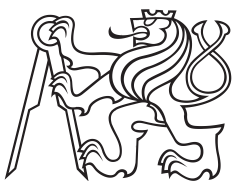


Master's Thesis



**Czech
Technical
University
in Prague**

F3

Faculty of Electrical Engineering
Department of Computer Science

System for analysis of Java language

Bc. Markéta Badalíková

May 2016

Supervisor: Ing. Martin Filipický

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Markéta Badalíková**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Systém pro analýzu jazyku Java**

Pokyny pro vypracování:

Proveďte rešerši současných nástrojů a knihoven pro lexikální a syntaktickou analýzu jazyku Java. Po dohodě s vedoucím práce vyberte některý z existujících nástrojů a rozšířte jej nebo navrhnete a implementujte nástroj nový.

Na výsledný systém jsou kladeny následující požadavky: provést základní lexikální analýzu zdrojového kódu, správně tokenizovat kód a syntakticky jej analyzovat. Syntaktická analýza obsahuje požadavek na tvorbu abstraktního syntaktického stromu (ASS), který vizualizujete. Nástroj bude přijímat zdrojový kód ve formátu JUnit frameworku. Výstupem ze syntaktické analýzy bude krom ASS také zvláštní seznam testů (tj. metod v dané třídě) a seznam kroků v daném testu (tj. příkazů v dané metodě).

Pro systém navrhnete a napíšete testy vytvořené pomocí Selenium WebDriver a JUnit.

Seznam odborné literatury:

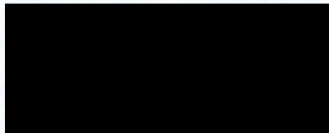
Herbert Schildt: Java: The Complete Reference (Complete Reference Series), Mcgraw-Hill Osborne Media; 9 edition

Ronald Mak: Writing Compilers and Interpreters: A Software Engineering Approach, Wiley; 3 edition

Torben & AElig;gidius Mogensen: Introduction to Compiler Design, Springer; 2011 edition

Vedoucí: Ing. Martin Filipický

Platnost zadání: do konce zimního semestru 2016/2017



doc. Ing. Filip Železný, Ph.D.
vedoucí katedry



prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 22. 10. 2015

Acknowledgement / Declaration

Nejprve bych ráda poděkovala svému vedoucímu Ing. Martinovi Filipskému za jeho neocenitelné rady, podporu a nezměrnou trpělivost. Mé díky patří také mé rodině a přátelům, kteří mi při psaní práce drželi palce a byli tu pro mě, i když jsem na ně neměla tolik času, kolik by si zasloužili. V neposlední řadě musím poděkovat také své kočce, která na mě při psaní této práce dohlížela svým bystrým okem, pokud tedy zrovna nepodřimovala.

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. května 2016

Abstrakt / Abstract

Cílem této práce je vytvořit systém pro lexikální a syntaktickou analýzu zdrojového kódu v jazyce Java a práci s jeho abstraktním syntaktickým stromem (AST). Práce obsahuje přehled existujících projektů podobného zaměření, z nichž byl pro tvorbu AST vybrán nástroj JavaParser, na nějž práce navazuje. Výsledný systém je napsaný v jazyce Java, poskytuje grafické uživatelské rozhraní vytvořené v JavaFX a umožňuje vizualizaci AST, jeho úpravy, filtraci podle typu uzlů a vyhledávání podle vzoru.

The goal of the thesis is to implement a system for lexical and syntax analysis of source code in Java language and manipulation with its abstract syntax tree (AST). The thesis contains a summary of existing projects with similar functions, from which JavaParser tool was chosen to provide generation of AST and to be used as a foundation of the application. The resulting system is written in Java, offers JavaFX graphic user interface and enables visualization of AST, modifications of it, filtration by node type and searching by patterns.

Contents /

1 Introduction	1	6.1.3 Tree traversal	29
2 Problem statement	2	6.1.4 Node removal	30
2.1 Assignment	2	6.1.5 Summary of issues	31
2.2 Goals	2	6.2 Data structure dictionary	32
2.2.1 Test automation	2	6.3 Main data structures and	
2.3 Content of the thesis	3	function providers	32
3 Parsing theory	4	6.3.1 TreeNodes	32
3.1 Purpose of parsing	4	6.3.2 Roles	33
3.2 Compilers and interpreters	4	6.3.3 Visitors	34
3.2.1 Phases of a compiler	4	6.3.4 Printers	35
3.3 Lexical analysis	6	6.3.5 Modes and settings	37
3.4 Syntax analysis	6	6.4 Core application	37
4 Existing solutions	11	6.4.1 Analyzer	37
4.1 Selection criteria	11	6.4.2 Settings management	37
4.2 Selected projects	12	6.4.3 Parsing	38
4.2.1 Compiler Tree API	12	6.4.4 Tree building	38
4.2.2 Eclipse JDT AST	12	6.4.5 History	39
4.2.3 ANTLR	13	6.4.6 Node deleting	39
4.2.4 Spoon	14	6.4.7 Searching	40
4.2.5 JavaParser	14	6.4.8 Pattern management	40
4.3 ANTLR and JavaParser		6.4.9 Searching by patterns	40
comparison	14	6.5 GUI elements	41
4.4 Summary	17	6.5.1 Stages	42
5 Design	18	6.5.2 Tree views and Mes-	
5.1 Requirements	18	sage pane	42
5.1.1 Functional require-		6.5.3 Code areas	42
ments	18	6.5.4 Graphs	43
5.1.2 Non-functional re-		6.5.5 Managers	43
quirements	19	7 Testing	44
5.2 Use cases	20	7.1 Test configuration	44
5.2.1 AST viewing	20	7.2 Test strategy	44
5.2.2 Node operations	21	7.2.1 Static code analysis	44
5.2.3 Tree editing	21	7.2.2 Unit, integration and	
5.2.4 File operations	22	system testing	45
5.2.5 Searching	22	7.2.3 Manual testing	46
5.3 Structure of the application	23	7.2.4 Negative testing	48
5.3.1 Layers and used li-		7.3 Test results	49
braries	23	8 Conclusion	50
5.3.2 Core application	25	8.1 Future work	51
5.3.3 Graphic user interface	26	References	52
5.3.4 Connection between		A CD content	53
components	27	B User manual	54
6 Implementation	28	C Productions used in syntax	
6.1 JavaParser features	28	analysis example	56
6.1.1 Types of nodes	28	D List of JavaParser node types	58
6.1.2 Visitors	29	E Class diagrams by packages	60

Tables / Figures

4.1. Available parsers overview	11
5.1. Results of pattern search example	19
6.1. Printer tree traversal steps	36
7.1. Parameters of negative testing of file inputs	48
3.1. Phases of a compiler	5
3.2. Natural language parse tree	7
3.3. Java code parse tree	9
3.4. Expression parse tree	10
4.1. Example of ANTLR visualization	13
4.2. AST parsed by ANTLR	15
4.3. AST parsed by JavaParser	16
5.1. Example of pattern search function	19
5.2. Main use cases diagram	20
5.3. Layers of the application and the employed libraries	24
5.4. Data structures of the application	26
5.5. GUI elements	26
5.6. Connections between core and GUI	27
6.1. JavaParser ast package	28
6.2. JavaParser visitors	29
6.3. Indistinguishable nodes example	31
6.4. TreeNode hierarchy	33
6.5. Printer example	35
6.6. Application layout	41
B.1. Test window	54
B.2. Pattern menu	55
E.3. Core package	60
E.4. File package	61
E.5. Pattern package	61
E.6. Treebuilder package	61
E.7. Visitor package	62
E.8. Gui package	62
E.9. Codearea package	63
E.10. Graph package	63
E.11. Stage package	63
E.12. Treeview package	63
E.13. Node package	64
E.14. Printer package	64
E.15. Settings package	64

Chapter 1

Introduction

Lexical and *syntax analysis* are processes essential to comprehension of any given language, programming or natural. Both the code used to give commands to a computer and the human speech exchanged between people have to follow certain rules, making it possible to transform ideas into words in a way the other party will understand, unless they want their communication to end up in misunderstanding and confusion.

In the context of natural languages, lexical analysis refers to identification of the parts of speech, as in which words are nouns, verbs, prepositions and so on. Similarly, programming languages need to determine keywords, variable names or literals. Syntax analysis then checks whether the words are arranged correctly in accordance with the *grammar rules*, ensuring there will be a verb in a sentence, or that a variable declaration will certainly not end up without the variable name. This process is often referred to as *parsing*.

Grammar rules are applied by finding plausible relationships between words in the text. These relationships are arranged in a hierarchical structure called *parse tree* or *abstract syntax tree* (AST). As a curiosity, it may be worth mentioning that Czech children learn to perform syntax analysis of sentences in their native language by drawing a parse tree in elementary school.

The goal of the thesis is to implement a tool performing lexical and syntax analysis of code in Java programming language, providing a visualization of the AST and means to modify it. To reach this objective, we will explore the parsing process in more depth firstly, following with a summary of existing projects offering similar functions. Eventually, we will choose one of them as basis to be extended by the tool. Next part will cover intended use cases and design of the structure of the application. After that will follow implementation description and finally the result will be subjected to testing.

Chapter 2

Problem statement

Before we delve into the world of parsing and parse trees, we should revise tasks defined by the thesis assignment, discuss its goals and summarize which topics it should cover.

2.1 Assignment

As stated in the assignment, the goal of the thesis is to implement an application for lexical and syntax analysis of Java source code.

The thesis should explore existing tools and libraries conducting lexical and syntax analysis of Java source code. If there are any found, we will evaluate them to determine whether any of them can be used and extended by the intended tool. In the case of no project being convenient enough, the tool will have to cover this functionality itself.

The tool should satisfy the following requirements: enable visualization of the AST, accept source code from the JUnit framework and, as the output, aside from the AST, provide a separate list of test methods in the given class and their statements. The tool should be tested by Selenium WebDriver and JUnit tests.

2.2 Goals

The purpose of the resulting tool is to help testers and software quality engineers during the process of *functional test automation* with conducting:

- linear test script analysis
- faster test modification based on patterns
- filtering of parameters affecting possibilities of searching for common functions

We will now explain the concept of test automation briefly.

2.2.1 Test automation

Automated testing is a strategy with the goal of replacing manual testing, lowering the costs of test execution. In contrast to manual tests, automated tests can be executed as many times and on many platforms as possible, without the need to employ a human tester. Furthermore, lacking the human factor, they can run for instance at night, making use of non-productive time slots.

However, these advantages come at a price: as expected, automation is not as flexible as a human being and tends to shatter with any change of the system under test. This adds costly maintenance to the already high preparation effort.

Especially challenging task in this domain is simulation of communication between the user and front-end of the tested system. It is usually solved by recording actions

performed by the test designer and then extending them by code, which, for example, asserts the expected results. This adds stability to the otherwise quickly obtained but not too steady product of recording. However, these improvements bring further high effort. In conclusion, we can say that test automation is about constantly seeking balance between those two approaches [1].

2.3 Content of the thesis

Firstly, in chapter 3, the task will be to gain a deeper understanding of lexical and syntax analysis, to become familiar with the terminology concerning the topic and take a closer look at how these are conducted by Java language.

Next step, described in chapter 4, would be to formulate criteria of selection of the existing tools and then conduct the research with the goal of selecting one of them. We will evaluate results of the selection according to the criteria, and eventually, we will choose the best candidate.

After that, chapter 5 will outline suggested design of the application. We will analyse the requirements to produce a list of use cases and a proposal of structure of the application components. This will be followed by a description of implementation details in chapter 6. The final part, chapter 7, will describe testing of the tool.

As we have mentioned, we will now continue to the next chapter explaining the theory regarding the lexical and syntax analysis and demonstrating the procedures on examples.

Chapter 3

Parsing theory

This chapter will discuss theoretical background regarding the lexical and syntax analysis, otherwise known as *parsing*. Initially, we will explain reasons and usage of parsing in compilers and interpreters, and then we will describe the individual steps of the processes, demonstrating their course on examples in natural language and Java.

3.1 Purpose of parsing

Parsing is a process determining whether a sequence of symbols abides to the rules of the corresponding language, and building a *parse* or *syntax tree* describing hierarchical structure of relationships between the symbols in the input text. In other words, parsing checks whether words arranged in a certain way make sense in the context of the given language.

Generally, we can consider two types of parsing: of natural languages and of programming languages. In both cases, the goal is to ensure that the source text is eventually understood either by other human being or a computer that executes the code.

Majority of programming languages used today produce so-called *high-level languages*. These are designed more to be understood by people than by computers and it is necessary to translate them into *low-level machine language* before the computer can use them. A bridge connecting these languages is a representation understood by a computer, usually a parse tree. Programs used to execute this translation are called *compilers* and *interpreters* [2].

3.2 Compilers and interpreters

Compilers and interpreters are *language processors* used to run computer code. Both have parsing as an essential part of their routine.

While a compiler uses this representation to generate low-level machine code, interpreter executes statements from the tree directly without generating any additional code. Due to this, interpreters are easier to implement and are a good choice for debugging, as they are able to run the program more quickly and with less effort, whereas their execution speed is slower than that of a machine language program [3].

3.2.1 Phases of a compiler

Figure 3.1 illustrates stages of a compiler and output of each stage. The first three phases are called the *front-end*, last three the *back-end*, with *intermediate code generation* in the middle. The front-end part is the same for the interpreter, stopping at the middle part and executing the code [2].

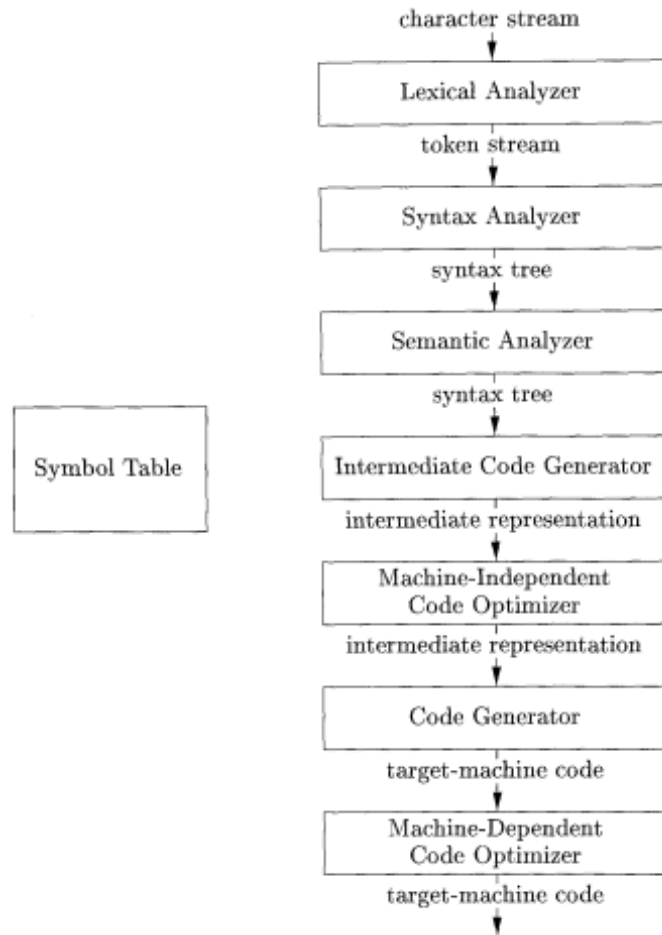


Figure 3.1. Phases of a compiler. Courtesy of [4].

We will now briefly describe the front-end stages, as they are related to our area of interest.

Lexical analyzer. Otherwise called *lexer*, *scanner* or *tokenizer*, the lexical analyzer reads text of the program letter by letter and converts the sequence of characters into *tokens*, symbols corresponding to the symbols of the language, filtering out white space and comments. Lexical analysis is often performed as a part of syntax analysis, however, it is actually better to keep them separate, as not to mix two different aspects of the language. Specifications of lexical grammar are usually formed by *regular expressions* [2].

Syntax analyzer. Alternatively called *parser*, this program performs the parsing process. Its purpose is to arrange the tokens to form a syntax tree, according to the *grammar* of the language in which the code is written.

Semantic analyzer. *Semantic analysis* may be called *contextual analysis* too. It looks for undeclared variable identifiers and performs *type checking*, for example examining whether the variables are used accordingly to their type. To store information about variables, a *symbol table* is employed [5]. We will not focus on semantic analysis any further.

We will now talk about lexical and syntactic analysis in the following sections.

3.3 Lexical analysis

This section will demonstrate lexical analysis using a natural language example, which will be compared against an example in Java. The natural language example was inspired by examples from [6].

As we already mentioned in the previous sections, lexical analysis reads the source text and transforms it into a sequence of tokens. In the context of natural languages, it is necessary to determine the *parts of speech*, which serve as tokens in this case. It might be worth mentioning that the name of the process refers to the fact the words are compared against the vocabulary, in other words, *lexicon* of the language, knowledge of which helps us to recognize the parts of speech.

Firstly, we will try to determine tokens in this simple sentence:

```
The cat sat on a mat.
```

- nouns - cat, mat
- verbs - sat
- prepositions - on
- articles - the, a

Now, we will apply the same process to analysis of a short Java code. Before we start, we should list token types used by the language, used instead of parts of speech, and examples of them:

- *identifiers* - names of variables, classes
- *keywords* - if, public, instanceof
- *literals* - numbers, strings
- *separators* - commas, brackets
- *operators* - less than, ampersand

Using this knowledge, we can easily determine tokens in this code snippet:

```
public class Example {  
    public void example(int a) {}  
}
```

- identifiers - Example, example, a
- keywords - public, class, void, int
- separators - {}, ()

The identified tokens will now serve as an input of the next stage, the syntax analysis.

3.4 Syntax analysis

In this step, the task is to arrange the tokens obtained by lexical analysis into a parse tree according to a grammar, illustrating the hierarchy of relationships between them. Only tokens arranged in a syntactically correct way should make sense in the context of the language.

The grammar consists of *production rules* in the form

$$N \rightarrow X_1 \dots X_n$$

where N is an abstract symbol called a *nonterminal* and $X_1 \dots X_n$ is a set of nonterminals and *terminals*. Terminals are the tokens. This way, the rules specify possible sequences of terminal symbols resulting from recursively replacing nonterminals by other symbols [2].

The process of rewriting nonterminals is called *derivation*. Beginning from the *start (goal) symbol* used as a root, by rewriting every nonterminal with corresponding right-hand sides of the productions and adding them as children of the nonterminal node until all of the leaves contain terminals, a syntax tree is built. Reading the leaves from right to left should result in obtaining the original string.

The rules must work in a way to make it possible for any valid text in the language to be representable by a syntax tree.

We will now demonstrate the process using examples from the previous section. We will begin with analysis of the syntax tree of our sentence in figure 3.2.

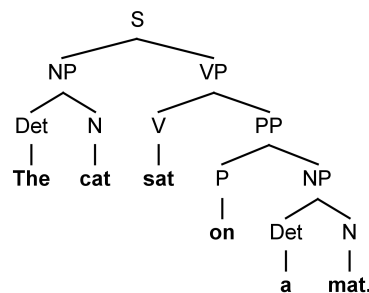


Figure 3.2. Natural language parse tree.

The tokens are grouped into *phrases*: *noun phrase* (NP) contains tokens specifying properties of the noun, *verb phrase* (VP) does the same for verbs and *prepositional phrase* (PP) connects other phrases with a preposition.

The main parts of a *sentence* (S) are *subject* and *predicate*, usually represented by nouns and verbs respectively. In this case, the pair is *cat* - *sat* and all remaining words are dependent on them. The sentence is divided into two parts at the root, where a noun phrase is related to the subject and verb phrase to the predicate.

By defining the structure, the rules of the language further state that some tokens should not be paired together, e.g. articles have to appear only before the nouns and nowhere else.

Following list contains productions used in figure 3.2.

```

S -> NP VP
PP -> P NP
NP -> Det N
VP -> VP PP
Det -> 'the' | 'a'
N -> 'cat' | 'mat'
V -> 'sat'
P -> 'on'

```

A production is formed by a type of node to the left of the arrow and types of children the node is allowed to have to the right of the arrow. The first node in the list should be the start symbol, which always has to be the root of the tree; in this case it is S, as in sentence. We can see that the words themselves are nonterminals, as there are no productions further deriving them.

When observing the productions, we can see how the parse tree was built according to them. As the start symbol is S, it will be the root, which should be followed by the derivation producing NP and VP and so on, until all of the tokens, or the words of the sentence, appear in the leaves, connected correctly with the rest of the tree. This approach reflects a parse tree building method called *predictive* or *top-down parsing*. Opposite approach, *bottom-up parsing*, does it the other way round, by looking for right-hand sides of productions and rewriting them using inverse derivation until they reach the start symbol [2].

Basically, the rules should be able to describe every possible parse tree corresponding to a grammatically correct sentence in the language. Of course, in reality the possibilities are much broader than in this example. Parsing of programming languages works very much the same, however, compared to the often highly ambiguous human speech, is incomparably more definite and easier to decipher for computers.

Now we can look at the Java example in figure 3.3.

As in the previous example, the tree follows rules given by the grammar. The root symbol, `CompilationUnit`, representing Java code placed in a file, is defined in the official grammar specification [7] like this:

```
CompilationUnit:
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}
```

Square brackets denote zero or one occurrence, curly brackets zero or more. This means a compilation unit may or may not contain one package declaration and an arbitrary number of import declarations and type declarations. To be sure what type declaration means, the specification can be consulted further:

```
TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration
  ;
```

Continuing like this leads to covering the whole tree. If the reader is interested in retracing the whole process, they can see the rather lengthy list of productions used in this example in appendix C.

We should note that the complete specification contains many seemingly unnecessarily detailed rules which may be not very intuitive to understand. For example, type of an expression is determined by consecutively comparing the expression against each of the expression types:

```
Expression:
  LambdaExpression
  AssignmentExpression

AssignmentExpression:
```

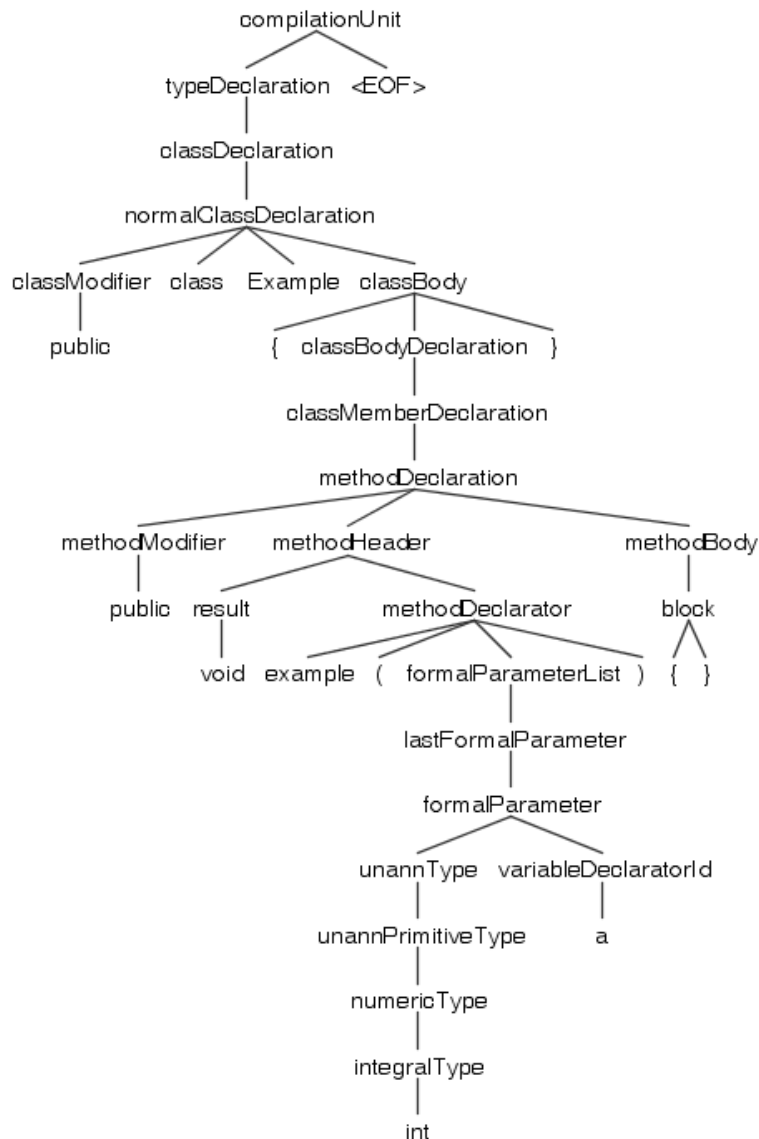



Figure 3.3. Java code parse tree.

```

ConditionalExpression
Assignment

```

```

ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression ? Expression : ConditionalExpression
ConditionalOrExpression ? Expression : LambdaExpression

```

```

ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression || ConditionalAndExpression

```

```

ConditionalAndExpression:
InclusiveOrExpression
ConditionalAndExpression && InclusiveOrExpression
...

```

Each of these decisions adds a level to the parse tree. Due to this, parsing some expressions might result in a tree similar to the one in figure 3.4 representing this code snippet:

```
if(a > 5) {}
```

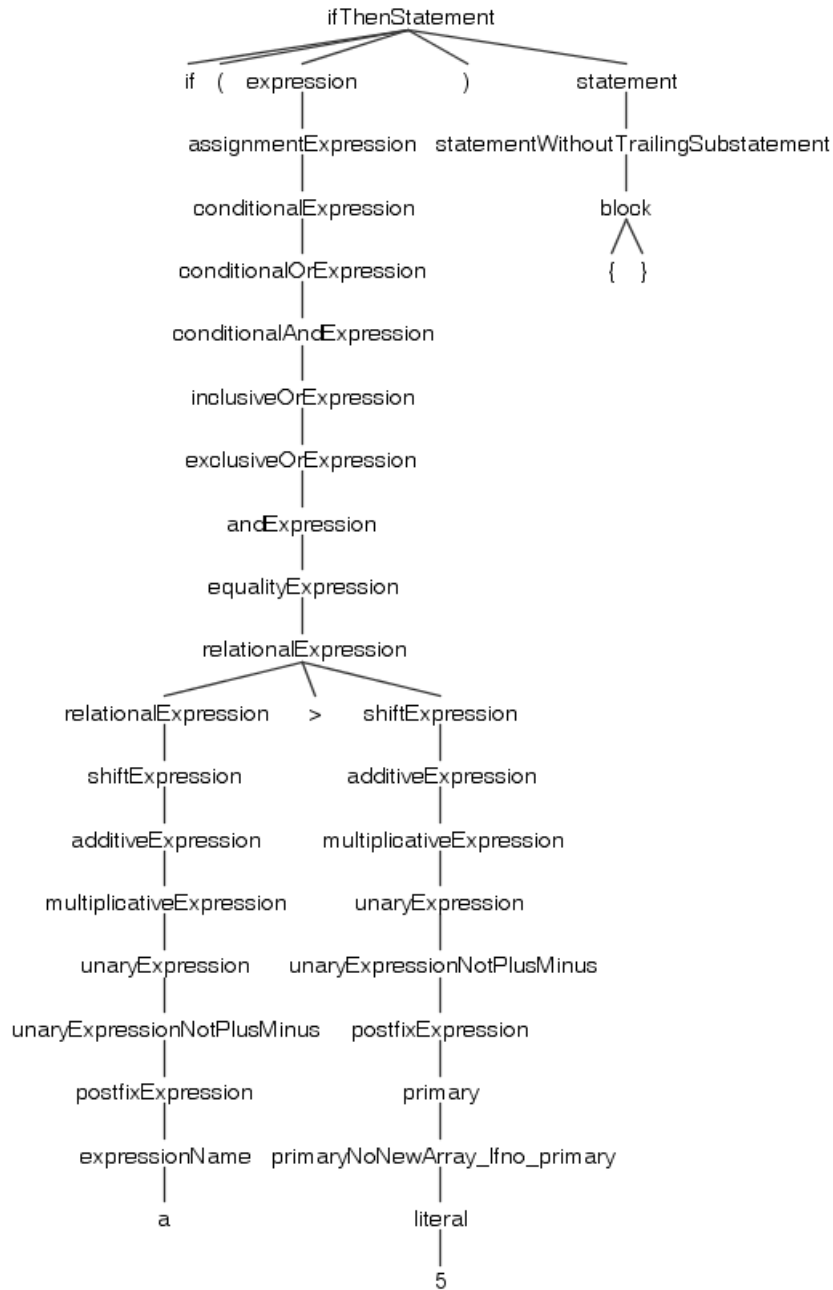


Figure 3.4. Expression parse tree.

This concludes the introduction to the lexical and syntax analysis. We will now continue with the research of existing parsers.

Chapter 4

Existing solutions

This chapter will evaluate existing projects dealing with parsing of the Java language and AST generation. The reason for doing this evaluation is to try to choose a solution that might be used by our intended application.

Firstly, we will state criteria of the selection and then we will introduce the individual projects. If there are more feasible solutions found, we will compare them to choose the more appropriate one.

4.1 Selection criteria

We decided the criteria of choice of parser to use in this project to be the following:

- must support **latest Java version**
- must be **open-source**
- should support **modification** of AST
- should be able to generate **visualization** of AST
- should be well **documented** and provide enough **examples** of use
- should be currently **active**

Seemingly, this functionality is not much often sought after, as examples of use of those projects and other resources regarding the matter are rather scarce and in many cases outdated.

We conducted research of this field of interest largely by exploring observations made by fellow programmers on blogs and forums. Information about projects that satisfied the criteria most closely is outlined in table 4.1.

parsers	Comp. Tree	Eclipse	JavaParser	Spoon	ANTLR
forks (16/5/16)	-	-	159	47	401
Java version	8	8	8	8	8
open source	yes	yes	yes	yes	yes
modification	no	yes	yes	yes	no
visualization	no	no	no	yes	yes

Table 4.1. Available parsers overview.

All tools are written in Java.

First two parsers are parts of larger projects. *Compiler Tree API* can be found in the primary Java compiler and *Eclipse JDT AST* is a part of toolbox Eclipse JDT.

The remaining examples are standalone projects hosting their code on GitHub. *Java-Parser* and *Spoon* are both Java oriented and both support modifications of the AST.

ANTLR is a very popular parser generator and there is Java grammar available for it, but it is impossible to use it to modify code.

The parser we eventually chose for this project was JavaParser. Reasons are described in the following sections.

4.2 Selected projects

This section will comment on the candidates in question. Most importantly, we will list their advantages and disadvantages.

4.2.1 Compiler Tree API

Compiler Tree API ¹⁾ is a part of *javac*, the primary java compiler found in Java Development Kit (JDK).

It can be used immediately after importing these packages:

- `com.sun.source.doctree`
- `com.sun.source.tree`
- `com.sun.source.util`

As it is a part of JDK, it is not necessary to download any additional libraries to use it. Its source code is publicly available, as is the rest of *javac*. However, the API is not very well documented and there are no official examples of usage. It is possible to find a few good unofficial ones ²⁾ ³⁾. But still, it is not well applicable to this project, as there are no indications of support of AST modification.

Advantages:

- not necessary to download further libraries
- as a part of an official tool should be reliable

Disadvantages:

- does not support modification of AST
- not well documented
- lack of examples of usage

4.2.2 Eclipse JDT AST

Eclipse JDT AST can be found in Eclipse Java Development Tools (JDT). It is not necessary to run it in Eclipse IDE. It has better documentation than Compiler Tree API and there is even an official manual, however, it is deprecated ⁴⁾. There are more examples to be found, one of the best ones would be a tutorial series featured in ⁵⁾.

¹⁾ *Compiler Tree API documentation*. <http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/>

²⁾ Jakub Holý: *Using Java Compiler Tree API to Extract Generics Types*. <https://dzone.com/articles/using-java-compiler-tree-api>

³⁾ Andrey Redko: *Java Compiler API*. <https://www.javacodegeeks.com/2015/09/java-compiler-api.html>

⁴⁾ *Eclipse Corner Article: Abstract Syntax Tree*. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

⁵⁾ *Eclipse JDT Tutorials*. <http://www.programcreek.com/2011/01/best-java-development-tooling-jdt-and-astparser-tutorials/>

Unfortunately, many of the examples are outdated and are not compatible with the current versions.

Advantages:

- supports modification of AST
- more tutorials available

Disadvantages:

- official resources are deprecated

■ 4.2.3 ANTLR

ANTLR ¹⁾ is a popular parser generator, meaning it builds a parser of any language it receives grammar specification of. There is a large list of available grammars of many programming languages, including Java. Being able to understand more languages would make it an ideal candidate, it cannot however modify the AST. It is well documented and the official homepage contains a number of tutorials, there is even an official reference book. It supports visualization of the AST both as a tree view and as a diagram.

Example of a custom grammar and its visualization can be seen in figure 4.1. The example is taken from ANTLR homepage.

Grammar:

```
grammar Expr;
prog: (expr NEWLINE)* ;
expr: expr ('*' | '/' ) expr
     | expr ('+' | '-' ) expr
     | INT
     | '(' expr ')';
NEWLINE : [\r\n]+ ;
INT      : [0-9]+ ;
```

Input:

```
100+2*34
^D
```

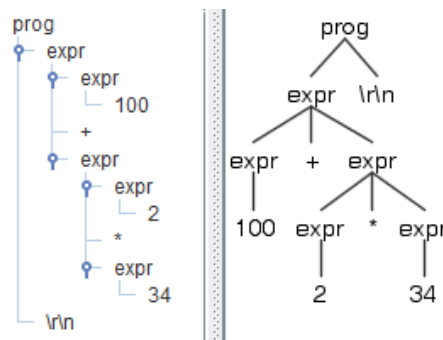


Figure 4.1. Example of ANTLR visualization.

¹⁾ ANTLR. <http://www.antlr.org/>

Advantages:

- large community
- well documented, many tutorials
- support of multiple languages
- supports AST visualization

Disadvantages:

- does not support modification of AST

■ 4.2.4 Spoon

Spoon ¹⁾ is a library dedicated to the analysis of Java code only, allowing modifications of the AST. It supports visualization in form of a tree view. Nevertheless, there were a few issues during the evaluation, such as the official page not updating examples with new versions or the parser not being able to parse the code without imports of libraries used by the parsed code. These issues were eventually fixed, but they contributed to favouring the next parser.

Advantages:

- supports modification of AST
- supports AST visualization

Disadvantages:

- examples tend to be not updated

■ 4.2.5 JavaParser

JavaParser ²⁾ is similar to Spoon, but does not support AST visualization. Aside from that, it has a few tutorials showing the essential functionality, which cover almost everything needed to work with it. Out of the evaluated parsers, it was the least problematic one to use with no major disadvantages.

Advantages:

- supports modification of AST
- fairly active community
- good official examples and documentation

■ 4.3 ANTLR and JavaParser comparison

As JavaParser cannot visualize the AST, there was an idea to use JavaParser to make changes in the AST and then let the resulting code be parsed by ANTLR to produce visualization. However, comparison of trees produced by both parsers proved a very

¹⁾ *Spoon - Source Code Analysis and Transformation for Java*. <http://spoon.gforge.inria.fr/index.html>

²⁾ *JavaParser by javaparser*. <http://javaparser.org/>

important point: that although parsers are based on the same official Java grammar specification, the results are not necessarily the same.

Figures 4.2 and 4.3 show visualized AST of the very simple following code parsed by ANTLR and by JavaParser.

```
package example;

public class Example {

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

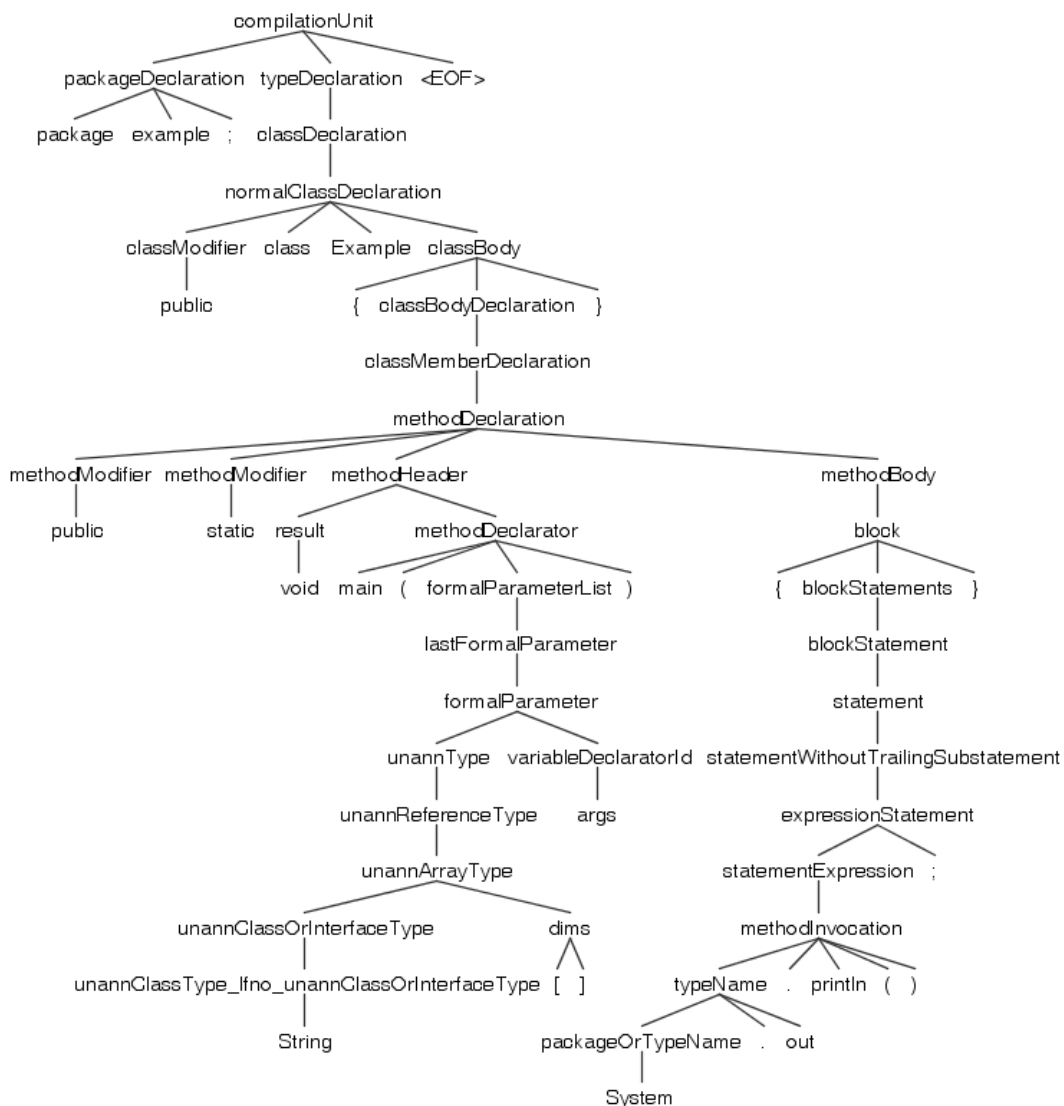


Figure 4.2. AST parsed by ANTLR.

It is evident that the tree provided by ANTLR contains far more information than the other one and on top of that there are many differences in how the nodes are organized.



Figure 4.3. AST parsed by JavaParser.

This is fault of ANTLR not being a Java parser exclusively. JavaParser is tuned up to process Java only, and thus omits some information, which ANTLR cannot allow to do. We can see the evidence by consulting the official Java specification [7], by comparing for example the structure of `MethodDeclaration` with the respective part of the tree:

```

MethodDeclaration:
{MethodModifier} MethodHeader MethodBody

MethodHeader:
Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

MethodDeclarator:
Identifier ( [FormalParameterList] ) [Dims]
  
```

ANTLR strictly follows the grammar: it lists under the `Method Declaration` node both modifiers of the method, its header and its body in nodes bearing the same label as their specification counterparts. On the other hand, JavaParser does not have nodes with modifiers and places return type and parameters directly below the `Method Declaration` node and not in `Method Header`.

4.4 Summary

In this chapter, we have conducted evaluation of the currently most popular open-source parsers of the Java language. After finding out that most of them do not provide enough resources explaining how to use them, the best choices ended up to be ANTLR and JavaParser.

ANTLR provides the most support, however, it cannot be used for modifications of the AST, which is an essential feature of our intended tool. However, not giving up completely on it, we considered using it for visualization of AST generated by JavaParser. This way, we found out that AST provided by ANTLR contains much more details, rendering the two representations incompatible.

This observation lead to the conclusion that JavaParser will have to be accompanied by a proper visualization provider. This might actually be a good thing, as the application will have its own graphic user interface and including foreign elements in it might prove to be problematic.

Currently, it is not quite possible to determine whether lighter AST provided by JavaParser will prove to be a good choice. For now, we can surely confirm that it is far more transparent, especially for people not familiar with the grammar specification. However, whether it really is better fitting will be proven by time.

Now that we have chosen our parser, we can advance to designing our application in the next chapter.

Chapter 5

Design

This chapter describes requirements the resulting application should accomplish, defines possible use cases and suggests structure of the components.

5.1 Requirements

In this section, we will elaborate requirements stated in the assignment and their further expansions, and suggest how they should be satisfied.

Some of the requirements were already mentioned in chapter 4, where we described criteria of selection of parsers. Our choice, JavaParser, fulfils the requirement of AST modification, however, it does not support AST visualization, for which our tool will have to implement its own means.

5.1.1 Functional requirements

Perform lexical and syntax analysis of the code and create AST.

This functionality will be provided by JavaParser.

Visualize AST.

The AST should be accessible from some kind of visualization, probably an expandable tree view and a diagram like the ones available in ANTLR (figure 4.1). Representations will enable selection of nodes and there should be always the same node selected in both of them, meaning they should be connected.

Present connection between nodes and their representation in code.

The source code should be viewable while working with AST and it should be possible to select AST nodes by clicking on their location in the code and vice versa.

Filter out nodes by type.

The user should be given an opportunity to choose types of nodes that will not be visible in the tree.

Render list of tests in the given class.

Tests in JUnit framework are denoted by `@Test` annotation. This means that the parser will process annotations, from which the application will choose the ones with the correct name.

Modify AST: create, edit or delete nodes.

Both create and edit operations will be administered by simply editing the code in a provided editor, which should be the most intuitive way, as there are many types of nodes and choosing and placing them in the right context would require decent

knowledge of the grammar, resulting in unnecessary complexity. Node removal can be executed by editing the code as well, but at the same time it will be possible to delete nodes by selecting them from the tree. To avoid mistakes and confusion, before deleting the nodes will be visibly marked in all of their representations (list, diagram, code).

There is no requirement to check undeclared variables.

Search in AST by criteria.

The application should make it possible to search for nodes meeting given criteria (for example node name).

Search in AST by patterns.

By being provided a sequence expected to be found, the application will search the AST for occurrences of the given sequence and visibly mark the results.

The pattern idea is illustrated in figure 5.1 and table 5.1. The required sequence in this example is BC.

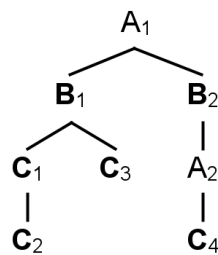


Figure 5.1. Example of pattern search function.

result 1	B_1C_1
result 2	B_1C_2
result 3	B_1C_3
result 4	B_2C_4

Table 5.1. Results of pattern search example.

■ 5.1.2 Non-functional requirements

Implement in Java.

The tool should be written in Java language.

Provide graphic user interface.

To make working with the tool as comfortable as possible, the application should be controlled through a graphic user interface (GUI).

Accept JUnit source code.

JUnit framework does not contain any elements uncommon for plain Java. It relies heavily on annotations, which JavaParser supports.

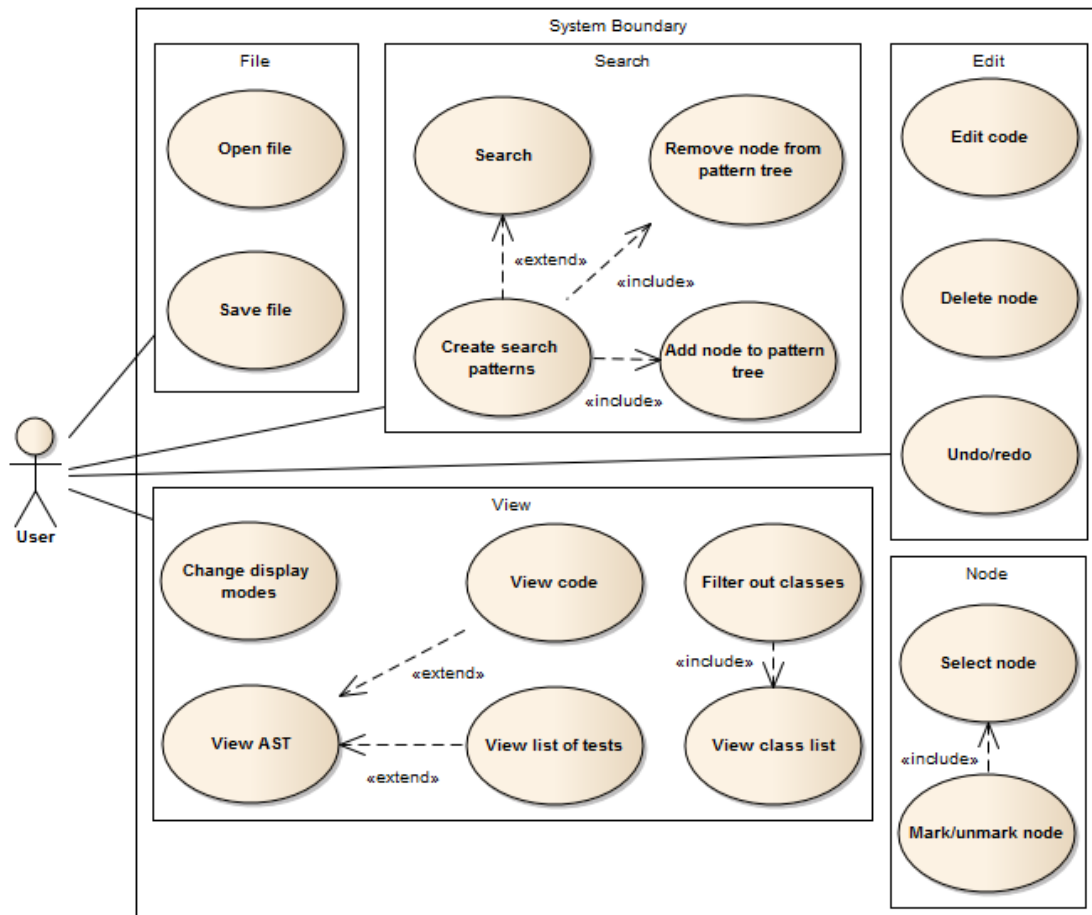


Figure 5.2. Main use cases diagram.

5.2 Use cases

We derived the set of main use cases from the requirements described in the previous section, appending some additional ideas. They are illustrated in figure 5.2. In the following sections, we will describe the use cases further and illustrate usage of the more complicated functions by scenarios.

5.2.1 AST viewing

View AST.

The user will be able to view the AST representation. It should be always visible in the application, without requiring any additional steps.

View code.

The user may browse the code belonging to the AST, connected to it in a manner that location in the code matching the currently selected node will be highlighted.

View list of tests.

The tests will be visible in the AST as any other node. On top of that, they may be viewed separately in a list containing only test methods.

View class list

The application will provide a list of node types currently present in the code.

Filter out classes.

The application should be able to filter out given types (classes) of nodes. This functionality will be administered by the class list. The user might use it to choose types of nodes that should not be visible in the AST.

Note: Nodes should be filtered out in a way that their children will be appended to their parent.

1. The user will choose types of nodes to filter out.
2. The user will confirm their choice.
3. The system will reload the representations of AST to match the selection.

Change display modes.

The application will provide the possibility of altering display of some elements:

- whether nodes should be represented by their type, name or both
- whether the class list should be ordered simply by the alphabet or by the type hierarchy

■ 5.2.2 Node operations**Select node.**

The user might select a node in either representation of the AST), which will highlight it in the other ones.

Mark or unmark node.

The nodes may be marked by one of the two types of markers:

- **delete** - marks node to be deleted or a node in the pattern tree that should be marked to be deleted when found
- **search** - marks either AST nodes that should serve as search roots or node types in the class list that should be searched for

■ 5.2.3 Tree editing**Edit code.**

The source code loaded in the application might be edited in an embedded editor. After editing, the code will be parsed again, updating the AST.

1. The user will edit the code in the editor.
2. The user will confirm their changes.
3. The system will reparse the code and reload the representations of AST to match the edited code.

In case the changes made by the user would render the code unparseable, the editor will announce the problem and will not save the changes.

Delete node.

The user might delete any node from the AST. They will first mark the node to be deleted and then confirm the selection, which will delete the nodes and reparse the AST.

1. The user will choose node or nodes to be deleted.
2. The user will confirm the selection of nodes to be deleted.
3. The system will delete the nodes and reload the representations of AST.

Undo or redo.

States of the AST between edits are stored and it should be possible to move backwards or forwards between them.

1. The user will attempt to go back or forward in history.
2. The system will take the desired AST and reload the representations of AST according to it.

In case it is not possible to undo or redo, the application will notify the user.

■ 5.2.4 File operations**Open file.**

The application should provide a way to open the file to be parsed and give appropriate feedback in case it is unparseable.

1. The user will choose a file to be opened.
2. The system will parse the code in the file and provide the AST representations.

If the file does not exist or the code in it is not valid Java code, the application will notify the user.

Save file.

Resulting code can be saved to a file.

1. The user will choose a name for the saved file.
2. The system will print the code representation of the AST root into the file.

■ 5.2.5 Searching**Search.**

The application should offer the user searching for nodes in the AST by the following criteria:

- name - identifier of the node
- class - node type
- root - node from which the search should start

Any of the criteria can have multiple values.

1. The user will enter the criteria in the following ways:

- name will be entered into a text field
- class will be marked in the class tree
- root will be marked in the AST

2. The system will return a list of nodes satisfying the criteria.

Furthermore, the user might search by patterns.

Create search patterns.

A pattern is a tree denoting a sequence which should be found in the AST. It will be possible to state whether the found node has to be a direct descendant of the node of the type stated by the previous pattern node. Node in a pattern can be marked for deletion, resulting in all found occurrences to be marked the same.

The user should be allowed to edit the pattern trees by adding or deleting nodes in them.

Add node to the pattern tree.

The user should be able to add a node to the pattern tree. Either a root node of a new pattern tree or a child of an existing pattern node can be added.

Remove node from the pattern tree.

The user might remove a node from the pattern tree. Children of the removed node will be connected to its parent.

■ 5.3 Structure of the application

This section will describe the structure of the main parts of the application: the **core application** and the **graphical user interface (GUI)**. The core will be usable independently from the GUI, which will use an instance of the core to perform its tasks.

Firstly, we will state layers of the application and external libraries used by them. After that, we will describe components of the main parts.

■ 5.3.1 Layers and used libraries

Layers of the application with external libraries employed by them are illustrated in figure 5.3. We will introduce them in the following sections, which also describe how and why they are used by the application.

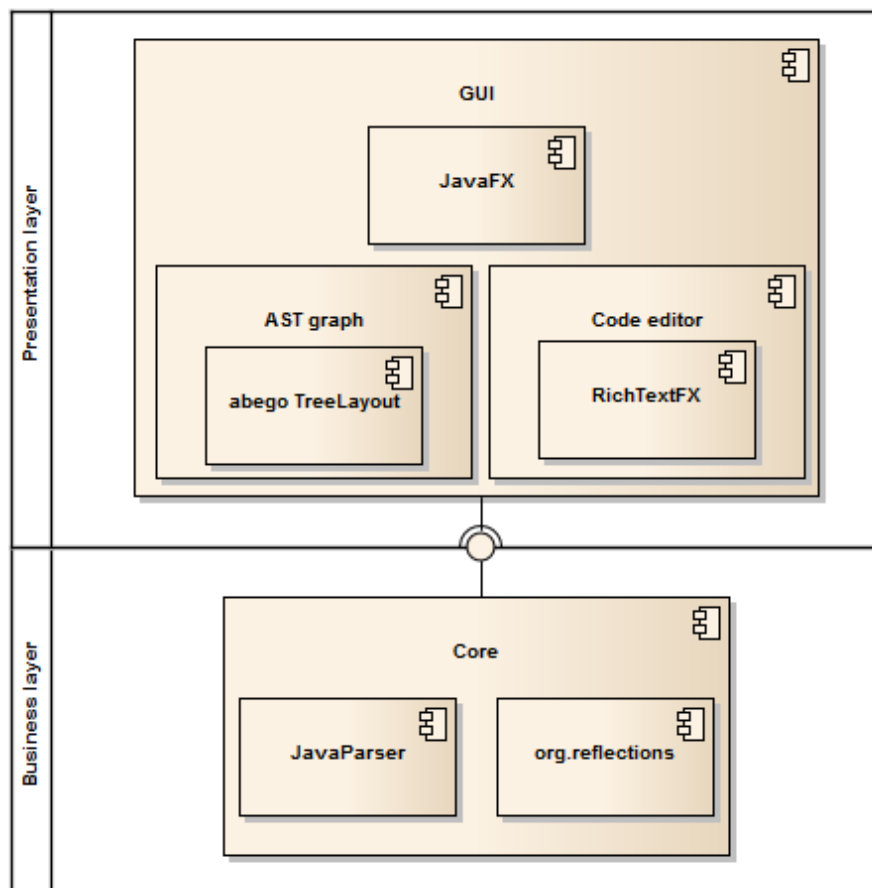


Figure 5.3. Layers of the application and the employed libraries.

■ 5.3.1.1 Business layer

As **JavaParser** was already discussed in 4.2.5, we will explain usage of **Reflections** ¹⁾ It is a Java library providing, for example, collection of subtypes of a given class or elements annotated by a given annotation.

Its usage looks like this:

```
Reflections reflections = new Reflections("package");
Collection<? extends Class> set = reflections.getSubTypesOf(C.class);
```

We need to employ this library to retrieve list of subtypes of **Node** class from **JavaParser**, which will be used, for example, in the class list used to choose node types to be filtered out.

■ 5.3.1.2 Presentation layer

GUI for the application will be written using **JavaFX** library, which is bundled with JDK. Its elements can be found in packages beginning with `javafx`. It is intended as a replacement for Swing [8].

To make work with the library more comfortable, it is possible to use FXML files, used to describe element arrangement in the layout using XML. Furthermore, FXML files

¹⁾ *Reflections library*.. <https://github.com/ronmamo/reflections>

can be managed by Scene Builder tool ¹⁾, which provides a GUI to design the layout in the WYSIWYG way.

We chose this platform for the GUI because it is a native Java library, which should allow for maximum compatibility with our Java application.

Abego Treelayout ²⁾ is a Java library that calculates layout of nodes of an arbitrary tree from input values like width and height of nodes and horizontal and vertical gaps between them. It guarantees the edges will not cross each other. Data obtained from the library can then be used to draw a tree, completely separating the drawing process from assembling the layout.

ANTLR parser generator mentioned in chapter 4.2.3 uses abego TreeLayout for its AST visualization. This made us choose this library, for we wanted to achieve a similar look of the tree. Another important point was that it is independent from the drawing provider, because other solutions usually relied on Swing.

RichTextFX ³⁾, written in Java, provides a base for rich-text editors and code editors. The application will use its **CodeArea** element for displaying and editing source code. Its configuration will be based on a demo from the GitHub page ⁴⁾.

As these functions were exactly what we needed, we decided to use the library in our application.

The editor provides exactly the functions we need in our application, for example:

- line number display
- undo and redo operations
- immediate keyword highlighting
- caret position retrieval

■ 5.3.2 Core application

Figure 5.4 lists the main data structures used by the business layer. When a new source code is loaded into the system, structures depicted in the Core application state component have to be reset. State of the application in this context will be defined as objects related directly to the AST currently managed by the application.

An exception are the two remaining structures pictured out of the state component: **List of filtered classes** remains the same for each session and the same is true for the **Pattern tree**. Both will be stored in a file to persist between the runs of the application.

Compilation unit is the root of the current AST, as provided by JavaParser.

AST is a structure encapsulating nodes of the JavaParser AST. There are two main reasons not to use the structure provided by JavaParser directly:

- it is necessary to store references to the node representations in the GUI (graph diagram, items in the tree view)
- tree filtering

¹⁾ *Scene Builder*. <http://treelayout.sourceforge.net/>

²⁾ *abego TreeLayout*. <http://treelayout.sourceforge.net/>

³⁾ *RichTextFX*. <https://github.com/TomasMikula/RichTextFX>

⁴⁾ *JavaKeywords.java.*: <https://github.com/TomasMikula/RichTextFX/blob/master/richtextfx-demos/src/main/java/org/fxmisc/richtext/demo/JavaKeywords.java>

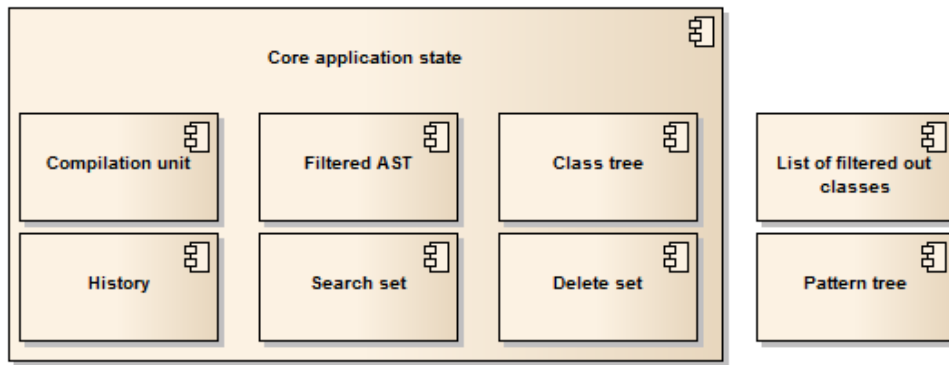


Figure 5.4. Data structures of the application.

We could theoretically evade the first issue by storing the references by the `setData` method the `JavaParser` nodes offer, however, managing proper structure should be easier.

The second reason is more insidious: it is necessary to deal with occasional holes in the tree, which will appear if a node is filtered out. We could solve this simply by skipping the nodes when a representation is being built, however, in case the filtered tree would be needed again, the same process would have to be repeated. This way, the filtered tree can be stored and reused.

Class tree is a tree holding node types appearing in the AST. It will be constructed after building the filtered AST, as through this process the present classes will be detected.

History is an object storing previous states of the AST.

Search set is a set of nodes that are currently marked to serve as search roots. A node should not be in the search set and delete set at the same time.

Delete set is a set of nodes that are marked for deletion.

■ 5.3.3 Graphic user interface

Analogous to the case of core application, figure 5.5 depicts elements of the presentation layer that store information about the state of the loaded AST.

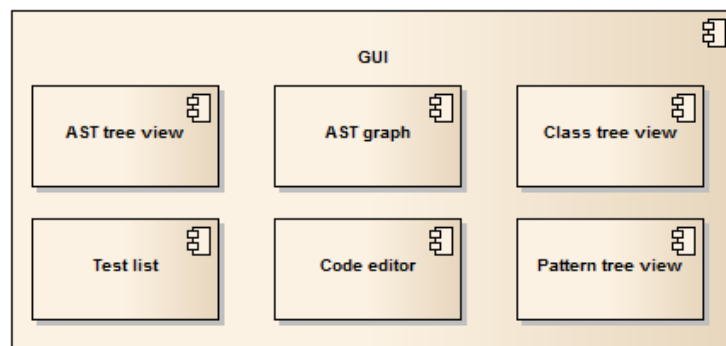


Figure 5.5. GUI elements.

AST tree view is an expandable tree view of the nodes in the tree.

AST graph is the diagram version of the tree view.

Class tree view contains the list of the appearing node types. It shall function as a checkbox form to filter out redundant classes.

Test list is a separate list containing only method declarations conforming to the JUnit test definition.

Code editor shows the parsed code and it should be possible to use it to edit the code.

Pattern tree view contains the list of saved patterns.

5.3.4 Connection between components

The core application will be independent from the GUI, whereas the GUI will use an instance of the core and mirror its structures. We can observe connections between core and GUI structures in figure 5.6.

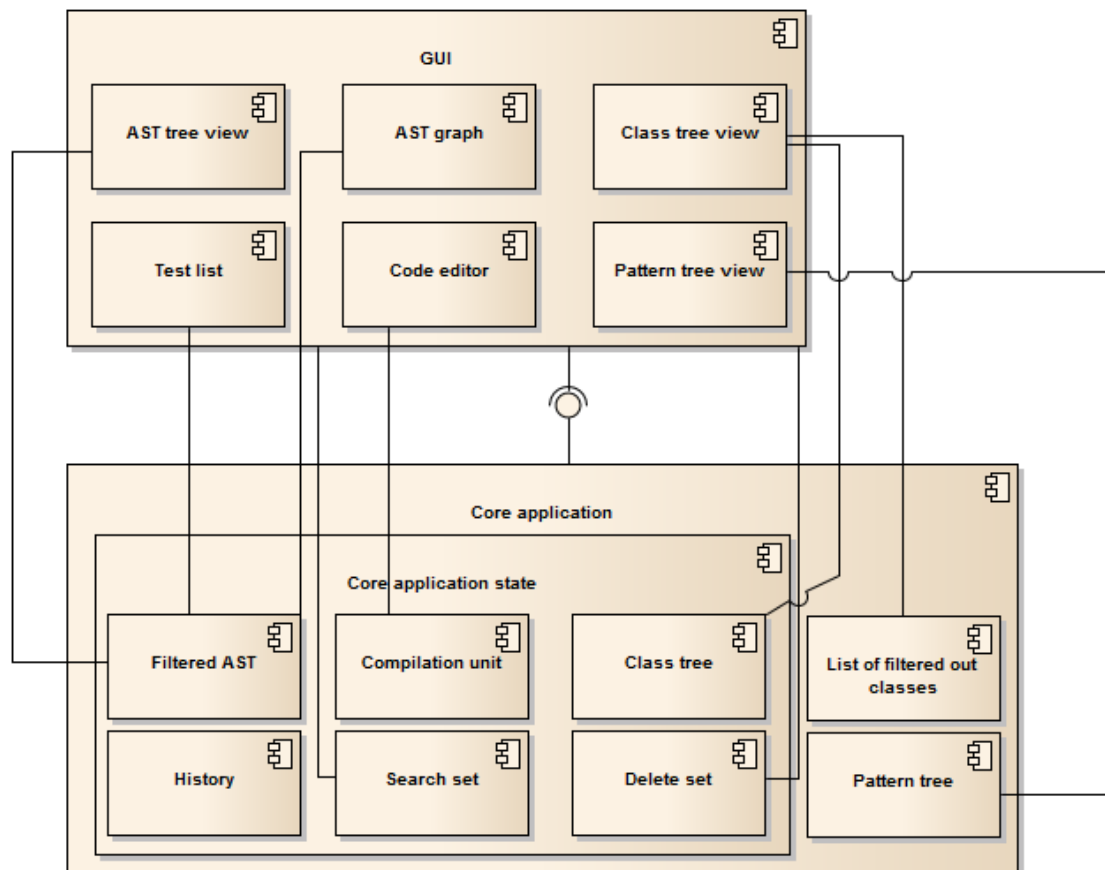


Figure 5.6. Connections between core and GUI.

Now that we know what our application should theoretically look like and what functions it should have, we can move onto the implementation process.

Chapter 6

Implementation

This chapter will describe implementation details of the final application. Firstly, we will explore JavaParser to find out about its functions and how we could employ them. After that, we will explain how our application is implemented, beginning with data structures and function providers used across the application. Then we will proceed to describing the core, building a new tree first and continuing with the remaining functions. Finally, we will explore the GUI and its elements and functions.

Class diagrams of each package can be found in appendix E.

6.1 JavaParser features

JavaParser is the most essential component of the application, as it provides the AST the rest of applications works with.

We will look into how JavaParser works and how we may incorporate its functionality in our application.

6.1.1 Types of nodes

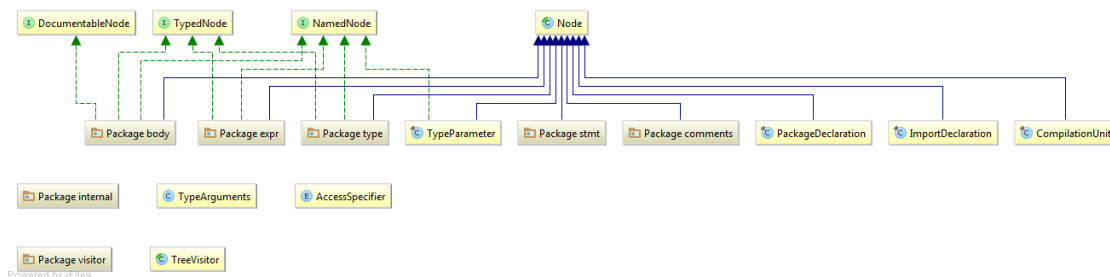


Figure 6.1. JavaParser ast package.

Nodes of the AST are represented by instances of the `Node` class. We can find their implementations in the `ast` package, diagram of which we can see in figure 6.1. The root node in every AST is of type `CompilationUnit`.

There are a few interfaces to make tasks with nodes having similar attributes easier: `DocumentableNode`, `NamedNode` and `TypedNode`. This project makes use of `NamedNode`, which is implemented by nodes that have `NameExpr` as their child.

Each node has a reference to its parent and owns a list of its children.

Nodes contain coordinates of the portion of the code belonging to it: *begin line*, *begin column*, *end line* and *end column*.

We can assign an Object to the node through `setData` method.

For reference, complete list of `Node` types can be found in appendix D.

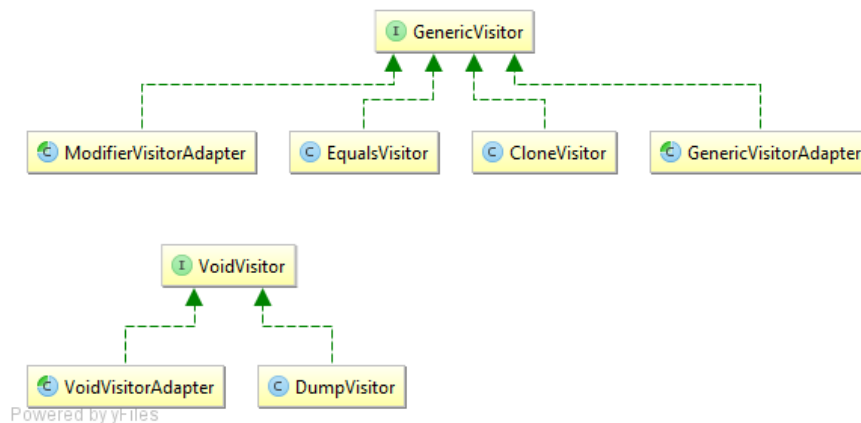


Figure 6.2. JavaParser visitors.

6.1.2 Visitors

Visitors, as the name suggests, use the *Visitor pattern* to be able to perform a specific action for every type of node. The interfaces have `visit` methods for each of the types and each type has an `accept` method to run the visitor with itself.

```
public <A> void accept(VoidVisitor<A> v, A arg) {
    v.visit(this, arg);
}
```

It is possible to create custom visitors implementing the *GenericVisitor* or *VoidVisitor* interface, or extending one of the *Adapter* classes if not all `visit` methods are to be overridden.

The `arg` argument can be used for an arbitrary object.

6.1.3 Tree traversal

There are two ways how to perform tree traversal.

The first approach is to use `getChildren` method of every node. This is employed by the class `TreeVisitor` (not included in the `visitor` package), which simply recursively calls the visiting method on all children of the node until there no unvisited children. We may use it when it is not necessary to consider type of the node, thus not being able to access specific attributes of the node without casting them to a subtype.

```
public void visitDepthFirst(Node node) {
    this.process(node);
    Iterator var2 = node.getChildrenNodes().iterator();

    while(var2.hasNext()) {
        Node child = (Node)var2.next();
        this.visitDepthFirst(child);
    }
}
```

The second approach uses visitors. To traverse the tree, they call the `accept` method on every type of their attributes separately. This way, we know the type of child in advance, unlike in the case of `getChildren` approach.

Example: visit method from VoidVisitorAdapter for ForeachStmt

```
@Override
public void visit(final ForeachStmt n, final A arg) {
    visitComment(n.getComment(), arg);
    n.getVariable().accept(this, arg);
    n.getIterable().accept(this, arg);
    n.getBody().accept(this, arg);
}
```

6.1.4 Node removal

The method of removing nodes from the AST suggested in JavaParser manual is to have the node found by a visitor extending the ModifierVisitorAdapter. Return value of methods of this visitor is the node on which the method has been called. Node can be removed by instead returning null.

Example from the official manual ¹⁾:

```
Task is delete a=20 from here:

class D {
    public int foo(int e) {
        int a = 20;
    }
}

class MyVisitor extends ModifierVisitorAdapter {

    @Override
    public Node visit(VariableDeclarator n, Node arg) {
        if (n.getId().getName().equals("a")
            && n.getInit().toString().equals("20")) {
            return null;
        }
        return n;
    }
}
```

In this manner, any type of node should be deleted. The easiest way to do this would be to make a visitor which deletes node accepting the visitor, equal to the node given as `arg`. This is necessary because the visitor has to be accepted by the parent of the node for the removal to actually happen, meaning that without the comparison the parent would be removed instead.

```
@Override
public Node visit(VariableDeclarator n, Node arg) {
    if (n.equals(arg)) {
        return null;
    }
    return n;
}
```

¹⁾ JavaParser manual. <https://github.com/javaparser/javaparser/wiki/Manual>

However, there is an issue: some nodes are tied to their parents in a way that deleting them disrupts the code and renders it unparseable. Unfortunately, this property is not defined by the type of the node, but rather by relationship between type of the node and type of its parent.

Example: Considering example in figure 4.3, there are two nodes of the type `NameExpr`: `example` and `System`. They appear in these parts of the code:

```
1) package example;
2) System.out.println();
```

If we remove `example`, the declaration will be syntactically incorrect, as `package` keyword has to be followed by its identifier. On the other hand, there is nothing wrong about `out.println()`.

Alternatively, it is possible to delete nodes without using `ModifierVisitorAdapter` by using setters of the parents to set the respective attribute to `null`, or, in case of attributes that are grouped in lists, by removing them from the collection.

To determine which node types are deletable, it is necessary to consult `ModifierVisitorAdapter` and see which attributes can be set to `null` by it.

It is impossible to delete nodes by removing them from the collection obtained by `getChildrenNodes`.

■ 6.1.5 Summary of issues

■ 6.1.5.1 Lack of universal delete method

As we already discussed in the Node removal section, there is no way to remove nodes uniformly and at the same time not damage the code. It will be necessary to develop a solution that will consider context from which the node is being deleted.

■ 6.1.5.2 Indistinguishable nodes

In some cases, there are no means of telling children nodes of the same type apart aside from their order. For example:

```
Scope.method(arg1, arg2);
```

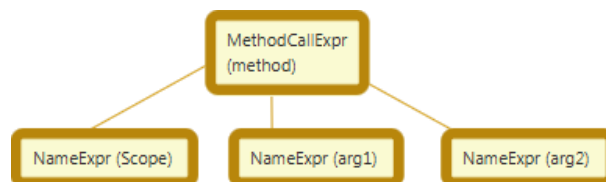


Figure 6.3. Indistinguishable nodes example.

As seen in figure 6.3, all of the children of the `MethodCallExpr` are `NameExpr` nodes. There is no way to retrieve the information that the first node represents scope and the other two are arguments; all of them could very well be arguments.

6.1.5.3 Necessity of double-parsing

JavaParser removes empty lines inside blocks, which is not an issue, however, this removal happens only when the nodes containing them are printed using `toString` method. The numbers of lines and columns stored are from the version with empty lines not yet removed. To keep up with this difference it is either necessary to use the original code instead of the printed one, or to parse the `CompilationUnit` again to update the numbers.

Similar problem happens with deleting. When a node is deleted, the numbers will not update until the `CompilationUnit` is parsed again.

6.2 Data structure dictionary

To allow the reader to have better orientation in various elements used by the application, we will provide a little dictionary. From now on, every element will be referred to by the name stated here.

- *Node* - `Node` instance used by JavaParser
- *TreeNode* - `TreeNode` instance encapsulating Nodes
- *CompilationUnit* - `CompilationUnit` instance, which is the root of the AST provided by JavaParser, formed by Nodes
- *Filtered AST* - AST representation consisting of TreeNodes
- *Class tree* - node type hierarchy consisting of TreeNodes
- *Pattern tree* - node representing the patterns consisting of TreeNodes
- *Filtered classes* - list of filtered out classes
- *Search set* - set of TreeNodes marked as search roots
- *Delete set* - set of TreeNodes marked for deletion
- *Undeletable set* - set of TreeNodes marked as undeletable. Appears in GUI only.

6.3 Main data structures and function providers

In this section, we will describe the data structures and function providers used at various places in the application. Firstly, we will describe `TreeNode`, which is the structure encapsulating Nodes. Then we will mention Roles, continuing with Visitors and Printers, used to transform TreeNodes into different representations. Finally, Modes and contents of `settings` package will be mentioned.

6.3.1 TreeNodes

TreeNodes are found in package `analyzer.node`, which contains their hierarchy.

Every node used in the application is either a `Node` or a `TreeNode`. They are used in these cases:

- Filtered AST
- Class tree
- Pattern tree

Each of these trees has a corresponding JavaFX `TreeView` in the GUI. Because of this, each `TreeNode` contains a reference to its `TreeItem`, which is an object appearing in the `TreeView`.

The main usage of the trees is to transmit them from the core, where they are formed, to the GUI, where they are processed to be displayed.

`TreeNodes` are connected to their GUI representations via various means, depending on the type of the `TreeNode`. We can see these connections in figure 6.4.

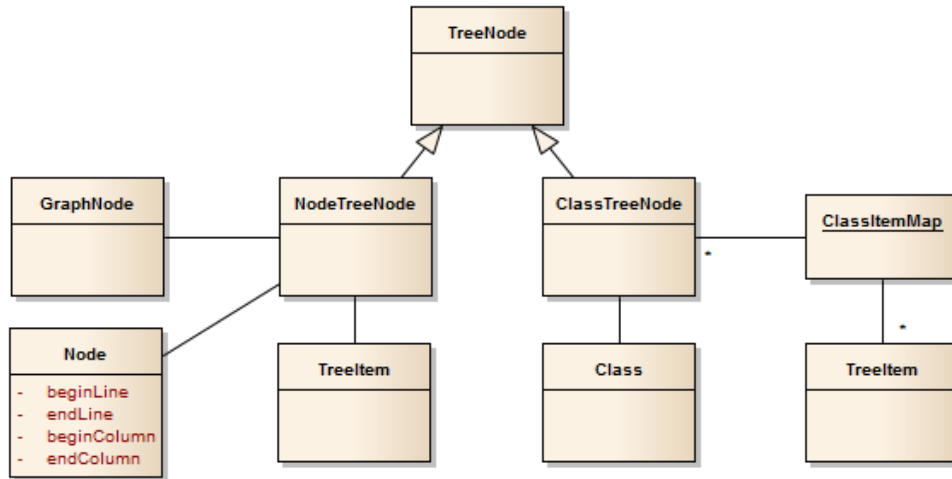


Figure 6.4. `TreeNode` hierarchy.

In GUI, `NodeTreeNodes` appear with titles in this format: `ROLE Class (name)`

NodeTreeNode instance encapsulates references to other node representations: `Node`, tree item node and graph node, and appears in the Filtered AST representation. These connections make it possible to highlight the other nodes when one of them is selected. As `Node` contains coordinates of the node in the code, the corresponding part can be highlighted in the code too.

If the `Node` of the `NodeTreeNode` has a feasible identifier, it is adopted by the `NodeTreeNode`. The identifier is obtained via `ToStringVisitor`.

They are comparable by their position in the code.

ClassTreeNode differs from `NodeTreeNode` by hosting a reference of a class instance. They are used in Class tree and Pattern tree. `ClassTreeNodes` in the Class tree are not allowed to have a name. In the Pattern tree, they can have the searched name.

6.3.2 Roles

Some `TreeNodes` may contain reference to a **Role** object, hosted in `analyzer.node`. Roles are the answer to one of the JavaParser issues described in 6.1.5: the indistinguishable nodes. If a node type has more attributes of the same type, the will be assigned a role, which will make it possible to tell them apart. For example, children of `MethodCallExpr` will receive `M_SCOPE` and `M_ARGUMENT` roles.

6.3.3 Visitors

Our visitors extend the JavaParser ones described in 6.1.2. The visitors are grouped into three packages: `analyzer.core.visitor.name`, `analyzer.core.visitor.role` and `analyzer.core.visitor.delete`. They provide functions that need to be different for every type of Node.

Visitors are always used with Nodes and not TreeNodes.

6.3.3.1 Name visitors

AbstractNamedNodeVisitor is a base for visitors used to operate with Node identifiers. This visitor visits Nodes that are implementing NamedNode interface, making access to their names uniform, as all of them implement the `getName` method. The `visit` methods are virtually the same:

```
@Override
public void visit(AnnotationDeclaration n, Node arg) {
    if (processNamedNodeAndContinue(n)) {
        super.visit(n, arg);
    }
}
```

The result of abstract method `processNamedNodeAndContinue` can decide whether children of the node should be visited by calling `super`.

ToStringVisitor retrieves name of the Node depending on its type. It extends `AbstractNamedNodeVisitor` and overrides further `visit` methods of Nodes that have attributes for which having a name makes sense. For example, a block will not have a name, while a binary expression can be represented by its operator.

Nodes without a name must have an empty `visit` method so they would not use the name of a descendant, called by the original method.

NameSearchVisitor, on the other hand, always has to visit all descendants, as its purpose is to find all Nodes of the given name or names. Aside from names it checks if the Node is of one of the given classes if they are provided.

6.3.3.2 Role visitors

CurrentRoleVisitor finds out role of the given node in the context of its parent.

AbstractChildrenNodesVisitor visits nodes which might have nodes with roles as their children. It is extended by **RoleSearchVisitor**, which, similarly to `NameSearchVisitor` searches for nodes with given roles, names and classes at the same time.

There is a separate search visitor for searching for given roles, as there are less node types with a role than those with a name, making the search more efficient.

6.3.3.3 Delete visitors

AbstractDeleteVisitor is a base of visitors used to delete nodes. `Visit` methods do not use `super` calls, as the visitor is always accepted by the parent of the node to be deleted and further propagation is not desired. They contain method calls to process collections, as these can be processed uniformly, unlike single attributes, which have to be accessed in the extending classes.

Example:

```
@Override
public void visit(AnnotationDeclaration n, Node arg) {
    processCollection(n.getAnnotations(), arg);
    processCollection(n.getMembers(), arg);
}
```

CheckDeletableVisitor checks whether a node is of a type that can be deleted from its parent. It does so by trying to find the node among the attributes of its parent and if it not found, it means that the node is impossible to delete.

DeleteVisitor deletes a node from its parent. Nodes contained in collections are removed from them, single attributes are set to null.

Example:

```
@Override
public void visit(AnnotationDeclaration n, Node arg) {
    if (n.getJavaDoc() != null && n.getJavaDoc().equals(arg)) {
        n.setJavaDoc(null);
    }
    super.visit(n, arg);
}
```

Super is called for the collections to be handled by `AbstractDeleteVisitor`.

6.3.4 Printers

Printers, placed in `analyzer.printer`, receive a `TreeNode`, which is a root of a tree, and build a new representation of it while traversing it.

All printers extend **AbstractPrinter** class, which performs DFS traversal using stack. DFS order is essential to keep connection between parents and children.

Unless order of the children of the node added in each loop is not reversed, the resulting tree will be mirror reversed. However, in majority of cases, the tree, which was build as a mirror image as well, will be passing through the printer, meaning it will return to its correct order. Due to this, children are not being reversed in this case.

Steps of the process are demonstrated in table 6.1, showing traversal of the tree in figure 6.5.

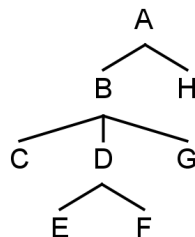


Figure 6.5. AbstractPrinter example.

expected	actual	stack
A	A	B H
B	H	B
C	B	C D G
D	G	C D
E	D	C E F
F	F	C E
G	E	C
H	C	

Table 6.1. Printer tree traversal steps.

StringPrinter prints text representation of the tree. It is used for writing the pattern to a file. Tree in figure6.5 would be printed like this:

```
A
-B
--C
--D
---E
---F
--G
-H
```

Number of dashes means depth of the node.

GenericItemPrinter produces `TreeItems` to be placed in a `TreeView`. The most important part of its algorithm determines in what relationship the current and the newly created node are. We can observe it in this code snippet:

```
if (depth > curNodeDepth) {
    currentItem.getChildren().add(item);
} else if (depth < curNodeDepth) {
    TreeItem<T> parent = currentItem.getParent();
    for (int i = 0; i < curNodeDepth - depth; i++) {
        parent = parent.getParent();
    }
    parent.getChildren().add(item);
} else {
    currentItem.getParent().getChildren().add(item);
}
```

In the first case, depth of the new node is greater than depth of the other. This means the new node is a child of the previous.

Second case shows situation where depth of the new node is lower. This means that its parent appeared earlier in the hierarchy. The difference of depths denotes which ancestor it is.

Third case, in which the depths are equal, means that the nodes are siblings.

NodeItemPrinter extends the previously described `GenericTreePrinter`. It fills the `nodesOnLine` map, which keeps trace of nodes appearing on each line of code. This enables to determine which portion of code should be highlighted when we click in the code editor.

CheckBoxItemPrinter prints the class `TreeView`. Additionally, it keeps map of classes pointing to their `TreeItems` (as it is impossible to place the reference in a `Class` object) and manages changes in the list of filtered out classes. Each `TreeItem` has a checkbox attached, which serves for managing filtered out classes.

ClassItemPrinter prints plain `TreeItems` used in pattern `TreeView`.

GraphPrinter fills the tree to be used by abego `TreeLayout`, which prepares the layout for the AST graph.

■ 6.3.5 Modes and settings

Modes hold the current state of the application modes. These are the following:

- `toString` - whether the `TreeNode`s should be represented by their type, name, or both
- `classListHierarchical` - whether the class tree should be hierarchical or flat
- `searchDirect` - whether search should look for nodes suiting the criteria only among children of the search root or not

Package `settings` contains various constants like file locations, CSS style classes, error messages and currently applied modes.

Files contains locations where the files with recently opened file, list of filtered classes and pattern tree should be found.

Gui contains locations of FXML and CSS files and icons, names of CSS style classes and dimensions used by the tree layout.

Messages consist of strings with messages notifying the user of success or failure of their actions.

■ 6.4 Core application

In this section, we will describe functions of the core application, which can be found in `analyzer.core` package.

■ 6.4.1 Analyzer

Main class of the core application is `analyzer.core.Analyzer`, serving as a front controller receiving all requests from the GUI and delegating them to responsible services. Moreover, it holds objects storing the current state of the AST. Their composition is almost identical to the one proposed in figure 5.4, with the only difference being it contains list of test methods and both flat and hierarchical class tree at the same time.

■ 6.4.2 Settings management

The application saves the following data into files to make them available between the runs of the application. Those are:

- location of the last loaded file
- list of the filtered out classes
- Pattern tree

To write them into a file, **FileSaver** is used. If they are deleted, they will be created anew.

SettingsLoader loads the files on a new run of the application. If a non-existent type name appears in the filtered out classes list, it will be discarded. Similarly, **Pattern-Loader** loads the saved patterns in StringPrinter format:

```
---ROLE Class (name)
```

Number of dashes means depth of the node. The algorithm to assemble the tree is similar to the one employed by `GenericTreePrinter`.

As the node title can have one to three words, it is necessary to determine what the words mean:

- 3 words: role, class, name
- 1 word: class
- 2 words: either role and class or class and name

■ 6.4.3 Parsing

Parsing is provided by **Parser** in `analyzer.core.file` class. It uses `JavaParser` to parse a provided file or a string. In case the input is unparseable, it returns `null` instead of `CompilationUnit`.

Aside from initial loading of the code, parsing occurs in these situations too:

- after a `Node` is removed
- after the code is edited by the code editor

Reasons to this were elaborated on in 6.1.5.

■ 6.4.4 Tree building

Tree builders can be found in `analyzer.core.treebuilder`. When `CompilationUnit` is ready, a `NodeTreeNode` filtered AST has to be built by **FilteredTreeBuilder**.

Similarly to the printers, the Node tree is traversed and rebuilt with `NodeTreeNodes`, omitting filtered out nodes. It is built as a mirror image of the original and by passing through a printer later, it is mirrored again.

During the process of building the filtered AST, a list of node types appearing in the tree is acquired. Also, the `NodeTreeNodes` are assigned roles using the `CurrentRoleVisitor`.

The Node tree traversing algorithm uses these objects:

- Filtered classes
- a stack
- collection of already visited nodes
- collection of present nodes
- parent `NodeTreeNode`

It begins with inserting the `CompilationUnit` into the stack. It follows these steps:

1. Get reference to the last node in the stack.

2. If the node is in the visited collection, remove it from the collection, pop it from the stack, set parent to its parent and continue to the next loop.
3. Otherwise add the current node to the visited collection, create new `NodeTreeNode` (unless it is filtered out), detect its role and append it to the parent node.
4. Add children of the node to the stack.

After the filtered AST is built, it is time to build the class tree with **ClassTreeBuilder**. To do this, the **ClassMap** is needed. It retrieves the Node subtypes using the Reflections library and stores them in a map connecting their names with their class objects. This eliminates the need to instantiate the classes from a string, which can be problematic.

To build the flat class tree, we just loop through the classes in the `ClassMap` and append them as new `ClassTreeNodes` to the root, in case they are in the list of present nodes.

When building the hierarchical class tree, we need to first obtain a full hierarchical class tree, which will be then filtered to contain only appearing nodes. This is conducted by looping through the list of classes and by retrieving superclass of each node, consecutively assembling the tree. After that, we will traverse the tree, removing the nodes that do not appear in the code.

While the class trees are being built, each of the classes is checked for having roles as their children. These are appended to the class nodes with a `ClassTreeNode` containing the role and `Node` class. Roles are obtained by **RoleDictionary**, which returns roles after receiving name of the parent class, as in this case we cannot use a visitor without a `Node` instance.

■ 6.4.5 History

`analyzer.core.History` stores the current `CompilationUnit` after there are any changes in the AST. It manages the states by owning the following objects:

- current `CompilationUnit`
- stack of the previous states
- stack of the following states

Ways the class works can be summarized in the following points:

1. When a new state is added to history, the current one goes to the previous states, making the new state the current one and emptying the following states.
2. Until the undo operation is executed, stack of following states remains empty.
3. When undo happens, current state is placed in the following states and the latest state from the previous states becomes the current one.

■ 6.4.6 Node deleting

Before deleting, nodes have to be visited by `CheckDeletableVisitor`, which will determine whether they can be deleted without causing the code to be unparseable. If the node is deletable, it will be added to `Delete` set.

If `delete` method in `Analyzer` is called, all nodes in the `Delete` set are removed by `DeleteVisitor`.

■ 6.4.7 Searching

SearchService in `analyzer.core` is responsible for retrieving Nodes corresponding to the criteria given by the user, which can be the following:

- roots
- names
- roles
- classes

Roots are `NodeTreeNode`s from which the search should start.

One of the search approaches is **single class or role search**. The result list is filled with all the nodes of the given class or role. Classes are retrieved by `ASTHelper.getNodesByType` from `JavaParser`, roles by `RoleSearchVisitor`.

The other approach is **search by criteria**. If there are no roots set, the search will start from the AST root, `CompilationUnit`. There are two variations: one searches among direct descendants of the roots and the other among all descendants.

Search among children is conducted by simply checking if any of the children Nodes meets the given criteria. Search among all descendants depends on the filled out criteria, as according to them the search will be conducted differently:

- If there are any roles, `RoleSearchVisitor` will be used.
- If there are any names and no roles, `NameSearchVisitor` will be used.
- If there are no roles and no names, `ASTHelper.getNodesByType` will be used, as only classes will remain.

This sequence is based on the fact that all Nodes have a class, less of them have names, and even less nodes have roles.

In case a resulting node is currently filtered out, the first not filtered out ancestor will be provided in the results.

Note: It should be said that search results will contain only the nodes having the marked class and role at the same time, not both items of the class and of the role (intersection is used rather than union). This can be confusing as items of both categories appear in the same list, implying they might be of the same search category.

■ 6.4.8 Pattern management

PatternManager from `analyzer.core.pattern` adds and removes nodes from the Pattern tree. Each pattern node is allowed to have only one child.

In the case we are adding node to a parent that already has a child, the new node will become its parent. Similarly, node removal appends its children to its parent.

■ 6.4.9 Searching by patterns

This function is provided by **PatternSearchService** in `analyzer.core.pattern`. Arbitrary node in the Pattern tree can be marked for deletion by setting their `marked` attribute to `true`, resulting in found nodes being marked as such too.

The algorithm has three parts. The first part conducts a search by criteria for every node in the Pattern tree, using results of the previous level as roots. Results of each level are saved to be used in the next part.

Plain Node NodeTreeNode in the pattern tree means that between its parent and its child might be any number of nodes. Depending on whether such node appears in the sequence, direct search by criteria will be used or not.

The second part takes the results of the last level. For each Node in it, it looks for its ancestor in a higher level, until the whole found pattern is reconstructed. This is performed because results in the last level are already final, while the previous levels might contain nodes that would not make it to the final round. During this process, suitable nodes are marked for deletion. Output of this part is a collection of stacks, each one of them containing a found pattern.

The final part is to add the marked nodes to the Delete set. The result stacks are scanned through to find the marked nodes and each one of them tries to get into the Delete set. After this procedure, list of undeletable nodes is returned.

6.5 GUI elements

We can proceed to describing GUI elements and functions. In case of interest in the complete list of actions possible in the application, the reader should consult appendix B.

The whole application is run by `analyzer.App`, which reads FXML file of the main window, called stage, and opens it. This stage is managed by `Controller` class, connected to the FXML file directly. It can access elements from the file by `@FXML` annotation. It communicates with the Analyzer class, the core application and contains handlers for each control in the main window, making it the largest class in the project.

For better orientation, we will refer to elements of the layout by names in figure 6.6.

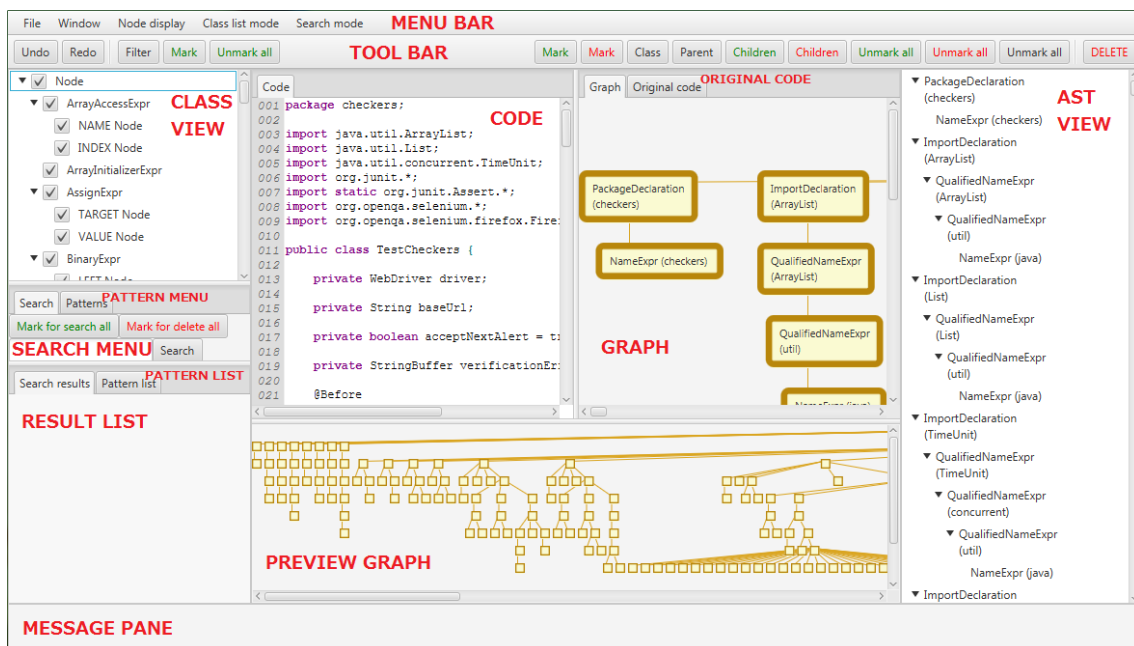


Figure 6.6. Application layout.

Controller manages connections of layout elements to their controllers in the initialization phase and handles refreshes of the AST. Similarly to the Analyzer class, it serves as a front controller, delegating input from the user to the Analyzer, processing its responses.

■ 6.5.1 Stages

Aside from the main stage, three additional stages can be opened. Their controllers can be found in `analyzer.gui.stage`. They use **StageInitializer** to set up the common properties.

GraphStageController is used to display the Graph in a separate window. This can be useful, as the graph can be very large.

TestStageController contains list of the test method and their statements. They are connected to their respective `NodeTreeNode`s.

EditStageController opens a code editor. After the user has finished editing, their changes are saved and the AST is refreshed.

■ 6.5.2 Tree views and Message pane

Tree views are AST view, Class view, Result list and Pattern list. Each of them has their own controller located in `analyzer.gui.treeview`. AST view is filled out by `NodeItemPrinter`, Class view by `CheckBoxItemPrinter` and Pattern list by `ClassItemPrinter`.

AST view has **AstChangeListener** as a handler of item selection, which enables connection between `GraphController`, `PreviewGraphController`, `MessageController` and `SelectionManager`. It selects graph nodes and part of the code corresponding to the selected AST node, writing its title in the Message pane, which is administered by **MessageController** in `analyzer.gui`. In certain cases, a message appears there to notify the user about success or failure of an operation. Failure notifications are red and appear for example when a Node cannot be deleted.

■ 6.5.3 Code areas

Code is managed by **CodeController** in `analyzer.gui.codearea`, which contains uneditable `RichTextFX CodeArea`. It handles recoloring of marked or selected parts of code. When a different node is selected or any node is unmarked, the whole area has to be restyled using the Search set, Delete set and Undeletable set.

The most remarkable function here is the one that chooses which part of code should be highlighted after a click on a line has been registered. It chooses item appearing on the position located the lowest possible in the AST, which is on the top of stack containing nodes on a given line, which was constructed by `NodeItemPrinter`.

To highlight Java keywords, **JavaKeywords** class from the `RichTextFX` GitHub page is used. This is employed by the editor in the separate stage too.

The reason for having two separate code editor is that if we edited the one linked to the AST, it would cause problems with highlighting, as for instance entire lines would be missing, disrupting the connection.

Aside from the main code area, there is Original code too. It is however not linked to the AST and serves only for reference. It is managed by **OriginalCodeController**.

■ 6.5.4 Graphs

Graph controllers are located in `analyzer.gui.graph`. There are two graphs: Graph and Preview graph, which serves as a possible shortcut between distant nodes.

They are managed by **GraphController** and **PreviewGraphController**, which both extend **AbstractGraphController**. The extending classes contain proper data of the two representations, as dimensions, and customized functions, as references to the graph nodes are saved as two different attributes of `TreeNode`.

We should mention that the abstract class makes it possible to scroll to the node when it is not currently visible in the viewport. This is achieved by obtaining coordinates of the node in the pane and dividing the x coordinate by width of the pane and the y coordinate by height.

Rendering of the graphs is provided by **GraphDrawer**. It receives layout data from `GraphPrinter` and uses them to draw the tree, stating from lines, which connect the nodes represented by buttons.

■ 6.5.5 Managers

SelectionManager from `analyzer.gui` serves for marking nodes. It uses **Marker** enumeration, which has values `SEARCH`, `DELETE` and `UNDELETABLE`. Undeletable nodes are stored in a set appearing in GUI only, as it serves to remember which nodes are marked as such, allowing for unmarking them. There are methods for marking and unmarking a single node or multiple nodes.

When a `TreeNode` is marked, it has to do the following:

- insert a corresponding icon next to the item in the `TreeView`
- add a style class to the Graph nodes
- highlight the code
- add it to the right set

Node removal is handled by **DeleteManager** like this:

1. If a `TreeNode` is to be marked for deletion, it sends the signal to the Analyzer.
2. Analyzer determines whether the inner Node can be deleted.
3. In case it is not possible, it gives the Node back. A message is produced to let the user know which node cannot be deleted, which is important especially in case of deletion of multiple nodes.
4. Otherwise it sends `null` back and the node is marked successfully.

This concludes the essential functions and implementation details of the application.

Chapter 7

Testing

This chapter will describe the testing process. Firstly, we will list the test configuration. After that, we will state the test strategy and then we will describe the individual testing methods, commenting on findings and related corrections.

Due to the nature of the tool, only the core application was tested programmatically. As the tool is a desktop application and not a web application how was originally intended, but was decided against for better integration with Java environment, it cannot be tested by Selenium WebDriver. Instead, with moderate application extent in mind, manual test scenarios were developed to find issues with the GUI.

7.1 Test configuration

Testing of the application, as well as the whole implementation process, were conducted using IntelliJ IDEA 15.0.5 IDE running inside of Windows 7.

The application uses the following versions of the required software:

- JDK 1.8.0_60
- JavaParser 2.4.0
- org.reflections 0.9.10
- RichTextFX 0.6.10
- abego TreeLayout 1.0.3
- JUnit 4.12

7.2 Test strategy

We placed main emphasis on static code analysis and unit testing, accompanied by integration and system testing. Tools used in this case were code analysis tools provided by IntelliJ IDEA IDE and JUnit framework.

As the tested application uses GUI, this part will be checked using manual testing and negative testing to enhance quality.

We will now present how the testing methods were conducted and what were their results.

7.2.1 Static code analysis

Static code analysis was used continuously during the development of the tool, as well as during testing. Code was analysed from the following points of view:

Dependency analysis made it easier to keep track of dependencies between classes, facilitating the process of minimizing dependencies.

Duplicates location was used to encounter code duplicities and remove them.

Code inspection helped with finding imperfections and possible bugs such as:

- which declarations can have weaker access
- which declarations can be final
- dead code
- duplicated code
- unused imports
- ignored method call results
- spelling errors

■ 7.2.2 Unit, integration and system testing

Unit testing focuses on the key functionality provided by the core application. The tests were designed mostly to check whether a valid input is handled in the expected way. In cases where invalid input may be entered, we checked whether it is ignored or produces corresponding feedback.

Only public methods were tested, as the private methods cannot be used in different context than with public methods that call them. If it was possible, we tried to maximize isolation of methods by providing mock objects instead of those acquired by methods dedicated to their creation.

We do not test `Parser` class, as it only retrieves `CompilationUnit` from `JavaParser`. Furthermore, `CompilationUnits` necessary for our tests will be provided by `Parser`, as creating mock ones would be very complicated, and more importantly needless, as we should not need to test an external library. Similarly, we will not test `ClassMap`, which retrieves node types, over which we have no control. It should be noted that tests relying on AST structure were constructed partially copying the results from the current state, as it would be hard to simulate it entirely.

Visitors are to be tested through methods utilizing them.

A set of functions to facilitate testing was implemented, providing file and input string handling, mock objects or comparison of `TreeNode` trees.

Tests for the following classes were created:

- `FileSaver` - compares string representations produced by it with expected strings.
- `SettingsLoader` - compares objects resulting from provided strings with expected objects. Provided strings contain invalid values, which are to be ignored by the methods.
- `PatternLoader` - is given an input string and the resulting pattern tree is compared against the expected one.
- `PatternManager` - checks creating and removal of pattern nodes. Focuses especially on the case when we add new node to a node that already has a child and when we remove a node that has a child, or in other words, when we have to insert or remove node surrounded from both sides.
- `PatternSearchService` - we provide `CompilationUnit` and result list according to our expectations and see whether the same patterns are found.
- `FilteredTreeBuilder` - to simplify definition of expected AST, a long list of filtered types is provided, minimizing the number of nodes. However, structure of the resulting AST relies heavily on the `JavaParser` grammar. To make up for this, we used the

actual result from the application, checked whether it makes sense and used it in the test, with hopes that the current state is correct. As we do not want to create mock Nodes, mock NodeTreeNodes do not contain Nodes and are compared with the real ones by name.

- **ClassTreeBuilder** - we provide list of present node types and expected class tree. For the hierarchical tree, we constructed the expected tree consulting the Node hierarchy (available in appendix D).
- **SearchService** - we provide CompilationUnit and expected result list. We focus on trying all possible combinations applicable to Search by criteria operation: direct and indirect search, names, roles, classes and root.

Integration tests are used for the same classes, but they test only methods that use mock objects, to see how the employed classes will interact.

Testing of **Analyzer** class, which is the main class of the core application, serving as a front controller, was conducted as system testing, as it is tied to the rest of the application. **History** class will be tested through it.

■ 7.2.3 Manual testing

Manual testing scenarios were designed to follow the use cases listed in 5.2. We will be referring to elements of the layout by names in figure 6.6.

Open file.

1. Run the application and choose a valid file to be opened.
2. Inspect if the source code and AST representations are rendered.

View AST + View code + Select node.

1. Click on an arbitrary node in the AST tree view.
2. Check whether the node is selected in both graphs and the corresponding code portion is highlighted.
3. Repeat the same procedure for every AST representation: graph, preview graph, code.

View list of tests.

1. Open a valid file containing JUnit test methods.
2. Click on **Tests** option in Window menu on the Menu bar.
3. Click on the items and check whether they are selected in every AST representation.

View class list + Filter out classes.

1. Uncheck arbitrary classes in the Class view.
2. Press Filter button in Tool bar.
3. Click on a filtered out class and confirm whether the Result list is empty.
4. In case of shorter code, inspect the AST visually.

Change display modes.

1. Select **class** option in Node display menu in the Menu bar.
2. Inspect whether the nodes do not have identifiers (there is nothing in brackets).

3. Select **name** option in Node display menu in the Menu bar.
4. Inspect if the nodes do not have node types (all visible titles are in brackets).

Mark or unmark node.

1. Select an arbitrary node and press Delete.
2. Check whether a red or black icon appears next to the node in AST view, corresponding code is red or black and the graph nodes are coloured red or black.
3. Press Delete while having selected the same node.
4. Check whether the node is unmarked.
5. Repeat the same with pressing Q and colour green.

Edit code.

1. Click on **Edit** option in Window menu on the Menu bar.
2. Change something in the code, keeping it valid.
3. Click on Done.
4. Check whether the changes appear in the code in the main window.

Delete node.

1. Select an arbitrary node.
2. Press Delete.
3. If the node is coloured black, select another one until it is coloured red.
4. Press Backspace.
5. Inspect the code to see whether code corresponding to the deleted node is missing.

Undo or redo.

1. Delete a node like in the previous scenario.
2. Click on Undo button in the Tool bar.
3. Inspect the code and the AST to see whether the deleted node has returned.
4. Click on Redo button in the Tool bar.
5. Inspect the code and the AST to see whether the deleted node was deleted again.

Save file.

1. Click on Save option in the File menu in the Menu bar.
2. Write an arbitrary name and click on Save.
3. Go to the location of file, open it and check if its code is the same as the one opened in the application.

Search by criteria.

1. Choose a name (in brackets) from the AST view.
2. Write it into input field in Search menu.
3. Click on Search.
4. Click on the results to see if they point to AST nodes with the same name.
5. Select the parent of one of the found nodes.
6. Press Q.
7. Click on Search button in Search menu.
8. Inspect whether only descendants of the marked node are in the Result list, and have the given name at the same time.

9. Select type of the selected AST node in the Class view.
10. Press Q.
11. Click on Search button in Search menu.
12. Check if the node is in the Result list, accompanied by prospective nodes with the same name, type and ancestor.

Create search patterns.

1. Select an arbitrary item in Class view.
2. Click on Add pattern in Pattern menu.
3. Observe whether a new item of the selected type appeared in the Pattern list.

Add node to the pattern tree.

1. Select a leaf node in the pattern tree.
2. Select an arbitrary item in Class view.
3. Click on Add child in Pattern menu.
4. Check if a new item was added.
5. Select parent of the new node.
6. Select a different item in Class view.
7. Click on Add child in Pattern menu.
8. Check if the newly added node became parent of the previously added child.

Remove node from the pattern tree.

1. Select a leaf node in the pattern tree.
2. Click on Remove in Pattern menu.
3. Check if the node was removed.
4. Select another node which is not a leaf node or root of a pattern.
5. Click on Remove in Pattern menu.
6. Check if the child of the removed node is now child of its parent.

7.2.4 Negative testing

Negative testing explores what happens, if the tested application receives invalid input. We performed a series of test to observe behaviour in such cases.

First we checked how the application handled invalid file inputs and files with a large number of lines. Parameters of the test can be observed in table 7.1. We measured how long processing and rendering of the AST took.

test	number of lines	contains Java	Java extension
1	0	no	no
2	1566	no	yes
3	1000	yes	yes
4	1500	yes	yes
5	2000	yes	yes

Table 7.1. Parameters of negative testing of file inputs.

Results of testing file inputs described in table 7.1:

1. Parsed normally (CompilationUnit with no children).

2. Parsing error.
3. Parsed normally after 15 seconds.
4. Parsed normally after 30 seconds.
5. Parsed normally after 60 seconds.

From these results, we can observe that the application does not consider file extension. As 60 seconds were deemed to be too long for loading, 1500 lines limit was added to the application.

In the next stage, the application was tested for unexpected mouse inputs by clicking on every possible spot. An issue found by this was that clicking on empty space in any `TreeView` resulted in `NullPointerException`, as associated methods did not receive any input. This issue was fixed by checking if there is really any item selected. Another similar case happened with clicking on Remove button in pattern menu if the pattern list was empty.

7.3 Test results

We will now summarize test results by presenting their findings. Many issues were encountered and fixed during the implementation phase, meaning there were not many critical errors to be found by the testing.

We will now list the most important findings:

- Trying to access parent of the root node resulted in `NullPointerException` in several places. A condition was added to check whether parent of the node is `null`
- `SettingsLoader` and `FileSaver` were changed not to access file locations by a static field, to allow testing with different files and avoid rewriting data in them.
- Searching for a role in an AST in which the node with the given role was filtered out resulted in the result list containing a `null` `TreeNode`. This was fixed by removing `null` values from the list.
- When searching by patterns, `CompilationUnit` would sometimes appear in the set of search roots. This was caused by the node being inserted into the root set, even if there were other roots.
- Saved file was missing `.java` extension.
- Larger files might take unbearably long to be processed. A condition was added to accept only source code with less than 1500 lines.
- Clicking on empty space in `TreeViews` resulted in `NullPointerException`. A condition was added to check whether selected item is not `null`.

Chapter 8

Conclusion

To conclude our work, we should summarize what we have achieved in each part of the thesis and revise the goals we have stated in the beginning, evaluating whether and how we have accomplished them.

We began our journey by studying theory concerning lexical and syntax analysis in chapter 3. We learned about the purpose of parsing in the world of programming languages as a tool to translate source code in a human-friendly programming language into machine code comprehensible by computers. To become more familiar with the process, we demonstrated similarities between parsing of natural and programming languages, realizing they are not that much different. We picked up how an abstract syntax tree (AST) is built.

As we needed to employ AST generation in our tool, our task in chapter 4 was to explore existing parsers that enable parsing of Java language and evaluate, according to our criteria, whether they are convenient for our usage. The best candidates were ANTLR and JavaParser, however, ANTLR does not support AST modification, which was one of our requirements. This resulted in favouring JavaParser, even though it does not provide AST visualization. We tried to use ANTLR for generating visualization of AST parsed by JavaParser, which lead to the discovery that the two programs generate AST differently. Thus, we had to implement our own visualization function.

Now that we had an AST provider, in chapter 5 we could revise our requirements and derive a set of use cases we wished our application to have. Then we decided the application should be divided into two main components, core and GUI, and listed what data structures they should contain. We also established external libraries the application should employ: the aforementioned JavaParser for the AST generation, Reflections for retrieval of node types in the AST, JavaFX for the GUI, abego TreeLayout for the layout of AST visualization and RichTextFX for code editor.

In chapter 6, having decided on how the application should be structured and its functions, it was finally time to start implementing. As the first step, we became familiar with functions of JavaParser, to determine how we could use it to its maximum potential. Aside from that, we found out about a few imperfections, which our application would have to improve. With this knowledge in mind, we started implementing our tool. At first, we introduced data structures and function providers used across the application. Then we described the core, starting from the process of building a new tree and continuing with the remaining functions. Next step was to describe the GUI, concerning which we described its elements and functions.

When our application was finished, in chapter 7 we proceeded to testing. We mentioned configuration used when testing the application and elaborated on our test strategy. Then we presented manual and negative testing scenarios and listed the most important findings the testing brought.

After this reminiscence, we shall revise how we fulfilled the requirements. Our tool performs lexical and syntax analysis of an arbitrary Java code and produces AST, which is then visualized in two representations, allowing for more flexibility while analysing the nodes. Furthermore, orientation is enhanced by providing links between the nodes and their respective areas in code. It is possible to filter out unnecessary nodes to make the viewed AST lighter. In case of JUnit code, we can display list of only the test methods separately, maintaining connection to the main AST. We can easily remove nodes and receive immediate feedback if it is not possible, and carry out further modifications by directly editing the code. Finally, we can search in the AST either by criteria or patterns, permitting bulk edits of the resulting nodes.

The intended purpose of the resulting tool was to aid testers with code processing, which should be satisfied, as thanks to the implemented functionality, the tool helps to achieve faster analysis and modification of code.

Event though the resulting application fulfilled the proposed requirements, the question whether the tool will meet the expectations placed on it remains unanswered. After the tool is being used for some time in practice, new proposals might appear, possibly changing concept of the tool in a certain degree. This is however not a bad thing, as it is always important to be open to changes.

8.1 Future work

As the final part of the thesis, we will propose a few ideas on how to make our application more capable in the future.

Offer parsing of more programming languages. We gave up on potential versatility, which could have been achieved by ANTLR, in favour of AST modification. As the required language to be processed was Java, it was not an issue. However, for the future it would be a great improvement and it would make the tool even more beneficial.

Support more complex patterns. Currently, the application allows only patterns in which each node can only have one child. Allowing addition of siblings would expand usability of this function. However, it would be better to first think over whether this function would be really needed.



References

- [1] FEWSTER, Mark and Dorothy GRAHAM. *Software Test Automation: Effective Use of Test Execution Tools*. 1st edition. Addison-Wesley Professional, 1999. ISBN 978-0201331400.
- [2] MOGENSEN, Torben. *Introduction to Compiler Design*. 1st edition. Springer-Verlag London, 2011. ISBN 978-0-85729-828-7.
- [3] MAK, Ronald. *Writing Compilers and Interpreters: A Software Engineering Approach*. 3rd edition. Wiley Publishing, 2009. ISBN 978-0-470-17707-5.
- [4] AHO, Alfred. V., Monica S. LAM, Ravi SETHI and Jeffrey D. ULLMAN. *Compilers: Principles, Techniques and Tools*. 2nd edition. Addison Wesley, 2006. ISBN 0-321-48681-1.
- [5] WATT, David A. and D. F. BROWN. *Programming Language Processors in Java: Compilers and Interpreters*. 1st edition. Pearson, 2000. ISBN 0-130-25786-9.
- [6] BIRD, Steven, Ewan KLEIN and Edward LOPER. *Natural Language Processing with Python*.
<http://www.nltk.org/book/>.
- [7] GOSLING, James, Bill JOY, Guy STEELE, Gilad BRACHA and Alex BUCKLEY. *The Java® Language Specification: Java SE 8 Edition*. 2015.
<http://docs.oracle.com/javase/specs/jls/se8/html/index.html>.
- [8] SCHILDT, Herbert. *Java: The Complete Reference, Ninth Edition*. 9th edition. McGraw-Hill Education, 2014. ISBN 978-0-071-80855-2.



Appendix A

CD content

Note: preferred way of obtaining external libraries is via Maven (`pom.xml` file is in `analyzer` folder). In case of not using Maven, `.jar` files can be imported instead from folder `libraries`.

- `thesis.pdf` - PDF version of the thesis
- `assignment.jpg` - scanned assignment
- `example.java` - Java source code that may be used as input
- `thesis/` - folder containing source code of the text of the thesis
- `analyzer/` - project file, contains source files, tests and settings files
- `libraries/` - external libraries employed by the program

Appendix B

User manual

This section provides brief introduction to how to control the application. We will be referring to elements of the layout by names in figure 6.6.

Open and save file. Menu bar - Open file/Save file. If file selected to be opened is invalid, a message will appear in Message pane.

Undo and redo. Tool bar - Undo/redo.

Select a node. Click on a node in either AST view, Graph view or Preview graph view. A node will be selected by clicking on the corresponding place in the code too. Message pane will display name of the selected node.

Mark selected node. Tool bar - Mark (right) (green for search, red for delete) or press Q (search) or Delete (delete). Nodes marked for search will be used as search roots. If a node marked for delete cannot be deleted, a message will appear in Message pane.

Select class of the selected node in the class view. Tool bar - Class

Go to parent of the selected node. Tool bar - Parent

Mark children of the selected node. Tool bar - Children (green for search, red for delete).

Unmark all marked nodes. Tool bar - Unmark all (right) (green for search, red for delete, black for undeletable).

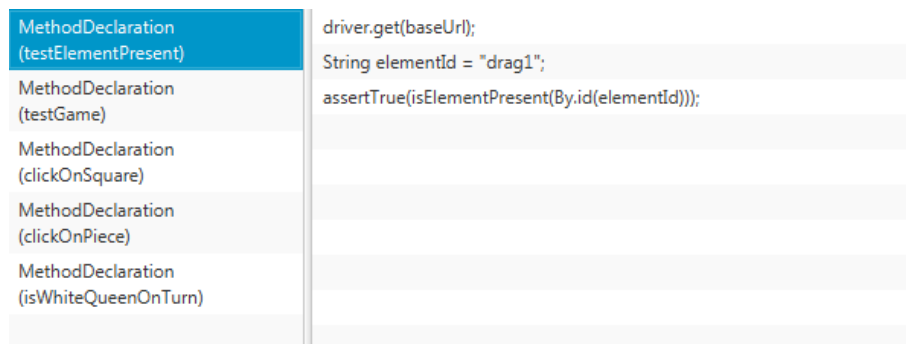
Delete nodes marked for delete. Tool bar - DELETE or press Backspace. A message informing of result of the operation will appear in Message pane.

Change node display mode. Menu bar - Node display.

Change class view mode. Menu bar - Class list mode.

View original code. Open Original code tab.

View graph in separate window. Menu bar - Window - Graph. Nodes will remain connected.



MethodDeclaration (testElementPresent)	driver.get(baseUrl);
MethodDeclaration (testGame)	String elementId = "drag1";
MethodDeclaration (clickOnSquare)	assertTrue(isElementPresent(By.id(elementId)));
MethodDeclaration (clickOnPiece)	
MethodDeclaration (isWhiteQueenOnTurn)	

Figure B.1. Test window.

View list of test methods. Menu bar - Window - Tests. List items are connected to the AST. List of test methods can be seen in figure B.1.

Edit code. Menu bar - Window - Edit. The main window will be disabled during editing.

Filter out nodes. Uncheck the node types to be filtered in Class view and press Tool bar - Filter.

Get list of all nodes of a type or role. Click on the type in the Class view, results will appear in Result list.

Select a node from the result list in AST. Click on the result item.

Search for nodes with a given name. Type the name into Search menu - input field and press Search. Multiple values are separated by comma.

Search for nodes of a given type or role. Select the type in Class view and press Tool bar - Mark (left) or press Q, then press Search. Multiple types can be selected.

Change search mode. Menu bar - Search mode.

Unmark all marked node types. Tool bar - Unmark all (left).

Mark all results. Search menu - Mark for search all/Mark for delete all.

Pattern menu and Pattern list can be seen in figure B.2

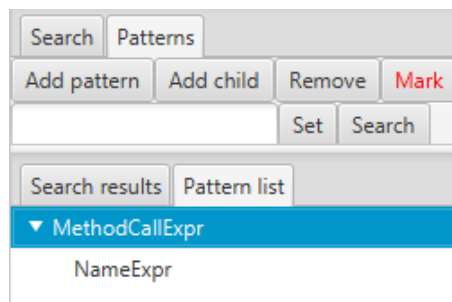


Figure B.2. Pattern menu.

Add new pattern. Pattern menu - Add pattern.

Add child to a pattern node. Select parent in the Pattern list and press Pattern menu - Add child. Child of Node type means its child does not have to be direct descendant of its parent if found in the AST.

Remove pattern node. Select node in the Pattern list and press Pattern menu - Remove.

Mark pattern node for delete. Select node in the Pattern list and press Pattern menu - Mark or press Delete.

Set name of pattern node. Type the name into Pattern menu - input field and press Set.

Search by pattern. Select root of the pattern in Pattern list and press Search.

Appendix C

Productions used in syntax analysis example

This appendix contains list of productions used in the example in 3.4.

```
CompilationUnit:
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration

ClassDeclaration:
  NormalClassDeclaration
  EnumDeclaration

NormalClassDeclaration:
  {ClassModifier} class Identifier [TypeParameters] [Superclass]
    [Superinterfaces] ClassBody

ClassModifier:
  (one of)
  Annotation public protected private
  abstract static final strictfp

Identifier:
  IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

ClassBody:
  { {ClassBodyDeclaration} }

ClassBodyDeclaration:
  ClassMemberDeclaration
  InstanceInitializer
  StaticInitializer
  ConstructorDeclaration

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;

MethodDeclaration:
  {MethodModifier} MethodHeader MethodBody
```



```

MethodModifier:
(one of)
Annotation public protected private
abstract static final synchronized native strictfp

MethodHeader:
Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
UnannType
void

MethodDeclarator:
Identifier ( [FormalParameterList] ) [Dims]

FormalParameterList:
ReceiverParameter
FormalParameters , LastFormalParameter
LastFormalParameter

FormalParameters:
FormalParameter {, FormalParameter}
ReceiverParameter {, FormalParameter}

FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
boolean

NumericType:
IntegralType
FloatingPointType

IntegralType:
(one of)
byte short int long char

VariableDeclaratorId:
Identifier [Dims]

MethodBody:
Block
;

Block:
{ [BlockStatements] }

```

Appendix D

List of JavaParser node types

```
Node
-BaseParameter
--MultiTypeParameter
--Parameter
-BodyDeclaration
--AnnotationMemberDeclaration
--ConstructorDeclaration
--EmptyMemberDeclaration
--EnumConstantDeclaration
--FieldDeclaration
--InitializerDeclaration
--MethodDeclaration
--TypeDeclaration
--TypeDeclaration
---AnnotationDeclaration
---ClassOrInterfaceDeclaration
---EmptyTypeDeclaration
---EnumDeclaration
-Comment
--BlockComment
--JavadocComment
--LineComment
-Expression
--AnnotationExpr
---MarkerAnnotationExpr
---NormalAnnotationExpr
---SingleMemberAnnotationExpr
--ArrayAccessExpr
--ArrayCreationExpr
--ArrayInitializerExpr
--AssignExpr
--BinaryExpr
--CastExpr
--ClassExpr
--ConditionalExpr
--EnclosedExpr
--InstanceOfExpr
--LambdaExpr
--LiteralExpr
---BooleanLiteralExpr
---NullLiteralExpr
---StringLiteralExpr
----DoubleLiteralExpr
----LongLiteralExpr
```

```
-----IntegerLiteralExpr
-----IntegerLiteralMinValueExpr
----CharLiteralExpr
----LongLiteralExpr
-----LongLiteralMinValueExpr
--MethodCallExpr
--MethodReferenceExpr
--NameExpr
---QualifiedNameExpr
--ObjectCreationExpr
--SuperExpr
--ThisExpr
--TypeExpr
--UnaryExpr
--VariableDeclarationExpr
-ImportDeclaration
-MemberValuePair
-PackageDeclaration
-Statement
--AssertStmt
--BlockStmt
--BreakStmt
--ContinueStmt
--DoStmt
--EmptyStmt
--ExplicitConstructorInvocationStmt
--ExpressionStmt
--ForStmt
--ForeachStmt
--IfStmt
--LabeledStmt
--ReturnStmt
--SwitchEntryStmt
--SwitchStmt
--SynchronizedStmt
--ThrowStmt
--TryStmt
--TypeDeclarationStmt
--WhileStmt
-Type
--ClassOrInterfaceType
--PrimitiveType
--ReferenceType
--UnknownType
--VoidType
--WildcardType
-TypeParameter
-VariableDeclarator
-VariableDeclaratorId
```

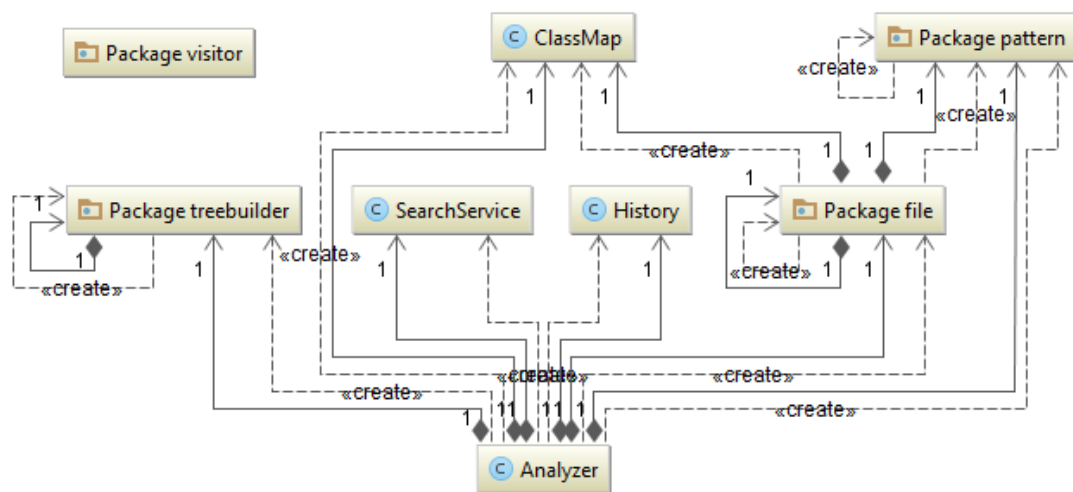
Appendix E

Class diagrams by packages

All packages are contained in **analyzer** package, which contains the following hierarchy. Each package name is accompanied by figure in which its contents and their dependencies can be seen.

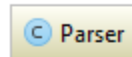
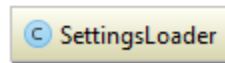
- core E.3
 - file E.4
 - pattern E.5
 - treebuilder E.6
 - visitor E.7
 - delete
 - name
 - role
- gui E.8
 - codearea E.9
 - graph E.10
 - stage E.11
 - treeview E.12
- node E.13
- printer E.14
- settings E.15

Now we will show diagrams of each package with dependencies.



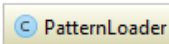
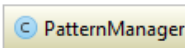
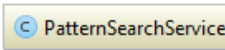
Powered by yFiles

Figure E.3. Core package.



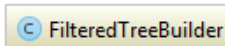
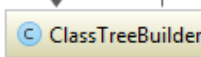
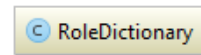
Powered by yFiles

Figure E.4. File package.



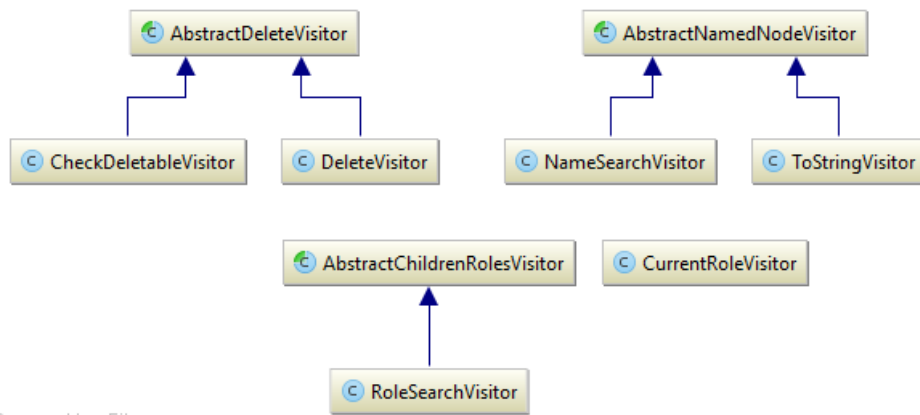
Powered by yFiles

Figure E.5. Pattern package.



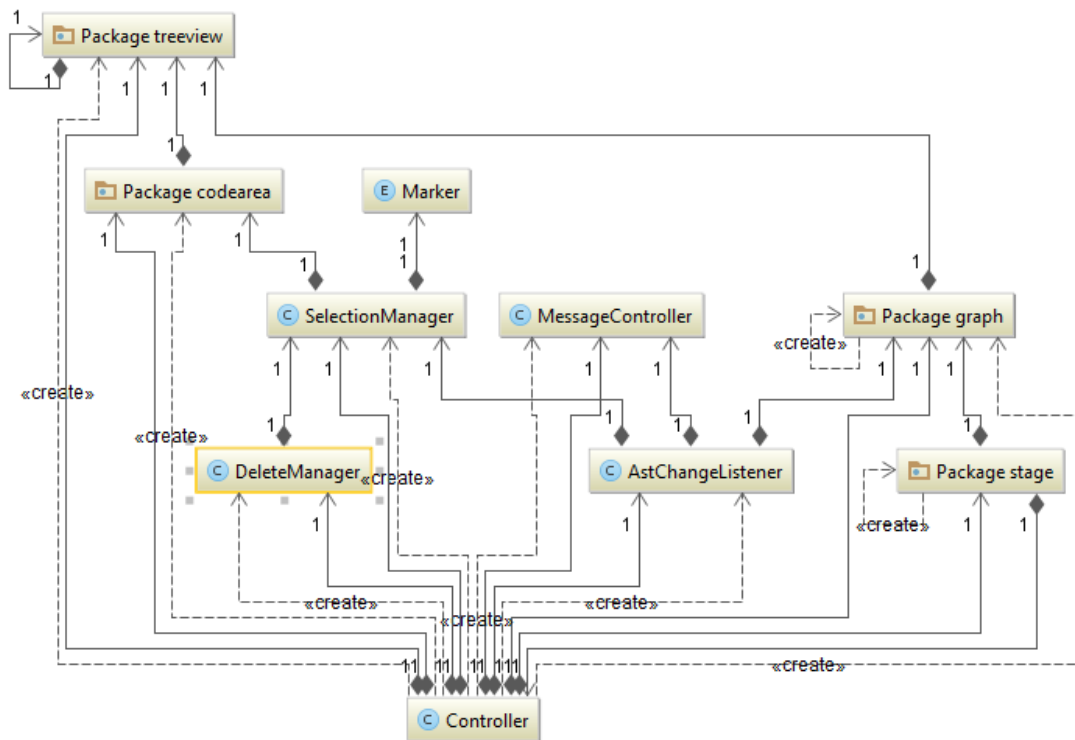
Powered by yFiles

Figure E.6. Treebuilder package.



Powered by yFiles

Figure E.7. Visitor package.



Powered by yFiles

Figure E.8. Gui package.



Figure E.9. Codearea package.

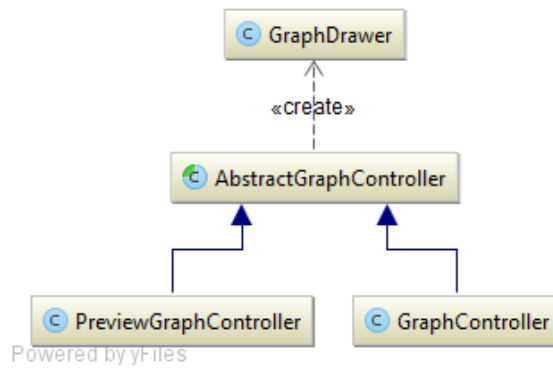


Figure E.10. Graph package.

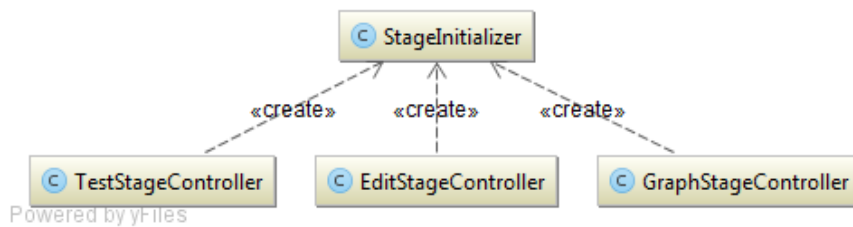


Figure E.11. Stage package.

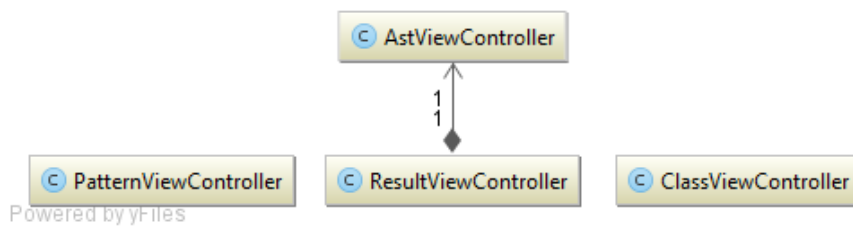


Figure E.12. Treeview package.

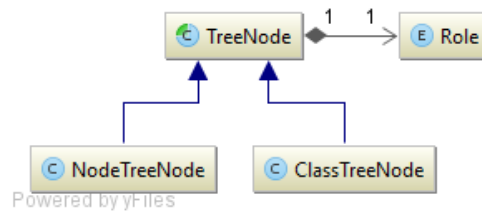


Figure E.13. Node package.

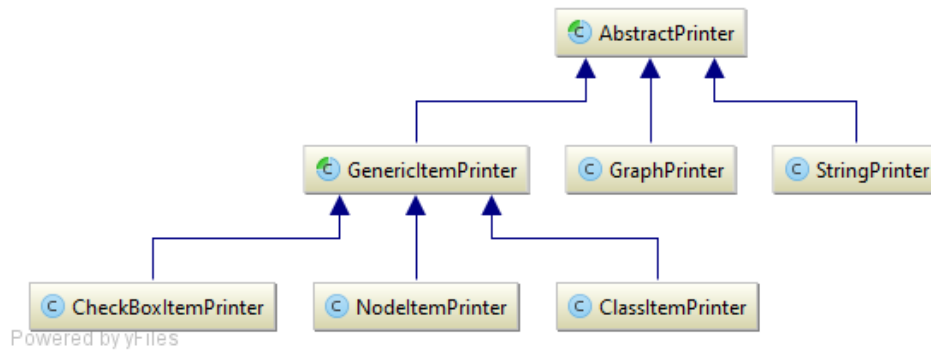


Figure E.14. Printer package.

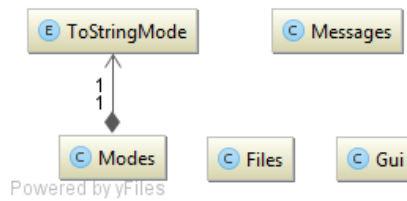


Figure E.15. Settings package.