only one researcher, me, so I created the following stages to quickly filter out papers that would not be very helpful to answer the review questions:

Stage 1 All papers that I selected from *Google Scholar* by searching the phrases enter Stage 1. In this stage, I directly apply the publishing

#	Keywords	Search phrase
1	automatic grading programming education	"automatic grading" AND "programming education"
2	auto-grader code quality (integration)	auto-grader OR "automatic grading" AND "code quality" integration
3	auto-grader code quality (component)	auto-grader OR "automatic grading" AND "code quality" component
4	(source) code quality	code OR "source code" quality
5	(source) code quality metrics	code OR "source code" metrics"
6	(source) code quality quantification	code OR "source code" quality quantification"
7	(source) code quality evaluation algorithm	code OR "source code" quality AND "algorithm" OR "evaluation"
8	(source) code standard	code OR "source code" standard"
9	(source) code readability	code OR "source code" readability
10	readable (source) code	readable code OR "source code"
11	code writing practices	"code writing" OR coding OR programming practices
12	code writing guidelines	"code writing" OR coding OR programming guidelines
13	code writing rules	"code writing" OR coding OR programming rules
14	teaching programming habits	teaching programming OR coding habits
15	(static) (source) code analysis code quality	"static" AND "code" OR "source code" analysis AND "code quality"

Table 1.1: Paper searching keywords and phrases

date restrictions: **(a)** For auto-graders in programming education, the papers must have been published within the last 15 years, a little long because the topic is relatively under-developed. **(b)** For code quality measurements that do not change drastically over time for a fixed context, the papers published in the last 40 years are considered. **(c)** For algorithms used to measure code quality, I use the papers published in the last 20 years.

Stage 2 The papers in Stage 1 are screened by title, abstract, and publisher/venue to quickly filter out the most unrelated ones.

1. Is its title or abstract directly related to any of the review questions?
2. If it is a paper regarding auto-graders, does it mention the methodology of the auto-graders discussed and where they have been applied?
3. If it is a paper about code quality measurements, does it talk about quality in general that can be taught in a programming course, or a very specific one in some specific context?
4. Is the publisher/publishing venue of the paper a reliable one? Famous science paper publishers like Springer and ScienceDirect are considered reliable. Venues in the list provided by [14] are considered reliable. However, because of the **under-development** of the topic, a paper that is not in the list but strongly addresses any of the RQs can be an exception.

Stage 3 The papers in Stage 2 as well as those that might or might not meet the criteria are screened by full text, with a focus on the Introduction

and Conclusion. Those that really can help answer the review questions enter Stage 3.

Of all the papers in Stage 1, 23 entered Stage 3, and the filtering process is demonstrated in Figure 1.1. On the left, the three rectangles in red represent the papers discarded during the filtering, and on the right, the four rectangles in dark green represent those that were kept.

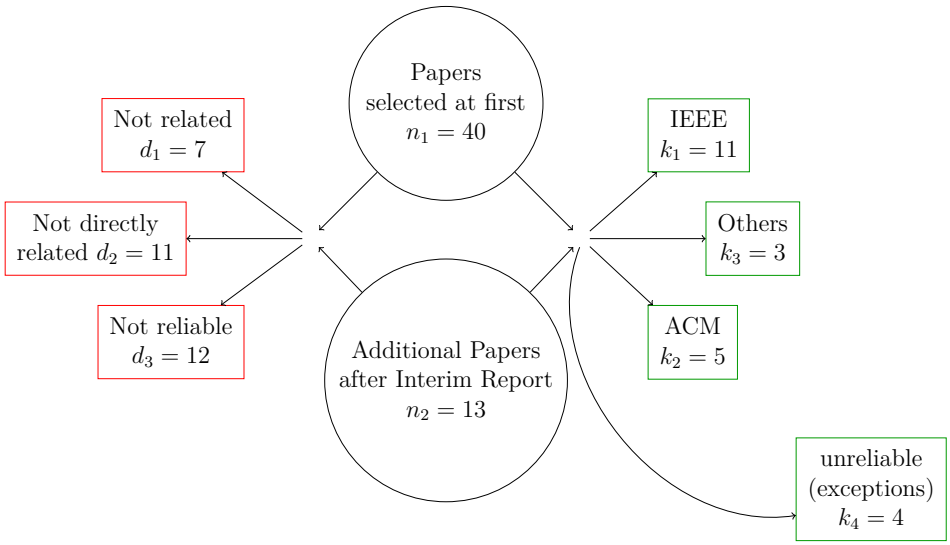

Figure 1.1: Result of paper filtering

1.2.3 Review Results

I will open this subsection by showing the result of paper selection and filtering (i.e. those that were kept and used). After that, I will present what I have concluded from reviewing these papers.

Result of paper selection and filtering

All papers that entered the final stage are listed in Table 1.2, where the papers from unreliable publishers are exceptions as they strongly addressed at least one of the review questions.

Current auto-grader literature

In Section 1.1.1, I briefly discussed some facts about contemporary auto-graders. In this subsection, I will give a more complete summary of the current literature.

Source	Venue/Publisher	Year	Source	Venue/Publisher	Year
[15]	Springer	2007	[16]	IEEE	2010
[17]	IEEE	2019	[18]	IEEE	2010
[19]	IEEE	2021	[20]	IEEE	2021
[21]	ACM	2014	[22]	IEEE	2010
[23]	IEEE	2016	[24]	IEEE	2016
[25]	Unreliable	2023	[26]	ACM	2023
[10]	ACM	2017	[27]	ACM	2021
[28]	Unreliable	2010	[29]	IEEE	2013
[30]	IEEE	2021	[31]	ScienceDirect	2012
[32]	Unreliable	2021	[33]	ACM	2011
[34]	IEEE	2016	[35]	Unreliable	2014
[36]	ScienceDirect	2022	-	-	-

Table 1.2: List of all papers kept in the review

A recent paper of literature summary exists. J. Tharmaseelan et al. revisited and summarised all auto-grader technologies in programming assignments before 2021, and categorised them into three categories [30]:

- **Dynamic Approach:** In that paper's term, the dynamic approach is, in reality, the most common "test-driven" approach used in most existing auto-graders. In this approach, the submitted code is executed against a series of predefined test cases, and the grade depends on the passed and failed test cases.
- **Machine Learning Approaches:** These approaches are mainly found in recent auto-graders. That paper points out that these approaches can differ greatly in strategy and has elaborated on three typical machine learning strategies: one using *Convolutional Neural Network*, one using *Syntax Graph*, and one being *Natural Language Based*.
- **Static Code Analysis:** In the broad sense, static analysis refers to analysing a piece of code statically without executing it. That paper mentions that the method appears mostly in modern IDEs (Integrated Development Environments), which utilise this method for various purposes: code highlighting, finding syntax errors, discovery of infinite loops, etc. Although IDEs are not part of auto-graders, they can assist students in writing code.

In addition to those techniques summarised in that paper, a few noteworthy yet different techniques can be observed. Some [10] propose the use

of program repair in auto-graders. Another [31] introduces fuzzy logic into the test cases of an auto-grader. Different from test-driven methods, some researchers present evaluating correctness by the semantic execution path [11]. More recent studies in 2023 [25, 26] have specifically addressed integrating machine learning approaches of judging code quality into auto-graders used in high-school-level programming courses.

Current code quality evaluation literature

There have already been numerous attempts to explore different source code quality measurements, and they have been applied in many different places regarding code style, readability, maintainability, and more.

An attempt [15] was made to address several problems in the then-conventional quality metrics by proposing a comprehensive framework that was novel at that time. However, its focus was on general software characteristics (e.g. Reliability, Efficiency, and Maintainability) and therefore, the work fails to satisfy the more specific and subtle quality metrics introduced in programming courses. General works like this can also be found, [34, 33], which summarised the overall practices and guidelines of coding, while the latter one specifically examined what separates good code from great code.

More specific studies have also been made. Since identifiers (names of variables, classes, etc.) are a major part of any source code, various problems related to the identifier names in code are investigated [16, 18]. Some studies research code readability [17, 22, 23, 24], each either researches on a high level or tries to give solutions to specific problems. Focusing on machine learning approaches, [19, 20, 22, 35] together cover a wide range of code quality problems and even program bugs, and the second of which particularly concludes most learning approaches. Apart from these technical studies, Simon et al. [21] looked into how well students accept certain programming practices.

I also want to highlight a few novel papers found in the literature review. **(a)** Y. Kanellopoulos et al. [28] propose some code quality evaluation methodology based on the *ISO/IEC-9126* [37] standard, which defines some software engineering product quality standards. **(b)** H. Chen et al. [36] showcases a scheme to quantify a student's contribution to code-quality in programming projects. **(c)** D. Steidl et al. [29] specifically target the quality of comments in source code.

Gaps and the research questions

As already pointed out in the Introduction, very few of these auto-graders treat source code quality as a factor to the marks they give. As for the few that do this, they mostly use machine learning approaches, which are arguably not suitable for education, as the prominent machine learning models like neural network are not transparent and easily explainable.

In the code-quality evaluation literature, despite a large number of papers' investigation into a broad range of code quality metrics and implementations, studies that inspect the context of programming courses can hardly be found. Besides, except for a few, none considered how the metrics could affect programming education, and they do not discuss which actions the teachers of programming courses should perform to ensure the code quality of their students.

Besides the aims and objectives mentioned in Section 1.1.2, I am also motivated to answer the following research questions derived from the results of the literature review:

RQ1 If transparency must be ensured, how well can I do (i.e. how many aspects of code quality can be covered) in developing my system?

RQ2 How my system's evaluation result on a piece of source code is going to help teachers assign marks to it?

Chapter 2

Methodology and Results

In this chapter, how I achieved my aims and objectives, as well as the results will be discussed.

2.1 Methodology

The section is divided into these parts: **(a)** I explain how my literature review results helped me develop my methodology. **(b)** I describe the scheme I proposed to evaluate source code quality so that my objectives *A1O1* and *A1O2* can be achieved and, **(c)** how I develop the software system so that my objectives *A2O1* and *A2O2* can be achieved. **(d)** I demonstrate the software testing methods I used to test my software system. **(e)** I mention the tools and environment I used to develop the software system.

2.1.1 How literature review helped my methodology

Reviewing the discussion/limitation section of papers about using machine learning to grade code or judge code quality confirms my assumption that most machine learning models cannot provide a transparent process of judging code quality. Additionally, reading the studies of static code analysis gave me an approximate idea of how many features I can achieve in my software.

2.1.2 Code-quality evaluation methods

Recall the two aims and four objectives of my project. *A1O1* requires me to judge the quality from many different perspectives, so I made a decision to categorise different pieces in a source file and evaluate them differently. For example, an identifier can be judged by its naming, while punctuation is judged by spacing around.

A1O2 implies that I need to design a deterministic algorithm instead of putting source code texts into black boxes like trained machine learning models.

Combining the implications of the two objectives, I devised a *Division-Evaluation-Summary-Verdict* strategy to judge the source code quality of a Java source file.

- The text in the source file is divided into different categories.
- For each instance of a category, the instance is evaluated to have a *result*.
- All evaluation results of a category are summarised in a *summary*.
- All categories have their summaries combined into a *verdict*.

The Evaluation in my Division-Evaluation-Summary-Verdict strategy is the key process. To carry out it, I follow a combination of two major approaches:

Text analysis In this approach, the source code inside a source file is treated as pure text with certain structures. I divide different text structures, calculate the length of spaces and names, and analyse them locationally (e.g. if a space character appears before or after some non-space; if a text symbol starts a new line or not).

Meaning analysis In this approach, I analyse the meaning of the Java source code through *syntax analysis* (also called *parsing*) and partial *semantic analysis*. The meaning of source code is crucial to judging source code quality as the meaning directs the structures of the code.

In the next subsection, I will elaborate on what they mean in detail and how these methods are implemented in my software system.

2.1.3 Implementation of the methods in detail

After devising my quality evaluation method, I have developed a software system to apply the method. My software system evaluates Java source files. A source file is evaluated individually and independently by the system. The user inputs the path to such a source file to the system, and then the system puts the file through a series of processes and finally produces a verdict on it.

Java language version

A noteworthy matter is about the version of the Java language that my system processes. Java has a history of at least 20 years and more. As a consequence, it has 22 versions as of March 2024 [38]. Despite the numerous improvements and new features in the late versions of Java, a leading development software company *JetBrains* reports [39] that Java version 8 was still the most used version in the entire year 2023. Hence, Java version 8 [40] is the Java standard that my system works with.

Text analysis

The first process that the source file goes through is the **text analysis** process. In this process, the text of the file is divided into *tokens* first based on a set of rules made for the Java language. For example, the Java standard [40, Section 3.8] defines an identifier to be a mixture of letters and numbers[1] (with the exception that the first character cannot be a number). Therefore, whenever my system sees such a mixture of letters and numbers, it will treat the mixture as a token.

After the text analysis process, the source code text becomes a continuous sequence of tokens. The tokens are preliminarily categorised into the below categories based on their text characters.

JavaDoc Such a token is a Java comment that starts with a `/**`

Other Comment Such a token is a Java comment that is not a JavaDoc comment. That is, comments starting with a `//` or a `/*`.

White space Such a token is a continuous block of white space characters, including line terminators (but not contained in other tokens) [40, Section 3.6].

Identifier A Java identifier, as described above [40, Section 3.8].

Keyword A Java keyword [40, Section 3.9].

Literals A number, a string literal, a null, and more [40, Section 3.10].

Punctuation Java punctuation, such as `,` and `;`. Also called separators, as they separate statements. [40, Section 3.11].

Operator A Java operator. For instance, `+`, `-`, and `=`. [40, Section 3.12].

[1] The specification have a restriction on which Unicode characters are letters, which I implemented in the system exactly as-is.

The last five categories are collectively called `CodeBlock` tokens in my system. According to the standard [40, Section 3.5], my categorisation covers all possible kinds of Java source elements.

Another important operation in the text analysis process arranges and stores the tokens in different data structures according to their locations and lines in the source file. This operation enables the rest of my system to quickly query locational information about the tokens. My system supports these queries to the locational information of tokens:

1. Fast sequential access to tokens:
   (a) Get the token before a given token.
   (b) Get the token after a given token.
2. Fast random access to tokens:
   (a) Get a token given a global index.
   (b) Get a token given a line number and an index within the line.
3. Fast information access:
   (a) Given a token, get its global index.
   (b) Given a token, get its line number and index within the line.

Meaning analysis

After text analysis, the sequence of tokens and all textual information about them are sent into the meaning analysis process.

Meaning analysis 1: parsing The first major task of the meaning analysis process is to parse the sequence of tokens against the language's grammar into a *parsing tree*. This is also called the syntax analysis. A parsing tree is an ordered tree that links the tokens with the syntax structure, so it is also linked with the grammar of the language. The grammar in Java specification is written in a form of a *context-free grammar* (CFG) in formal computing language theory [40, Section 2.1].

A Java source file is defined as a `CompilationUnit` [40, Section 7.3], which is the *start symbol* (or *goal symbol*) of a CFG, indicating the root of the paring tree, where the parsing starts. The start symbol is a special `non-terminal` symbol. A non-terminal in CFG has a number of *productions*, which are the indented lines following the non-terminal in the Java specification. Each production defines an alternative that the non-terminal can have. According

to the productions, non-terminals are eventually expanded to a series of *terminals*, which are the basic characters for this grammar.

Like any other programming language, the grammar of Java must be unambiguous to ensure the existence of only one interpretation of a piece of valid source code. This unambiguity is in fact a very important property that ensures the uniqueness of the parsing tree of the sequence of tokens for the source code. J. Hopcroft, R. Motwani, and J. Ullman have defined in their famous textbook on computational languages that

> "A CFG $G = (V, T, P, S)$[2] is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labelled S and yield w." [41, p. 208]

To give an example, consider the *simplified* definition of some expressions in Java [40, Section 15.18]:

```
AdditiveExpression:
    MultiplicativeExpression (1)
    AdditiveExpression + MultiplicativeExpression (2)
    AdditiveExpression - MultiplicativeExpression (3)

MultiplicativeExpression:
    UnaryExpression (1)
    MultiplicativeExpression * UnaryExpression (2)
    MultiplicativeExpression / UnaryExpression (3)
    MultiplicativeExpression % UnaryExpression (4)
```

For the sake of exemplification, the term `UnaryExpression` is understood simply as an identifier in this example, and `AdditiveExpression` is used as the start symbol.

Given a simple input `a - b + c * d`, the unique parsing tree corresponding to it is drawn in Figure 2.1.

My system will match the sequence of tokens for an entire Java source file against the start symbol of the Java grammar, `CompilationUnit`, and produce a parsing tree with root labelled `CompilationUnit`, which typically is a much larger tree than that in the example.

Meaning analysis 2: partial semantic analysis After the parsing of the token sequence for a Java source file is complete, the second major task in my meaning analysis is performed. In this task, I walk the parsing tree

[2] where V is the set of non-terminals, T the set of terminals, P the set of productions, and S the start symbol.

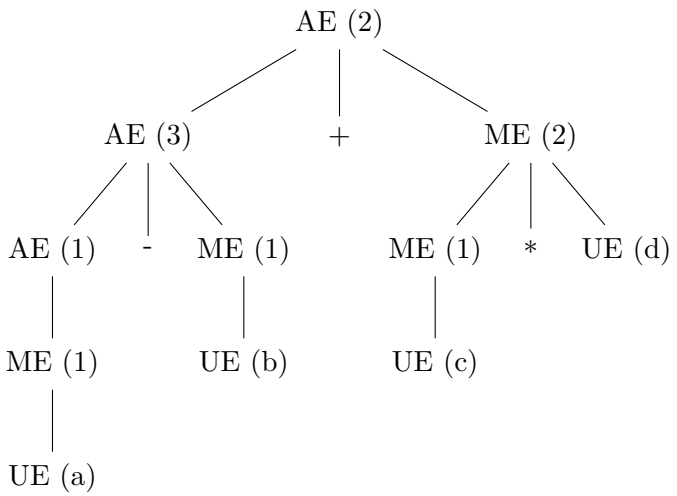

Figure 2.1: Parsing tree for `a-b+c*d` given the grammar

generated by the previous task to understand the meaning of the Java source code to some extent. Then, I assign the information gathered in this process to different kinds of code constructs that are the evaluated in the *Evaluation* step in my Division-Evaluation-Summary-Verdict strategy.

When I walk on the parsing tree, I traverse in a *depth-first* manner. That is, starting from the root, I traverse as deep as possible before I backtrack. Whenever I visit an interior (not a leaf) node of the tree (i.e. node of a non-terminal), I am notified of it and have information about the non-terminal, including which production is used for it. When I have completed the visit to such an interior node (i.e. all nodes in the subtree formed by the node and its children have been visited), I am notified as well. Finally, whenever I visit a leaf node (i.e. node of a sequence of terminals), I am notified, too.

Walking the parsing tree reveals the exact meaning of the source code. For example, the example parsing tree mentioned above in Figure 2.1 tells that the expression `a-b+c*d` in the above grammar results in the sum of `a-b` and `c*d`.

The division made by the text analysis is further divided after the meaning analysis. In particular, Identifiers and Operators are subdivided. I can know which identifiers are class names, method names, variable names, etc. After this division, the system will be ready for evaluating these divided primitives.

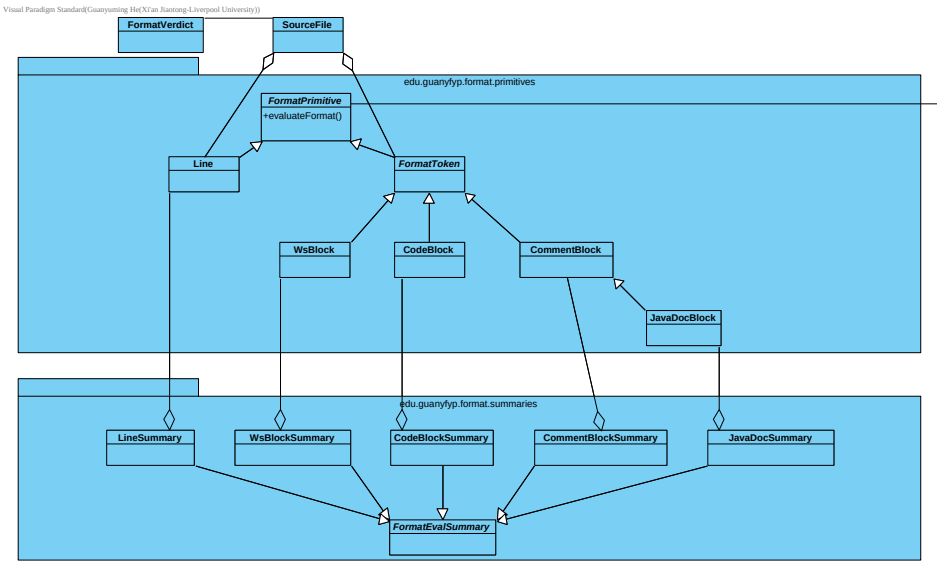

Figure 2.2: The core architecture of the system

Evaluation-Summary-Verdict

When the text analysis and then meaning analysis are complete, I finally have enough information to perform the last three steps, evaluation, summary, and verdict, in my Division-Evaluation-Summary-Verdict strategy.

I call all different kinds of primitive code constructs a `FormatPrimitive`. It is the unit on which an evaluation is performed. Then I define `FormatPrimitiveSummary` classes to give summary for each kind of primitives. Finally, the summary classes are combined in the main class, `SourceFile`, where the final verdict is given. Figure 2.2 is the class diagram of these classes, which form the core architecture of my software system.

In a *Summary*, all evaluation results for the same type of `FormatPrimitive` is summarised:

- Classifying all different sorts of code quality problems encountered in the evaluation results. For example, a `CodeBlock` may be subject to naming, length, spacing, and more problems.
- Putting the classified problems into different data structures for the verdict to use.

In the *Verdict*, the frequencies of encountering different kinds of problems are calculated. Because of the time limit, no further actions are taken in the current state of the system. However, in Section 2.2 I will explain how merely these frequencies will be significantly helpful for many further actions.

Software reuse — ANTLR 4

Parsing the Java language is not only difficult but also time-consuming. Java is a modern language with numerous features, and writing a parser involves many advanced algorithms.

Because of the time limit considerations and the fact that I am the sole developer, I decided to reuse some existing pieces of software to help me parse the Java language. After gathering enough information about contemporary tools, I chose ANTLR (version 4) [42] (Licensed with *The BSD License*, so I can use it as long as I reproduce the copyright of the BSD license, which is included in Appendix B.2 and in the license of my code repository on GitHub). ANTLR is a parser generator that generates a *parser*, given a Context Free Grammar written in the form of an ANTLR grammar. A parser transforms a sequence of text tokens into a parsing tree. In addition, ANTLR supports dividing text into tokens according to the grammar (this is also called the lexical analysis process).

The ANTLR grammar I employ in my project for Java version 8 is from an open source ANTLR grammar library [43]. The library is not licensed so it is available in public domain. I modified the grammar to remove some language features from later versions (e.g. Modules and Records), and also edited the parts of the grammar related to lexical analysis to ensure the tokens are categorised as I have described earlier.

Using the ANTLR grammar, ANTLR will generate three key classes for me:

1. `JavaLexer`, a *lexer*, which helps me convert a source file text into tokens.

2. `JavaParser`, a *parser*, which helps me parse the tokens into a parsing tree.

3. `JavaParserBaseListener`, a set of listener methods that help me respond to visiting/exiting different nodes of the parsing tree.

With these classes' help, I am able to create some classes to record the syntax structures and partial meaning (semantics) of the analysed source code. First, I create the class `SyntaxStructureBuilder`, which inherits from `JavaParserBaseListener` to build the syntax structure for the whole source code. A syntax structure is represented by the class `SyntaxStructure` and has some individual components like `SyntaxScope` and `SyntaxContext`. A class diagram of all these classes is illustrated in Figure 2.3.

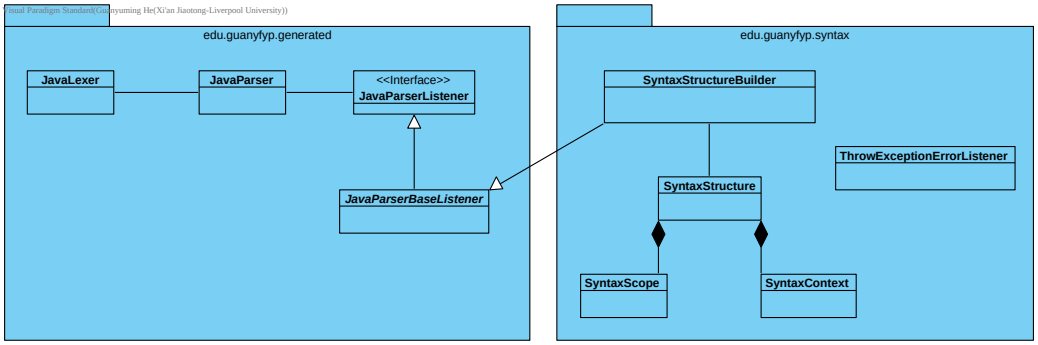

Figure 2.3: Classes related to text, parsing, and source meaning

The processes a source file will go through

Now that you know all the important classes of my system and their roles, I will introduce to you what a Java source file will undergo in my system.

1. The Java program starts in the `main` method of class `Main`, in which a `SourceFile` is initialised with the path to the source file.
2. Inside the constructor of the `SourceFile`,
   (a) Text analysis is performed. The text is divided into tokens, that is, `FormatTokens` in my system. Other primitive variables for format evaluation, like `Lines` are also created.
   (b) Syntax analysis is performed. The system uses a `JavaParser` to generate a parsing tree for the sequence of tokens.
   (c) I walk the parsing tree with `SyntaxStructureBuilder` to perform partial meaning analysis (also including partial semantic analysis), and the result of which is stored in a `SyntaxStructure` obtained from the builder.
   (d) The `SyntaxStructure` obtained can give information stored in smaller classes, like `SyntaxScope` and `SyntaxContext` (the context of a `FormatPrimitive`).
3. Now the `SourceFile` is constructed and has obtained all the necessary information. The `main` method calls its `analyse` method, which
   (a) For each `FormatPrimitive`, evaluates it.
   (b) After all the evaluations, summarises different types of them into summaries.

   (c) After all the summaries are available, composes them into a `Format Verdict`.

4. The `analyse` method returns the `FormatVerdict`, which is printed by the `main` method.

5. The system exits normally.

2.1.4 Software testing methodology

The software system has been thoroughly tested along with my development.

Test approaches

More than half of the tests are black-box tests done as unit testing. The rest of the tests are combinations of white-box testing, integration testing, and unit testing.

To perform black-box unit testing, I first define the specifications of various methods in my system as comments. Each of these methods is a *unit* in my testing. A such unit is tested separately and independently in some *test cases* dedicated to it. In each test case, inputs are given according to the specification for that method, and the outputs of the tested method are compared with the expected value from the specification.

Testing in units has several benefits for my project:

1. Because units are tested independently, I can easily refactor some units without needing to modify others.

2. Testing in units can boost problem discovery, thus allowing me to iterate my code more quickly.

Some pieces of code do not have large amounts of inputs but instead have drastic side effects. To test them, instead of black-box unit testing, I combine white-box testing, integration testing, and unit testing to ensure that all the side effects can be covered by my tests throughout most of the situations.

Test procedure After each update to the system (typically a Git commit), I would ascertain that new functionalities work correctly and old functionalities still work.

1. Add new tests for added or changed functionalities.

2. Run the new tests added in the update as well as all previous tests.

Test data

For test data, because the inputs to my software system are Java source code, I created Java source files inside the `test_data` directory of my project. There are a total of 22 different Java source files in the folder. They vary from boundary test data such as an empty file, to data for complex tests that contains a large amount of different Java code constructs.

Figure 2.4 displays the contents in the `test_data` directory as well as a part of a test data file.

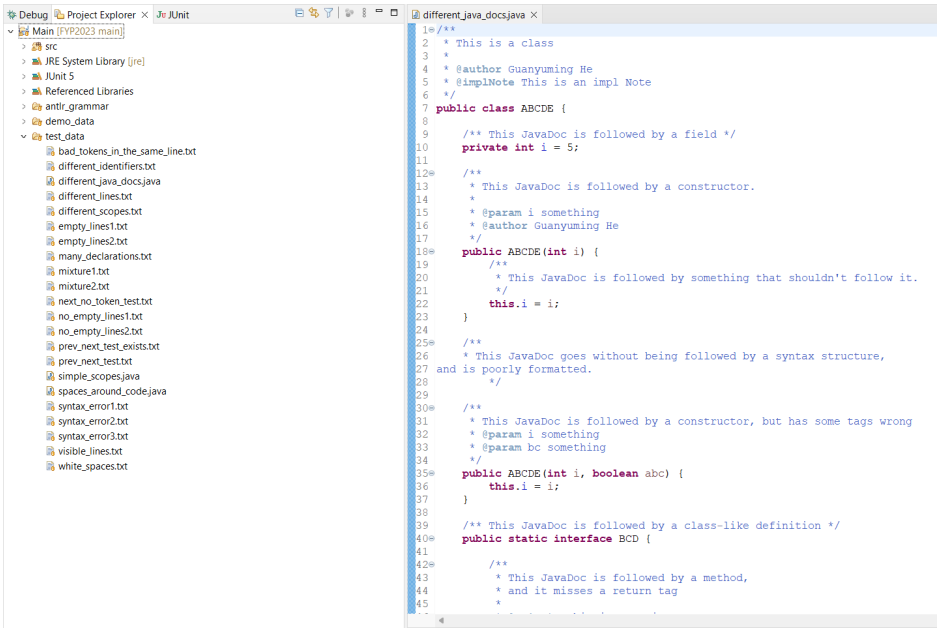

Figure 2.4: Demonstration of the test data

Test tools

JUnit 5 [44] was used as the framework of my software testing. It provides powerful utilities to help me in

1. Comparing outputs with expected values
2. Organising tests in units
3. Automatic execution of tests (with a few clicks)

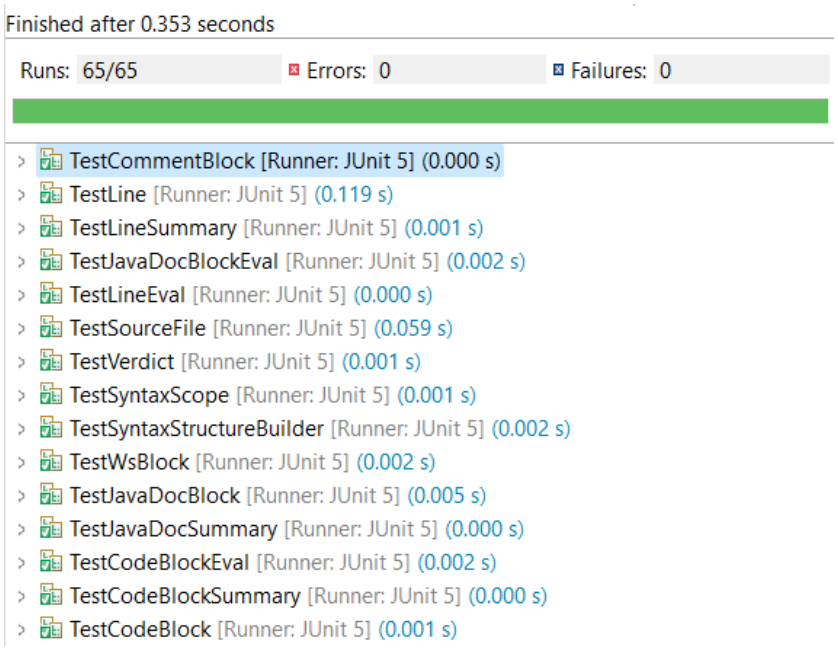

Figure 2.5: All 65 test cases can pass

Test coverage

For each important class, there is at least one test class with some test cases to cover its various methods. There are a total of 65 test cases, which can all pass now, as recorded in Figure 2.5.

If a class uses many other classes, then the test cases for that class are usually integration tests. Otherwise, they are usually unit tests.

2.1.5 Development tools and environment

The software development happened on two computers. One is my personal laptop, and another is my personal desktop. Most of the development was done on the desktop and the rest was done on the laptop. Table 2.1 shows the development environment of the two computers.

The whole development was carried out in *Eclipse IDE for Java developers, 2023-06* [45]. As mentioned earlier, two external Java libraries were used to assist the development and testing. The tool *ANTLR 4* [42] was used to help my system understand the meaning of Java source code, while another tool *JUnit 5* [44] was used to test my system.

The development progress was carried between the two computers by the version control system Git [46] and a remote repository on GitHub. After

Component	Desktop	Laptop
CPU	Intel Core i5 13600K	AMD Ryzen 9 7940HS
GPU	NVIDIA RTX 4070Ti	AMD Integrated Graphics
Memory	64 GB DDR5	32 GB DDR5
OS	Windows 11 64bit	Windows 11 64bit
IDE	Eclipse IDE for Java	Eclipse IDE for Java

Table 2.1: Environment of the development machines

completing some work on one computer, I execute

```
cd <PROJECT_ROOT>
git add .
git commit -m "<COMMIT MESSAGE>"
git push origin <BRANCH_NAME>
```

to push the work onto the remote repository on GitHub. When I am on the other computer, I execute

```
git fetch origin
git switch <BRANCH_NAME>
git pull origin <BRANCH_NAME>
```

to synchronise with the work I pushed to the remote repository.

The above tools have been used for my development, and I used a different set of tools to compose my thesis, that is, this paper. The paper is typeset in LaTeX, with the LaTeX package Ti*k*Z's help to draw most of the simple graphs and trees, BibTeX's help to make the references, and the software Visual Paradigm [47] has been used to produce the class diagrams shown in the thesis.

The process to generate my paper will not be shown here.

2.2 Results

My devised code quality evaluation methods and implemented software system have satisfied all of my goals and objectives defined in Section 1.1.2. In this section, I explain how the results have achieved these objectives. Then, I showcase the result of running the system on a piece of code full of code quality problems. At length, I use the Git history to give a glimpse of how much work has been done to develop the software system.

2.2.1 I have achieved my *A1O1* and *A1O2*

- *A1O1*: The resulting software system can judge code quality from these aspects:
  - Length
    - If a line is too long.
    - If an identifier is too long, or if it is too short in some contexts. For example, a one character identifier `i` is commonly used as a loop variable, but is a bad choice for a class field name.
    - Note that the standard for "too long" and "too short" can be customised, as will be explained later.
  - Spacing
    - If there is a space before/after most operators so that it does not appear crowded. For example, `a = b` is generally more good looking than `a=b`.
    - If there is no space before/after some operators. For instance, `i++` is normal while `i ++` looks strange.
    - If there is a space/newline after some punctuation. `void f(int a, int b, int c)` is generally more good looking than `void f(int a,int b,int c)`.
    - Note that which operators follow which conventions can be customised, as will be explained later.
  - Naming
    - If a (generic or not) class, interface, or enum name is Pascal named ("*nouns, in mixed case with the first letter of each internal word capitalized*"), which is defined by Oracle's Java naming convention [48, Classes, Interfaces].
    - If a variable is camel named. They are "*in mixed case with a lowercase first letter. Internal words start with capital letters*" [48, Variables].
    - If a constant is named in "*all uppercase with words separated by underscores*" [48, Constants].
    - Note that all these naming standards can be customised, as will be explained later.
  - Indentation
    - A line's indentation in my system is calculated as the visual length of all the white spaces before the first non-space character in that line.

    - A line's indentation is correct if and only if it is $4 \times s$, where s is the scope level the line is in. If the line is not in any scope, then $s = 0$. Otherwise, s is the number of scopes enclosing it.

- JavaDoc Comments
    - If a piece of source code has comments at all. Not having any comment is a bad coding habit.
    - If a JavaDoc has all the tags (a description starting with a @) it should have. For example, if the system finds that a JavaDoc is followed by a function with parameters `a, b`, then the JavaDoc should have tags `@param a` and `@param b`.
    - If a JavaDoc has none of the tags it should not have. For instance, a JavaDoc should not have a `@return` tag if it is followed by a class definition.
- Coding Style
    - The system is able to decide if some coding style is consistent throughout a source file. To exemplify, if the system finds

```
public void f() {
 ...
}
...
public voiod g()
{
 ...
}
```

      , then the system will conclude that the scope coding style is inconsistent in the source file.

From the above discussion, it is clear that all those approaches are transparent and deterministic, which achieves my *A1O2*.

2.2.2 I have achieved my *A2O1* and *A2O2*

By developing a software system that employs these methods, which I explained in Section 2.1.3, I have achieved my *A2O1* — developing a software for my proposed code-quality evaluation methods.

As for *A2O2*, I have ensured that *many* of the achieved features listed above can be customised and extended. To support customisability, in

each customisable `FormatPrimitive` class, I have created a nested static `Settings` class, which contains various settings entries that will affect how the `evaluateFormat()` method of that `FormatPrimitive` class works.

For example, in class `CodeBlock` I have the following `Settings` class.

```
public static final class Settings
{
 // Default values come from the Java coding convention by oracle.

 // Identifier settings
 public int longestIdentifierLength = 15;
 public int shortestIdentifierLength = 2;
 public NamingStyle desiredClassNamingStyle = NamingStyle.PASCAL_CASE;
 public NamingStyle desiredMethodNamingStyle = NamingStyle.CAMEL_CASE;
 public NamingStyle desiredVariableNamingStyle = NamingStyle.CAMEL_CASE;
 public NamingStyle desiredConstantNamingStyle = NamingStyle.UPPERCASE_UNDERSCOR

 // Punctuation settings
 public boolean checkPunctuationSpacesAround = true;

 // Operator settings
 public boolean checkOperatorSpacesAround = true;
 public boolean checkSpaceAroundIncDec = true;
}
```

Reading the code, it is evident that simply altering the value of some settings entries would change the behaviours of length, naming, spacing evaluations within `CodeBlock`. However, these entries still do not make the whole evaluation process customisable. Nevertheless, one can extend the evaluation and in turn, extend the settings. That is related to the system's extensibility.

In fact, the extensibility is immediately achieved through the framework that I have established in the core architecture of the system, which I have demonstrated in previous sections in Figure 2.2. To extend and modify an evaluation of some type of `FormatPrimitive`, one simply has to inherit from that type's class and override its `evaluateFormat()` method. Besides, one can then add a new `Settings` class in the subclass created to extend the customisation.

This customisability and extensibility achieve my *A2O2*.

2.2.3 Results of running the system on some code

In this subsection, I showcase how the system will work on a piece of Java source code full of code quality problems, which I manually crafted for demonstration.

Figure 2.6 shows the piece of code with the annotations I made manually according to the system's result on it. In the figure, each different colour represents a different type of code quality problem that the system found on the code, and each occurrence of such a problem is circled using the colour. Figure 2.7 records the output of the system on the same piece of code.

Figure 2.6: Annotation of code quality problems found on the example

In contrast, running it on a piece of code without code quality problems gives zeros on all problem frequencies in the verdict, as shown in Figure 2.8.

Explanation of the output

The output in Figure 2.7 is the `toString` result of the verdict given for the bad source code. It

1. Prints the frequencies of the various problems.
2. Prints all of the summaries. Each summary will
   (a) Print all the problems in the summary, in categories.

```
Verdict:
identifierLengthProblemFrequency = 0.3125
identifierNamingProblemFrequency = 0.1875
spacingProblemFrequency = 0.2
inconsistentScopeStyleFrequency = 1.0
badJavaDocFrequency = 1.0
lineLengthProblemFrequency = 0.041666668
lineIndentationProblemFrequency = 0.16666667
hasCommentsAtAll = true

CodeBlocks summary:
Identifier length problems:
        CodeBlock "badly_named_class" at 13, line 7
        CodeBlock "sssssssssssssssssssssssssssssssssssssssssssssss" at 15, line 12
        CodeBlock "badly_named_class" at 11, line 18
        CodeBlock "i" at 16, line 8
        CodeBlock "i" at 33, line 18
Badly named identifiers:
        CodeBlock "badly_named_class" at 13, line 7
        CodeBlock "Integer2" at 20, line 10
        CodeBlock "badly_named_class" at 11, line 18
Space and newlines around problems:
        CodeBlock "," at 48, line 18
        CodeBlock ";" at 22, line 21
        CodeBlock "=" at 36, line 21
        CodeBlock "}" at 4, line 23
Inconsistent scope styles found.
Scopes in new line style:
        CodeBlock "{" at 4, line 19
Scopes in old line style:
        CodeBlock "{" at 31, line 7

Lines summary:
Too long lines:
        Line 24
Badly indented lines:
        Line 10
        Line 12
        Line 21
        Line 23

JavaDocs summary:
Bad JavaDocs:
        JavaDoc Comment at 0, line 1
        JavaDoc Comment at 4, line 14
```

Figure 2.7: Output of the system on the example

```
Verdict:
identifierLengthProblemFrequency = 0.0
identifierNamingProblemFrequency = 0.0
spacingProblemFrequency = 0.0
inconsistentScopeStyleFrequency = 0.0
badJavaDocFrequency = 0.0
lineLengthProblemFrequency = 0.0
lineIndentationProblemFrequency = 0.0
hasCommentsAtAll = true
```

Figure 2.8: Output of the system on good code

(b) For each category, first print a description.

(c) Then print each problem in the category. A problem will

i. Print what it is,

ii. and which `FormatPrimitive` caused it (its location, index, etc.).

2.2.4 How much work did I put in the software

The version control system Git can give a glimpse of how much work I have done into developing the system. Running the following commands on the directory of my project will instruct Git to compare to current state of my source files with the initial state (when I first created the project).

```
git switch main
git diff --stat b154f0f
```

, where `b154f0f` is (the hash of) my initial commit calculated by Git.

The output concludes that `77 files changed, 26933 insertions(+), 1 deletion(-)`, comparing the current state with the initial state. It indicates that, excluding the files generated by ANTLR (inside `.../generated/`, having about 15,000 lines of code), I have written more than 10,000 lines of code to develop the system.

Unfortunately, this Git command can only compare two versions of my source repository, which cannot show how many lines of code I have changed and deleted during the process (it only shows `1 deletion(-)` in the output). Nevertheless, it is proof of my hard work put into developing the system.

Chapter 3

Conclusion and Future Work

In this chapter, I will conclude all my works in the first section and reflect on what I have learnt during the year, working on the project. In the second section, I will point out the significant limitations of my system and suggest possible future improvements and related works based on it.

3.1 Conclusion

In conclusion, during the two semesters, I have successfully

1. Conducted a literature review on the current literature and devised the project based on the gaps and methods identified in the review.
2. Invented a Division-Evaluation-Summary-Verdict strategy to transparently and fairly evaluate source code quality.
3. Developed a software system that applies my Division-Evaluation-Summary-Verdict strategy to evaluate the source code quality of Java 8.
4. Nearly thoroughly tested the software system.

3.1.1 What I learnt from the project

In this project, both research tasks (the literature review and proposing code-quality evaluation methods) and engineering tasks are quite heavy. Whether I want to be a researcher or an engineer in the future, if I want to be a good one, then I must go through trial by fire, and this final year project is part of it for me.

Researching perspective

From a researching perspective, conducting the literature review and coming up with the ideas helped me in these aspects:

1. Most obviously, I gained the state-of-the-art knowledge in the automatic grading field.

2. Less obviously, they improved my academic writing skills drastically. To clarify, my theoretical knowledge of academic writing was enough before reading this amount of papers, but one year before I definitely could not write so good a paper as this thesis. Why?

   I believe one critical reason is that I was not experienced enough to stand in a reader's position to proofread my papers. As I fully review so many papers during this final year project, I formed a deeper understanding of how a well-written paper could make it extremely easy for a reader to understand its contents, even if the reader is not an expert in the field of the paper.

   Therefore, my Interim Report was written better than my previous academic writings, and you can witness how this thesis paper of mine improved over my Interim Report (e.g. try comparing the Introduction Section here with that in my Interim Report).

3. Finally, it is one of the few times that I have used my knowledge to solve real-world problems. Another time was during my summer SURF last year, although that time I was working with other people as well as some PhD students who came up with most of the ideas.

   This time, I am the only researcher, who is responsible for all the ideas (the research direction was provided by my supervisor, who also helped me in forming some ideas). This is definitely challenging, but as I have successfully pulled it off, my insight in creative work will increase, which is very important for a researcher.

Engineering perspective

This software system is definitely large for the current me. In fact, it is larger (in terms of lines of code) than any group project I have done. Having successfully developed such a system helps me in these aspects:

1. I acquired more experience in object-oriented programming. Although the resulting classes are not flawless, I am confident to say they are well-modularised and have clear and separate roles.

2. Learning ANTLR while developing was definitely challenging. If you examine my Git history, then you will notice a big refactoring happening during the winter vacation. I made many significant changes to the system during the year, and these experiences will reinforce my engineering ability in dealing with changes and unknown factors in the future.

3. Testing the whole system was another notable challenge. Trying to complete the development and testing within the time limit, I had to carefully test important methods first, instead of blindly following theoretical test procedures that would have drained my time without having good coverage.

   Unlike a researcher, an engineer cannot follow theories only. Coping with these practical constraints enhanced my ability to balance between theories and reality.

3.2 Future Work

3.2.1 Current limitations

Despite the success of my Division-Evaluation-Summary-Verdict strategy and the software system, because of the time limit and other various reasons, it has some significant drawbacks.

1. Although it covers a broad range of code quality problems, it still cannot be said to cover most of them, as there are so many different code quality problems. For example, although my system can already obtain the list of *modifiers* (e.g. public, private, static, and final) for an identifier, it does not do anything about them currently. One code quality problem could be that all fields of a class are public, which violates the abstraction principle.

2. The Verdict now only calculates the frequency of different kinds of problems, although it is a good foundation for future work.

3. Because of the time limit, the system now only supports Java version 8.

4. Although the system achieves transparency compared with machine learning approaches, it requires the developers to develop new classes for supporting different programming languages, because they have different syntax structures from Java. However, machine learning systems

may only require some tweaking in parameters to work successfully on a new language.

5. Its speed has not been specifically optimised. However, even if its speed is optimised, parsing is still a time consuming process. Therefore, its speed may be lower than that of some machine learning systems.

6. Although one motivation of the project is to improve existing auto-graders, I have not had the time to integrate the software into an existing auto-grader. Nevertheless, because the system is independent, it should be easy to do so in the future.

3.2.2 Possible future work directions

Here I suggest some possible directions to improve the aforementioned drawbacks in the future.

1. Despite the fact that my system does not cover most of the large set of code quality problems, one important contribution of my final year project is the Division-Evaluation-Summary-Verdict strategy and the framework of my software, which enables people to easily add more problem coverage using the framework. By cloning the source code of my project on GitHub https://github.com/Little-He-Guan/FYP2023, people who need it can use the framework and easily add the coverage for code quality problems they need to address.

2. The frequencies of different kinds of problems is a good foundation on marking the code quality. This can be as simple as assigning weight to different categories of code quality problems and subtracting the product of the weight and frequencies from the marks.

3. Because of the nature of the system, making it work on different languages will require:

   - An ANTLR grammar written for it
   - Study the syntax so that one knows how to analyse it when traversing the parsing tree.

   Fortunately, the open source project where I obtained the ANTLR grammar for Java [43] contains about 100 different ANTLR grammars for programming languages, making the first step as trivial as a few minor modifications to the grammar. Still, one would have to study the

syntax and contemplate how to analyse the meaning when traversing the parsing tree. That is kind of inevitable.

4. Unfortunately, parsing, or finding a way to derivate a parsing tree from a context-free grammar (CFG) for a given string, is quite complex by nature. Theoretically, one of the fastest parsing algorithms for CFG has a complexity of about $O(n^{2.37})$ [49], but that requires the grammar to be in a special form, Chomsky normal form, which is not the case for the grammars of many programming languages, including Java.

   Current prominent parsers usually use a LL-parsing algorithm. The software I use for parsers, ANTLR, uses a variation of this algorithm, *ALL*, which can be shown to have a complexity of $O(n^4)$, but which can "perform linearly on grammars used in practice" [50]. Unless there appears an algorithm with much less time complexity, the speed of my system cannot be drastically improved in the future.

5. Although the software system is not integrated into any auto-grader, its independent nature makes integration easy to perform. One simple solution is to process a piece of source code individually and simultaneously by both an existing auto-grader and my system, and finally combine the marks given by both of them in some way.

References

[1] Oxford English Dictionary, *Digitalization, N. (2), Sense 2.* Oxford University Press, 2023, https://doi.org/10.1093/OED/4102068919.

[2] M. Castells, *The rise of the network society.* John wiley & sons, 2011.

[3] J. Brodny and M. Tutak, "Assessing the level of digitalization and robotization in the enterprises of the european union member states," *PloS one*, vol. 16, no. 7, p. e0254993, 2021.

[4] V. G. Khalin and G. V. Chernova, "Digitalization and Its Impact on the Russian Economy and Society: Advantages, Challenges, Threats and Risks," *Administrative Consulting*, no. 10, 2018. [Online]. Available: https://ideas.repec.org/a/acf/journl/y2018id943.html

[5] M. Muktiarni, I. Widiaty, A. G. Abdullah, A. Ana, and C. Yulia, "Digitalisation trend in education during industry 4.0," *Journal of Physics: Conference Series*, vol. 1402, no. 7, p. 077070, dec 2019. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1402/7/077070

[6] T. Camp, W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker, and S. Zweben, "Generation cs: the growth of computer science," *ACM Inroads*, vol. 8, no. 2, p. 44–50, may 2017. [Online]. Available: https://doi.org/10.1145/3084362

[7] J. Tims, S. Zweben, and Y. Timanovsky, "Acm-ndc study 2016–2017: fifth annual study of non-doctoral-granting departments in computing," *ACM Inroads*, vol. 8, no. 3, pp. 48–62, 2017.

[8] G. E. Forsythe and N. Wirth, "Automatic grading programs," *Communications of the ACM*, vol. 8, no. 5, pp. 275–278, 1965.

[9] J. Caiza and J. Del Alamo, "Programming assignments automatic grading: Review of tools and implementations," in *INTED2013 Proceedings*,

ser. 7th International Technology, Education and Development Conference. IATED, 4-5 March, 2013 2013, pp. 5691–5700.

[10] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, "Automatic grading and feedback using program repair for introductory programming courses," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 92–97. [Online]. Available: https://doi.org/10.1145/3059009.3059026

[11] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: An approach based on formal semantics," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2019, pp. 126–137.

[12] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 110–115. [Online]. Available: https://doi.org/10.1145/3059009.3059061

[13] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," in *Proceedings of the 20th Australasian Computing Education Conference*, ser. ACE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 73–82. [Online]. Available: https://doi.org/10.1145/3160489.3160500

[14] Cornell University. Top publication venues in computer science. Accessed 24th Sept. 2023. [Online]. Available: https://www.cs.cornell.edu/andru/csconf.html

[15] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, and H. Watanabe, "A framework for measuring and evaluating program source code quality," in *Product-Focused Software Process Improvement*, J. Münch and P. Abrahamsson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 284–299.

[16] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *2010*

14th European Conference on Software Maintenance and Reengineering, March 2010, pp. 156–165.

[17] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, May 2019, pp. 2–12.

[18] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 31–35.

[19] N. Akhter, S. Rahman, and K. A. Taher, "An anti-pattern detection technique using machine learning to improve code quality," in *2021 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*, Feb 2021, pp. 356–360.

[20] V. Raychev, "Learning to find bugs and code quality problems - what worked and what not?" in *2021 International Conference on Code Quality (ICCQ)*, March 2021, pp. 1–5.

[21] Simon, B. Cook, J. Sheard, A. Carbone, and C. Johnson, "Student perceptions of the acceptability of various code-writing practices," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 105–110. [Online]. Available: https://doi.org/10.1145/2591708.2591755

[22] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.

[23] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[24] T. Sedano, "Code readability testing, an empirical study," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, April 2016, pp. 111–117.

[25] S. M. Tisha, "Machine-learning approaches for developing an autograder for high school-level cs-for-all initiatives," 2023.

[26] S. M. Tisha, R. A. Oregon, G. Baumgartner, F. Alegre, and J. Moreno, "An automatic grading system for a high school-level computational thinking course," in *Proceedings of the 4th International Workshop on Software Engineering Education for the Next Generation*, ser. SEENG '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 20–27. [Online]. Available: https://doi.org/10.1145/3528231.3528357

[27] E. Hegarty-Kelly and D. A. Mooney, "Analysis of an automatic grading system within first year computer science programming modules," in *Proceedings of 5th Conference on Computing Education Practice*, ser. CEP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 17–20. [Online]. Available: https://doi.org/10.1145/3437914.3437973

[28] Y. Kanellopoulos, P. Antonellis, D. Antoniou, C. Makris, E. Theodoridis, C. Tjortjis, and N. Tsirakis, "Code quality evaluation methodology using the iso/iec 9126 standard," *International Journal of Software Engineering & Applications*, vol. 1, no. 3, p. 17–36, Jul. 2010. [Online]. Available: http://dx.doi.org/10.5121/ijsea.2010.1302

[29] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 83–92.

[30] J. Tharmaseelan, K. Manathunga, S. Reyal, D. Kasthurirathna, and T. Thurairasa, "Revisit of automated marking techniques for programming assignments," in *2021 IEEE Global Engineering Education Conference (EDUCON)*, April 2021, pp. 650–657.

[31] F. Jurado, M. A. Redondo, and M. Ortega, "Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice," *Journal of Network and Computer Applications*, vol. 35, no. 2, pp. 695–712, 2012, simulation and Testbeds. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804511002050

[32] F. K. Sinambela, "An integrated automatic-grading and quality measure for assessing programming assignment," *JATISI (Jurnal Teknik Informatika dan Sistem Informasi)*, vol. 8, no. 3, pp. 1508–1514, 2021.

[33] R. Green and H. Ledgard, "Coding guidelines: Finding the art in the science: What separates good code from great code?" *Queue*, vol. 9, no. 11, p. 10–22, nov 2011. [Online]. Available: https://doi.org/10.1145/2063166.2063168

[34] I. Bergström and A. F. Blackwell, "The practices of programming," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2016, pp. 190–198.

[35] V. Barstad, M. Goodwin, and T. Gjøsæter, "Predicting source code quality with static analysis and machine learning," in *Norsk IKT-konferanse for forskning og utdanning*, 2014.

[36] H.-M. Chen, B.-A. Nguyen, and C.-R. Dow, "Code-quality evaluation scheme for assessment of student contributions to programming projects," *Journal of Systems and Software*, vol. 188, p. 111273, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000358

[37] "ISO/IEC 9126:2001 Software engineering Product quality," International Organization for Standardization, Geneva, CH, Standard, 2001.

[38] Oracle, "Oracle Releases Java 22," 2024, accessed 29 April 2024. [Online]. Available: https://www.oracle.com/news/announcement/oracle-releases-java-22-2024-03-19/

[39] JetBrains, "Developer Ecosystem 2023: Java," accessed 2 March 2024. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/java/

[40] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2015) The Java Language Specification, Java SE 8 Edition. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[41] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., M. Hirsch, M. Goldstein, and K. Harutunian, Eds. Greg Tobin.

[42] T. Parr, "ANTLR," 2023. [Online]. Available: https://www.antlr.org/

[43] GitHub contributors. (2023) Grammars-v4, a collection of formal grammars written for ANTLR v4. [Online]. Available: https://github.com/antlr/grammars-v4

[44] The JUnit team, "JUnit 5," 2024. [Online]. Available: https://junit.org/junit5/

[45] The Eclipse Foundation, "Eclipse IDE for Java Developers," 2023. [Online]. Available: https://www.eclipse.org/downloads/packages/release/2023-06/r/eclipse-ide-java-developers

[46] The Git Community, "Git," 2024. [Online]. Available: https://git-scm.com/

[47] The Visual Paradigm Team, "Visual Paradigm," 2024. [Online]. Available: https://www.visual-paradigm.com/

[48] Oracle, "9 — Naming Conventions," april 1999. [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

[49] L. G. Valiant, "General context-free recognition in less than cubic time," *Journal of Computer and System Sciences*, vol. 10, no. 2, pp. 308–315, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000075800468

[50] T. Parr, S. Harwell, and K. Fisher, "Adaptive ll(*) parsing: the power of dynamic analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 579–598. [Online]. Available: https://doi.org/10.1145/2660193.2660202

Appendix A

How to Get Source Code

The source code for my project is completely contained in this GitHub repository of mine https://github.com/real-guanyuming-he/FYP2023. To download this repository to your local machine, make sure you have Git installed locally, and execute the following command:

```
git clone https://github.com/real-guanyuming-he/FYP2023.git
```

Appendix B

Software Licenses

This appendix contains the license of my software system as well as the licenses of all the software systems I reused (until now) in my project. Theoretically I do not need to include the license of mine here, but I do need to reproduce part of the license of the systems I utilised in the project. For consistency, I also included my license here.

B.1 My project License

B.2 ANTLR 4 License