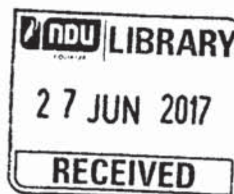
Towards an Automated Analysis of the Quality of Source Code Comments

By
Mireille J. Haddad

Faculty of Natural and Applied Sciences
Department of Computer Science
Notre Dame University – Louaize

A thesis submitted in partial fulfillment of the requirements for the Master of Science in Computer Science

June 2017


NDU LIBRARY
27 JUN 2017
RECEIVED

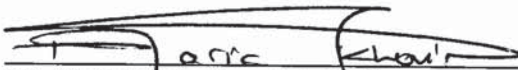
Towards an Automated Analysis of the Quality of Source Code Comments

By

Mireille J. Haddad

Approved by:

Dr. Pierre A. Akiki: Assistant Professor of Computer Science
Advisor.



Dr. Marie Khair: Associate Professor of Computer Science
Member of Committee.

Dr. Nazir Hawi: Associate Professor of Computer Science
Member of Committee.

Date of Thesis Defense:
June 22, 2017

Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Lebanon, June 2017

Mireille J. Haddad

Abstract

Maintenance is the most costly phase of the software life cycle. The maintenance cost of a program is estimated to be over 80% of its total life cycle costs (Erlikh, 2000). Since most of the maintenance time is devoted to understanding the program itself, program comprehension becomes essential. Often, a large fraction of the maintenance time is spent on reading code to understand what functionality of the program it implements.

An insufficiently documented source code can be challenging for developers to understand and maintain. A clear and concise documentation can help developers to inspect and understand their programs. Unfortunately, one of the major problems faced by developers during maintenance is that documentation is often not available or not useful.

This thesis provides a heuristic approach for an automatic analysis and assessment of source-code comments by parsing by using a parser generator tool called ANTLR. This approach measures the semantic similarity between the comment content and its corresponding entity identifier name. An algorithm was developed for splitting identifiers into component terms and computes the similarity percentage between the useful content of the comment and the identifier. The developed approach categorizes comments as follows: Scary noise, noise, normal with minor similarity, probably meaningful, empty, and TODO.

A study was carried out to evaluate the ability of the proposed approach to adequately assess source-code comments. In this study the source code of the Eclipse open source Integrated Development Environment (IDE) was parsed. The results showed that more than 50% of the comments fall into the category of empty comments and spread over 62% of the whole project files. Only 18% of the comments were of a high quality and around 20% of the files contain noise comments. Most Class and Interface identifiers have comments while more than 50% of the methods lack comments.

Keywords: software maintenance, program comprehension, source-code comments, source-code parsing, ANTLR.

Table of Contents

Abstract iv
Table of Contents v
List of Figures viii
List of Tables x
List of Abbreviations xi
Acknowledgments xii

Chapter 1: Introduction and Problem Definition 1

1.1 Introduction to the General Problem 1
1.2 Problem Definition 3
1.3 Research Objectives 4
1.4 Approach and Main Results 5
1.5 Thesis Organization 6

Chapter 2: Background and Motivation 7

2.1 Introduction 7
2.2 Parsing Source Code to Improve Maintainability 9
2.2.1 Modernizing Legacy Software Systems 10
2.2.2 Refactoring 10
2.3 Tools for Parsing Source Code 11
2.3.1 ANTLR 12
2.3.2 JavaCC 12
2.3.3 SableCC 13
2.4 Another Tool for Language Recognition (ANTLR) 13
2.4.1 Lexical Rules Implementation 15
2.4.2 Syntactic Rules Implementation 16
2.4.3 Semantic Checking and Error Handling 17
2.5 Previous Work on Comment Analysis 18
2.6 Research Motivation 20

Chapter 3: Comment Parsing and Analysis 22

3.1 Introduction 22

3.2 Categorization of Comments....23
3.2.1 Documentation Comments....23
3.2.2 Copyright Comments....24
3.2.3 Task Comments....24
3.3 Analysis Heuristics for Source Code Comments....24
3.3.1 Identifier Names Tokenization Heuristic....24
3.3.2 Word Match and Similarity Heuristic....25
3.3.3 Location Heuristic....25
3.3.4 Lines Count Heuristic....25
3.4 Parsing Source Code....25
3.4.1 Parsing Concepts....26
3.4.2 Parsing Algorithm....28
3.4.3 Parse Tree....28
3.5 Parser Generator....30
3.5.1 Generating Lexers and Parsers with ANTLR....30
3.5.2 Grammar Writing....32
3.5.3 Building Grammar....33
3.5.4 Parser of Java Language....35
3.6 Interpreter Implementation....36
3.6.1 ANTLR Generated Files....37
3.6.2 Interpreter File....37
3.6.3 Error Handling....39

Chapter 4: Applying Comment Parsing and Analysis to an Open-Source Software....41

4.1 Introduction....41
4.2 Retrieving Information....41
4.3 Semantic Similarity Approach....43
4.3.1 Preprocessing....43
4.3.2 Stemming....45
4.3.3 Computing Similarity....45
4.4 Testing the Approach with an Open Source Software....47
4.4.1 Analyzing the Source Code....47
4.4.2 Results....49
4.4.3 Result Discussion....55
4.4.4 Threats to Validity and Limitations....56

Chapter 5: Conclusions and Future Work....57

5.1 Main Problem Addressed in this Thesis....57
5.2 Contributions....58
5.3 Future Work....59

Bibliography 61
Appendix A: Grammar File...65

List of Figures

Figure 2.1: The overall framework of ANTLR (Liu et al., 2008) 14
Figure 2.2 Abstract Syntax Tree (AST) Construction (Turuntaev, 2014, p. 44) 14
Figure 2.3 Declaration of Identifier (Parr & Harwell, 2013) 16
Figure 2.4 Syntactic Rule of Java Interface (Parr & Harwell, 2013) 17
Figure 2.5 Calling ANTLR's Interface Functions (Parr, 2013, p. 239) 18
Figure 3.1 A Simple Java Program 26
Figure 3.2 Fragment of Java Grammar 26
Figure 3.3 Parse Tree Fragment Produced by a Parser 29
Figure 3.4 The Relationship between Characters, Tokens, and ASTs (Parr, 2013, p. 10) ... 31
Figure 3.5 Grammar File General Overview 33
Figure 3.6 HelloWorld Class Example 34
Figure 3.7 ANTLR Console 36
Figure 3.8 Implementation Phases 36
Figure 3.9 Grammar Global Variables Declaration 39
Figure 3.10 Example with Syntax Error 40
Figure 3.11 Syntax Error Message 40
Figure 4.1 Retrieving Information upon Parsing 43
Figure 4.2 Cleaning Messy Comment 45
Figure 4.3 Algorithm for Semantic Similarity Computation 46
Figure 4.4 Support Tool – Selecting the Target Project 48
Figure 4.5 Support Tool – Showing the Output after Parsing and Analysis 49
Figure 4.6 Comment Category Percentage 49
Figure 4.7 Noise Comment Example 50
Figure 4.8 Scary Noise Comment Example 50
Figure 4.9 Empty Comment Example 51
Figure 4.10 Distribution of the Different Types of Comments across the Source-Code Files 52
Figure 4.11 Number of Each Type of Comment per Source-Code File 53

Figure 4.12 Descriptive Statistics of Comment Types per File 54
Figure 4.13 Classes with/without Comments by Percentage 54
Figure 4.14 Interface with/without Comments by Percentage 55
Figure 4.15 Methods with/without Comments Percentage 55

List of Tables

Table 2.1 Strengths and Shortcomings of Previous Research on Code Comment Analysis 20
Table 3.1 ANTLR EBNF Operators 27
Table 4.1 Comments Categories 47

List of Abbreviations

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
BNF	Backus Naur Form
EBNF	Extended Backus–Naur Form
CFG	Context-Free Grammar
LL	Leftmost derivation
LR	Rightmost derivation
LALR	Look-Ahead LR
LHS	Left-Hand Side
RHS	Right-Hand Side
AWB	Architect's Workbench
POS	Part of Speech
IDE	Integrated Development Environment
GUI	Graphical User Interface

Acknowledgments

I would like to express my sincere gratitude and deep appreciation to Dr. Pierre Antoine Akiki for his unlimited guidance, assistance, encouragement and tremendous patience throughout this research. Besides my advisor, I would like to thank the other members of my thesis committee for their constructive criticism and insight.

I am deeply and forever indebted to my husband for his endless love, support and encouragement throughout my entire life. I would like to thank my sweet children, Natalia and Sami, for their unconditional support and love. Finally, I wish to express my love and gratitude to my beloved families who have shown unending understanding through the duration of my studies.

Chapter 1: Introduction and Problem Definition

A maintainable software system would have a competitive edge in the software market, due to its ability to accommodate changes more efficiently. Maintainability is a critical property that allows the continuity and growth of software systems. A wide range of metrics that focus on different notions of complexity and code readability have been proposed for measuring maintainability.

The complexity of software systems continues to grow. In order for maintainers to accomplish their tasks, the software code should be readable and understandable. It would be very difficult and exhaustive to understand code lacking good documentation. Comments are an important source for software system documentation and are usually written in natural language meant to provide a clear description of the implementation. Good code comments can provide software developers with a better understanding of a software system's implementation, thus reducing the cost of maintenance.

In this chapter, the impact of program understanding on software maintenance is introduced. Then the importance of source-code comments in enriching program understandability is discussed. An approach to a detailed, quantitative and qualitative analysis of source-code comments is proposed. A parsing technology using ANTLR in analyzing comments is presented.

1.1 Introduction to the General Problem

Software engineering processes and tools have increasingly evolved over the past few years. However, software maintenance is one of the software development life cycle phases that still requires a lot of resources. Before the early 1990s, software maintenance required almost half of the resources (Coleman et al., 1994). Studies showed that 80 to 95% of the budget allocated to Information Systems is spent on maintenance activities (Erlikh, 2000). Many software maintenance problems arise from having to spend a long time on

comprehending the given system or program. This comprehension involves understanding the basic ideas and the design of the affected parts of the system, and determining where changes should be made. This is particularly evident if maintenance is not being carried out by the software developer who originally wrote the code. Moreover, a code review process is very expensive and hard. Developers spend a lot of time reviewing changes of others (Bosu & Carver, 2013). It is not only a significant effort in terms of time spent but it also forces the reviewer to switch context away from his or her current work. On the other hand, source-code comments almost always remain unchanged during maintenance activities. As a result, a comment may not properly describe the implementation—i.e., the information provided in the comment of a method and in its corresponding implementation may not be coherent with each other. Therefore, without prior knowledge of the system's implementation or the availability of suitable documentation, program understanding becomes an extremely time-consuming process.

Program comprehension is a software engineering activity that helps in understanding how a program solves a given problem to be able to fix errors or perform modifications. Program comprehension is a study of techniques and tools that try to automatically extract the needed information from a program providing necessary help to programmers who need to understand it (Freitas et al., 2012). Semantic information extracted from source code is not enough to fully understand a program. Hence, researchers started exploring the importance of semantic information that can be found in source-code comments, which are written in natural language. Comments vary according to programmers, thereby increasing the difficulty of the analysis process. General trends and rules could be derived for the commenting habit in order to be used in program documentation. Besides, not every comment is useful and can be used for documentation or comprehension purposes (Haouari et al., 2011). Additionally, in contrast to code, natural-language comments cannot be bound and controlled by syntax conventions which hamper the analysis of source-code comments.

Many previous researches that worked on analyzing the quality of software systems either totally ignore source-code comments or only results with quantitative claims (Oman & Hagemeister, 1992). However, metrics such as counting the number of lines containing a comment without differentiating between different types of comments seems too simple to

be meaningful. It does not take into account that some comments such as copyrights, do not add any value to system understandability or enhance the quality of system documentation.

1.2 Problem Definition

Even though programmers know the importance of good comments, commenting on the source code is often neglected due to development pressure conducted by urgent delivery dates and release deadlines (Van De Vanter, 2002). Without proper comments, understanding a program becomes a very difficult and time-consuming task that negatively affects software maintenance, testing and debugging.

Comments may reveal important information such as the reason for adding new lines to the source code, knowing about the progress of a collective task, or even why relevant changes were performed. Thus, comments may be used to describe issues that may require work in the future, and give notice about emerging problems and which decisions need to be taken (Maalej & Happel, 2010; Shokripour et al., 2013). These descriptions give us human readability and provide additional information that summarizes the developer context. Code comments and the source code itself are an important documentation to help us understand a system (de Souza et al., 2006). A survey of professional programmers found that the source code itself is the primary documentary artifact, with comments being the second most used one when seeking to understand a source code (de Souza et al., 2005).

Comments add a kind of refreshment to the source code. While good comments impact the readability of source code, adding an excessive amount of comments is discouraged (Fowler, 1999; Kernighan & Pike, 1999). If a source-code file needs a lot of comments to improve its readability, it might contain code fragments that are complex and hard to understand—i.e., bad code. Fowler mentioned that thickly commented code can be a sign of "code smell" to be refactored (Fowler, 1999, p. 71). Therefore, many comments are recommended to make a code more stable, but frequently adding describing comments is discouraged in terms of the code stability (Aman & Okazaki, 2008).

Ideally speaking, what cannot be determined by reading the source code should be commented. Unfortunately, while code comments are frequently used as a crutch,

programmers do not always maintain the related code comments while maintaining the source code itself. This results in outdated and fuzzy comments that badly and inaccurately describe the corresponding code. Such out-of-sync comments fall into the bad-comments category (Martin, 2008, p. 59) that need to be analyzed and optimized. Martin emphasized the fact that comments are a certain sign of bad code (Martin, 2008, p. 53). Hence, instead of commenting on bad code, developers should try to rewrite the code by enhancing its understandability. But comments are nevertheless indispensable. Commenting code is a necessity even when the code is clean.

Many of the existing techniques for measuring the quality of software systems do not take into consideration the quality of source-code comments. This might be due to several reasons; some are related to the fact that developers comment out their code for debugging purposes or as noting marks for later reuse, which might not be distinguishable from the real comments. Others are related to the fact that some types of comments such as copyrights do not facilitate system understandability or enhance comment quality. Additionally, developers expect comments to give them deep insights about the functionality or implementation details behind the source code, but some types of comments are useless as they are no clearer than the code. Others are noise and scary noise that add nothing to the source code (Martin, 2008, p. 64).

1.3 Research Objectives

Each programming language has its own coding conventions or standards. However, there are several general points that the developer should follow to ensure that his or her code is well organized so that it can be easily understood by others. Comments represent the main source for system documentation and hence a key for source-code understanding with respect to development and maintenance. They are parts of the code that are ignored by the compiler and are in no way mandatory but seek to make reading, reusing and maintaining the code easier. Comments play an important role in the design, maintenance, and use of software programs (Tan et al., 2007).

Writing high-quality source-code comments requires succinct technical writing and a deep understanding of the source code. When confronted with large programs, it is easier to read

a well-written comment than it is to trawl through the lines of code trying to understand the function of every piece of code. Comments should not be too thick or long. If the developer had followed a coding standard through the development phase, it should not be too difficult for other developers to read his or her own code. Otherwise, the code will look messy.

Comments should not be too little, nor should they be on every line. Most coding standards recommend commenting functions and objects rather than loops, variable assignments or every single line.

Comments also should not be too complicated or useless. It is important to keep the comments short, simple and to the point.

Hence, as part of a software quality audit, an analytical study is conducted by analyzing the existence of various types of comments of identifiers (classes, methods, types, variables, etc.) in the source code. This work presents a heuristic approach for comment classification and categorization that provides better insights about system documentation quality (Steidl et al., 2013). Based on comment classification, a quantitative and qualitative evaluation of comment quality will be provided. Such a comment quality model looks like the quality models in maintenance (Deissenboeck et al., 2007). Regarding coherence with source code and usefulness to the reader, our aim is to provide an assessment of comment quality and detect comments that are confusing and not helpful. Hence, for a better comment quality analysis an automatic approach will be provided based on heuristics to enforce the model in practice and test the results on open-source projects.

1.4 Approach and Main Results

This work presents an approach towards an automated analysis and assessment of the quality of source-code comments. It is considered as part of a software quality audit.

First, comment categorization based on a heuristic approach is performed. By differentiating between different comments categories, quantitative claims about how many comments potentially contribute to system understanding and how many comments are missed or serve other purposes are discussed. A comprehensive quality model for source-

code comments is also defined, with the main goal of capturing the semantic similarity between the comment content and its corresponding entity-identifier name. The model reveals the direct impact of source-code comments in helping software engineers understand and improve their products.

Second, decoding the source code by parsing is the major task of the used analysis tool. The resulting parse trees expose the syntactic structure of the source-code comments, making this information available for further analysis.

Third, for an assessment of comment quality, a heuristic approach is implemented and the results of the applied approach to an open-source project for detecting confusing or unhelpful comments are evaluated.

1.5 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2: gives an overview of software maintenance and program comprehension. A review on parsing source code for improving maintainability is conducted. The adopted parser generator tool (ANTLR) and other alternatives are presented. The related work pertaining to comment parsing and analysis is also presented and discussed.

Error! Reference source not found. describes different comment categories and analysis heuristics. The relevant parsing concepts and algorithms are presented. A description of a parser generator approach and its implementation using ANTLR are discussed.

Chapter 4: explains the proposed approach for analyzing source code. The results of a study that applies the proposed approach to existing open-source software are presented. The viability of the approach is tested, and insights that help in understanding how developers use comments are given.

Chapter 5: summarizes this work and provides directions for future research.

Chapter 2: Background and Motivation

After development, a program enters a maintenance phase in order to keep its functionality up-to-date and conformant to various changes. In order for the maintenance to be carried out efficiently and effectively, the code should be maintainable. Hence, the internal quality of software systems is an important area of research in software engineering.

Software engineers are constantly trying to improve the practice of software development by enhancing the readability of source code. Readable source code allows programmers to quickly and accurately obtain critical information about a program. Therefore, readable programs are easier to maintain. Source-code documentation is an important artifact that helps software developers comprehend and maintain software systems.

This chapter primarily addresses the importance of source-code readability and understandability throughout the long-term phase of software's life cycle. Software maintenance is introduced in Section 2.1. Section 2.2 highlights the use of parsing in analyzing source code and improving software maintainability by providing examples about modernization and refactoring. Section 2.3 provides some examples of automatic parser generator tools while the used tool in this thesis is described in detail in Section 2.4. Previous works on comment analysis are presented in Section 2.5, whereas the motivation behind this work is shown in Section 2.6.

2.1 Introduction

As software systems are becoming larger and more complicated, a major part of the software life-cycle resources are dedicated to maintenance (Erlikh, 2000). Thus, there is a serious need for advanced tools that aid exploration and comprehension of today's software in order to reduce the cost of maintenance. One of the prerequisites of successful development and maintenance of any software product is its source-code readability. In an effort to achieve this, programmers try to make the code more understandable and logical

by properly documenting variables, methods, types, and classes using comments. Therefore, documentation represented by natural-language comments plays a major role in program identifiers to improve software maintainability (Shepherd et al., 2007). The work presented in this thesis particularly focuses on locating and analyzing comments on identifiers (e.g., classes, methods, types, and variables).

As mentioned earlier, maintenance is a long-term phase in software's life cycle. Maintenance tends to have a relatively longer duration than all the previous life-cycle phases combined, thereby requiring much more effort.

Software maintainability is the facility with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment (IEEE1219, 1998). Hence, producing software that is not flexible enough to adapt to changes could result in a loss of time and resources. Maintainability is affected by the readability and understandability of both source code and documentation. A high level of software maintainability indicates that less effort would be required during the maintenance phase. Despite the fact that software maintenance is essential and challenging, it varies from one software system to another based on various measurement criteria and it might be poorly managed. One reason for poor management is the absence of a good measure of software maintainability. There is a need for standard rules or protocols for measuring software maintainability (Aggarwal et al., 2002). During maintenance, changes are made to the source code to fix bugs or upgrade functionalities. Today programs consist of thousands or even millions of lines of code. Hence, it can be difficult to modify such code especially if it is written by someone else. The assimilation and understanding of source code is almost impossible if it is not well supported by meaningful comments. Writing comments on source code became a standard practice in programming, because comments improve software maintainability by helping developers understand code. Nevertheless, the source code of many software systems lacks adequate comments (de Souza et al., 2005). Comments are important as they are used to convey the main intent behind design decisions, along with some implementation details. Therefore, comments are critical aspects to comprehensibility and maintainability of source code. Green and Petre mention the importance of supporting a secondary notation—e.g., comments—in programming languages to convey meaning to the human reader (Green & Petre, 1996).

Maintainability is related to understandability, modifiability, and testability. A good measurement of maintainability is a good measure of those concepts. Although the quality of code comments plays a minor role in assessing software maintainability, it is commonly agreed that poor documentation negatively affects maintainability (Hartzman & Austin, 1993). For example, the number of lines of comments is an understandability measure. While the original developers are not always the people who maintain the software, poor documentation increases the time of understanding and maintaining the software where the maintainer might find it easier to rewrite the code than to understand or modify the existing one. Like the source code, the maintenance activities will degrade comments unless they are treated properly and not allowed to become inaccurate.

The main goal of commenting source code is making it easier to understand by someone who did not write it (Kernighan & Pike, 1999, p. 23) or for the original developers who want to modify and reuse their own code later on. Although code comments are necessary even when the code is clean, the existence of some types of comments or their frequency in the code might be a strong indicator of low code quality. Even Martin (Martin, 2008, p. 55), who asserts that the only good comment is the comment you found a way not to write, admits that good comments exist when they explain the developers' intentions, emphasize the importance of something, or warn of consequences. Also in his work on comment analysis, Raskin (Raskin, 2005) concludes that self-documenting code and automatic documentation are not enough, and that inline comments on the same line as the code are too brief and that several lines should be used.

Thus, for the purpose of code quality analysis and assessment, a parsing tool can be used to extract the needed information from the program by parsing the code and producing an Abstract Syntax Tree (AST) (Strein et al., 2007).

2.2 Parsing Source Code to Improve Maintainability

Software engineering is supported by various program analysis tools that can be used in software maintenance tasks. Such tasks include: refactoring, program comprehension for understanding unknown code, bug fixing or enhancement based on change requests made by the user, code quality assessment during code reviews, and reverse engineering. For this

purpose, computer scientists have developed a wide range of declarative languages based on context-free grammar (CFG) formalism, and parser generators that produce efficient parsers for their descriptions. An essential part of software engineering research and practice is the ability to parse the source code in order to extract information from it using analysis tools (Binkley, 2007). Parsing is a process whereby a given program is matched against grammar rules to determine whether or not it is syntactically correct. Hence, source-code analysis plays an important role in the quality assessment of software projects. Two examples are the modernizing of legacy software systems and refactoring.

2.2.1 Modernizing Legacy Software Systems

Today, many companies around the world still use production software that is written in COBOL, Pascal, and other legacy computer programming languages. These organizations are highly motivated to renovate their software systems for many reasons including the difficulties of maintaining the legacy systems and in hiring developers who are skilled in legacy programming languages (Mitchel & Keef, 2012). However, this renovation or migration process is often either unaffordable or produces poorly eligible software with maintenance difficulties (McAllister, 2010). Hence, one of the major maintenance concerns is the problem of porting and adapting the existing applications into modern technologies. Therefore, tools and techniques for parsing legacy languages play an important role in modernizing legacy systems while preserving their functional integrity.

2.2.2 Refactoring

Software must be designed to be resilient or easily changeable in order to meet new requirements. As the software is frequently changed, it becomes increasingly difficult to add new features without rethinking its design. Refactoring is a way of restructuring code in order to improve its design while preserving the original functionality (Fowler, 1999). Invisible Java Compiler Compiler (IJACC) is an example of a refactoring parser tool for Java source files, which is used to scan the code for compiler construction ("Invisible Jacc Version 1.1," 1997). Moreover, software reusability is an important prerequisite for improving software quality and reducing maintenance cost. For example, measuring the coupling level between system modules is one of the essential aspects where heavy

coupling decreases the reusability. Hence, for analyzing the coupling measures of object-oriented systems, a parser called "Design Analyzer" has been developed to define the design patterns of the system (Hasan & Hasan, 2010). Additionally, developers frequently reuse fragments of source code by performing "copy and paste" during the development phase. Therefore, code parsing is also used in detecting such code clones that are annoying during maintenance (Maeda, 2009).

Descriptive naming conventions that programmers should use for methods, fields, and classes help newcomers to capture the functionality of a method or field more precisely and better understand what the code does. The Software Word Usage Model (SWUM) is a lexical approach that captures the conceptual information about a program through both its natural language identifiers and program structure (Hill, 2010). But using extremely descriptive identifier names that accurately describe an entity lead to very long identifier names. Longer names can actually reduce code readability rather than increase it (Liblit et al., 2006). For this purpose, Relf implemented the naming style guidelines in a tool to help programmers create high-quality identifiers and to refactor existing identifiers (Relf, 2005). By following the Java conventions, identifier names begin with a lower-case character and consist of one or more words, and use internal capitalization to mark the beginning of every second word. For accessing the quality of identifier names, Butler et al. provide an approach for improving the tokenization algorithm (Butler et al., 2011). Fowler, in his original list of refactoring, includes the renaming of method where the name does not reflect the purpose of the method (Fowler, 1999). Abebe et al. developed a system to recognize "lexicon bad smells" in identifiers, thereby identifying a wide range of identifier names for possible refactoring (Abebe et al., 2009).

2.3 Tools for Parsing Source Code

In order to help software engineers understand and improve their products, several tools and techniques have been implemented for acquiring and delivering software information. Since software is written in a programming language, it is necessary to decode the source code by parsing for its further analysis. A parser implements the mapping from source code in string representation to a tree representation; it splits the input into tokens and finds a

hierarchical structure of the input. The implementation of any programming language usually begins with a description of the syntax of the language in any form. A language is defined by a grammar that contains all the information needed to develop a parser for that language.

Parsers can be developed manually, but this is a time-consuming process for programming languages with complex grammars. Alternatively, existing automatic parser generators can be used for more complex parsing algorithms. Several context-free parsing algorithms exist, the most widely used being the pure top-down Earley's parser (Earley, 1970, pp. 94–102). Parsing algorithms could be differentiated based on the way they implement the problem of parsing whether in the direction they read the input or whether they construct parse trees from the top down or the bottom up. Hence, for parsing purposes the LL (Left to right/Leftmost derivation => top down) or LR (Left to right/Rightmost derivation => bottom up) grammars are often used (Sippu & Soisalon-Soininen, 2013). For constructing such grammars, language-specific lexical and syntactic analyzers can be automatically performed by means of a suitable tool.

Some examples of automatic parser generator tools are:

2.3.1 ANTLR

Another Tool for Language Recognition (ANTLR) is a robust and popular generator tool which supports several programming languages such as: C, Java, C #, Python, and Ruby (Parr, 2013). It is used specifically in this thesis as a parser generator tool in achieving the main goal and is discussed in more detail in Section 2.4.

2.3.2 JavaCC

JavaCC (JavaComplierCompiler) is a top-down parser generator tool (Kodaganallur, 2004). It is used in many applications and much like ANTLR, but it has few features. The AST tree has a separate tool called JJtree which is combined with JavaCC tool. However, it is only a Java generator tool. Furthermore, the documentation is poor when compared to ANTLR.

2.3.3 SableCC

SableCC (SableCompilerCompiler) is a bottom-up LALR(1)-based compiler-compiler parser generator which takes object-oriented methodology for constructing parsers (Gagnon & Hendren, 1998). SableCC is different from the other tools in that it does not allow any semantic actions in the input specification, only syntax. It maintains easy code for generated parser as a result. However, SableCC has some performance issues. It generates or supports C++ and Java.

2.4 Another Tool for Language Recognition (ANTLR)

ANTLR is a language-based tool that allows designing compilers, interpreters, recognizers and translators generated in any one of a variety of formal languages (Yu et al., 2008) based on self-written grammar rules similar to Extended Backus–Naur Form (EBNF) expression (Garshol, 2008; Parr, 2007). ANTLR can automatically generate a program that parses an input stream of symbols and determines whether that input conforms to a grammar. Moreover, it provides support for building abstract syntax trees (ASTs), tree walking and translation. It also provides a convenient means for automatic error recovery and reporting. ANTLR uses LL (k) grammar, and the generated program is very intuitive and easy to debug (Parr, 2007). By using ANTLR as a compiler, it provides an appropriate syntax for specifying lexers, parsers, and tree parsers through three class templates: Lexer, Parser, and TreeParser. The overall framework of ANTLR is shown in Figure 2.1 (Liu et al., 2008).

ANTLR generates the corresponding lexical analyzer (Lexer) and the parser (Parser) from customized lexical and parser grammar rules. A lexer's role is to perform lexical analysis—i.e., scanning and decomposing the input stream into individual symbols called Tokens. Then, the lexer classifies and organizes the tokens into a unified format. Consequently, the parser feeds off this token stream and tries to recognize the input structure (class, function, variable, etc.). Hence, it is mainly used for the inspection of proper grammar rules that match the input.

There are many types of translators. Some are simple and immediately execute actions to get results. Others are more complicated and use the parser for AST construction (Parr,

2005). The procedure of input stream analysis for AST construction is shown in Figure 2.2 (Turuntaev, 2014, p. 44). AST traversal is a defined TreeParser in ANTLR. A TreeParser works from left to right and uses depth-first rules in the AST traversal (tokens as nodes).

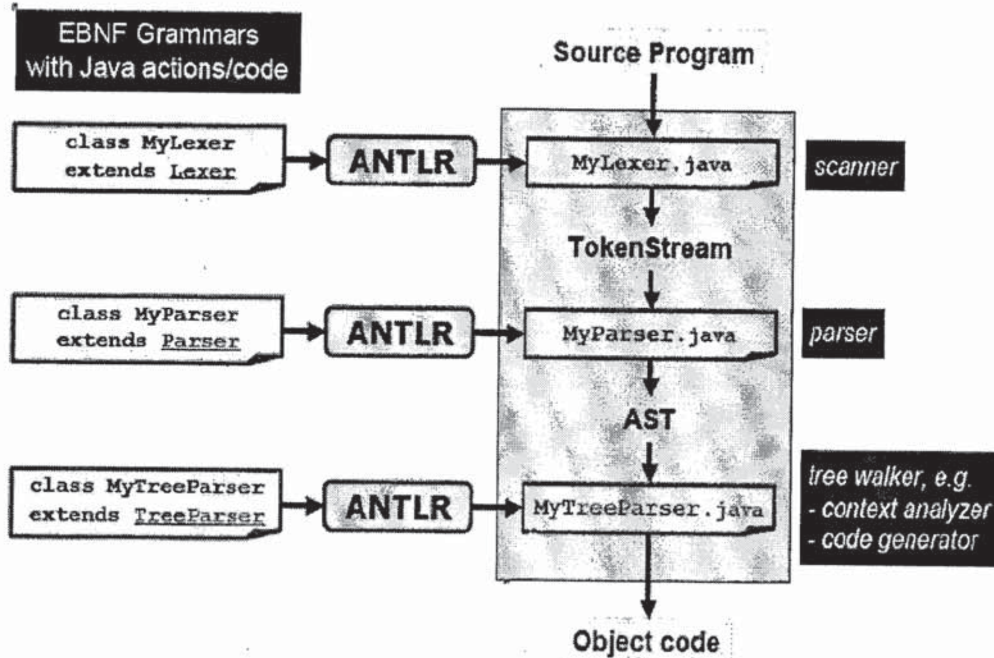


Figure 2.1: The overall framework of ANTLR (Liu et al., 2008)

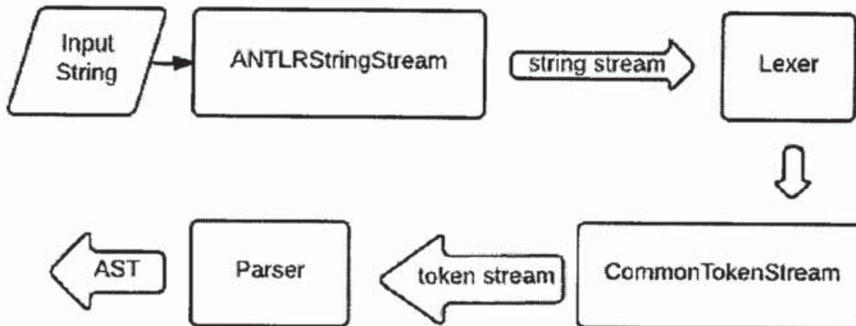


Figure 2.2 Abstract Syntax Tree (AST) Construction (Turuntaev, 2014, p. 44)

With ANTLR, an input Java file can be broken down into lexical analysis, syntax analysis, and semantic checking (to be discussed later in detail). The lexer breaks up the input stream into tokens based on the lexical rules pre-defined in the grammar document. Then, the

parser feeds off this token stream and tries to recognize the input structure (Parr, 2007) based on the pre-defined parser rules of the grammar. Finally, the semantic checking tests the validity of the results generated by the parser. After performing the three above-mentioned steps, the input file can be implemented.

More precisely, lexical analysis scans input source-code (e.g., Java) file entered by the users from left to right, and generates a stream of tokens in order as inputs to the syntax analysis (Parr, 2005). In other words, Lexer translates input streams into tokens according to the lexical rules defined in the ANTLR grammar file. The tokens recognized by the Lexer are as follows: keywords, identifiers, operators, separators, NULL, literals, etc. Additionally, lexical analysis removes blank spaces from a source-code file. Lexical analysis is ANTLR's interface with external clients. If a character stream cannot be recognized as one of the defined tokens, the lexer will generate error messages.

Syntax analysis determines whether the syntax of the input files corresponds to the defined grammar rules on the basis of tokens of Lexer. These grammar rules are written based on EBNF expressions. The ISO standard "ISO/IEC14977" enacted the international standard of EBNF in 1997. ANTLR automatically generates a parser according to a grammar rules file. Thus, syntactic analysis—e.g., JavaParser—gets the words or tokens from lexical analysis, and it checks the grammatical correctness of token sequences according to the syntactic rules defined in an ANTLR grammar file. If the input file can be totally parsed by a sequence of the defined grammar rules starting from a specific token, then the syntactic analysis is successful, otherwise the JavaParser will generate syntactic error messages. Syntactic analysis and lexical analysis complement each other. When the parser needs more tokens, the lexical analyzer sequentially scans the current input to identify the next token and then returns it to the parser. Finally, after the syntax analysis succeeds, the parser performs a semantic checking of the input file's integrity and correctness.

2.4.1 Lexical Rules Implementation

According to ANTLR programming specification, the lexical rules and syntactic rules are written in a grammar file (.g4 file extension) in which the lexical rules and syntactic rules are represented in upper-case and lower-case letters, respectively. The grammar file generally includes a header block, an options block, a parser (Parser), and a lexical scanner

(Lexer) as stated in order. According to the defined grammar file, ANTLR automatically generates an executable lexical analyzer (JavaLexer.java) and syntactic parser (JavaParser.java).

The main task of the lexical analysis part is to recognize the words of the input file and identify the correspondent stream of tokens. In other words, the lexer's job is to transform a meaningless stream of characters into tokens that would have meaning when processed by the parser. For example, according to our grammatical syntax of ANTLR, the key identifiers are defined using the regular expression where the defined lexical rules are shown in Figure 2.3 (Parr & Harwell, 2013).

```
1039 // §3.8 Identifiers (must appear after all keywords in the grammar)
1040
1041 Identifier
1042     :   JavaLetter JavaLetterOrDigit*
1043     ;
1044
1045 fragment
1046 JavaLetter
1047     :   [a-zA-Z$_] // these are the "java letters" below 0xFF
1048     |   // covers all characters above 0xFF which are not a surrogate
1049         ~[\u0000-\u00FF\uD800-\uDBFF]
1050         {Character.isJavaIdentifierStart(_input.LA(-1))}?
1051     |   // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
1052         [\uD800-\uDBFF] [\uDC00-\uDFFF]
1053         {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2), (char)_input.LA(-1)))}?
1054     ;
1055
1056 fragment
1057 JavaLetterOrDigit
1058     :   [a-zA-Z0-9$_] // these are the "java letters or digits" below 0xFF
1059     |   // covers all characters above 0xFF which are not a surrogate
1060         ~[\u0000-\u00FF\uD800-\uDBFF]
1061         {Character.isJavaIdentifierPart(_input.LA(-1))}?
1062     |   // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
1063         [\uD800-\uDBFF] [\uDC00-\uDFFF]
1064         {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2), (char)_input.LA(-1)))}?
1065     ;
1066
```

Figure 2.3 Declaration of Identifier (Parr & Harwell, 2013)

This definition of an "Identifier" lexical rule can be directly used to identify any word in the syntactic analysis. The Lexer also has the following alternative names: scanner, lexical analyzer, and tokenizer.

2.4.2 Syntactic Rules Implementation

The Parser, also called a syntactical analyzer, checks to see if the tokens conform to the syntax of the language defined by the grammar. Similar to the lexical rules, syntactic rules are also implemented by defining each syntactic block using EBNF. For example,

according to the Java interface declaration each of the keywords is defined as an EBNF, and each EBNF of the keywords can be expressed by one or more sub-paradigms as shown in Figure 2.4 (Parr & Harwell, 2013). Starting from the top, every EBNF is defined layer by layer. Finally, all of the EBNFs are built into a complete syntax tree, which defines the Java interface. So syntactic analysis recognizes the input tokens as any correct sequence of EBNFs to complete a syntax tree.

```
163⊖ interfaceDeclaration
164      :   'interface' Identifier typeParameters? ('extends' typeList)? interfaceBody
165      ;
166
167⊖ typeList
168      :   type (',' type)*
169      ;
```

Figure 2.4 Syntactic Rule of Java Interface (Parr & Harwell, 2013)

As Figure 2.4 shows, the interface declaration is defined by the lexical rule "Identifier" and the sub-rules: "typeParameters," "typeList" and "interfaceBody." Similarly, the three sub-rules are defined by other lexical and syntactic rules. Therefore, according to the syntactic tree top-down analysis approach, the rules are defined one by one from the top.

2.4.3 Semantic Checking and Error Handling

While defining rules, ANTLR supports embedding programming language—e.g., Java—code to perform semantic actions. These codes are used to generate the error handling procedures. The codes are implemented by the ANTLRWORKS tool (Parr, 2005), which is the ANTLR graphical development environment.

Based on the above discussion, Java grammar rules are defined by EBNF and a Java parser is implemented based on ANTLR technology, which is made of lexical analysis, syntax analysis, and semantic checking. Then, through the interface functions provided by ANTLR as shown in Figure 2.5 (Parr, 2013, p. 239), the lexical and syntactic analyzer can be easily called, in order to preclude repetitive coding and improve the reliability and reusability of the software.

```
ANTLRFileStream stream = new ANTLRFileStream(resultFileList.get(f));
JavaLexer lexer = new JavaLexer(stream);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
String result = parser.compilationUnit().expr ;
```

Figure 2.5 Calling ANTLR's Interface Functions (Parr, 2013, p. 239)

2.5 Previous Work on Comment Analysis

Although there is no consensus about what constitutes a good/bad comment, in his book "Clean Code" Martin agrees that the only source of truly accurate information is the code itself (Martin, 2008, p. 53). However, he claimed that sometimes comments are beneficial, and he classified the comments into two categories: good and bad. Accordingly, good code comments are the legal comments (e.g., copyright and authorship statements), informative comments, explanation of intent that provides the intent behind a decision, clarification comments that translate the meaning of some obscure, TODO comments, or a well-described javadocs public API. On the other hand, examples of bad comments include: mumbling, redundant, misleading, mandated, noise. and scary noise comments.

Several approaches proposed models, metrics, and tools to analyze and assess source-code comments for improving understandability, modifiability, and testability.

Code comments and documentation help testers to carry out the desired tests without having to perform an extensive reading of the source code. Therefore, parsing code and finding comments was used to promote a better understanding of the source code and improve software testing based on existing documentation (Arrington, 2009).

JavadocMiner is a tool for analyzing the quality of inline documentation (Khamis et al., 2010). This tool mainly targets inline-documentation in the form of Javadoc comments by evaluating the quality of the language and the consistency between source code and its comments, based on a set of heuristics. For measuring the quality of the language, heuristics such as counting the number of tokens, verbs and nouns, calculating the average number of words, or counting the number of abbreviations are used. Additionally, Fog index or the Flesch reading ease level heuristics are used for measuring the readability of comments. Moreover, for the purpose of analyzing the consistency between source code

and its comments, this approach provides heuristics for ascertaining that all aspects are documented (for example, aspects of a method such as parameter or return type) and for computing the ratio of identifiers with Javadoc comments to the total number of identifiers. SYNC (Boolean indicating synchronization) is a heuristic that finds return types, parameters, and thrown exceptions that are no longer up-to-date—e.g., due to changes in the code. Finally, the authors observed in a case study of JavadocMiner on different releases of ArgoUML and the IDE Eclipse that the modules with the highest quality of code comments contain the least amount of bugs and vice versa.

An empirical study was also conducted to explore the role of task annotations in source code (Storey et al., 2008). After surveying software developers, analyzing the code of open-source projects, and gathering data from personal interviews, the authors found out that most task comments are those that contain @TODO tags. Although "TODO" comments are often used for documenting small tasks to be completed in the future, there is a big risk that the developers will forget to revisit them. Therefore, the authors in their analysis suggested several implications for tool designers such as supporting metadata within task comments, introducing ad-hoc task cleanup wizards or providing a filtering mechanism for task views. Additionally, a preliminary study was performed by manually analyzing task comments in Architect's Workbench (AWB), an internal IBM code base (Ying et al., 2005). The authors came up with a detailed categorization of task comments such as tasks used for communication, pointers to change requests, bookmarks on past tasks, current tasks, or future tasks. However, there is no automatic assessment of task comment quality.

Nurvitadhi et al. investigated whether class comments or method comments were more helpful in comprehending a Java program (Nurvitadhi et al., 2003). In a study of 103 people beginning to learn programming, they found that class comments did not help in fostering a high-level understanding of the program. However, method comments did help in understanding the functionality of the methods.

Furthermore, according to Tan et al. a significant percentage of comments relates to hot topics—e.g., memory allocation and synchronization (Tan et al., 2007). With a simple keyword search ("lock," "alloc," "signal," "thread"), the authors detected hot comments in different Linux modules. They also conducted a preliminary study that analyzes the

synchronization-related comments in Linux based on a combination of natural language processing and some heuristics (i.e., keyword searches). With their technique, the authors were able to detect 12 bugs in Linux, two of them being confirmed by developers. As a result, the authors explored the feasibility and benefits of an automatic analysis of comments to detect bugs in code and bad comments.

2.6 Research Motivation

Since no all-encompassing quality model of code comments has been developed so far, the above-mentioned works give an overview of previous research about source-code comments.

Although JavadocMiner attempts to measure the quality of code comments, which is also the target of this thesis, it uses very basic heuristics that do not differentiate between different comment types (Khamis et al., 2010).

Storey et al. specifically target task comments (Storey et al., 2008). However, in this thesis our aim is to provide a general assessment of comment quality and categorization of multiple types of comments including those that are related to tasks.

The study of Tan el al. is tailored to the specific topic of synchronization and memory allocation (Tan et al., 2007). In contrast, our approach analyzes comments independent of the context.

The strengths and shortcomings of the previous research about code comment analysis are shown in Table 2.1.

Table 2.1 Strengths and Shortcomings of Previous Research on Code Comment Analysis

Existing Research	Strengths	Shortcomings
JavadocMiner (Khamis et al., 2010)	• measures the quality of code comments by analyzing the quality of inline documentation	• uses very basic heuristics • does not differentiate between different comment types
Task comments @TODO (Storey et al., 2008)	• explore the role of task annotations in source code • result in a detailed categorization of task	• particularly focus on task comments • no automatic assessment of task-comment quality

	comments	
Hot Comments (Tan et al., 2007)	• tailored to the specific topic of synchronization comments	• context specific (keyword searches) • no automatic comment analysis

Chapter 3: Comment Parsing and Analysis

This chapter provides an accessible exposition of our work towards an automated analysis of the quality of source-code comments. Different comment categories are described based on semantic assumption in addition to the heuristics used in assessing the usefulness of the comments. Relevant parsing concepts and the suitable parsing algorithm are also presented alongside a parser generator approach that is based on ANTLR.

3.1 Introduction

In order to analyze and assess the quality of source-code comments, there is a need for a precise definition of comment quality. Most of the previous researches attempt to give recommendations on how to write good code comments, without specifying the quality model of code comments. One motivation for the focus on identifier names in analyzing comments is that most of the application-domain knowledge that programmers possess when writing code is captured by identifier names' mnemonics (Antoniol et al., 2002). In computer programs, Knuth noted that descriptive identifiers are a clear indicator of code quality and comprehensibility (Knuth, 2003).

The main goal behind source-code commenting is to improve its understandability. In this work, a quality model in maintenance will be introduced to specify the usefulness behind the existence of each type of comment in helping the developer understand code. With the same underlying syntax of the different comment categories, no parser or compiler can perform comment classification based on grammar rules. Therefore, a heuristic approach is required.

3.2 Categorization of Comments

For the purpose of this work, comments are grouped into different categories by evaluating the actual content of their textual information. Syntactically, comments in Java are written either between the delimiters "/*" and "*/" and are declared as block comments or after the delimiter "//" and are known as single-line comments. In this thesis, only block comments are analyzed and single-line comments (except TODO comments) are left for future work. In order to analyze and assess the quality of source-code comments, a quality model is defined based on the semantics (i.e., the information conveyed by the comments) for classifying comments. The assumption is that if the code is of good quality, the comments provide a good description of the code particularly of the identifier name (i.e., comments share many similar terms with the identifier names). Therefore, different types of comments are distinguished as the following.

3.2.1 Documentation Comments

Documentation comments describe and provide information about the neighboring entities of the source code. Most of the existing comments in any program are documentation comments, which may be block or single-line comments. They may appear before or after the referred identifier or even in the same line. These comments may either repeat the code verbatim or compactly summarize it. For example, consider the following code of variable declaration:

```
/**
 * The page number
 */
public int pageNumber;
```

Its documentation comment essentially repeats the code which is a kind of noise comment type (Martin, 2008, p. 64). A more useful type of documentation comment is a summary comment which describes the code in a concise manner as the following:

```
/**
* This method is used to add two integers.
* @param numA This is the first paramter to addNum method
* @param numB This is the second parameter to addNum method
* @return int This returns sum of numA and numB.
```

```
 */
public int addNum(int numA, int numB) {
        return numA + numB;
      }
```

3.2.2 Copyright Comments

Due to coding conventions, every file should begin with a copyright comment that provides a brief description on what the file does and a short extract of the license.

3.2.3 Task Comments

Task comments are a note for the developer about an unfulfilled task and usually begin with "//TODO."

3.3 Analysis Heuristics for Source Code Comments

Different types of comments are used for documenting the existing identifiers inside a file. For example, a class comment should provide insight on the high-level knowledge of a program—e.g., which services are provided by the class, and which other classes make use of these services (Nurvitadhi et al., 2003). On the other hand, a method comment should provide a low-level understanding of its implementation (Nurvitadhi et al., 2003). For the purpose of software quality assessment, the similarity between comments and the accompanying identifier names is measured. Hence, a heuristic approach is required and a set of heuristics were implemented for assessing the quality of every comment.

3.3.1 Identifier Names Tokenization Heuristic

Every programming language has conventions that constrain the content and form of identifier names. Java follows Camel Case syntax for naming the identifier. If the name is combined with several words, every second word will always start with uppercase letter—e.g., pageNumber. Another way is to delimit separate words using an underscore ("_") character. This heuristic means of splitting conventionally constructed identifier names is based on internal capitalization and special characters to indicate word boundaries.

3.3.2 Word Match and Similarity Heuristic

This heuristic detects the number of words matching (longest match) by comparing the identifier name (after tokenizing, if needed) with its corresponding comment content. A similarity percentage will also be calculated for a more precise evaluation of comment semantics. To quantify the similarity ranking scale, the highest rank is assigned to 1.0 and the lowest to 0.0. For example, given the comment `/* The page number */` for pageNumber variable declaration, 2 matching words ("page" and "number") are found and the similarity factor is 0.8. Such a high similarity factor indicates that the comment adds nothing but noise.

3.3.3 Location Heuristic

Where is the comment located? Comments can be written at different locations in the code, such as for a class, a field, a method, a parameter, an exception or a subset of statements within a method. This heuristic extracts the identifier to which the comment is related.

3.3.4 Lines Count Heuristic

This heuristic means pertains to detecting the use of well-formed sentences within the comment. It counts the total number of lines within the comment.

3.4 Parsing Source Code

Parsing is a technique that is used to convert software source code from text files into a form that facilitates automated analysis. It is a process of analyzing the source code of a program in order to determine its grammatical structure. There are many ways to parse a sentence using a computer, most of which involve the use of grammar rules. More specifically, a parser implements the mapping from source code in string representation to a tree representation. In this section, our main goal is to provide an exposition of the relevant ideas, rather than detailing the parsing theory. Sub-Section 3.4.1 reviews the relevant parsing concepts while the suitable parsing algorithm is described in Sub-Section 3.4.2. Sub-Section 3.4.3 shows the mapping of input tokens into a parse tree based on the parsing algorithm.

3.4.1 Parsing Concepts

In this section, different parsing-related concepts are briefly covered. A very simple Java program is shown in Figure 3.1, and a fragment of the Java grammar used to parse the given program and construct its parse tree is shown in Figure 3.2. The complete grammar defines the entire syntax of the Java programming language.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Figure 3.1 A Simple Java Program

```
typeDeclaration
    :   classOrInterfaceModifier* classDeclaration
    |   classOrInterfaceModifier* enumDeclaration
    |   classOrInterfaceModifier* interfaceDeclaration
    |   classOrInterfaceModifier* annotationTypeDeclaration
    |   ';'
    ;

classDeclaration
    :   'class' Identifier typeParameters?
        ('extends' typeType)?
        ('implements' typeList)?
        classBody
    ;

classOrInterfaceModifier
    :   annotation       // class or interface
    |   (   'public'     // class or interface
        |   'protected'  // class or interface
        |   'private'    // class or interface
        |   'static'     // class or interface
        |   'abstract'   // class or interface
        |   'final'      // class only -- does not apply to interfaces
        |   'strictfp'   // class or interface
        )
    ;

classBody
    :   '{' classBodyDeclaration* '}'
    ;

classBodyDeclaration
    :   ';'
    |   'static'? block
    |   modifier* memberDeclaration
    ;

memberDeclaration
    :   methodDeclaration
    |   genericMethodDeclaration
    |   fieldDeclaration
    |   constructorDeclaration
    |   genericConstructorDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;
```

Figure 3.2 Fragment of Java Grammar

Like most programming language grammars, our grammar belongs to a class known as context-free grammars (CFGs) or Backus Naur Form (BNF). CFGs consist of a set of rules in which each rule has a left-hand side (LHS) consisting of a single nonterminal, and a right-hand side (RHS) that consists of several symbols, where a symbol is either a terminal or a nonterminal. It is important here to distinguish between terminals that are found in a grammar, and tokens that are the lexical units to be parsed. Each token in the input string corresponds to one terminal in the grammar.

The set of all terminals used in a language is known as the language's alphabet. These terminals are the text between quotation marks ('') that must match a token of the input stream. In our case, the alphabet of Java includes the terminals: public, class, implements, etc. The grammar rules define the set of ways in which the terminals may be composed to produce a meaningful sentence in the Java programming language.

Extended BNF (EBNF) syntax provides more extensions to BNF typically by allowing optional clauses and repeated (zero-or-more or one-or-more) clauses on the RHS. Hence, the main difference is that EBNF is more expressive and is therefore used more often by language designers. The Java grammar fragment in Figure 3.2 makes use of two features of EBNF syntax (the "?" character) to specify the optional symbol and (the "*" character) to specify the repeated zero or more times symbol. Though it is not mentioned in this example, the ("+" character) denotes that the preceding rule is repeated one or more times. The ("|" character) denotes a choice between rules and brackets that is used for grouping syntax rules. The operators that ANTLR uses in its EBNF syntax are summarized in Table 3.1.

One and only one nonterminal in a grammar is denoted as the start symbol. By convention, it is the first nonterminal symbol, which is the LHS symbol of the first grammar rule. The start symbol represents the root of all parse trees for that grammar.

For any programming language, a well-formed CFG could be defined based on the following restrictions:

- Every RHS nonterminal of a grammar rule must appear as an LHS of another grammar rule.
- Every LHS nonterminal must appear at least once either on the RHS of a grammar rule or it must be the start symbol.
- The grammar is not cyclic. It must not contain any useless rules that allow an LHS to be the same as an RHS resulting in an infinite number of parse trees for a given input.
- Every single nonterminal must be resolved to a terminal.

Table 3.1 ANTLR EBNF Operators

ANTLR / EBNF	Explanation
A B	Matches A followed by B.

A | B	Matches A or B.
A?	Matches zero or one occurrences of A.
A+	Matches one or more occurrences of A.
A*	Matches zero or more occurrences of A.
( )	Parenthesis can be used to group several elements and can be treated as one single token.

3.4.2 Parsing Algorithm

The parser scans the input going from left to right. It identifies whether it is a leftmost or rightmost derivation. The parser uses the grammar rules for finding the appropriate derivation. In our study, a context-free grammar is utilized and the leftmost top-down processing strategy is used.

The top-down parser begins with the single start symbol. The parser starts to expand the start symbol into a stream of symbols using any applicable rule from the grammar. It continues to expand each nonterminal symbol in the string until the string consists of only terminal symbols. This resulting string is compared to the words of the input stream, and if they match the input is a legal construct. Otherwise, the parser backtracks and tries new expansions of the nonterminal until the given input parses or an error arises, meaning the input is semantically incorrect. Top-down parsers are sometimes called predictive parsers because of the way they predict the rules to use in a derivation.

Concerning our work on comment parsing, comments with their location information from the source code are extracted. This is done based on a top-down leftmost parsing algorithm. Then, the comments are analyzed by grouping them into categories based on the classification described in Section 3.2 and by implementing the set of heuristics described in Section 3.3.

3.4.3 Parse Tree

A parser determines whether a given input belongs to a grammar by outputting a Boolean result. It constructs a parse tree that spans the entire sequence of input tokens. The leaf nodes of a parse tree contain the tokens of the input, and the branch nodes contain sub-trees

that match the RHS of a grammar rule. Thus, each branch node corresponds to one LHS nonterminal that has been recognized. The root node is a branch containing the start symbol. Hence, a parse tree diagrammatically shows how the start symbol of a grammar derives an input stream of tokens.

An Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of the program. It is constructed by grouping the input tokens according to a grammar. A parse tree groups together adjacent words or phrases to form longer phrases, until a sentence that spans the entire input is found. The legal groupings, and consequently the possible structures of parse trees, are defined by a grammar. A fragment of a parse tree is illustrated in Figure 3.3. This parse tree was constructed from the tokens that were extracted by parsing the example shown in Figure 3.1 using the grammar rules shown in Figure 3.2.

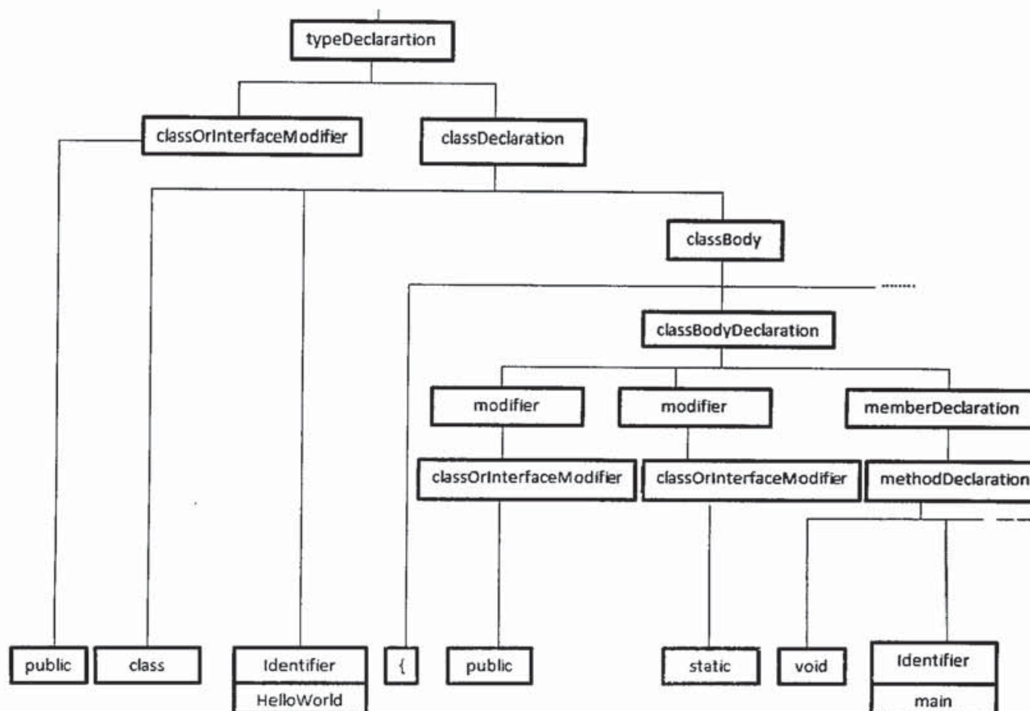


Figure 3.3 Parse Tree Fragment Produced by a Parser

When the parser receives its first token, it will try to match this token with a start rule of the EBNF. In this example, the start rule is the *typeDeclaration* rule. When entering the *typeDeclaration* rule the parser will first have to match one of the RHSs shown in Figure 3.2. The parser will match the RHS *classOrInterfaceModifier * classDeclaration.* Then, the parser applies the *classOrInterfaceModifier* rule to match the 'public' token. When the parser reaches the end of the *classOrInterfaceModifier* rule, it applies the *classDeclaration* rule. According to EBNF the next token should be 'class' which matches the given input, *Identifier* rule and *classBody* rule (the optional clauses in the RHS of the *classDeclaration* rule do not match the given example). The parser will match the 'HelloWorld' token with the *Identifier* rule, which ends here and the parser continues with the *classBody* rule. According to EBNF the next token should be '{' which matches the given input, *classBodyDeclaration* rule and the '}' token that is the closing braces of the given class. Now the parser continues with the *classBodyDeclaration* rule in a similar way as the previous rules until reaching the terminal tokens.

3.5 Parser Generator

In this thesis, a parser generator called Another Tool for Language Recognition (ANTLR) is used. One of the key features of ANTLR is that it uses a syntax very similar to EBNF to specify the syntax of a language. This section briefly introduces ANTLR and provides an overview on how to write grammar rules for it.

3.5.1 Generating Lexers and Parsers with ANTLR

The principles and techniques of writing compilers are quite diverse. Conceptually, the compilation process includes the following phases:

- Lexical analysis: sequence of code symbols converted into a sequence of tokens
- Syntactic analysis: sequence of tokens converted into a parse tree
- Semantic analysis: parse tree processed for establishing its semantics (meaning)
- Optimization

The process of building a parser by hand is a very time-consuming task and does not always guarantee a good result. A parser is usually a large-scale system making it difficult

to debug even for a simple input language. There are a number of tools to help in this process, and ANTLR is the one that is used in this thesis.

ANother Tool for Language Recognition (ANTLR) is an open-source framework written in Java. It allows designing compilers, interpreters, and translators with a variety of programming languages (e.g., Java, C++, and C#) based on self-written grammar rules similar to Extended Backus–Naur Form (EBNF) (Garshol, 2008). ANTLR uses a top-down parser generator that uses LL(*) parsing algorithm with EBNF notation.

An interesting feature of ANTLR is that it provides a convenient means of error recovery and reporting systems. Moreover, ANTLR has many advantages including: an object-oriented design, an integrated lexical and syntactic analysis, and the ability to generate readable and maintainable code.

The work conducted in this thesis is based on ANTLR 4.4. The customized lexical analysis (lexer) and syntax analysis (parser) rules are written in a grammar document, which is a text file with (.g4) extension. Based on this grammar document, ANTLR's workflow is defined as follows: a Lexer class containing the tokens is initially created, a Parser class containing various grammar rules recursively defined using EBNF expressions, and an optional TreeParser class defined based on the AST generated by the Parser class. The relationship between characters, tokens, and ASTs is shown in Figure 3.4.

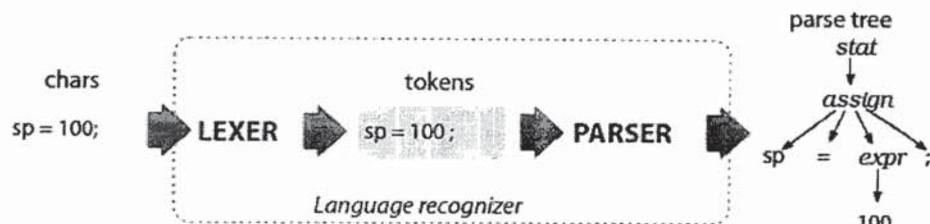


Figure 3.4 The Relationship between Characters, Tokens, and ASTs (Parr, 2013, p. 10)

Lexers and parsers are generated with ANTLR as follows:

1. A text file named 'Java.g4' is created. The file's contents include Java-class lexical analysis and syntax analysis rules defined by EBNF. A snippet of this file is shown in Figure 3.2.

2. ANTLR is used to generate Java source code of Lexer, Parser, and TreeParser (Listener) classes according to 'Java.g4.'
3. A management system is designed and implemented by calling the starting parser rule and taking lexical analysis, syntax analysis and semantic checking, respectively, by the use of Lexer, Parser, and TreeParser classes in that system.

3.5.2 Grammar Writing

A grammar file (GrammarName.g4) always begins with the statement "*grammar GrammarName;*." The GrammarName is the name of the filename. In other words, the filename containing grammar X must be called X.g4.

Except for the statement "*grammar GrammarName;*", the rest of the grammar file defined below this line such as rules and patterns is fully functional. An ANTLR grammar consists of two logical parts: a header and a body. In the header, there is the meta-data about the grammar and details that help the code generator. The body consists of the lexical and syntactic rules of the grammar.

The header is the user-defined code section otherwise known as named global actions (Parr, 2013, p. 175). It consists of header and member actions. To specify a header action, @header {...} is used in our grammar where package/import statements are injected. To inject fields or methods into the generated code, @members {...} is used. These named actions apply to both the parser and the lexer in the grammar.

The body is the set of rules which are written in an EBNF-like syntax and defined recursively. The format of an EBNF rule is as follows:

a : b ;

The symbol ':' separates the LHS from the RHS and the semi-colon ';' appears at the end of every statement/rule of the grammar file. The above rule representation denotes that the symbol 'b' on the RHS can be placed instead of the symbol 'a'. This replacement process is repeated until there is no LHS symbol referencing a rule in the grammar. All the symbols are denoted as tokens. The symbols on the RHS may be unlimited—for example:

a : b | c ; (either the symbol 'b' or symbol 'c' is chosen to be replaced in the place of symbol 'a').

The ANTLR tool generates recursive parsers from the grammar rules beginning with the start symbol. The start symbol becomes the root of the generated parse tree. The parsing begins at the root of the parse tree and proceeds towards the leaves (tokens). This type of parsing is called top-down parsing. Parser rule names must start with a lower-case letter, and lexer rules must start with a capital or upper-case letter. To denote the end of file token inside the ANTLR rules, simply use EOF, which means the parsing ends once all the input has been matched.

As a summary of what is stated in this section, any grammar file will look like Figure 3.5.

```
grammar GrammarName;

@header {
    // package/import statements
}
@members {
    // variables and methods declaration
}

// parser rules
startRule : ruleName; // startRule is the start symbol

ruleName : EndRule EOF;

// LEXER rules
EndRule : 'end' ;
```

Figure 3.5 Grammar File General Overview

3.5.3 Building Grammar

By following the syntax described in Section 3.5.2, here in the thesis the filename is given as Java. Hence, our grammar file (Java.g4) should begin with a "*grammar Java;*" statement. In the header action, some standard Java utility classes such as HashMap for storing (key, value) pairs to implement identifier-comment analysis need to be imported whereas the HashMap field declaration will be put in the members action as the following:

```
grammar Java;

@header {
      import java.util.HashMap;
```

```
}
@members {
        public HashMap<String,String> classCollectorCmnt = new HashMap<String,
String>();
}
```

In our case '*compilationUnit*' is the start symbol, which is the root of the generated parse tree. The start rule comes as follows:

```
compilationUnit
    :   packageDeclaration? importDeclaration* typeDeclaration* EOF
    ;
```

Here, the characters '?' (zero or one time) and '*' (zero or more times) means that the symbol to the left of the operator is optional.

Let us take the simple example shown in Figure 3.6 after adding a comment to the previously parsed Java class in Section 3.4.1 and see how to add/update the grammar rules for parsing the block comment of the given class.

```
/**
 * The HelloWorld class implements an application that
 * simply prints "Hello World!" to standard output.
 */
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 3.6 HelloWorld Class Example

First, new grammar rules (parser and lexer rules) for the comments, whether block or TODO, have to be declared by taking into consideration that a comment might not exist (by using the optional operator *). Such rules come as follows:

```
// parser rules
comments // handles block and TODO comment
 : (TODO_COMMENT {todo = 1; theCmnt = $TODO_COMMENT.text;} | COMMENT {blockCmnt
= 1; theCmnt = $COMMENT.text;})*
 ;

// LEXER rules
COMMENT // handles nested comments
    :    '/*' (COMMENT| . )*?  '*/'
    ;
```

```
TODO_COMMENT
    :   '// TODO' ~[\r\n]*
    ;
```

Second, by adding the LHS of the *comments* rule in front of the *typeDeclaration* symbol in the start rule *compilationUnit*, the parser will successfully parse any comment if it exists prior to any class declaration.

```
compilationUnit
      :   packageDeclaration? importDeclaration* comments typeDeclaration* EOF
    ;
```

3.5.4 Parser of Java Language

To implement a language, an interpreter that reads and recognizes all the valid sentences, phrases, sub-phrases and input symbols of that language should be implemented. Having already defined the parsing concepts (Section 3.4) and made use of the ANTLR library (Section 3.5), the parser should be developed. By following the structure of ANTLR, one has to translate the syntax of the Java language into lexer and parser rules of ANTLR for the purpose of the grammar file construction.

First, all the lexer rules that were to be used during the composition of the parser rules should be defined. Most of these were keywords, separators or operators (e.g. "{", "(", and "||"), while others were combinations of characters—for example, the definition of the lexer rule for white spaces including space, new line, etc.

```
WS  :  [ \t\r\n\u000C]+ -> skip
    ;
```

Having defined all the lexer rules, the parser rules should be defined. Given a simple Java program, it can be composed of classes, inner classes, methods, method body, variables, and so forth. Also, the method body can be further composed of various statement blocks. Each statement block can be further composed of statements. Sample statements are assignment statement, IF Statement, and For Statement.

To capture the entire structure of the Java source code, all the necessary parser rules in the grammar file must be defined. Later, with the help of the compiler, the necessary information can be retrieved and the Abstract Syntax Tree (AST) can be populated. The

Java grammar file from the ANTLR specification is provided by Parr and Harwell (Parr & Harwell, 2013).

Comments are parts of the source-code that are ignored by the compiler. Once the grammar file is defined for the Java source-code parsing, an update to the parser rules can be done for capturing and parsing all of the comments. The next step is to build the actual compiler to do the parsing of source-code comments, and build an interpreter for retrieving and analyzing the necessary information.

3.6 Interpreter Implementation

After constructing the grammar, different files are automatically generated by ANTLR once the grammar file (Java.g4) has been saved. A successful build of the workspace is seen on the ANTLR console as shown in Figure 3.7. For invoking the implementation of the output results, an interpreter is implemented using the Java programming language by calling the "`parseComments(String FILE_DIR, String OUTPUT_FILE)`" method in a class called JavaAnalyzeComments.java. The implementation phases are shown in Figure 3.8.

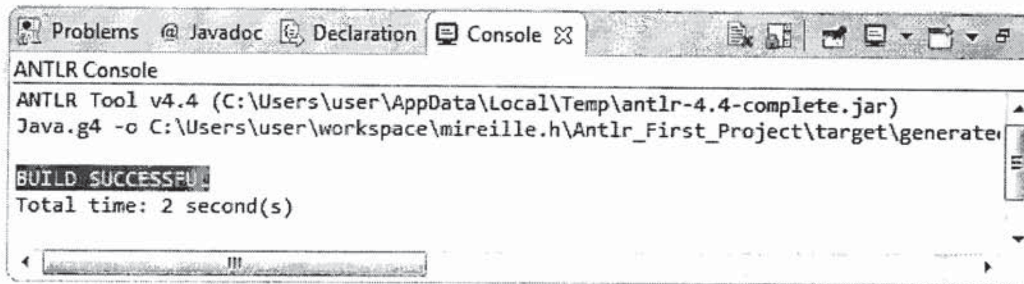


Figure 3.7 ANTLR Console

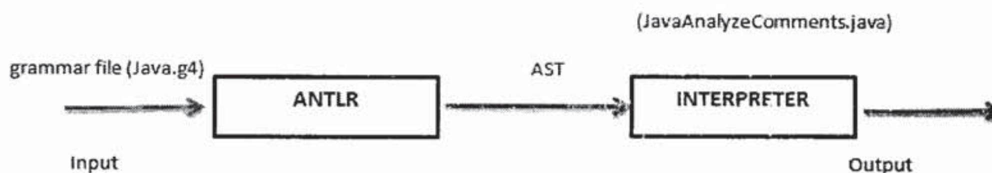


Figure 3.8 Implementation Phases

3.6.1 ANTLR Generated Files

From the grammar file *Java.g4* in a Java project in Eclipse, ANTLR generates many files with an extension .java, .tokens, .class such as: JavaParser.java, JavaLexer.java, JavaBaseListener.java, and JavaListener.java. The following sub-sections explain what each file contains.

3.6.1.1 JavaParser.java

This file contains the parser class definition, which is specific to Java grammar which recognizes the language syntax. It also contains a method for each rule in the grammar.

3.6.1.2 JavaLexer.java

This file contains the lexer class definition generated for analyzing each and every lexical rule and literal in the grammar. The lexer tokenizes the input breaking up into vocabulary symbols.

3.6.1.3 Java.tokens

For each defined token, this file contains a token type number. These values are needed when larger grammars must be split into multiple small grammars so that ANTLR can synchronize all the token type numbers.

3.6.1.4 JavaListener.java and JavaBaseListener.java

By default, AST is automatically built up by ANTLR parsers. ANTLR has a walker class, ParseTreeWalker, which knows how to walk these parse trees, triggers, events, or callbacks in the listener implementation object that is created. The JavaListener is an interface containing enter and exit methods for each rule in the parse grammar. JavaBaseListener is a set of empty implementations of all listener interface methods.

3.6.2 Interpreter File

Referring to Figure 3.8, the next step is to implement the interpreter of the Java programming language where an AST is given as an input with some of the methods of the subclass JavaBaseListener are going to be called.

3.6.2.1 JavaAnalyzeComments.java

This file is created for integrating a generated parser into a Java program. The "`parseComments(String FILE_DIR, String OUTPUT_FILE)`" method in this file invokes parser initialization and prints out the results into an external output file (OUTPUT_FILE). The most important contents of this file are as follows:

```
ANTLRFileStream stream = new ANTLRFileStream(resultFileList.get(f));
```

This line creates an InputStream of characters for the lexer that is used to read from the standard input. In our case the input is a file 'f' from a list of directory files (FILE_DIR) with an extension .java only.

```
JavaLexer lexer = new JavaLexer(stream);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
```

Next, through the above lines of code the lexer and parser objects are created derived from the Java grammar file and a TokenStream Pipes between them.

```
parser.compilationUnit()
```

This line launches the parser that starts parsing from the start symbol rule of the Java grammar file *'compilationUnit'*.

An important step is writing the output results to an output file (OUTPUT_FILE). This output text file summarizes the results with a comma (,) which serves as separation for further statistical excel representation. Such representation illustrates information about: directory location of the input file that was parsed, entity of the source code to which the parsed comment is related, whether the parsed comment is a block or a TODO comment, number of lines of a comment (LOC), number of words matched, similarity factor, and so forth.

First, the lines of code shown below open the output file and write the headers inside it to be filled with the parsed results.

```
FileOutputStream fos = new FileOutputStream(new File(OUTPUT_FILE));

String cvsTitles =
"FILE_PATH,CLASS_COPYRIGHT,CLASS_NAME,INTERFACE_NAME,METHOD_NAME,CONSTR_NAME,FIE
LD_NAME,ENUM_NAME,SL_COMMENT,ML_COMMENT,LOC,WORDS_MATCH,SIMILARITY_FACTOR";

fos.write(cvsTitles.getBytes());
```

Second, global public variables are declared before any grammar rule in the Java grammar file for hosting the acquired results. For every source code entity (class, constructor, method, field, enumeration, etc.), one Hashmap (key, value) is declared for a block comment type where the *key* is the entity name and the *value* is the comment itself (/* comment */) to be used for counting the number of its lines or for any further analysis in the interpreter file JavaAnalyzeComments.java. Another hashmap is created for the names of the entities that do not have any comments. An example showing the global variables that are declared for the class entity is given in Figure 3.9.

```
grammar Java;
@header {
   import java.util.HashMap;
}
@members {
   public int blockCmnt = 0;
   public String theCmnt = "";
   /* Class */
   public HashMap<String,String> classCollectorCmnt = new HashMap<String, String>();
   public HashMap<String,String> classCollectorNoCmnt = new HashMap<String, String>();
}
```

Figure 3.9 Grammar Global Variables Declaration

Calling such global variables is straightforward after launching the parser as follows:

```
HashMap<String, String> hClassMap = new HashMap<String,
String>(parser.classCollectorCmnt);
```

Given a key and a value stored in a Map object, it can be easily used to retrieve the values using the Map method and to write the corresponding results into the output text file.

3.6.3 Error Handling

ANTLR is very flexible and capable of producing better error recovery messages. It usually emits syntax error messages when it encounters a token mismatch. Such helpful error messages provide detailed information by describing a specific error as much as possible. This includes the line on which the variable is declared, the line(s) it is referenced to, the lexical scope to which it belongs, and between which lines of the source code it is

encountered. ANTLR automatically stores that information on every token in the token stream making it easy for the parser to extract the required details for faster semantic analysis. Consider the example shown in Figure 3.10 where an error is recognized. The error in this case is a missing semicolon at the end of line 8.

```
1  /**
2   * The HelloWorld class implements an application that
3   * simply prints "Hello World!" to standard output.
4   */
5  public class HelloWorld {
6
7      public static void main(String[] args) {
8          System.out.println("Hello World!")
9      }
10 }
```

Figure 3.10 Example with Syntax Error

Upon parsing the closing bracket of the "`println`" function call, ANTLR expects a semicolon to end the statement. However, the next token in the stream is a brace "}". As ANTLR has not found what it was looking for, it gathers the necessary information and generates the error message shown in Figure 3.11.

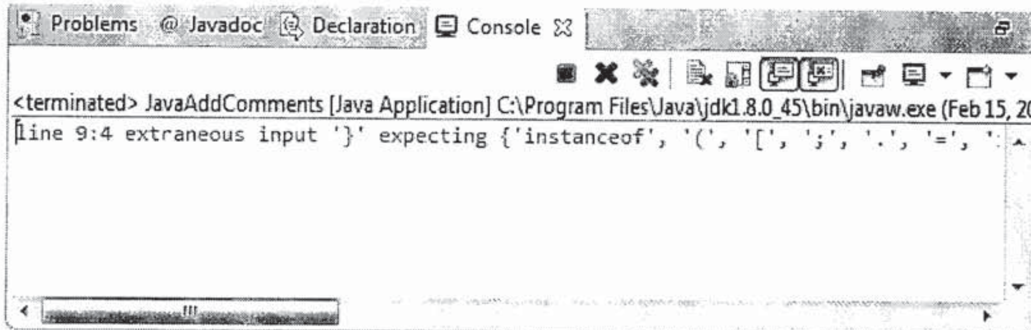


Figure 3.11 Syntax Error Message

Chapter 4: Applying Comment Parsing and Analysis to an Open-Source Software

This chapter presents a broad overview of the semantic analysis approach proposed in the present thesis with the technical details on its implementation. The source code of Eclipse, an open-source Integrated Development Environment (IDE) written in Java, was used to test the proposed approach.

4.1 Introduction

As a good practice, software programmers are required to select meaningful identifier names. Identifiers constructed by developers may contain useful information that is often the starting point for program comprehension activities. Identifiers are considered one of the most important sources of information about system concepts (Caprile & Tonella, 2000). Furthermore, by writing comments, programmers attempt to describe the source code with useful and meaningful information. Therefore, analyzing and matching the similarity between the identifier names and comments in the Java classes can be used to measure how effectively they facilitate program comprehension, testing, reusability, and maintainability.

The approach presented in this thesis was developed using Java (JDK 1.8), Eclipse 4.4 (JEE Luna), and ANTLR v4.4 (antlr-4.4-complete.jar). The Java language parser supports all original source code of JDK 1.2 and higher. The graphical user interface (GUI) of the supporting tool was implemented using Swing.

4.2 Retrieving Information

Before a meaningful semantic analysis can be performed on Java-file comments written in natural language, the Java file first needs to be parsed. This process involves chunking the

sentences of the file and tagging the individual words of a given phrase with its corresponding part-of-speech (POS). POS tagging means that every token is tagged with its corresponding grammar rule.

A way of retrieving the necessary information upon parsing a Java source-code file is by using HashMap data structures, where keys represent the identifier-names and values represent the corresponding comments. Such HashMaps will be put in the member action as was mentioned in Section 3.5.3.

For each type of identifier (Class, Interface, Constructor, Method, Field, or Enumeration), 2 HashMaps are declared. The first stores the identifier-comment pairs if the comment exists, and the second stores the identifiers without comments for further warning about uncommented identifiers.

For parsing and retrieving information about the existence of TODO comments (//TODO), HashMaps are also used for storing the location of such comments within the Java file. The location is the associated identifier that is revisited by the programmer later on to show that there is still work to be done on the marked piece of code.

Therefore, in order to generate output and update data structures, actions have to be embedded in the grammar. After the parser matches the entire grammar rule, the identifier-comment pair is stored in the HashMap. For example, the action "$Identifier.text" refers to the text matched by the Identifier reference and ANTLR translates it to getText(). The snippet of the grammar file presented in Figure 4.1 shows how 2 HashMaps are used to retrieve class-comment information.

To launch the parser, the usual code sequence is to create an input stream, attach a lexer to it, create a token stream attached to the lexer, and then create a parser attached to the token stream (as shown in Figure 2.5). Then, by simply calling the start symbol rule of the grammar, the global HashMap variables can be directly loaded for invoking the implementation of the semantic approach on the output results by performing some computations based on our Java programming language knowledge.

```
@header {
    import java.util.HashMap;
}
@members {
  public int blockCmnt = 0;
  public int todo = 0;

  public String theCmnt = "";

  /* Class */
  public HashMap<String,String> classCollectorCmnt = new HashMap<String, String>();
  public HashMap<String,String> classCollectorNoCmnt = new HashMap<String, String>();
}
comments
 : (TODO_COMMENT {todo = 1; theCmnt = $TODO_COMMENT.text;} | COMMENT {blockCmnt = 1; theCmnt = $COMMENT.text;})*
 ;

typeDeclaration
    :   comments classOrInterfaceModifier* classDeclaration
    |   comments classOrInterfaceModifier* enumDeclaration
    |   comments classOrInterfaceModifier* interfaceDeclaration
    |   classOrInterfaceModifier* annotationTypeDeclaration
    |   ';'
    ;
classDeclaration
    :   'class' Identifier {if(blockCmnt==1) {classCollectorCmnt.put($Identifier.text,theCmnt); blockCmnt = 0;}
                           else if(todo==1){classCollectorTODO.put($Identifier.text,theCmnt); todo=0;}
                           else classCollectorNoCmnt.put($Identifier.text,"");}
        typeParameters?
        ('extends' type)?
        ('implements' typeList)?
        classBody
    ;
```

Figure 4.1 Retrieving Information upon Parsing

4.3 Semantic Similarity Approach

Computing the semantic similarity between the identifier name and its corresponding comment is not a trivial task, due to the variance of natural language expressions. This section explains the different steps that were used for computing this similarity.

4.3.1 Preprocessing

First, the output stream of tokens is obtained for both the identifier names and their corresponding comment text. The main goal of this step is to reduce the forms of tokens to a common base form for further analysis. In this section, the preprocessing steps are discussed.

4.3.1.1 Tokenization

The tokenizer splits the identifier name into chunks. Different output token streams (sequences of chunks of input text) are obtained depending upon the type of tokenizer that

is used. Camel case is a naming convention in which a name is formed of multiple words that are joined together as a single word. The first letter of each word is capitalized so that the words that make up the name can be easily read. For example, "getDayOfMonth" and "GPSState" follow camel case rules.

For splitting the identifier names, there are two possible cases to consider:

1. Character at position i is lower case and character at position i+1 is upper case (e.g., getString)
2. Character at position i is upper case and character at position i+1 is lower case (e.g., IDNumber)

Case 1 is a straightforward Camel case, and the splitting occurs before the upper-case letter where the output stream of tokens is as follows: ("get" and "String"). However, the splitting in case 2 occurs before the last upper-case letter where the output stream of tokens is as follows: ("ID" and "Number").

The underscore ("_") character is also used as a word separator in the tokenization process.

4.3.1.2 Stop Words

Stop words such as: "the," "at," "is," and "and" are removed to focus on the significant semantic words in the context.

4.3.1.3 Normalization

After the first two steps, lower-case folding and alphabetical sorting is applied on the obtained output stream of tokens of both the identifier names and their corresponding comment text. This is done to achieve the highest similarity comparison.

4.3.1.4 Cleaning Messy Comment

All the idioms that do not add valuable semantic meaning to a comment are removed like tags, all the lines that start with the character '@', and so on. For example, the highlighted parts shown on the comment presented in Figure 4.2 should be removed before any further analysis.

```
/**
 * Returns a new object of class '<em>Dawn Generator</em>'.
 * <!-- begin-user-doc --> <!-- end-user-doc -->
 *
 * @return a new object of class '<em>Dawn Generator</em>'.
 * @generated
 * @since 1.0
 */
```

Figure 4.2 Cleaning Messy Comment

4.3.2 Stemming

The process of reducing a word to its root is called stemming. This process is performed for achieving the highest semantic similarity between the tokens. For example, words like "search," "searched," and "searching" are semantically related to the word "search." In this phase, the greatest common prefix with at least 3 characters is matched when comparing the tokens of both output streams.

4.3.3 Computing Similarity

With the first two phases, a tokenizer is used to produce tokens from the given input, apply different normalization processes, and use stemmers to reduce the tokens. Hence, two sets of keywords that truly represent the original text are produced. These keywords can be treated as sets of unique tokens. To calculate the similarity between the two streams, it would be better to represent the similarity in terms of a number that represents how similar two contents are on a 0 (not similar) to 1 (completely similar) scale. The total number of common words is also indicated as an additional semantic similarity measure. The algorithm for computing similarity between identifier names and comments is illustrated in Figure 4.3.

```
/**
 * returns the greatest prefix match between 2 strings
 */
public String greatestCommonPrefix(String a, String b) {
    int minLength = Math.min(a.length(), b.length());
    for (int i = 0; i < minLength; i++) {
        if (a.charAt(i) != b.charAt(i)) {
            return a.substring(0, i);
        }
    }
    return a.substring(0, minLength);
}

/**
 * Create a array of words (lowercase) and sort them in alphabetical order
 * @param str
 * @return
 */
public String[] normalize(String str)
{
     String[] result = str.split(" ");
      for (int i = 0; i < result.length; i++)
          result[i] = result[i].toLowerCase();
      Arrays.sort(result);
      return result;
}

public String[] compare(String comment, String identifier) {
      int nbWordsMatch = 0;
      float simFactor = 0;
      // Normalize comment
      String[] commentArray = normalize(comment);
      // Normalize Identifier
      String[] identifierArray = normalize(identifier);

      int p0 = 0, p1 = 0;
      while (p0 < commentArray.length && p1 < identifierArray.length)
      {
        String comPref = greatestCommonPrefix(commentArray[p0],identifierArray[p1]);
        if(comPref.length() > 2) // if at least 3 characters are matched; stemming
        {
            commentArray[p0] =  comPref;
            identifierArray[p1] =  comPref;
        }
        int comp = commentArray[p0].compareTo(identifierArray[p1]);
        if (comp == 0) { // same word
          nbWordsMatch++;
          p0++;
          p1++;
        }
        else if (comp < 0) { // commentArray word is before identifierArray word
          p0++;
        }
        else if (comp > 0) { // commentArray word is after identifierArray word
          p1++;
        }
      }
      simFactor = (2.0f * nbWordsMatch) / ( commentArray.length + identifierArray.length );

      String[] returnResult = new String[2];
      returnResult[0] = Integer.toString(nbWordsMatch);
      returnResult[1] = String.valueOf(simFactor);

     return returnResult;
  }
```

Figure 4.3 Algorithm for Semantic Similarity Computation

4.4 Testing the Approach with an Open Source Software

This section presents a study that was carried out to test the approach presented in the present thesis. Comment parsing and analysis was applied to the source code of Eclipse, an open-source IDE.

4.4.1 Analyzing the Source Code

In order to analyze the quality of source-code comments and evaluate the similarity measures, the source code of Eclipse was downloaded from one of the online repositories (Stepper, 2015), and the Java files were extracted to a local folder. The content of every Java file (*.java) of the Eclipse project was analyzed, based on the retrieval algorithm described in Section 4.2 and the semantic similarity approach described in Section 4.3. The output results were exported to an Excel workbook for producing charts. To evaluate the results, comments were grouped into 6 categories based on the similarity match value as shown in Table 4.1.

Table 4.1 Comments Categories

Category	Similarity Factor Value	Description
1	>= 0.8	Scary Noise comment
2	>= 0.4 and <= 0.79	Noise comment
3	>= 0.01 and <= 0.39	Normal comment with minor similarity
4	= 0	Probably Meaningful (0% similarity)
5	= -1	Empty comment
6		TODO comments

Note that the "Empty" comments are the ones that do not offer a semantic explanation regarding the functionality of the method or class, but can offer other details (using the annotations that start with @). Such details can include the version, the parameter list, the return values, the thrown exception or the generated annotation as shown in Figure 4.9.

Therefore, this type of comment is empty of any semantic information to be compared with its corresponding identifier name and thus its similarity factor value is set to a constant equal to -1.

The tool that was developed for parsing and analyzing the comments is shown in Figure 4.4. The "Browse…" button is clicked for selecting the folder of the target project and the "Save File…" button is clicked for selecting the file where the output results are to be stored. Then, the "Start" button is pressed to generate the output results in the grid. The rows in the results grid are colored based on the value of the last column "SIMILARITY_FACTOR" highlighting the comment categories mentioned in Table 4.1 as shown in Figure 4.5. Scary noise and noise comments are colored in RED, probably meaningful comments in ORANGE and empty comments in GREY. The total time to run and parse the selected Eclipse open-source project (744 Java files) is around 10 seconds.

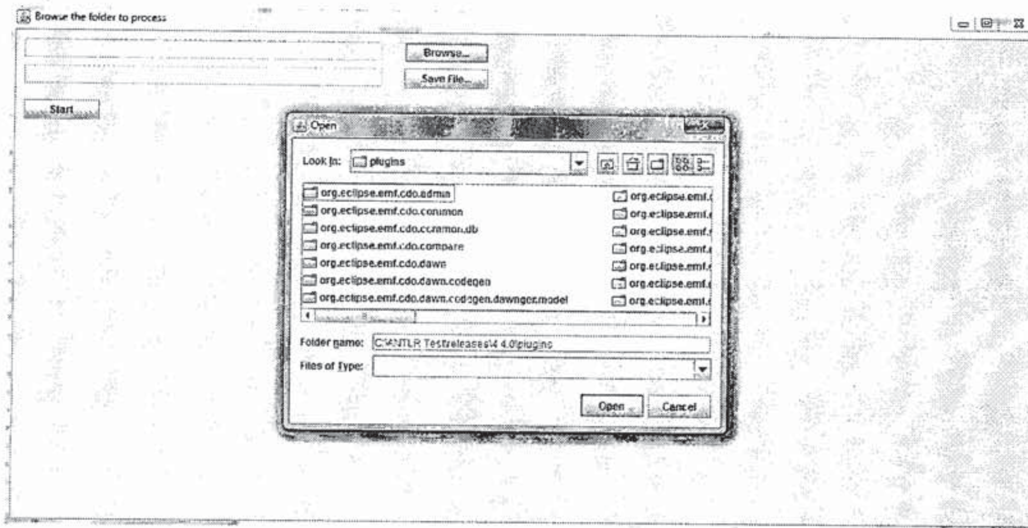


Figure 4.4 Support Tool – Selecting the Target Project

Figure 4.5 Support Tool – Showing the Output after Parsing and Analysis

4.4.2 Results

The distributions percentages of the previously defined comment categories (Table 4.1) are presented in Figure 4.6 for all of Eclipse's source code.

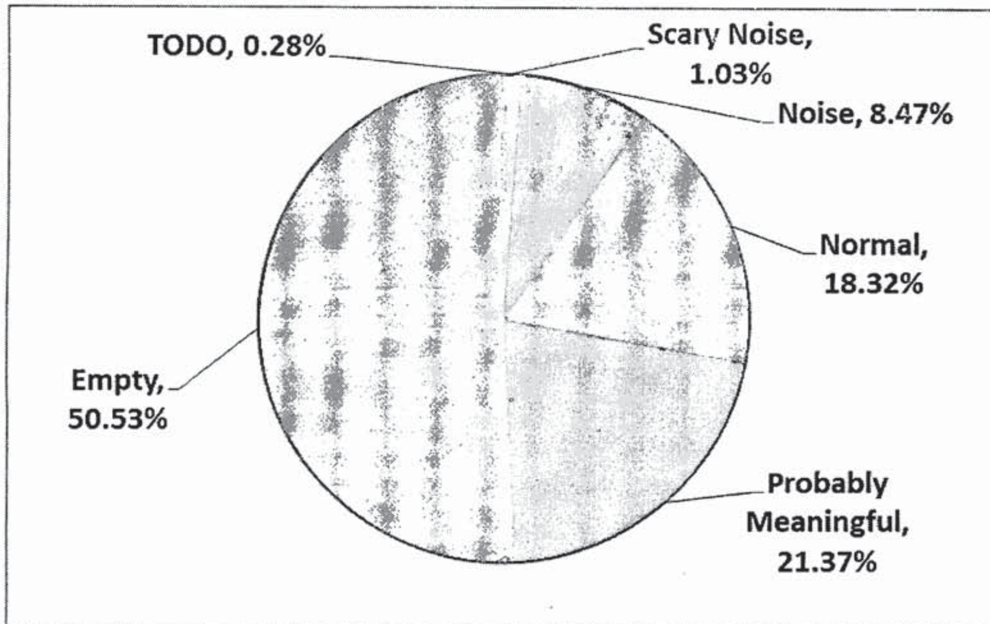


Figure 4.6 Comment Category Percentage

The percentage was determined by the proportion of comments in the data set that are covered by the calculated similarity factor bounds. As indicated in the chart shown in Figure 4.6, more than 8% of the comments were categorized as noise comments. This is not a good quality indicator, especially if it is considered that the selected Eclipse distribution is a small version with only 744 Java files. The code snippets shown in Figure 4.7 and Figure 4.8 demonstrate examples of noise and scary noise comments, respectively. These comments seem to have been automatically generated by a tool and neglected afterwards.

```
/**
 * Get the file from the page. <!-- begin-user-doc --> <!-- end-user-doc -->
 *
 * @generated
 */
public IFile getModelFile()
{
  return newFileCreationPage.getModelFile();
}
```

Figure 4.7 Noise Comment Example

```
/**
 * Runs the wizard in a dialog.
 *
 * @generated
 */
public static void runWizard(Shell shell, Wizard wizard, String settingsKey)
{
  IDialogSettings pluginDialogSettings = EcoreDiagramEditorPlugin.getInstance().getDialogSettings();
  IDialogSettings wizardDialogSettings = pluginDialogSettings.getSection(settingsKey);
  if (wizardDialogSettings == null)
  {
    wizardDialogSettings = pluginDialogSettings.addNewSection(settingsKey);
  }
  wizard.setDialogSettings(wizardDialogSettings);
  WizardDialog dialog = new WizardDialog(shell, wizard);
  dialog.create();
  dialog.getShell().setSize(Math.max(500, dialog.getShell().getSize().x), 500);
  dialog.open();
}
```

Figure 4.8 Scary Noise Comment Example

Moreover, a significant rate of more than 50% of the comments fall into the category of empty comments. Hence, more than half of the existing comments do not add any helpful or meaningful semantic explanation for the implementation of their corresponding entities. The code snippet presented in Figure 4.9 demonstrates examples of empty comments. The first comment only mentions that the method has been part of the software since version 4.4, the second comment only rewrites the parameter and the thrown exception, and the

third comment only contains the generated annotation. But none of the three comments add an explanation regarding what the function does.

```
/**
 * @since 4.4
 */
public static CDOBranchPoint normalizeBranchPoint(CDOBranchPoint branchPoint)
{
  long timeStamp = branchPoint.getTimeStamp();
  if (timeStamp == CDOBranchPoint.UNSPECIFIED_DATE)
  {
    return branchPoint;
  }

  CDOBranch branch = branchPoint.getBranch();
  return doNormalizeBranchPoint(branch, timeStamp);
}

/**
 * @param javaProject
 * @throws JavaModelException
 */
private void addJREContainerToProject(IJavaProject javaProject) throws JavaModelException
{
  addToClasspath(javaProject, JavaRuntime.getDefaultJREContainerEntry());
}

/**
 * @generated
 */
public ResourceSetInfo getResourceSetInfo(Object editorInput)
{
  return (ResourceSetInfo)super.getElementInfo(editorInput);
}
```

Figure 4.9 Empty Comment Example

A deeper analysis for the high rate of empty comments is represented by the number of files where these comments exist. As shown in Figure 4.10, empty comments exist in 466 out of 744 files. This indicates that empty comments span more than 62% of all the project files. Additionally, there is a rate of around 20% (148 files) of the files containing noise comments and a rate of more than 4% (32 files) of scary noise comments. These numbers provide important insights and highlight the importance of having a tool that can automatically analyze the quality of source-code comments.

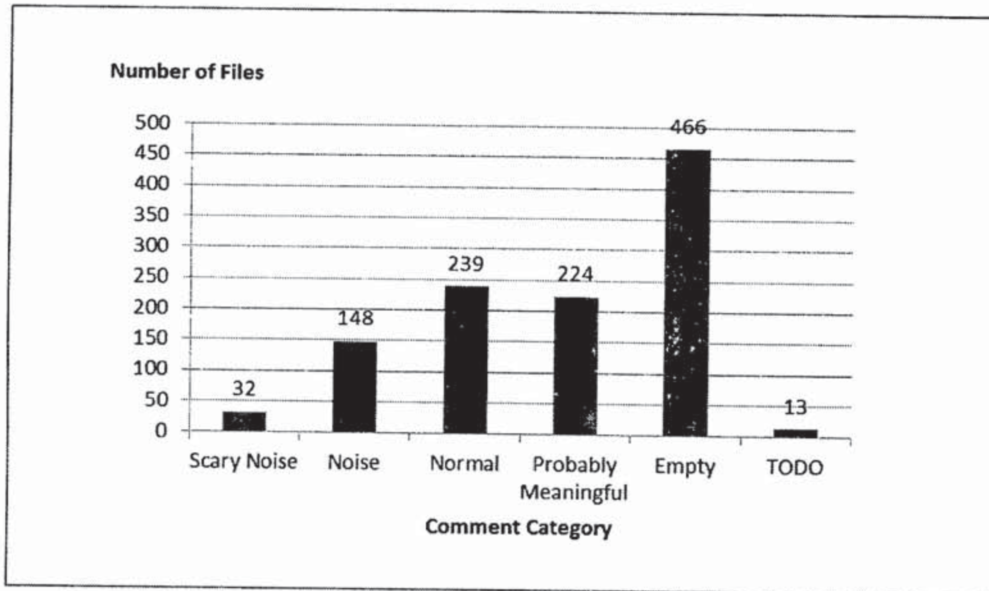


Figure 4.10 Distribution of the Different Types of Comments across the Source-Code Files

For a better interpretation of comments' dispersion across source-code files, a boxplot was used for representing the number of each type of comment per file. The results shown in Figure 4.11 only cover the files that have comments. A primary analysis shows a majority of extreme outliers in all of the empty, noise, normal and probably meaningful comments. Outliers are often bad data points that merit careful investigation.

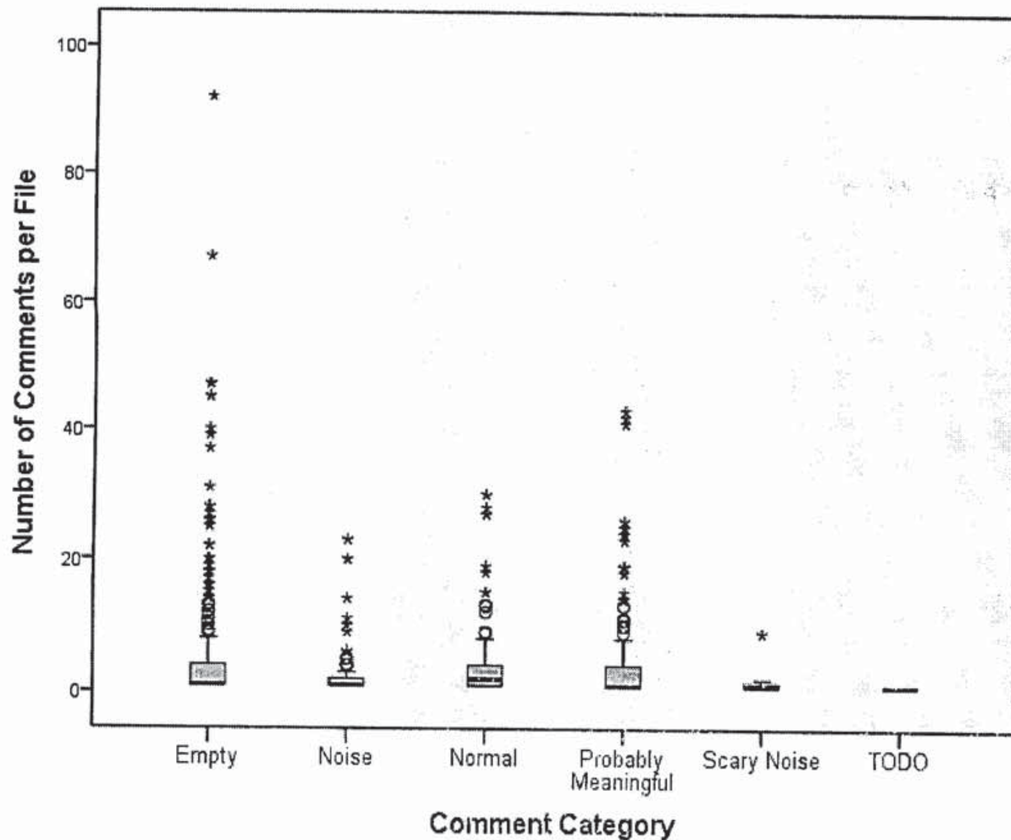


Figure 4.11 Number of Each Type of Comment per Source-Code File

Descriptive statistics of the data presented in Figure 4.11 are reported in the table of Figure 4.12. Obviously, an excessive number of bad comments (23 noise or 9 scary noise) in particular files is simply noisy and provides no meaningful documentation. Moreover, an average (Mean) of 3 noise comments and 2 scary noise comments per file needs improvement. On the other hand, a redundant amount of 92 or 67 empty comments (all only with @generated annotation) in a single file highlights the importance of having a tool that can automatically remove all these empty comments and leave/create that of the main class to differentiate user-written code from the whole class-file generated code. Additionally, by analyzing the 43 probably meaningful comments in a single file, it was observed that more than 50% of the comments are totally meaningless.

Comment Category	Mean	Median	Std. Deviation	Minimum	Maximum	N	% of Total N
Empty	4.92	1.00	9.187	1	92	466	41.5%
Noise	2.59	1.00	3.673	1	23	148	13.2%
Normal	3.48	2.00	4.516	1	30	239	21.3%
Probably Meaningful	4.33	1.00	6.935	1	43	224	20.0%
Scary Noise	1.47	1.00	1.436	1	9	32	2.9%
TODO	1.00	1.00	.000	1	1	13	1.2%
Total	4.05	1.00	7.191	1	92	1122	100.0%

Figure 4.12 Descriptive Statistics of Comment Types per File

Finally, as shown in Figure 4.11, most of the class identifiers have comments.

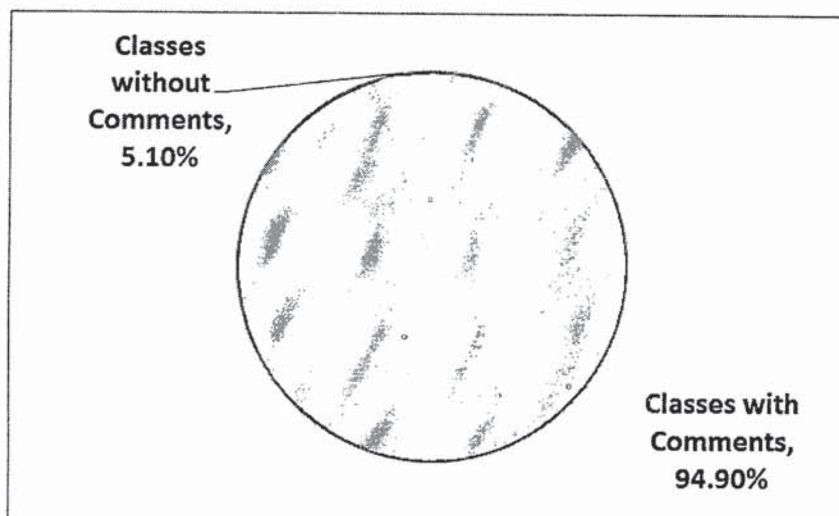


Figure 4.13 Classes with/without Comments by Percentage

Similarly, most interface identifiers have comments as shown in Figure 4.12.

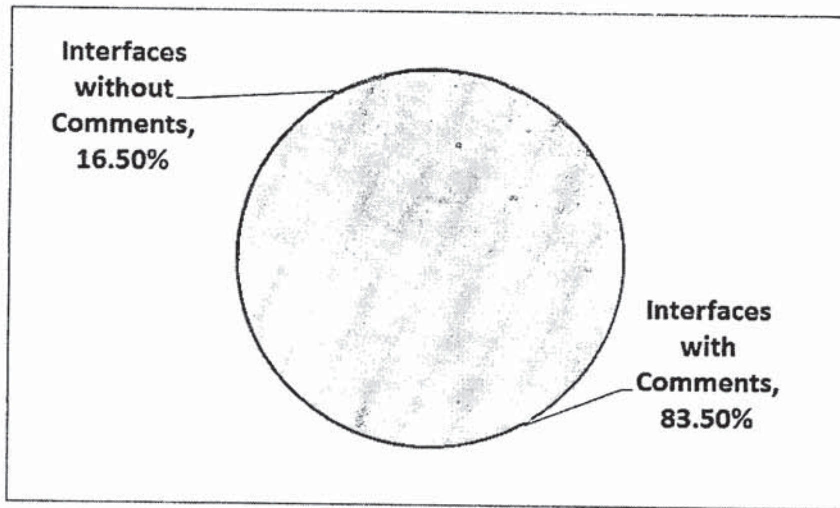


Figure 4.14 Interface with/without Comments by Percentage

On the other hand, more than 50% of the methods lack comments as shown in Figure 4.13.

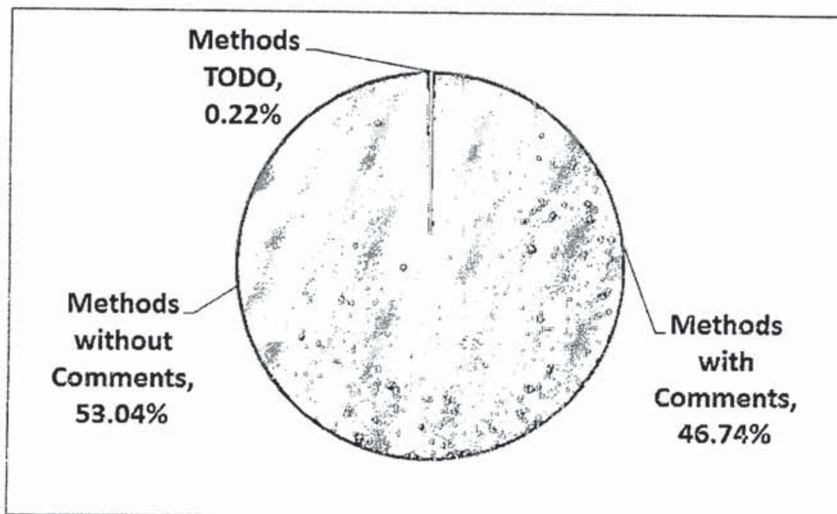


Figure 4.15 Methods with/without Comments Percentage

4.4.3 Result Discussion

After observing the results, one can see that although comments are used extensively throughout the project, as shown in Figure 4.10 to Figure 4.15, the quality of these comments requires improvement. Hence, to improve the software's maintainability, it

would be better to remove empty comments that do not convey any meaning. It would also be better if the identifier names are made more meaningful instead of using noise or scary noise comments. Giving a warning about the lack of comments could be helpful in making the source-code more understandable.

4.4.4 Threats to Validity and Limitations

Threats to validity concern the relation between the theory and the observation. In this thesis, a threat to validity lies in discerning the developers' intent while writing source-code comments. Therefore, the proposed analysis approach is based on extracting semantic information from the corresponding identifiers for the possibility of deducing the developers' likely intent.

The diversity of developers' skills and different code styles also produces a limitation in constructing and splitting the identifiers. The proposed approach is limited to a simple string matching based on a Camel case splitter. Yet identifiers could have been split in different ways by the developers who originally created them.

The outcome of the study presented in this thesis could be used for analyzing developers' behavior and practices when using comments. However, the results of this analysis cannot be generalized because the study only considered one software project. If the approach is applied to the source code of other software projects in the future, such as an analysis could be conducted and the results would be more generalizable.

Chapter 5: Conclusions and Future Work

To conclude this thesis, the present chapter recalls the main problem tackled herein. It also gives a summary of the results and contributions as well as an overview of the possible future work.

5.1 Main Problem Addressed in this Thesis

Software systems may typically contain thousands and even millions of lines of code. These systems are developed and maintained by a large number of developers. The person performing the maintenance is not always the developer who originally wrote the code. Therefore, without proper documentation of the source code, software maintenance becomes prohibitively expensive. Generally, software maintenance is defined as the adaptation and modification of a software product after delivery for a number of motives. Without exaggeration, 80% to 95% of the budget allocated to software is spent on its maintenance (Erlikh, 2000). Programmers spend more time reading and understanding code (often written by others) than writing it. Following the release and delivery of the product to the end users, software developers keep maintaining the software by updating it based on change requests and changes occurring in the environment, in order to keep it up to date. Well-written code comments help in reducing the effort of understanding software's source code.

Comments are crucial for understanding software programs. Comments can preserve the intentions behind the source code and highlight the implementation details. On the other hand, sometimes a comment just does not mean anything. Such comments may contain repeated and meaningless information and hence do not promote source-code understanding. Moreover, identifiers in the code are a rich source of information. The proper choice of identifiers improves software understandability and evolution. If carefully chosen, identifiers reflect the semantics and role of the named entities (Lawrie et al., 2007).

Thus, the analysis of source-code comments with the identifier names can help in capturing information that represents the developers' knowledge while writing the code.

5.2 Contributions

This thesis presents an approach for the analysis of the quality of source-code comments by using the ANTLR parser generator.

ANTLR is a powerful tool that takes the grammar of a programming language—e.g., Java—as input and produces an output containing selected information from the source code. As part of the proposed approach, comments were categorized primarily based on the level of semantic similarity between them and their corresponding identifier names. This categorization serves as a heuristic for automated analysis of the quality of source-code comments. The proposed approach employs the semantic information extracted from the comment and its corresponding identifier name to realize a set of heuristics for a quality assessment model.

The proposed comment parsing and analysis approach is supported by a tool that is explained in detail in Appendix A:

The proposed approach was evaluated by analyzing the source code of the open-source IDE Eclipse. The results showed that most classes and interfaces, and half the methods had comments. However, more than 50% of the existing comments are empty. These comments were spread over 62% of the source-code files. On the other hand, only 18% of the comments were of a high quality. Furthermore, 8% of the comments were very low-quality noise comments, which are not a good quality comment indicator.

As software continues to increase in size and complexity, software engineers need better source-code documentation to help them understand and maintain their products. This work constitutes an approach towards a detailed, quantitative, and qualitative analysis and assessment of source-code comments. Such assessment of comment quality plays a part in auditing software quality. Enriching the source-code comments with meaningful descriptive information and eliminating useless comments improves program comprehension.

The results of the study demonstrate the importance of having an automated approach for analyzing the quality of source-code comments. This approach has significant potential in improving code quality and thereby facilitating software maintenance.

5.3 Future Work

Additional heuristics can be realized in the future to enhance the automated quality analysis of source-code comments. Some examples include the following:

- Comparing the number of lines in each comment to the number of lines of code in their corresponding entity could help in determining whether a comment is too long or too short. For example, writing very long comments for a few lines of code might result in low-quality comment.
- It is possible to analyze more types of low-quality comments based on Martin's classification (Martin, 2008, p. 59).
- Single-line comment analysis was not part of the case study conducted in this thesis and could be an interesting addition in the future.
- Inline comment analysis could be added in the future by identifying these comments and relating them to the corresponding entity name. Only block comments were analyzed in this thesis.
- More complex strategies other than Camel case for splitting identifier names could be implemented. One example could be Samurai (Ensler et al., 2009).

Moreover, the proposed tool can be enhanced to automatically remove low-quality source-code comments or modify comments and identifier names to make them more understandable and helpful. This tool could also be integrated in an existing IDE such as Eclipse or Visual Studio to make it more directly accessible to developers.

Code refactoring activities can be another way to improve the quality of source-code comments. Hence, the tool can be enhanced to automatically refactor thickly commented code by applying activities such as Extract Method, Introduce Assertion, Replace Parameter with Method and others.

The work presented in this thesis analyzed source-code comments in Java code files. It will be interesting to see how this assessment transfers to different programming languages.

Since this work only focuses on comments, other Java coding conventions such as naming consistency of identifiers may also be assessed in the future.

The case study conducted in this thesis involved parsing the source-code of a single software project. Parsing additional open-source projects could reveal different comment-quality benchmarks and help in further understanding how developers use comments.

Bibliography

Abebe, S. L., Haiduc, S., Tonella, P., & Marcus, A. (2009). Lexicon Bad Smells in Software. In *IEEE 16th Working Conference on Reverse Engineering* (pp. 95–99).

Aggarwal, K. K., Singh, Y., & Chhabra, J. K. (2002). An Integrated Measure of Software Maintainability (pp. 235–241). Annual Reliability and Maintainability Symposium.

Aman, H., & Okazaki, H. (2008). Impact of Comment Statement on Code Stability in Open Source Development (pp. 415–419). Knowledge-Based Software Engineering, M. Virvou and T. Nakamura (Eds). IOS Press.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering Traceability Links Between Code and Documentation. *IEEE Transactions on Software Engineering*, *28*(10), 970–983.

Arrington, C. L. (2009). Improving Software Testing through Code Parsing. *Hampton University*, 5.

Binkley, D. (2007). Source Code Analysis: A Road Map (pp. 104–119). Future of Software Engineering, 2007. FOSE '07.

Bosu, A., & Carver, J. C. (2013). Impact of Peer Code Review on Peer Impression Formation: A Survey. In *7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 133–142).

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2011). Improving the Tokenisation of Identifier Names (pp. 130–154). Presented at the 25th European Conference on Object-Oriented Programming.

Caprile, B., & Tonella, P. (2000). Restructuring Program Identifier Names. In *International Conference on Software Maintenance.*

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using Metrics to Evaluate Software System Maintainability. *Computer*, *27*(8), 44–49.

Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., & Girard, J.-F. (2007). An Activity-Based Quality Model for Maintainability (pp. 184–193). IEEE International Conference on Software Maintenance.

de Souza, S. C. B., Anquetil, N., & de Oliveira, K. M. (2005). A Study of the Documentation Essential to Software Maintenance. In *International Conference on Design of Communication: Documenting and Designing for Pervasive Information* (pp. 68–75).

de Souza, S. C. B., Anquetil, N., & de Oliveira, K. M. (2006). Which Documentation for Software Maintenance?, *12*(3), 31–44.

Earley, J. (1970). *An Efficient Context-Free Parsing Algorithm* (Vol. 13). Communications of the ACM.

Enslen, E., Hill, E.,.Pollock, L., & Vijay-Shanker, K. (2009). Mining Source Code to Automatically Split Identifiers for Software Analysis. IEEE 6th International Working Conference on Mining Software Repositories.

Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional*, *2*(3), 17–23.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman.

Freitas, J. L., da Cruz, D., & Henriques, P. R. (2012). A Comment Analysis Approach for Program Comprehension (pp. 11–20). 35th Annual IEEE Software Engineering Workshop.

Gagnon, E. M., & Hendren, L. J. (1998). SableCC, an Object-Oriented Compiler Framework. In *Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176)* (pp. 140–154).

Garshol, L. M. (2008). BNF and EBNF: What are they and how do they work? Retrieved from http://www.garshol.priv.no/download/text/bnf.html

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments : A "Cognitive Dimensions" Framework, *7*, 131–174.

Haouari, D., Sahraoui, H., & Langlais, P. (2011). How Good is your Comment? A study of Comments in Java Programs. International Symposium on Empirical Software Engineering and Measurement.

Hartzman, C. S., & Austin, C. F. (1993). Maintenance Productivity: Observations Based on an Experience in a Large System Environment. In *Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering* (Vol. 1, pp. 138–170).

Hasan, K. M., & Hasan, M. S. (2010). A Parsing Scheme for Finding the Design Pattern and Reducing the Development Cost of Reusable Object Oriented Software. *International Journal of Computer Science & Information Technology (IJCSIT)*, *2*(3), 40–54.

Hill, E. (2010). *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration* (PhD thesis). University of Delaware.

IEEE1219 (1998). IEEE STD 1219: Standard for Software Maintenance.

Invisible Jacc Version 1.1. (1997). Retrieved from http://www.invisiblesoft.com/jacc

Kernighan, B. W., & Pike, R. (1999). *The Practice of Programming*. Boston, MA: Addison-Wesley Longman.

Khamis, N., Witte, R., & Rilling, J. (2010). Automatic Quality Assessment of Source Code Comments: The JavadocMiner (pp. 68–79). Presented at the 15th International Conference on Applications of Natural Language to Information Systems, Concordia University, Montreal, Canada.

Knuth, E. (2003). Selected Papers on Computer Languages. Presented at the CSLI Lecture Notes, no. 139, Stanford, California: Center for the Study of Language and Information.

Kodaganallur, V. (2004). Incorporating Language Processing into Java Applications: A JavaCC Tutorial. *IEEE Software*, *21*(4), 70–77.

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2007). Effective Identifier Names for Comprehension and Memory, *3*(4), 303–318.

Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive Perspectives on the Role of Naming in Computer Programs. Presented at the 18th Annual Psychology of Programming Workshop.

Liu, S., Zhang, R., Wang, D., Sun, H., Chen, Y., & Li, L. (2008). Implementing of Gaussian Syntax-Analyzer Using ANTLR.

Maalej, W., & Happel, H. J. (2010). Can Development Work Describe Itself? (pp. 191–200). 7th IEEE Working Conference on Mining Software Repositories (MSR).

Maeda, K. (2009). Code Clone Detection Using Parsing Actions (pp. 762–763). 9th International Symposium on Communications and Information Technology.

Martin, R. (2008). *Clean Code.*

McAllister, A. J. (2010). Automation-Enabled Code Conversion. In *International Conference on Software Engineering Research and Practice* (pp. 11–17).

Mitchel, R. L., & Keef, M. (2012). The Cobol Brain Drain,ComputerWorld, *46,*(10), 18–25.

Nurvitadhi, E., Leung, W. W., & Cook, C. (2003). Do Class Comments Aid Java Program Understanding? (Vol. 1, pp. T3C–13 – T3C–17). Presented at the 33rd Annual Frontiers in Education.

Oman, P., & Hagemeister, J. (1992). Metrics for Assessing a Software System's Maintainability. In *Conference on Software Maintenance* (pp. 337–344).

Parr, T. (2005). ANTLRWorks. Retrieved from http://www.antlr3.org/works/

Parr, T. (2007). *The Definitive Antlr Reference*. Oveilly & Associates Inc.

Parr, T. (2013). *The Definitive ANTLR 4 Reference.*

Parr, T., & Harwell, S. (2013). antlr/grammars-v4. Retrieved from https://github.com/antlr/grammars-v4/blob/master/java/Java.g4

Raskin, J. (2005). Comments Are More Important Than Code, *3*(2), 62–64.

Relf, P. A. (2005). Tool Assisted Identifier Naming for Improved Software Readability: An Empirical Study (pp. 53–62). Presented at the IEEE International Symposium on Empirical Software Engineering.

Shepherd, D., Pollock, L., & Vijay-Shanker, K. (2007). Case Study: Supplementing Program Analysis with Natural Language Analysis to Improve a Reverse Engineering Task. In *7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering* (pp. 49–54).

Shokripour, R., Anvik, J., Kasirun, Z. M., & Zamani, S. (2013). Why So Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation (pp. 2–11). 10th Working Conference on Mining Software Repositories.

Sippu, S., & Soisalon-Soininen, E. (2013). *Parsing Theory: Volume II LR(k) and LL(k) Parsing.*

Steidl, D., Hummel, B., & Juergens, E. (2013). Quality Analysis of Source Code Comments (pp. 83–92). San Francisco, CA, USA, 2013 IEEE 21st International Conference.

Stepper, E. (2015). Eclipse CDO Project. Retrieved from https://git.eclipse.org/c/cdo/cdo.git/

Storey, M. A., Ryall, J., Bull, R. I., & Myers, D. (2008). TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers (pp. 251–260).

Strein, D., Lincke, R., Lundberg, J., & Lowe, W. (2007). An Extensible Metamodel for Program Analysis. *IEEE Transactions on Software Engineering*, *33*(9).

Tan, L., Yuan, D., & Zhou, Y. (2007). HotComments: How to Make Program Comments More Useful? In *11th Workshop on Hot Topics in Operating Systems (HOTOS).*

Turuntaev, I. S. (2014). *Upgrading the Computational Core of the Distance Learning System*. Retrieved from https://www.google.com.lb/?gfe_rd=cr&ei=FpgQWceOGczA8gfW6bioAw&gws_rd=ssl#q=ISTuruntaev_Diploma_Full

Van De Vanter, M. L. (2002). The Documentary Structure of Source Code. *Information and Software Technology*, *44*(13), 767–782.

Ying, A. T. T., Wright, J. L., & Abrams, S. (2005). Source Code that Talks: An Exploration of Eclipse Task Comments and their Implication to Repository Mining. In *International Workshop on Mining Software Repositories* (pp. 1–5).

Yu, D., Yan, H., & Wang, J. (2008). Design and Implementation of NC Code Compiler Based on ANTLR. *Computer Applications*, *28*(2), 522–527.

Appendix A: Grammar File

This appendix presents the grammar (Java.g4) of the Java language (Parr & Harwell, 2013), which is used with ANTLR. The highlighted parts were added to the grammar as part of the work done in this thesis on parsing source-code comments.

```
grammar Java;

@header {
    import java.util.HashMap;
    import java.util.ArrayList;
    import java.util.List;
}

@members {
        /* flags */
  public int blockCmnt = 0;
  public int todo = 0;
  /*  the below variable is used locally for multiple constructors,methods,.... having same name but
with different parameters*/
  public int cmnt_var = 0;

  public String theCmnt = "";

  public List<String> myList = new ArrayList<String>();

  /*Copyright*/
  public List<String>  classCopyrightCmnt = new ArrayList<String>();
  /* Class */
  public HashMap<String,String> classCollectorCmnt = new HashMap<String, String>();
  public HashMap<String,String> classCollectorNoCmnt = new HashMap<String, String>();
  public HashMap<String,String> classCollectorTODO = new HashMap<String, String>();
  /* Interface */
  public HashMap<String,String> interfaceCollectorCmnt = new HashMap<String, String>();
  public HashMap<String,String> interfaceCollectorNoCmnt = new HashMap<String, String>();
  public HashMap<String,String> interfaceCollectorTODO = new HashMap<String, String>();
  /* Method */
  public HashMap<String,String> methodCollectorCmnt = new HashMap<String, String>();
  public HashMap<String,String> methodCollectorNoCmnt = new HashMap<String, String>();
  public HashMap<String,String> methodCollectorTODO = new HashMap<String, String>();
  /* Constructor */
  public HashMap<String,String> classConstructorCmnt = new HashMap<String, String>();
  /* Field */
  public HashMap<String,String> fieldCollectorCmnt = new HashMap<String, String>();
  /* Enumeration */
  public HashMap<String,String> enumCollectorCmnt = new HashMap<String, String>();
}

comments
 : (TODO_COMMENT {todo = 1; theCmnt = $TODO_COMMENT.text;} | COMMENT {blockCmnt = 1; theCmnt =
$COMMENT.text;})*
 ;

commentToPassBy : (COMMENT | TODO_COMMENT)*;

// starting point for parsing a java file
compilationUnit
```

```
    :   comments packageDeclaration? commentToPassBy importDeclaration*
                       {if(blockCmnt==1) {classCopyrightCmnt.add(theCmnt); blockCmnt = 0;}}
        typeDeclaration* EOF
    ;

packageDeclaration
    :   annotation * 'package'  qualifiedName  ';'
    ;

importDeclaration
    :   'import'  'static'?  qualifiedName  ('.' '*')? ';'
    ;

typeDeclaration
    :   comments classOrInterfaceModifier* classDeclaration
    |   comments classOrInterfaceModifier* enumDeclaration
    |   comments classOrInterfaceModifier* interfaceDeclaration
    |   classOrInterfaceModifier* annotationTypeDeclaration
    |   ';'
    ;

modifier
    :   classOrInterfaceModifier
    |   (   'native'
        |   'synchronized'
        |   'transient'
        |   'volatile'
        )
    ;

classOrInterfaceModifier
    :   annotation       // class or interface
    |   (   'public'      // class or interface
        |   'protected'  // class or interface
        |   'private'    // class or interface
        |   'static'     // class or interface
        |   'abstract'   // class or interface
        |   'final'      // class only -- does not apply to interfaces
        |   'strictfp'   // class or interface
        )
    ;

variableModifier
    :   'final'
    |   annotation
    ;

classDeclaration
    :       'class'  Identifier  {if(blockCmnt==1)  {classCollectorCmnt.put($Identifier.text,theCmnt);
blockCmnt = 0;}
                                                  else
if(todo==1){classCollectorTODO.put($Identifier.text,theCmnt); todo=0;}
                                                  else
classCollectorNoCmnt.put($Identifier.text,"");}
        typeParameters?
        ('extends' type)?
        ('implements' typeList)?
        classBody
    ;

typeParameters
    :   '<' typeParameter (',' typeParameter)* '>'
    ;

typeParameter
    :   Identifier ('extends' typeBound)?
    ;

typeBound
```

```
    :   type ('&' type)*
    ;

enumDeclaration
    :       ENUM   Identifier  {if(blockCmnt==1)  {enumCollectorCmnt.put($Identifier.text,theCmnt);
blockCmnt = 0;}}
          ('implements' typeList)?
         '{' enumConstants? ','? enumBodyDeclarations? '}'
    ;

enumConstants
    :   commentToPassBy enumConstant (',' commentToPassBy enumConstant)*
    ;

enumConstant
    :   annotation* Identifier arguments? classBody?
    ;

enumBodyDeclarations
    :   ';' classBodyDeclaration*
    ;

interfaceDeclaration
    :                                'interface'          Identifier          {if(blockCmnt==1)
{interfaceCollectorCmnt.put($Identifier.text,theCmnt); blockCmnt = 0;}
                                                                  else
if(todo==1){interfaceCollectorTODO.put($Identifier.text,theCmnt); todo=0;}
                                                                  else
interfaceCollectorNoCmnt.put($Identifier.text,"");}
          typeParameters? ('extends' typeList)? interfaceBody
    ;

typeList
    :   type (',' type)*
    ;

classBody
    :   '{'  classBodyDeclaration* commentToPassBy '}'
    ;

interfaceBody
    :   '{' interfaceBodyDeclaration* '}'
    ;

classBodyDeclaration
    :   ';'
    |    commentToPassBy 'static'? block
    |    comments modifier* memberDeclaration
    ;

memberDeclaration
    :   methodDeclaration
    |   genericMethodDeclaration
    |   fieldDeclaration
    |   constructorDeclaration
    |   genericConstructorDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;

/* We use rule this even for void methods which cannot have [] after parameters.
   This simplifies grammar and we can consider void to be a type, which
   renders the [] matching as a context-sensitive issue or a semantic check
   for invalid return type after parsing.
 */
methodDeclaration
    :   (type |'void') Identifier {if(blockCmnt==1) {cmnt_var = 1;  blockCmnt = 0;}}
```

```
        formalParameters                                                          {if(cmnt_var==1)
{methodCollectorCmnt.put($Identifier.text+$formalParameters.text,theCmnt); cmnt_var = 0;}
                                                    else
if(todo==1){methodCollectorTODO.put($Identifier.text+$formalParameters.text,theCmnt); todo=0;}
                                                    else
methodCollectorNoCmnt.put($Identifier.text+$formalParameters.text,"");}
                                                        ('[' ']')*
        ('throws'  qualifiedNameList )?
        (   methodBody
        |   ';'
        )
    ;

genericMethodDeclaration
    :   typeParameters methodDeclaration
    ;

constructorDeclaration
    :   comments Identifier {if(blockCmnt==1) { cmnt_var = 1;  blockCmnt = 0;}}
        formalParameters                                                          {if(cmnt_var==1)
{classConstructorCmnt.put($Identifier.text+$formalParameters.text,theCmnt);    cmnt_var    =    0;}}
('throws' qualifiedNameList)?
        constructorBody
    ;

genericConstructorDeclaration
    :   typeParameters constructorDeclaration
    ;

fieldDeclaration
    :   type variableDeclarators ';'
    ;

interfaceBodyDeclaration
    :    comments modifier* interfaceMemberDeclaration
    |   ';'
    ;

interfaceMemberDeclaration
    :   constDeclaration
    |   interfaceMethodDeclaration
    |   genericInterfaceMethodDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;

constDeclaration
    :   type constantDeclarator (',' constantDeclarator)* ';'
    ;

constantDeclarator
    :   Identifier ('[' ']')* '=' variableInitializer
    ;

// see matching of [] comment in methodDeclaratorRest
interfaceMethodDeclaration
    :   (type|'void') Identifier  {if(blockCmnt==1) {cmnt_var = 1;  blockCmnt = 0;}}
        formalParameters                                                          {if(cmnt_var==1)
{methodCollectorCmnt.put($Identifier.text+$formalParameters.text,theCmnt); cmnt_var = 0;}
                                                                                   else
if(todo==1){methodCollectorTODO.put($Identifier.text+$formalParameters.text,theCmnt); todo=0;}
                                                                                   else
methodCollectorNoCmnt.put($Identifier.text+$formalParameters.text,"");}
        ('[' ']')*
        ('throws' qualifiedNameList)?
        ';'
    ;
```

```
genericInterfaceMethodDeclaration
    :   typeParameters interfaceMethodDeclaration
    ;

variableDeclarators
    :   variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    :   variableDeclaratorId ('=' variableInitializer)?
    ;

variableDeclaratorId
    :    Identifier {if(blockCmnt==1){fieldCollectorCmnt.put($Identifier.text,theCmnt); blockCmnt =
0;}} ('[' ']')*
    ;

variableInitializer
    :   arrayInitializer
    |   expression
    ;

arrayInitializer
    :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '}'
    ;

enumConstantName
    :   Identifier
    ;

type
    :   classOrInterfaceType ('[' ']')*
    |   primitiveType ('[' ']')*
    ;

classOrInterfaceType
    :   Identifier (typeArguments)? ('.' Identifier (typeArguments)? )*
    ;

primitiveType
    :   'boolean'
    |   'char'
    |   'byte'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;

typeArguments
    :   '<' typeArgument (',' typeArgument)* '>'
    ;

typeArgument
    :   type
    |   '?' (('extends' | 'super') type)?
    ;

qualifiedNameList
    :   qualifiedName (',' qualifiedName)*
    ;

formalParameters
    :   '(' (formalParameterList)? ')'
    ;

formalParameterList
```

```
    :   formalParameter (',' formalParameter)* (',' lastFormalParameter)?
    |   lastFormalParameter
    ;

formalParameter
    :   variableModifier* type variableDeclaratorId
    ;

lastFormalParameter
    :   variableModifier* type '...' variableDeclaratorId
    ;

methodBody
    :   block
    ;

constructorBody
    :   block
    ;

qualifiedName
    :   Identifier ('.' Identifier)*
    ;

literal
    :   IntegerLiteral
    |   FloatingPointLiteral
    |   CharacterLiteral
    |   StringLiteral
    |   BooleanLiteral
    |   'null'
    ;

// ANNOTATIONS

annotation
    :   '@' annotationName ( '(' ( elementValuePairs | elementValue )? ')' )?
    ;

annotationName : qualifiedName ;

elementValuePairs
    :   elementValuePair (','elementValuePair)*
    ;

elementValuePair
    :   Identifier '='elementValue
    ;

elementValue
    :   expression
    |   annotation
    |   elementValueArrayInitializer
    ;

elementValueArrayInitializer
    :   '{' (elementValue (',' elementValue)*)? (',')? '}'
    ;

annotationTypeDeclaration
    :   '@' 'interface' Identifier annotationTypeBody
    ;

annotationTypeBody
    :   '{' (annotationTypeElementDeclaration)* '}'
    ;

annotationTypeElementDeclaration
    :   modifier* annotationTypeElementRest
```

```
    |   ';' // this is not allowed by the grammar, but apparently allowed by the actual compiler
    ;

annotationTypeElementRest
    :   type annotationMethodOrConstantRest ';'
    |   classDeclaration ';'?
    |   interfaceDeclaration ';'?
    |   enumDeclaration ';'?
    |   annotationTypeDeclaration ';'?
    ;

annotationMethodOrConstantRest
    :   annotationMethodRest
    |   annotationConstantRest
    ;

annotationMethodRest
    :   Identifier '(' ')' defaultValue?
    ;

annotationConstantRest
    :   variableDeclarators
    ;

defaultValue
    :   'default' elementValue
    ;

// STATEMENTS / BLOCKS

block
    :   '{' (blockStatement)* '}'
    ;

blockStatement
    :   comments localVariableDeclarationStatement
    |   statement
    |   typeDeclaration
    ;

localVariableDeclarationStatement
    :    localVariableDeclaration ';'
    ;

localVariableDeclaration
    :   (variableModifier)* type variableDeclarators
    ;

statement
    :   block
    |    comments {myList = new ArrayList<String>();} ASSERT expression (':' expression)? ';' {
if(blockCmnt==1) {methodCollectorCmnt.put(myList.get(0),theCmnt); blockCmnt = 0;}}
    |   commentToPassBy 'if' parExpression statement ('else' statement)?
    |   'for' '(' forControl ')' statement
    |   commentToPassBy 'while' parExpression statement
    |   'do' statement 'while' parExpression ';'
    |   commentToPassBy 'try' block (catchClause+ finallyBlock? | finallyBlock)
    |   'try' resourceSpecification block catchClause* finallyBlock?
    |   'switch' parExpression '{' switchBlockStatementGroup* switchLabel* '}'
    |   commentToPassBy 'synchronized' parExpression block
    |   commentToPassBy 'return' expression? ';'
    |   commentToPassBy 'throw' expression ';'
    |   'break' Identifier? ';'
    |   commentToPassBy 'continue' Identifier? ';'
    |   ';'
    |    comments {myList = new ArrayList<String>();} statementExpression ';' { if(blockCmnt==1)
{methodCollectorCmnt.put(myList.get(0),theCmnt); blockCmnt = 0;}}
    |   Identifier ':' statement
    ;
```

```
catchClause
    :   'catch' '(' variableModifier* catchType Identifier ')' block
    ;

catchType
    :   qualifiedName ('|' qualifiedName)*
    ;

finallyBlock
    :   'finally' block
    ;

resourceSpecification
    :   '(' resources ';'? ')'
    ;

resources
    :   resource (';' resource)*
    ;

resource
    :   variableModifier* classOrInterfaceType variableDeclaratorId '=' expression
    ;

/** Matches cases then statements, both of which are mandatory.
 *  To handle empty cases at the end, we add switchLabel* to statement.
 */
switchBlockStatementGroup
    :   switchLabel+ blockStatement+
    ;

switchLabel
    :   'case' constantExpression ':'
    |   'case' enumConstantName ':'
    |   'default' ':'
    ;

forControl
    :   enhancedForControl
    |   forInit? ';' expression? ';' forUpdate?
    ;

forInit
    :   localVariableDeclaration
    |   expressionList
    ;

enhancedForControl
    :   variableModifier* type variableDeclaratorId ':' expression
    ;

forUpdate
    :   expressionList
    ;

// EXPRESSIONS

parExpression
    :   '(' expression ')'
    ;

expressionList
    :   expression (',' expression)*
    ;

statementExpression
    :   expression
    ;
```

```
constantExpression
    :   expression
    ;

expression
    :   primary
    |   expression '.' Identifier
    |   expression '.' 'this'
    |   expression '.' 'new' nonWildcardTypeArguments? innerCreator
    |   expression '.' 'super' superSuffix
    |   expression '.' explicitGenericInvocation
    |   expression '[' expression ']'
    |   expression '(' expressionList? ')'
    |   'new' creator
    |   '(' type ')' expression
    |   expression ('++' | '--')
    |   ('+'|'-'|'++'|'--') expression
    |   ('~'|'!') expression
    |   expression ('*'|'/'|'%') expression
    |   expression ('+'|'-') expression
    |   expression ('<' '<' | '>' '>' '>' | '>' '>') expression
    |   expression ('<=' | '>=' | '>' | '<') expression
    |   expression 'instanceof' type
    |   expression ('==' | '!=') expression
    |   expression '&' expression
    |   expression '^' expression
    |   expression '|' expression
    |   expression '&&' expression
    |   expression '||' expression
    |   expression '?' expression ':' expression
    |   <assoc=right> expression
        (   '='
        |   '+='
        |   '-='
        |   '*='
        |   '/='
        |   '&='
        |   '|='
        |   '^='
        |   '>>='
        |   '>>>='
        |   '<<='
        |   '%='
        )
        expression
    ;

primary
    :   '(' expression ')'
    |   'this'
    |   'super'
    |   literal
    |   Identifier {myList.add($Identifier.text);}
    |   type '.' 'class'
    |   'void' '.' 'class'
    |   nonWildcardTypeArguments (explicitGenericInvocationSuffix | 'this' arguments)
    ;

creator
    :   nonWildcardTypeArguments createdName classCreatorRest
    |   createdName (arrayCreatorRest | classCreatorRest)
    ;

createdName
    :   Identifier typeArgumentsOrDiamond? ('.' Identifier typeArgumentsOrDiamond?)*
    |   primitiveType
    ;
```

```
innerCreator
    :   Identifier nonWildcardTypeArgumentsOrDiamond? classCreatorRest
    ;

arrayCreatorRest
    :   '['
        (   ']' ('[' ']')* arrayInitializer
        |   expression ']' ('[' expression ']')* ('[' ']')*
        )
    ;

classCreatorRest
    :   arguments classBody?
    ;

explicitGenericInvocation
    :   nonWildcardTypeArguments explicitGenericInvocationSuffix
    ;

nonWildcardTypeArguments
    :   '<' typeList '>'
    ;

typeArgumentsOrDiamond
    :   '<' '>'
    |   typeArguments
    ;

nonWildcardTypeArgumentsOrDiamond
    :   '<' '>'
    |   nonWildcardTypeArguments
    ;

superSuffix
    :   arguments
    |   '.' Identifier arguments?
    ;

explicitGenericInvocationSuffix
    :   'super' superSuffix
    |   Identifier arguments
    ;

arguments
    :   '(' expressionList? ')'
    ;

// LEXER

// §3.9 Keywords

ABSTRACT      : 'abstract';
ASSERT        : 'assert';
BOOLEAN       : 'boolean';
BREAK         : 'break';
BYTE          : 'byte';
CASE          : 'case';
CATCH         : 'catch';
CHAR          : 'char';
CLASS         : 'class';
CONST         : 'const';
CONTINUE      : 'continue';
DEFAULT       : 'default';
DO            : 'do';
DOUBLE        : 'double';
ELSE          : 'else';
ENUM          : 'enum';
EXTENDS       : 'extends';
FINAL         : 'final';
```

```
FINALLY       : 'finally';
FLOAT         : 'float';
FOR           : 'for';
IF            : 'if';
GOTO          : 'goto';
IMPLEMENTS    : 'implements';
IMPORT        : 'import';
INSTANCEOF    : 'instanceof';
INT           : 'int';
INTERFACE     : 'interface';
LONG          : 'long';
NATIVE        : 'native';
NEW           : 'new';
PACKAGE       : 'package';
PRIVATE       : 'private';
PROTECTED     : 'protected';
PUBLIC        : 'public';
RETURN        : 'return';
SHORT         : 'short';
STATIC        : 'static';
STRICTFP      : 'strictfp';
SUPER         : 'super';
SWITCH        : 'switch';
SYNCHRONIZED  : 'synchronized';
THIS          : 'this';
THROW         : 'throw';
THROWS        : 'throws';
TRANSIENT     : 'transient';
TRY           : 'try';
VOID          : 'void';
VOLATILE      : 'volatile';
WHILE         : 'while';

// §3.10.1 Integer Literals

IntegerLiteral
    :   DecimalIntegerLiteral
    |   HexIntegerLiteral
    |   OctalIntegerLiteral
    |   BinaryIntegerLiteral
    ;

fragment
DecimalIntegerLiteral
    :   DecimalNumeral IntegerTypeSuffix?
    ;

fragment
HexIntegerLiteral
    :   HexNumeral IntegerTypeSuffix?
    ;

fragment
OctalIntegerLiteral
    :   OctalNumeral IntegerTypeSuffix?
    ;

fragment
BinaryIntegerLiteral
    :   BinaryNumeral IntegerTypeSuffix?
    ;

fragment
IntegerTypeSuffix
    :   [lL]
    ;

fragment
DecimalNumeral
```

```
    :   '0'
    |   NonZeroDigit (Digits? | Underscores Digits)
    ;

fragment
Digits
    :   Digit (DigitOrUnderscore* Digit)?
    ;

fragment
Digit
    :   '0'
    |   NonZeroDigit
    ;

fragment
NonZeroDigit
    :   [1-9]
    ;

fragment
DigitOrUnderscore
    :   Digit
    |   '_'
    ;

fragment
Underscores
    :   '_'+
    ;

fragment
HexNumeral
    :   '0' [xX] HexDigits
    ;

fragment
HexDigits
    :   HexDigit (HexDigitOrUnderscore* HexDigit)?
    ;

fragment
HexDigit
    :   [0-9a-fA-F]
    ;

fragment
HexDigitOrUnderscore
    :   HexDigit
    |   '_'
    ;

fragment
OctalNumeral
    :   '0' Underscores? OctalDigits
    ;

fragment
OctalDigits
    :   OctalDigit (OctalDigitOrUnderscore* OctalDigit)?
    ;

fragment
OctalDigit
    :   [0-7]
    ;

fragment
OctalDigitOrUnderscore
```

```
    :   OctalDigit
    |   '_'
    ;

fragment
BinaryNumeral
    :   '0' [bB] BinaryDigits
    ;

fragment
BinaryDigits
    :   BinaryDigit (BinaryDigitOrUnderscore* BinaryDigit)?
    ;

fragment
BinaryDigit
    :   [01]
    ;

fragment
BinaryDigitOrUnderscore
    :   BinaryDigit
    |   '_'
    ;

// §3.10.2 Floating-Point Literals

FloatingPointLiteral
    :   DecimalFloatingPointLiteral
    |   HexadecimalFloatingPointLiteral
    ;

fragment
DecimalFloatingPointLiteral
    :   Digits '.' Digits? ExponentPart? FloatTypeSuffix?
    |   '.' Digits ExponentPart? FloatTypeSuffix?
    |   Digits ExponentPart FloatTypeSuffix?
    |   Digits FloatTypeSuffix
    ;

fragment
ExponentPart
    :   ExponentIndicator SignedInteger
    ;

fragment
ExponentIndicator
    :   [eE]
    ;

fragment
SignedInteger
    :   Sign? Digits
    ;

fragment
Sign
    :   [+-]
    ;

fragment
FloatTypeSuffix
    :   [fFdD]
    ;

fragment
HexadecimalFloatingPointLiteral
    :   HexSignificand BinaryExponent FloatTypeSuffix?
    ;
```

```
fragment
HexSignificand
    :   HexNumeral '.'?
    |   '0' [xX] HexDigits? '.' HexDigits
    ;

fragment
BinaryExponent
    :   BinaryExponentIndicator SignedInteger
    ;

fragment
BinaryExponentIndicator
    :   [pP]
    ;

// §3.10.3 Boolean Literals

BooleanLiteral
    :   'true'
    |   'false'
    ;

// §3.10.4 Character Literals

CharacterLiteral
    :   '\'' SingleCharacter '\''
    |   '\'' EscapeSequence '\''
    ;

fragment
SingleCharacter
    :   ~['\\]
    ;
// §3.10.5 String Literals
StringLiteral
    :   '"' StringCharacters? '"'
    ;
fragment
StringCharacters
    :   StringCharacter+
    ;
fragment
StringCharacter
    :   ~["\\]
    |   EscapeSequence
    ;
// §3.10.6 Escape Sequences for Character and String Literals
fragment
EscapeSequence
    :   '\\' [btnfr"'\\]
    |   OctalEscape
    |   UnicodeEscape
    ;

fragment
OctalEscape
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
    |   '\\' ZeroToThree OctalDigit OctalDigit
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

fragment
```

```
ZeroToThree
    :   [0-3]
    ;

// §3.10.7 The Null Literal

NullLiteral
    :   'null'
    ;

// §3.11 Separators

LPAREN          : '(';
RPAREN          : ')';
LBRACE          : '{';
RBRACE          : '}';
LBRACK          : '[';
RBRACK          : ']';
SEMI            : ';';
COMMA           : ',';
DOT             : '.';

// §3.12 Operators

ASSIGN          : '=';
GT              : '>';
LT              : '<';
BANG            : '!';
TILDE           : '~';
QUESTION        : '?';
COLON           : ':';
EQUAL           : '==';
LE              : '<=';
GE              : '>=';
NOTEQUAL        : '!=';
AND             : '&&';
OR              : '||';
INC             : '++';
DEC             : '--';
ADD             : '+';
SUB             : '-';
MUL             : '*';
DIV             : '/';
BITAND          : '&';
BITOR           : '|';
CARET           : '^';
MOD             : '%';

ADD_ASSIGN      : '+=';
SUB_ASSIGN      : '-=';
MUL_ASSIGN      : '*=';
DIV_ASSIGN      : '/=';
AND_ASSIGN      : '&=';
OR_ASSIGN       : '|=';
XOR_ASSIGN      : '^=';
MOD_ASSIGN      : '%=';
LSHIFT_ASSIGN   : '<<=';
RSHIFT_ASSIGN   : '>>=';
URSHIFT_ASSIGN  : '>>>=';

// §3.8 Identifiers (must appear after all keywords in the grammar)

Identifier
    :   JavaLetter JavaLetterOrDigit*
    ;

fragment
JavaLetter
    :   [a-zA-Z$_] // these are the "java letters" below 0xFF
```

```
    |   // covers all characters above 0xFF which are not a surrogate
        ~[\u0000-\u00FF\uD800-\uDBFF]
        {Character.isJavaIdentifierStart(_input.LA(-1))}?
    |   // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
        [\uD800-\uDBFF] [\uDC00-\uDFFF]
        {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2),
(char)_input.LA(-1)))}?
    ;

fragment
JavaLetterOrDigit
    :   [a-zA-Z0-9$_] // these are the "java letters or digits" below 0xFF
    |   // covers all characters above 0xFF which are not a surrogate
        ~[\u0000-\u00FF\uD800-\uDBFF]
        {Character.isJavaIdentifierPart(_input.LA(-1))}?
    |   // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
        [\uD800-\uDBFF] [\uDC00-\uDFFF]
        {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2), (char)_input.LA(-
1)))}?
    ;

//
// Additional symbols not defined in the lexical specification
//

AT : '@';
ELLIPSIS : '...';

//
// Whitespace and comments
//

WS  :  [ \t\r\n\u000C]+ -> skip
    ;

COMMENT /// handles nested comments
    :   '/*' (COMMENT| . )*? '*/' //-> skip
    ;

TODO_COMMENT
    :   '// TODO' ~[\r\n]*
    ;

LINE_COMMENT
    :   '//' ~[\r\n]* -> skip
    ;
```