Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate DiSTA

Corso di Laurea Triennale in Informatica

The Development of a Tool to Analyse Java Code Quality

Relatore: Prof. Sandro Morasca

Autore dell'elaborato:

Dario Bertolino

Matricola: 724118

Anno Accademico 2016 - 2017

Contents

1 Introduction

The aim of this work is the development of a java software tool that can recognise and compute software engineering metrics from java code. These values are useful to estimate software quality and they can refer how the code is written syntactically or the software structure as well.
Accurate measurement is a prerequisite for all engineering disciplines, and software engineering is not an exception. For decades, engineers and researchers have sought to express software features with numbers, in order to facilitate software quality assessment. A large body of software quality metrics have been developed, and numerous tools exist to collect metrics from program representations, e.g. VizzAnalyser, NTools, JHawk, SonarQube, sciTool. This large variety of tools allows a user to select the best suited one, e.g., depending on its handling, tool support, or price. However, this assumes that all metrics tools compute / interpret / implement the same metrics in the same way. From a practical point of view, engineers and managers should be aware that the metrics tool measurements do not necessarily follow the reasoning and intention of those who defined the metrics. For instance, the focus on maintenance of critical classes, where critical is defined with a metrics based assessment would be relative to the metrics tool used. What would be the right decision then? Scientifically, the validation or relevance of certain metrics is still an open issue. Controlled experiments involving metrics-based and manual assessment of a variety of real world software systems are costly. However, such a validation would then not support/reject the validity of a software metrics set but rather the validity of the software metrics tool used. Thus, the results of such an experiment cannot be compared or generalized and the costs could not be justified. Software engineering practitioners - architects, developers, managers - must be able to rely on scientific results. Especially research results on software quality engineering and metrics should be reliable. They are used during forward engineering, to take early measures if parts of a system deviate from the given quality specifications, or during maintenance, to predict effort for maintenance activities and to identify parts of a system needing attention. To provide reliable scientific results, quite some research has been conducted in the area of software metrics. Some of the metrics have been discussed and reasoned about for years, but only few metrics have even been validated experimentally to have correlations with certain software qualities. Moreover, software engineering practitioners should be able to rely on the tools implementing these metrics, to support them in quality assessment and assurance tasks, to allow to quantify software quality, and to deliver the information needed as input for their decision making and engineering processes. In

order to rest on the scientific discussions and validations, i.e., to safely apply the results and to use them in practice, it would be necessary that all metrics tools implement the suggested metrics the way they have been validated. We know that metrics tools deliver different results given the same input and at least some tools do not implement the metrics as intended. For rather simple metrics, like the Number of Children (NOC), most tools compute the same or very similar results. For other metrics, e.g., the Coupling Between object Classes (CBO) or Lack of Cohesion of Methods (LCOM), the results present a much bigger variation. from a practical point of view, software engineers need to be aware that the metrics results are tool dependent, and that these differences change the advice the results imply. Especially, metrics based results cannot be compared when using different metrics tools. From a scientific point of view, validations of software metrics turn out to be even more difficult. Since metrics results are strongly dependent on the implementing tools, a validation only supports the applicability of some metrics as implemented by a certain tool. More effort would be needed in specifying the metrics and the measurement process to make the results comparable and generalizable.

1.1 Organization of the Thesis

In the Chapter **??** we will talk about the theory of automata and formal languages explaining what are grammars, non-deterministic pushdown automatons and Parsing trees. These concepts are fundamental to understand a front-end compilation process and consequently how a code analysis is done. Section **??** is really important describing the Lexer and the Parser which are compiler's components that make up the main core of the created software. Chapter **??** is dedicated to ANTLR4, a tool that can generate a Lexer and Parser from a programming language grammar. These generated components offer mechanisms to extract information from the compilation process like parse tree listeners and visitors. Since our scope was to extract specific data from a compilation process, ANTLR4 was used to generate a Lexer and a Parser from an open source Java 8 grammar obtaining a solid base to develop the software. Thanks to this useful tool most part of the work was dedicated to find algorithms to calculate not only simple metrics like lines of code and number of methods, but also more complex ones like McCabe's cyclomatic complexity, depth of Inheritance Tree or Coupling between objects. Chapter **??** is dedicated to the project design explaining which elements are involved in the developed execution model and describing the software structure which support multithreading. Every .java file that is part of the analysed code has a dedicated thread which is responsible for its compilation and extraction of data. Taking advantage of the Barrier

design pattern once all launched threads has finished, the collected data are used to calculate the desired metrics. Chapter **??**, which is the most important one, describes all metrics and the way they are calculated. In Chapter **??** we will also discuss the possibility to extend the software implementing new metrics calculation or other ways to show the results. Others programming languages could be analysed with a software created in the same way since only an ANTLR4 grammar is requested and the open source world offers many.

2 Code Analysis

2.1 Grammars

A Grammar G is defined by a quadruple

$$G =< V, T, P, S > \tag{1}$$

In which:

- V represents the non-terminal symbols set. (variables, syntactic categories)
- T the terminal symbols set. (Sentence's components)
- P the set of production rules.
- S the non-terminal initial symbol, which marks the beginning of a sentence construction.

A production rule is a fundamental function to determine the grammar type. If a grammar contains production rules like (T U V)* → (V U T)* , the grammar is context-sensitive, but if it contains production rules like V → (V U T)* , the grammar is context-free.
For example, B is a non-terminal symbol, 0 and 1 are terminal symbols for a binary language:

- 0B → 01 and 1B → 11 are context-sensitive production rules because B becomes 01 or 11 based on what precede it, a 0 or a 1. So it is based on the context in which B is located.
- B → B0 and B → 0 are context-free production rules.

In 1956, Noam Chomsky described a grammars hierarchy:

Grammar	**Language**	**Automaton**	**Production rules**
Type 0	Recursively-enumerable	Turing Machine	No restrictions
Type 1	Context-sensitive	Linear-Bounded	(T U V)* → (V U T)*
Type 2	Context-free	Non-deterministic pushdown automaton	V → (V U T)*
Type 3	Regular	Finite state automaton	V → (V)T

Every regular language is context-free, every context-free language is context-sensitive, every context-sensitive language is recursive and every recursive language is recursively enumerable. These

are all proper inclusions, meaning that there exist recursively enumerable languages that are not context-sensitive, context-sensitive languages that are not context-free and context-free languages that are not regular. In order to understand what is a recursively-enumerable language we should talk about Turing machines and the problem of indecisiveness. However we are not interested in such arguments.

2.2 Context-free Languages, PDAs & Parsing trees

For the scope of this work, we are interested in context-free languages, as most programming languages are, and so far we know that they are recognised by non-deterministic pushdown automata (PDA).

A PDA is a non-deterministic finite state machine with a memory stack available.

In the Theory of Automata & Formal Languages an automaton runs on some given sequence of inputs composed by symbols from an input alphabet. A finite sequence of these symbols is called a word. At any given time, an automaton is in a state, belonging to a finite set of states. At each time step when the automaton reads a symbol, it transitions to another state that is decided by a function that takes the current state and symbol as parameters.

A PDA transition function also has another parameter, a symbol from a stack alphabet and at each step in addition to the state change, also the stack is modified.

A non-deterministic pushdown automaton is created by seven elements:

$$P =< Q, \Sigma, \Gamma, \delta, q_0, Z_0, F > \tag{2}$$

In which:

- Q is for states set.
- Σ is for input alphabet.
- Γ is for stack alphabet.
- δ is for the transition function: (Q x Σ x Γ) $\rightarrow$ (Q x Γ^*),
  in which an input modify the automaton state and stack.
- q_0 is for initial state.
- Z_0 is for initial symbol of the stack.
- F is for final states set.

Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines. Non-deterministic pushdown automata can recognize all context-free languages. Saying that an automaton accepts an input it means that it has recognized it as a word or construction of a specific context-free language. A PDA accepts its inputs in two different ways. The former is when a computation reaches a final state and the latter is when a computation reaches a situation of empty stack. For a given language A, final state acceptation and empty stack acceptation are not the same, but for language accepted for final state always exists an equivalent one accepted for empty stack.

Why is this important? We are going to define a Parsing Tree and we will see how the visit of this tree is the key to check the correctness of programs statements. A program statement is the equivalent of a language phrase and every time the compiler recognises one, a part of the Parsing Tree corresponding the program is generated. A Grammar is ambiguous if a phrase of the language can be generated by two Parsing trees.

A Parsing Tree is a data structure, which meets the following conditions for a given Grammar:

- Every inner node has a label with a non-terminal symbol in V.
- Every leaf has a label with a non-terminal symbol, a terminal symbol or an empty leaf. In the last case it has to be the only son of the father node.
- When a node has a label with a non-terminal symbol A, if we read his sons from left to right, the passage from A to the sons set is a production role. (A = s1, s2, . . . , Sn)

Figure **??** contains an example of how a java instruction is trasformed in a parsing tree.

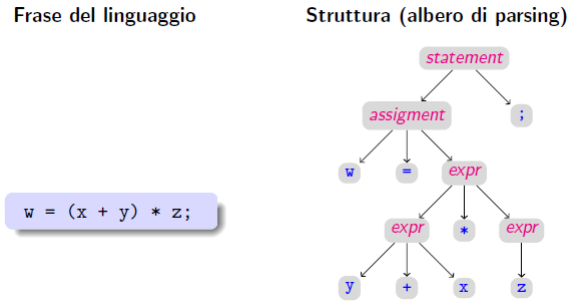


Figure 1: A simple parsing tree

2.3 Compilation Process

In a compilation process a component called Reader carries out code analysis. The work is done by two sub-components called Lexer and Parser, producing an intermediate representation (IR). Back-end Generators convert the IR in assembly language or, for example, in bytecode for java. In figure **??** a simple execution flow through the compiler components is shown.

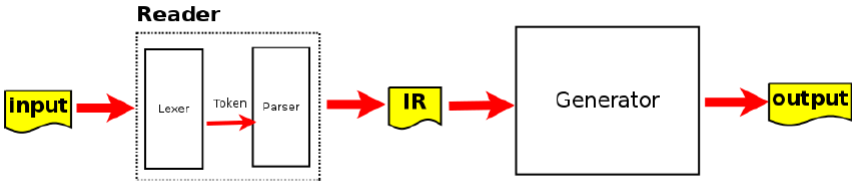


Figure 2: Compiler components

As explained in figure **??** a character stream is received in input by the Lexer, which recognises different elements of the language and organises them in Tokens (aggregated characters). Then the created token stream is analysed by the Parser, whose task is to arrange phrases in Syntax trees, or Parsing trees. So, Lexer finds words while ignoring comments, spaces, blank lines and creates tokens for key words, identifiers, symbols and operators of a desired language. On the other hand, Parser controls phrases from a syntactic point of view, following Grammar rules.

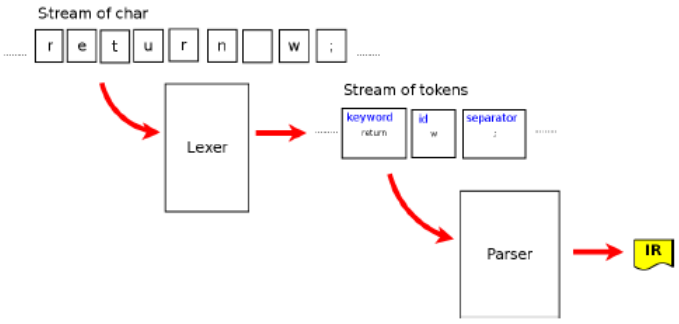


Figure 3: Lexer and Parser functions

Reader analyses code in two phases:

- Check of **syntactic rules**.
- Check of **static Semantic**, rules that cannot be defined in a grammar.

To understand the process of recognising static semantics, we need to consider two new elements of the Reader, the **Symbol Table** and the **Semantic Analyzer**. Figure **??** how these components are related to the Lexer and the Parser.

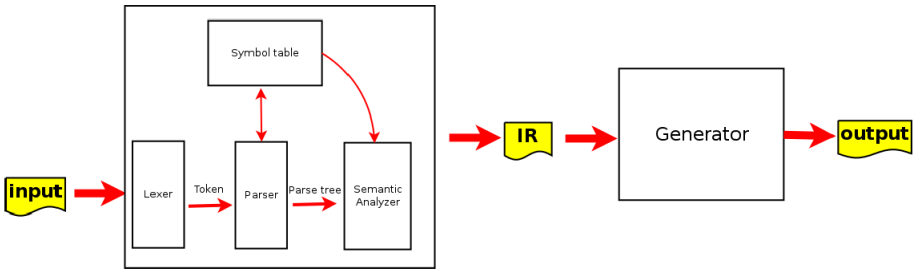
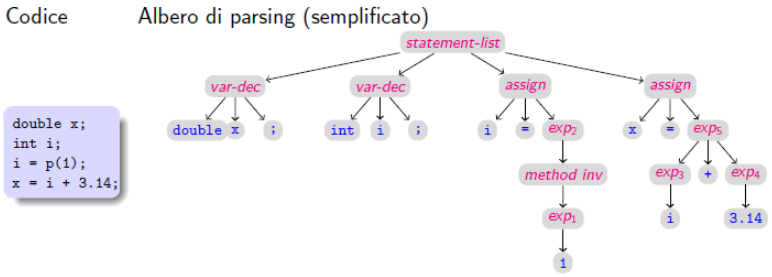


Figure 4: Symble table and semantic analyser

The Symbol Table is generated by the Parser to store variables, methods and their types. Once a Parsing tree is generated, it is used as an input for the Semantic Analyser, whose task is to annotate it for example with expression's types or needed conversions during the execution. This amount of information is offered by the Symbol Table. In figure **??** we can observe a simple example of how this process works for four simple instructions in Java language.



Symbol table

Token	Tipo	Altre informazioni
p	metodo	proptotipo int → int
...	...	...
x	var	tipo double
i	var	tipo int

Annotazioni albero di parsing

exp_1	tipo int
exp_2	tipo int
exp_3	tipo int, conversione a double
exp_4	tipo double
exp_5	tipo double

Figure 5: Simple example for Symbol Table and Semantic Analyzer

After the annotation process of the Parsing tree, Intermediate Representation is ready to be analysed by a Generator. Some compilers also provide a middle-end component, which optimizes the Parsing tree. In this work, we are not interested in the Back-end function of a compiler and furthermore it would be really difficult to carry out a general treatise about it because it is usually created for precise goals like generation of bytecode for JVM. Tools for generating Parsers are divided into two main families. The former generates parser top-down or recursive descendent parser, like ANTLR4 and the latter generates parser bottom-up, like Javacc and Yacc.

2.4 Recursive Descendent Parsers

The Parser's task is to verify phrase correctness by creating a Parsing tree, which is checked for acceptance in a process that is highly similar to PDA's acceptance for empty stack. A Recursive descendent parser, as the name suggests, verifies the acceptance through a top-down visit of the Parsing Tree. Figure **??** contains a simple example.

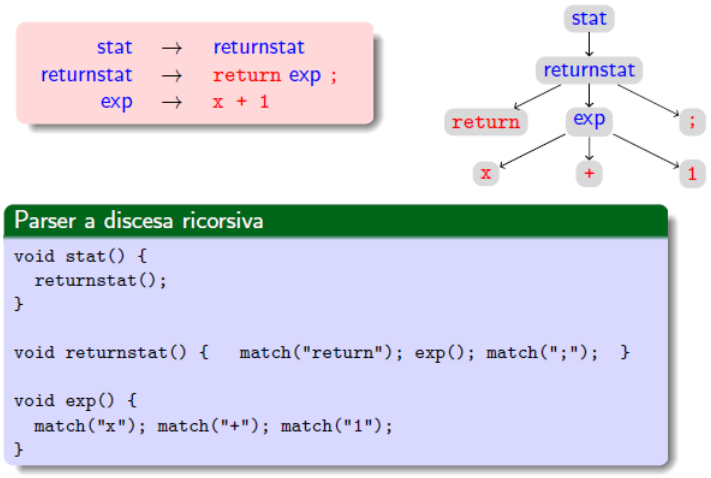


Figure 6: Simple example of implementing a visit for a top-down Parser

Consider the input stream of token as a stack: **return x + 1 ;** We need two types of function:

- **stat(), returnstat()** and **exp()** implement the production rule for a non-terminal symbol;
- **match(param)** removes from the stack the terminal symbol passed as a parameter.

If the entire visit leads to an empty stack, the input is accepted, because it satisfies the grammar rules. What was described until now cannot be considered a general mode of operation for a descendent recursive parser, and indeed not all grammars can produce one. Effectively context-free grammar for which it is possible to implement a top-down Parsers are LL(k), where k¿1. In these grammars the input is read Left-to-right and the Parsing Tree is created with a Leftmost derivation, that is at each step the left most non-terminal symbol is analysed.
Three criteria must be satisfied:

- Non-determinism has to be solvable in the application of production rules through the lookahead token, which is a procedure that carries out a forward visit of the input for at most k symbols, without consuming it.

- No left-recursion in production rules, because that is cause of infinite loops. This problem can be solved, but the process transforms the grammar in a more complicated one.
- The grammar cannot be ambiguous.

Figure 7: Example of left-recursion

Figure 8: Left-recursion solved

In figure **??** is shown a very simple example of left recursion and in fuigure **??** we can observe how it is resolved. Thus, we have a fairly precise idea of how a front-end component of a compiler works and we have the instruments to understand the very important role of ANTLR4 in the Java measure sensor development process.

3 ANTLR4

ANTLR v4 is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files. It's widely used in academia and industry to build all sorts of languages, tools, and frameworks. From a formal language description called a grammar, ANTLR generates a parser for that language that can automatically build parse trees. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code. If you give your ANTLR-generated parser valid input, the parser will always recognize the input properly, no matter how complicated the grammar. Of course, it's important to make sure the grammar accurately describes the language in question.

3.1 Parse-tree Listeners

To walk a tree and trigger calls into a listener, ANTLR's runtime provides class ParseTreeWalker.
To make a language application, we build a ParseTreeListener implementation containing application-specific code that typically calls into a larger surrounding application. ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign, for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign(). The tree diagram shown below shows ParseTreeWalker performing a depth-first walk, represented by the thick dashed line, in Figure **??**.

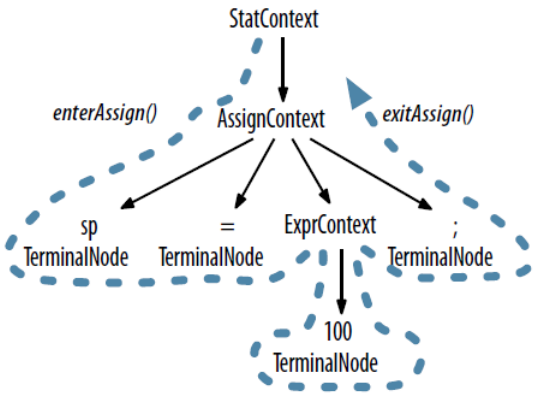


Figure 9: Parse tree walker path example

It also identifies where in the walk ParseTreeWalker calls the enter and exit methods for rule assign. The diagram in Figure **??** shows the complete sequence of calls made to the listener by ParseTreeWalker for our statement tree. The beauty of the listener mechanism is that it is all automatic. We do not have to write a parse-tree walker, and our listener methods do not have to explicitly visit their children.

Figure 10: Parse tree walker call sequence

3.2 Parse-tree Visitors

There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children. Option-visitor asks ANTLR to generate a visitor interface from a grammar with a visit method per rule. Figure **??** shows the familiar visitor pattern operating on our parse tree:

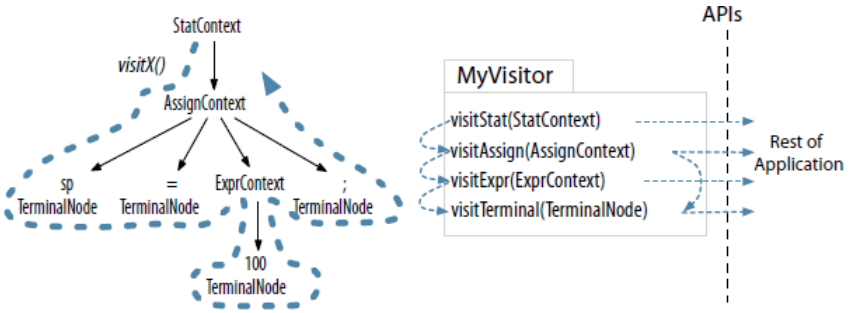


Figure 11: Personalised visitor sequence

The thick dashed line shows a depth-first walk of the parse tree. The thin dashed lines indicate the method call sequence among the visitor methods.

ANTLR gives us a leg up over writing everything ourselves by generating the visitor interface and providing a class with default implementations for the visitor methods. This way, we avoid having to override every method in the interface, letting us focus on just the methods of interest.

3.3 Implementing Applications with Parse-Tree Listeners

To build language applications without entangling the application and the grammar, the key is to have the parser create a parse tree and then walk it to trigger application-specific code. We can walk the tree using our favourite technique, or we can use one of the tree-walking mechanisms that ANTLR generates. In this section, we're going to use ANTLR's built-in ParseTreeWalker to build a listener-based simple example. In figure **??** we can observe a really simple grammar, in figure **??** a file related to the grammar and in figure **??** a Parsing tree generated for the File.

listeners/PropertyFile.g4

```
file : prop+ ;
prop : ID '=' STRING '\n' ;
```

Figure 12: Simple grammar to define properties.

listeners/t.properties

```
user="parrt"
machine="maniac"
```

Figure 13: File for the property grammar

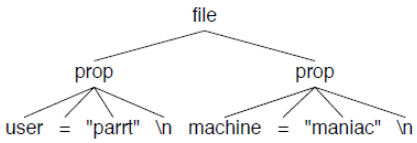


Figure 14: Parsing tree generated by the Parser for the property file.

Once we have a parse tree, we can use ParseTreeWalker to visit all of the nodes, triggering enter and exit methods. Now move on figure **??** and take a look at listener interface PropertyFileListener that ANTLR generates from grammar PropertyFile. ANTLR's ParseTreeWalker triggers enter and exit methods for each rule subtree as it discovers and finishes nodes, respectively. Because there are only two parser rules in grammar PropertyFile, there are four methods in the interface.

listeners/PropertyFileListener.java

```
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.Token;

public interface PropertyFileListener extends ParseTreeListener {
        void enterFile(PropertyFileParser.FileContext ctx);
        void exitFile(PropertyFileParser.FileContext ctx);
        void enterProp(PropertyFileParser.PropContext ctx);
        void exitProp(PropertyFileParser.PropContext ctx);
}
```

Figure 15: Java listener class generated by ANTLR4.

The FileContext and PropContext objects are implementations of parse-tree nodes specific to each grammar rule. They contain useful methods that we'll explore as we go along. As a convenience, ANTLR also generates class PropertyFileBaseListener with default empty implementations. The default implementations let us override and implement only those methods we care about. For example, in figure **??** a reimplementation of the property file loader that has a single method like before, but using the listener mechanism:

listeners/TestPropertyFile.java

```
public static class PropertyFileLoader extends PropertyFileBaseListener {
    Map<String,String> props = new OrderedHashMap<String, String>();
    public void exitProp(PropertyFileParser.PropContext ctx) {
        String id = ctx.ID().getText(); // prop : ID '=' STRING '\n' ;
        String value = ctx.STRING().getText();
        props.put(id, value);
    }
}
```

Figure 16: Java listener class in which we override methods from the listener.

There are a lot of interfaces and classes in flight here, so let's look at the inheritance relationship between the key players in figure **??**.

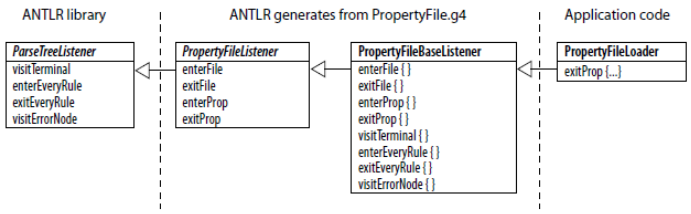


Figure 17: Inheritance relationship.

Interface ParseTreeListener is in the ANTLR runtime library and dictates that every listener respond to events visitTerminal(), enterEveryRule(), exitEveryRule(), and (upon syntax errors) visitErrorNode(). ANTLR generates interface PropertyFileListener from grammar PropertyFile and default implementations for all methods in class PropertyFileBaseListener. The only thing that we're building is the PropertyFileLoader, which inherits all of the blank functionality from PropertyFileBaseListener. Method exitProp() has access to the rule context object, PropContext, associated with rule prop. That context object has methods for each of the elements mentioned in rule prop (ID and STRING). Because those elements are token references in the grammar, the methods return TerminalNode parse-tree nodes. We can either directly access the text of the token payload via getText(), as we've done here, or get the Token payload first via getSymbol(). And now for the exciting conclusion. Let's walk the tree, listening in with our new PropertyFileLoader.

```
$ antlr4 PropertyFile.g4
$ ls PropertyFile*.java
PropertyFileBaseListener.java   PropertyFileListener.java
PropertyFileLexer.java          PropertyFileParser.java
$ javac TestPropertyFile.java PropertyFile*.java
$ cat t.properties
user="parrt"
machine="maniac"
$ java TestPropertyFile t.properties
{user="parrt", machine="maniac"}
```

Figure 18: Property application test.

Our test program in figure **??** successfully reconstitutes the property assignments from the file into a map data structure in memory. A listener-based approach is great because all of the tree walking and method triggering is done automatically.

4 Project Design

4.1 UML

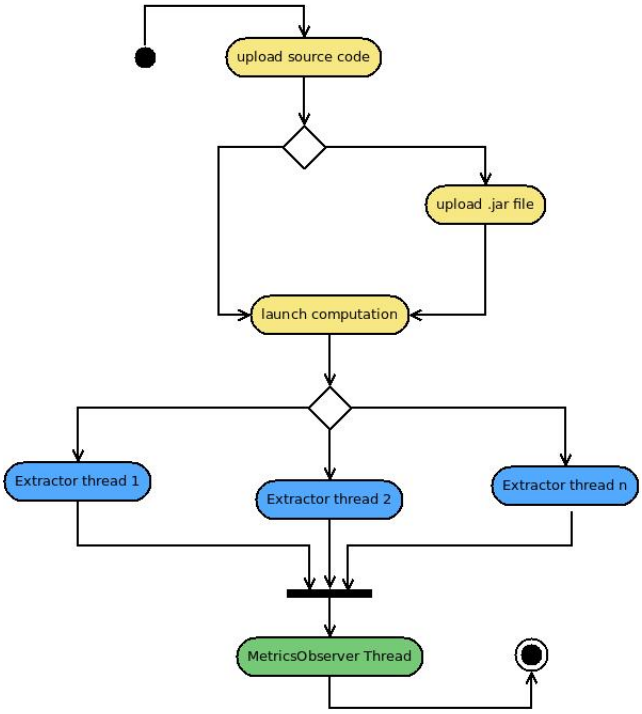


Figure 19: Activity diagram.

The software execution in figure **??** is composed of three different phases:

- Crawling (yellow)

  First of all a directory containing the .java file to be analysed and an optional .jar file containg the related compiled classes are chosen. Only if the directory contains .java files and if the .jar file effectively contains the related compiled classes, the crawling phase is passed.

- Single file analysis (light blue)

  For every .java file found, a MetricsExtractor thread is launched in order to compile code and extract a lot of useful data. All these threads are implemented with the Barrier concurrent pattern.

- General analysis (green)

  When the MetricsExtractors have reached the barrier, one last Thread called MetricsObserver is launched to compute final metrics.

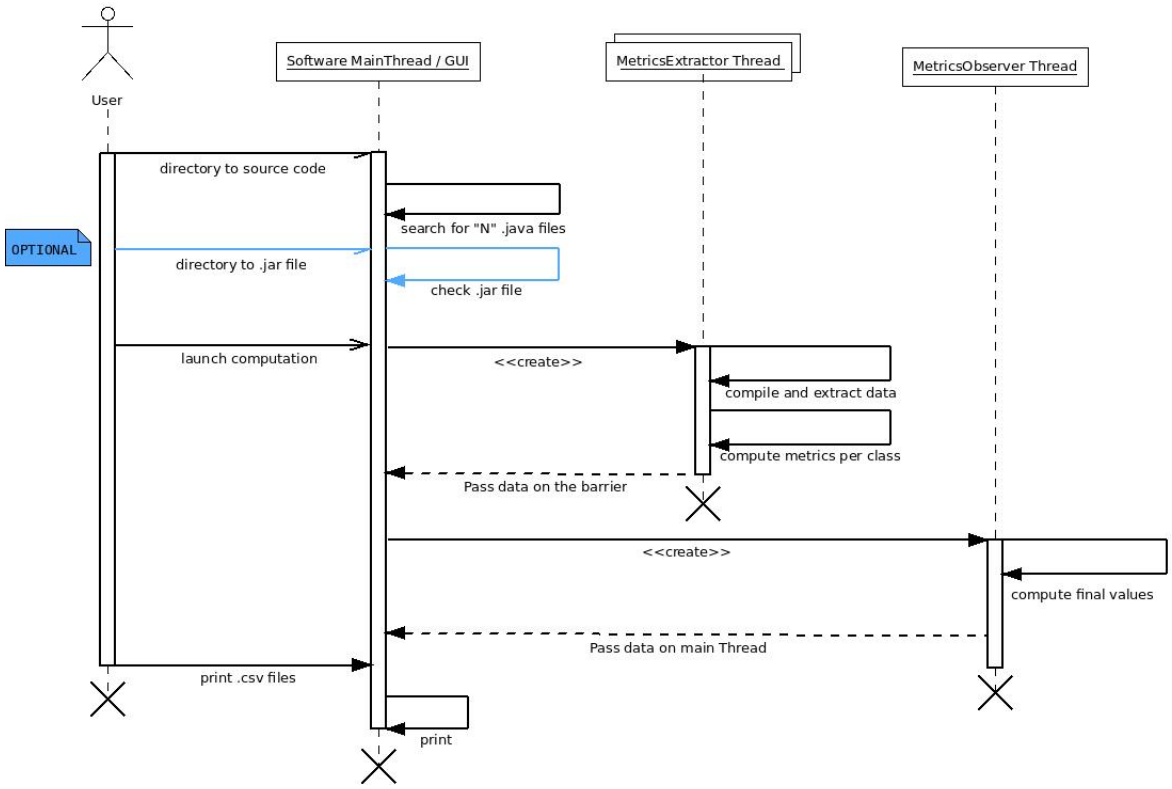


Figure 20: sequence diagram.

The sequence diagram in figure **??** is trying to explain the execution model in details. A user communicates with the main thread through the GUI. When he launches a computation successfully the MainThread creates the needed threads in order to compute the desidered metrics on the uploaded java code. The threads pass the extracted and calculated data to the Mainthread before dying. Once the computation ends the user can ask the MainThread to print the results in three different .csv files which will be explained in chapter 6.

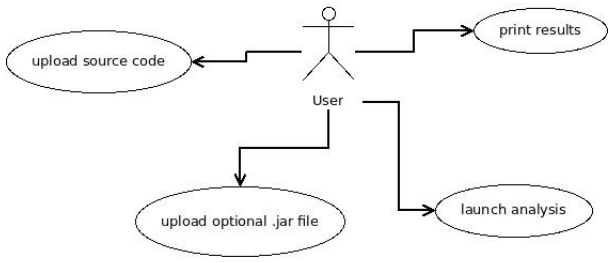


Figure 21: use case diagram.

The use case diagram in figure **??** shows what a user can do with the software from an high level of abstraction. Uploading a .jar file is optional because it is requested only if we want to calculate also structure metrics like "detph of inheritance tree", "number of children" and "coupling beetween objects". The Computation of these metrics is way easier with a compiled code available. For example in order to calculate DIT for a class with just the Java code you have to collect all the classes names and understand which class extends another. After that you need to compare all the collected data and count the depth of a class in the inheritance tree observed before. We also need to consider a plus extension for the Object class that is super class of any class by default. However a java class can extend a class that is located in an imported library. In a situation like that it could be very difficult to trace back the entire inheritance tree of a class. If a compiled code is available instead, calculate the depth of inheritance tree of class is simple as write a recursive method. Indeed a compiled class contains all the needed data. More details about how all the metrics are calculated will be revealed in chapter 5.

In order to create a solid main structure of the software it was implemented using the Model-View-Controller pattern. This choice was made also to simplify the realization of a user friendly GUI with JavaFX, but it will be explained better in the next few pages. In figure **??** we can find a package diagram. The mainPackage contains a class and five different packages:

- generated package

  Containing all the classes generated by ANTLR4.

- data package

  Containing all the classes in which the data extracted during the compilation process are stored.

- logic package

  Containing all the classes that creates the developed execution model, e.g. threads, barrier pattern.

- view package

  Containing all .fxml files of the GUI and the relatives controller classes.

- util package

  Containing all classes with a specific scope. For example the CSVMaganer class is responsible for the results printing in .csv files.

The ComponentsCtrl class represents the main core managing all the GUI scenes, threads and data during the execution. For example, methods in the GUI's controller classes can launch the needed threads or access the extracted data in order to print them through a static reference to an instance of a DataCtrl class (util package).
We will now try to show the software structure in the details benefiting of three different class diagrams, the first for the generated package, the second for the data package and the last one for the logic package.

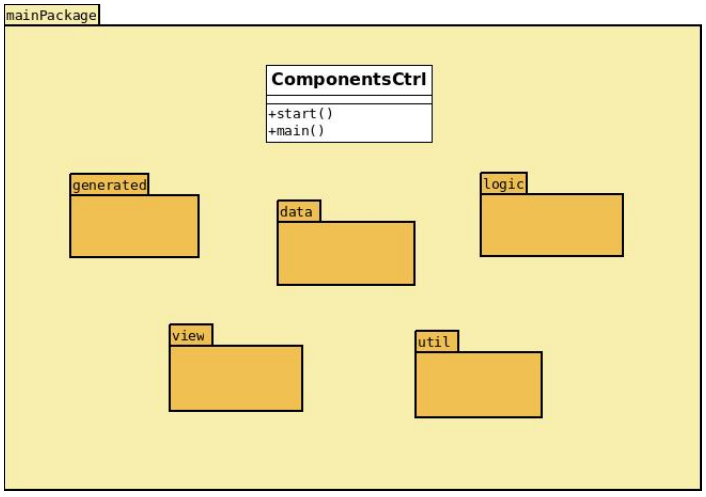


Figure 22: Packages diagram.

The four classes in figure **??** are generated by ANTLR4. The Java8Parser class and the Java8Lexer class are used by every single MetricsExtractor thread in order to compile the code. The Java8Listener interface provides a number of methods equals to the double of different node's type that a java 8 parsing tree can include. Every type of node is related to the possible invocation of two methods during the visit of the parsing tree. One is invoked when the visit is entering the node and the other is invoked when exiting from it.

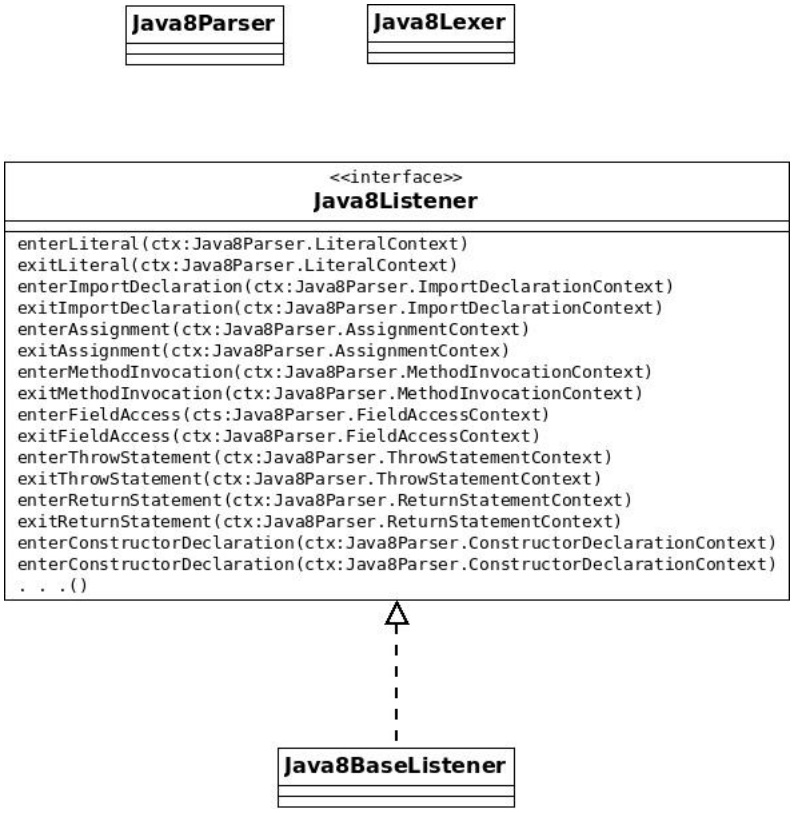


Figure 23: generated package's class diagram.

The class diagram shown in figure **??** is designed to allow the collected data to be passed between the threads in a simple, protected and extensible way. The abstract DataPack class is the idealization of the group of data collected by a MetricsExtractor thread. A number of DataPack instances are aggregated to a GeneralsPack class which represents the group of data related to the MetricsObserver thread. In order to create a more dense analysis of the code also a class corresponding the data of a single function was created. The generalizations for the ExtractorPack class and the GeneralsPack class differentiate the computation with .jar file from the computation without one.

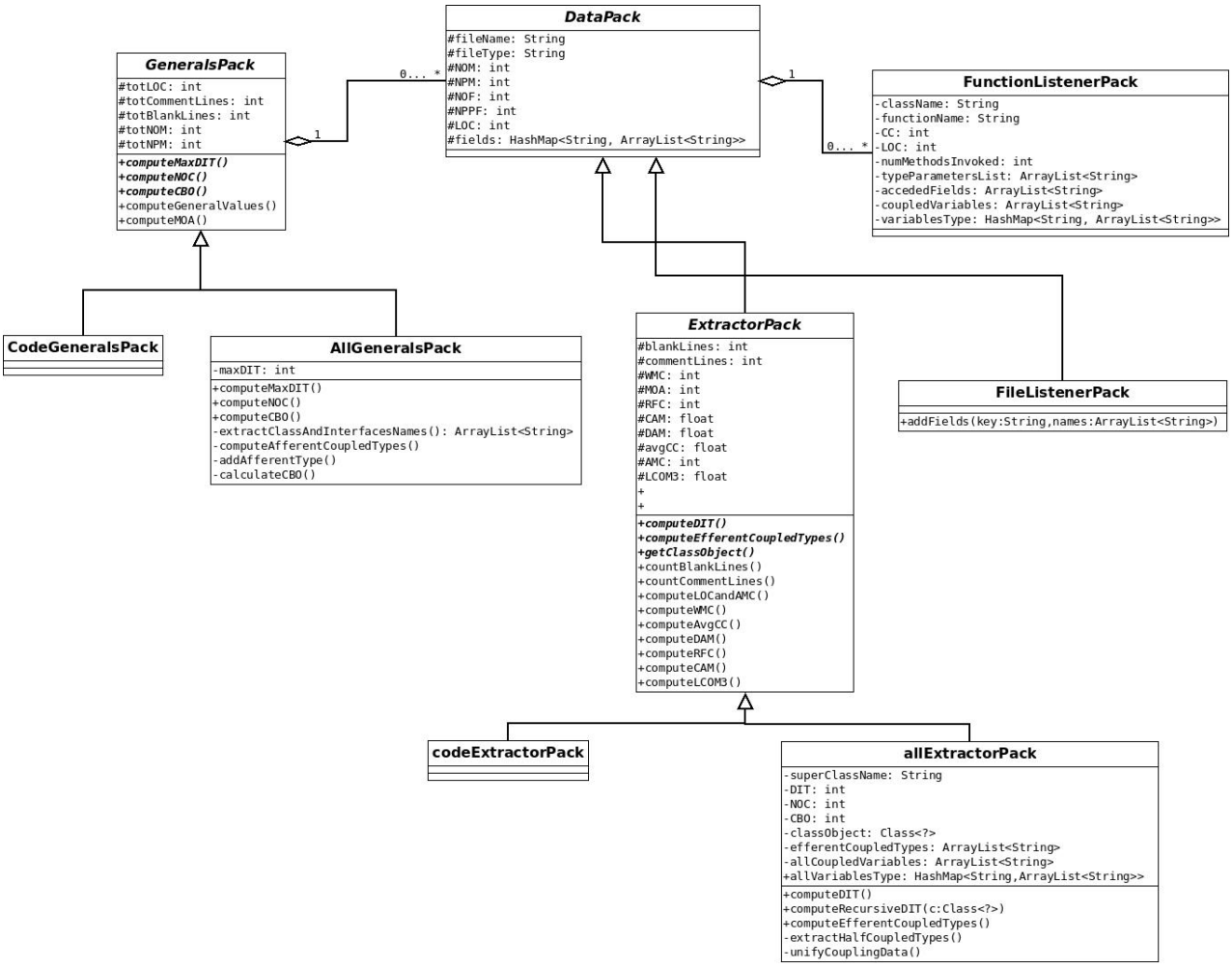


Figure 24: data package's class diagram.

The last class diagram shown in figure **??** includes all the classes which are responsible for the modellation of the lanched threads. MetricsObserver, Barrier and MetricsExtractor classes realized the Barrier concurrent pattern. During the crawling fase the number of the .java files founded in the source code is saved on the Barrier instance. Every single MetricsExtractor thread launched for a .java file is associated to that Barrier instance and when a MetricExtractor computation is done all the relatives ExtractorPacks are passed to it and a counter is incremented. When all the MetricsExtractor have finished their work, the Barrier instance is able to notice it because of the number of .java files saved before compared to the counter. At this point the MetricsObserver thread is launched carrying all the ExtractorPacks collected before. The generalizations of MetricsObserver and MetricsExtractor classes differentiate the computation with .jar file from the computation without one as in the data package. The FileListener and the FucntionListener have an important role because all Java8Listener's overrided methods are implemented in them. These classes are responsible for the direct extraction of data from the compilation process. As anticipated before every class in this package has its own dedicated class in the data package, e.g MetricsObserver has GenerealsPack, MetricsExtractor has ExtractorPack.

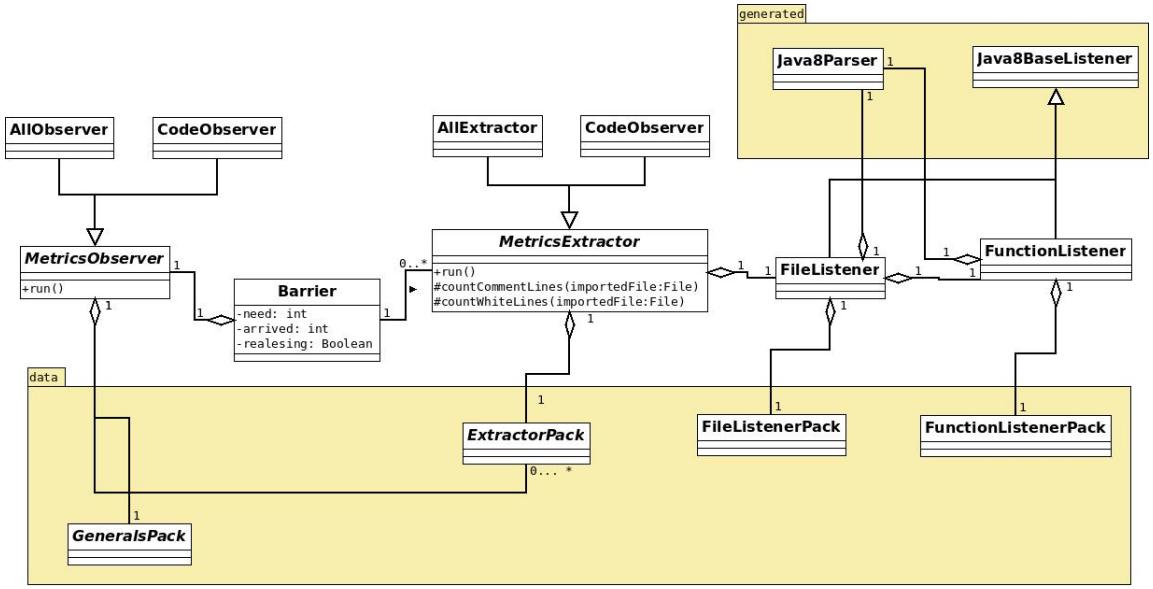


Figure 25: logic package's class diagram.

4.2 GUI

The graphic user interface was obtained thanks to JavaFX and SceneBuilder. The second one is a software to graphically design a scene generating a .fxml file which contain a JavaFX markup language similar to XML. The components in the scene like textfields, charts or links are related to a java object in the corresponding controller class of the scene. For example the InputScene.fxml is controlled by the InputSceneCtrl.java and the connections between these files are made through SceneBuilder. The containers of all scenes (VBox, HBox, Panels, ..) are controlled by ComponentsCtrl class which provides static methods to load and set the .fxml files. An example in figure **??**.

```
/**
 * This method set set as the center of the structure layout BorderPane, a VBox which represents
 * the scene for the input request.
 */
public static void showInputSceneLayout(){
    try{
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ComponentsCtrl.class.getResource( name: "view/InputScene.fxml"));
        inputLayout = loader.load();

        structureLayout.setCenter(inputLayout);
    } catch (IOException e){
        e.printStackTrace();
    }

}
```

Figure 26: Example of loader method in ComponentsCtrl.

The input scene in figure **??** gives a user the posiibilty to choose a directoriy to the source code and one to jar file. All the calculated metrics are shown too.

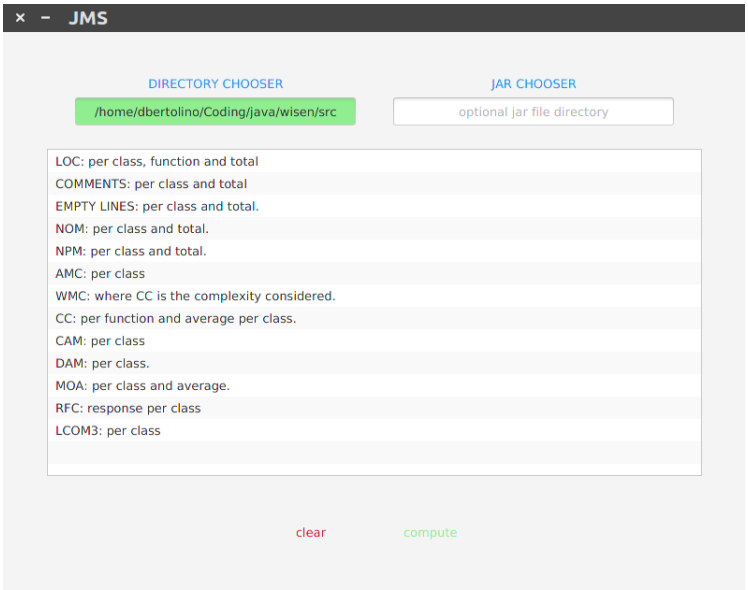


Figure 27: Input scene.

A progress was created to follow the computation and it is shown in figure **??**. Every time a MetricsExtractor thread finish his work it notifies the fact to the MainThread which is responsible for the progress bar incrementation. Same discussion for the MetricsObserver thread.

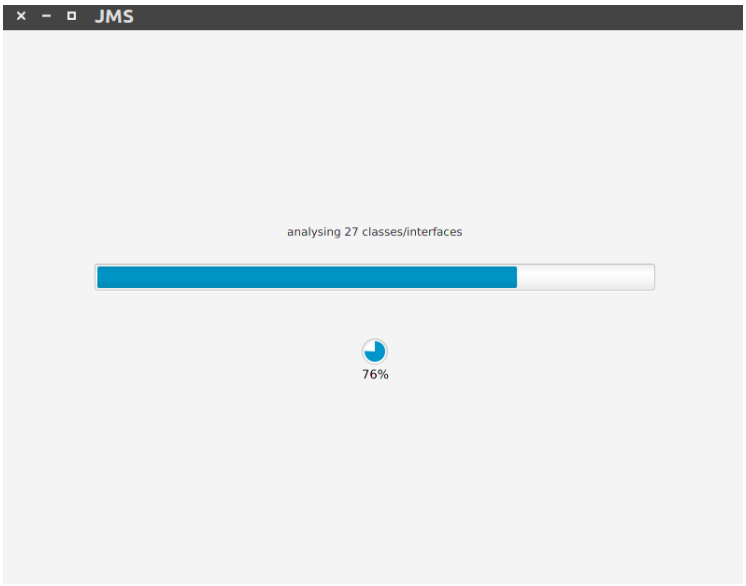


Figure 28: Progress scene.

Another significant scene of the GUI will be shown in chapter 6 to discuss how the results are presented.

5 Calculated Metrics

5.1 Lines of Code - LOC

Lines of code are counted from java binary code and it is the sum of number of fields, number of methods and number of instructions in every method of given class. This metric is useful to determine the size of a java program.

Collecting data and computation: LOC are counted by compiler listeners, while the number of blank lines and comment lines are extracted by a simple word-processor task executed directly on the code. Class Listener counts a line of code for every field, method and constructor declared, at the same time the Function Listener count a line of code for every local variable declaration statement, expression statement, synchronized statement, throws statement, return statement and every possible iteration or selection cycle. Expressions are counted as one line of code even if the expression is composed by two field accesses and cycles like switch-case or try-catch, counts number of lines of code equals to the number of case/catch labels. Sum of lines of code per class is calculated during the MetricsExtractorThread life, invoking the computedLOCandAMC() method on the ExtractorPack. Sum of total values is calculated during the MetricsObserver thread life, invoking the computeTotalValues() method excecuted on the GeneralsPack. Word-processor task is invoked directly on every single file by MetricsExtractors.

5.2 Number of Methods and Public Methods - NOM/NPM

Number of methods metric is used to calculate the average count of all class operations per class while number of public methods metrics can be used to measure the size of an API provided by a package.

Collecting data and computation: Methods and public methods are counted by the compiler listener in the same way, with the difference of a condition on the method declaration modifier. Methods per class are simply counted by every file listener, while the total number is calculated during the MetricsObserver thread life, invoking the computeTotalValues() method excecuted on the GeneralsPack.

5.3 Average Method Complexity - AMC

Size of a method is equal to the number of java binary codes in the method and the average method complexity for a class is calculated by dividing the sum of complexity of all methods to the total number of methods in that class. The assumption behind this metric is that a large method, which contains more code, tends to introduce more faults than a small method.

Collecting data and computation: This is a simple sum of the lines of code of every function in class, calculated during the MetricsExtractorThread life, invoking the computedLOCandAMC() method on the ExtractorPack.

5.4 Cyclomatic Complexity - CC

Cyclomatic Complexity is a graph-theoretic complexity and it can be used to manage and control program complexity through the count of possible paths in the execution flow. We now explain how it works and how it will be calculated in the JMS. Even if any program with a backward branch potentially has an infinite number of path, so using the total number of paths is impractical, this theory defines a set of algebraic expressions that give the total number of possible paths through a structured program. The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is,

$$V(G) = e - n + p \tag{3}$$

Theorem I: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

For example, consider a program associated with a directed graph that has unique entry and exit nodes and where each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is known as the program control graph and it is assumed that each node can be reached by the entry node and each node can be reach the exit node. An example of directed graph is shown in figure **??**.

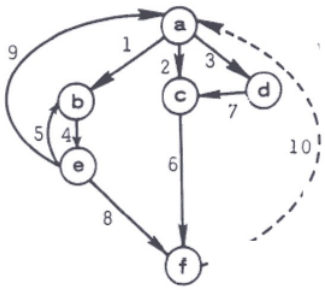


Figure 29: Directed graph.

If we imagine that the exit note (f) branches back to the entry node (a), the graph became a strongly connected one and we can apply Theorem I. Therefore, the maximum number of linearly independent circuits in G is equal to 9-6+2. In the general case of one connected component, we therefore have

$$V(G) = e - n + 2 \tag{4}$$

Consider now five linearly independent circuits in G.

$$B1 : (abefa), (beb), (abea), (acfa), (adcfa) \tag{5}$$

It follows that B1 forms a basis for the set of all circuits in G and any path through G can be expressed as a linear combination of circuits from B1. For instance,

$$(abeabebebef) = (abea) + (2beb) + (abefa) \tag{6}$$

To better understand how this works, we can associate a vector to each member of B1

Path **Branches**

	1	2	3	4	5	6	7	8	9	10
(abefa)	1	0	0	1	0	0	0	1	0	1
(beb)	0	0	0	1	1	0	0	0	0	0
(abea)	1	0	0	1	0	0	0	0	0	0
(acfa)	0	1	0	0	0	1	0	0	0	1
(adcfa)	0	0	1	0	0	1	1	0	0	1

The path $(abea(be^2)fa)$ corresponds to the vector 2004200111 and the vector addition of $(abefa)$, $(2beb)$ and $(abea)$ yelds the desired result. Now, in using Theorem I we can choose a basis set of circuits that correspomd to paths through the program. The set B2 is a basis of program paths:

$$B2 : (abef), (abeabef), (abebef), (acf), (adcf) \tag{7}$$

Linear combination of path in B2 will also generate any path, for example:

$$(abea(be^2)fa) = 2(abebef) - (abef) \tag{8}$$

The strategy just described can be used to measure the complexity of a program a program by computing the number of linearly independent paths v(G), control the "size" of programs by setting an upper limit to v(G) (instead of use just physical size), and use the cyclomatic complexity as the basis for a testing methodology.
Several properties of the cyclomatic complexity:

- $V(G) \geq 1$
- $V(G)$ is the maximum number of linearly independent paths in G; it is the size of a basis set. Inserting or deleting functional statements to G does not affect $V(G)$
- $V(G)$ depends only on the decision structure of G.

We now discuss the role of p in the computation formula for the cyclomatic complexity can be calculated of a strongly connected graph, i.e., V(G) = e – n + p, We said that p is the number of connected components in the graph and the way we define a program control graph would result in all control graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having control structure shown in Figure **??**

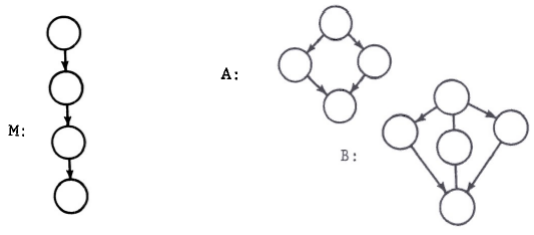


Figure 30: Control structures.

The total graph shown before with 3 connected components as $M \cup A \cup B$ and we calculate complexity as:

$$v(M \cup A \cup B) = e - n + 2p = 13 - 13 + (2 \times 3) = 6 \tag{9}$$

This method with $p \neq 1$ can be used to calculate complexity of a collection of programs, particularly a hierarchical nest of subroutines as shown above.
Notice that $v(M \cup A \cup B) = v(M) + v(A) + v(B) = 6$. In general, the complexity of a collection C of control graphs with k connected components is equal to the sum of their complexities. To see this let $C_i, 1 \leq i \leq k$, denote the k distinct connected components, and let e_i and n_i be the number of edges and nodes in the ith connected component.

$$v(C) = e - n - 2p = \sum_{1}^{k} e_i - \sum_{1}^{k} n_i + 2k = \sum_{1}^{k} (e - n - 2p) = \sum_{1}^{k} v(C_i) \tag{10}$$

Since the calculation of the cyclomatic complexity seen until now can be quite complex to implement in software, an effort has been made to simplify it, creating a calculation to be done in terms of program syntactic constructs.

Mills theorem: Structured programming paradigms include procedural and functional programming and, as a consequence, Object Oriented programming like java. If the number of function (Φ), predicate (π) and collecting nodes (Υ) in a structured program and e is the number of edges of the program control graph, then:

$$e = 1 + \Phi + 3\pi \tag{11}$$

Assuming p=1 and substituting in v=e-n+2 we get:

$$v = (1 + \Phi + 3\pi) - (\Phi + 2\pi + 2) + 2 = \pi + 1 \tag{12}$$

Cyclomatic complexity is equal to the number of predicates plus one for a structured program, so we are now able to calculate it simply counting the number of predicates and not having to deal with the program control graph. In practice compound predicates as IF "C1 AND C2" THEN, are treated as contributing two to complexity since without the connective AND we would have IF C1 THEN IF C2 THEN, which has two predicates. For this reason it has been found to be more convenient to count conditions instead of predicates when calculating complexity.

For the CASE construct with N cases the count is N-1 because the simulation of case with if-then will have N-1 conditions.

Collecting data and computation: Decision points are counted by the compiler listener for each function. Every FunctionPack has a starting CC value of 1, which is incremented by the compiler listener every time a decision point is found, during the MetricExtractor thread life.

5.5 Weighted Methods per Class - WMC

Weighted methods per class is simply the sum of the complexities of its methods. As a measure of complexity we use the cyclomatic complexity. It is an indicator of how much effort is required to develop and maintain a particular class.

Collecting data and computation: The sum of the CC of every method is done by the MetricExtractor thread invoking the computeWMC() method.

5.6 Cohesion Among Methods of Class -CAM

This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation (S) of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types (t) in whole class and number of methods (n):

$$V(G) = \frac{S}{t \times n} \tag{13}$$

In a range from 0 to 1 a metric value close to 1.0 is preferred.

Collecting data and computation: FunctionListeners collect type parameters of every function in the class, then CAM is calculated during the MetricsExtractorThread life, invoking the computeCAM() method on the ExtractorPack. This method controls all different types in class and sums all different types per function. If functions have no parameters the value assumed for CAM is 1.

5.7 Data Access Metric - DAM

Data access metric indicates the level of data hiding (encapsulation) in the class and it is the ratio of the number of private or protected attributes to the total number of attributes declared in the

class. In a range from 0 to 1 a high value for DAM is desired. It .

Collecting data and computation: FileListeners collect number of fields (NOF) and number of private/protected fields (NPPF) of every class, then DAM is calculated during the MetricsExtractorThread life, invoking the computeDAM() method on the ExtractorPack. If Class has no fields, the value assumed for DAM is 1.

5.8 Measure Of Aggregation - MOA

This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations (class fields) whose types are user defined classes.

Collecting data and computation: The Compiler listener populates an HashMap<String>, ArrayList<String> in the FileListenerPack in which the key is a type name and the ArrayList contains all fields names declared for that type in the class. MOA is calculated during the MetricsObserver life cycle, invoking the computeMOA() method on the GeneralsPack. This method simply controls which type of the fields declared corresponds to a Java class or interface defined by the programmer and count the related variables.

5.9 Response for a Class - RFC

The Response for a class metric measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). Ideally, we would want to find for each method of the class, the methods that that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method's call graph. This process can however be both expensive and quite inaccurate. We calculate a rough approximation to the response set by simply inspecting method calls within the class's method bodies. The value of RFC is the sum of methods called within the class's method bodies and the number of class's methods. This simplification was also used in the 1994 Chidamber and Kemerer description of the RFC metric.

Collecting data and computation: The number of method declared in class are already calculated for the NOM metric, so JMS simply count the method calls within every method using the FunctionListener. The sum is computed during the MetricsExtractor lifecyle calling the computeRFC() method on the ExtractorPack.

5.10 Lack of Cohesion in Methods - LCOM3

A cohesive class performs one function. Lack of cohesion means that a class performs more than one function. This is not desirable. If a class performs several unrelated functions, it should be split up. High cohesion is desirable since it promotes encapsulation. Low cohesion indicates inappropriate design and high complexity. It has also been found to indicate a high likelihood of errors. The class should probably be split into two or more smaller classes. In a normal class whose methods access the class's own variables, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). When $LCOM3 = 0$, each method accesses all variables. This indicates the highest possible cohesion. $LCOM3 = 1$ indicates extreme lack of cohesion. In this case, the class should be split. When there are variables that are not accessed by any of the class's methods, $1 < LCOM3 \leq 2$. This happens if the variables are dead or they are only accessed outside the class. Both cases represent a design flaw. The class is a candidate for rewriting as a module. Alternatively, the class variables should be encapsulated with accessor methods or properties. There may also be some dead variables to remove. If there are no more than one method in a class, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as zero.

LCOM3 varies between 0 and 2 and values from 1 to 2 are considered alarming.

$$LCOM3 = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1 - m} \tag{14}$$

- m - number of procedures (methods) in class
- a - number of variables (attributes in class)
- $\mu(A)$ - number of methods that access a variable (attribute)

Collecting data and computation: Number of procedures and number of fields have already been calculated for other metrics, JMS simply needs to count the number of methods that access every fields. FunctionListener populate an $ArrayList < String >$ for every function with all the names of the variables acceded within the function. During the MetricsExtractor thread life cycle is called the computeLCOM3() method, which calculate the sum of the method that access a field and compute LCOM3 for that class.

5.11 Coupling Between Objects - CBO

The coupling between object classes metric represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions. A class's afferent couplings is a measure of how many other classes use the specific class and class's efferent couplings is a measure of how many other classes is used by the specific class.

Collecting data and computation: Part of the efferent coupled types of a class are extracted during the MetricsExtractor thread invoking the computeEfferentCoupledTypes() method on the AllPack object. This method call the private extractHalfCoupledTypes() method in figure **??** first, in which are extracted all Exceptions, return and superclass types from the .jar file.

```
private void extractHalfCoupledTypes(){
    for(Method m : this.classObject.getDeclaredMethods()){

        for(Class<?> type : m.getExceptionTypes()) {
            if (!(this.efferentCoupledTypes.contains(type.getSimpleName()))) {
                this.efferentCoupledTypes.add(type.getSimpleName());
            }
        }
        Class<?> type = m.getReturnType().getClass();
        if(!(type.getSimpleName().equals("Class")) && !(type.getSimpleName().equals("Object"))){
            if(!(this.efferentCoupledTypes.contains(type.getSimpleName()))){
                this.efferentCoupledTypes.add(type.getSimpleName());
            }
        }

    }
    if(this.classObject.getSuperclass() != null){
        String superType = this.classObject.getSuperclass().getSimpleName();
        if(!(superType.equals("Class")) && !(superType.equals("Object"))){
            if(!(this.efferentCoupledTypes.contains(superType))){
                this.efferentCoupledTypes.add(superType);
            }
        }
    }
```

Figure 31: Code to extract coupled types from .jar file.

Every time the FunctionListener bumps into a method invocation, it stores into the FunctionListenerPack the name of the variable on which is invoked the espressions passed as argumets. Meanwhile every time it bumps into field access, it stores into the FunctionListenerPack the name of the variable acceded and it also save all the local variable names (per type) encountered.
So a FunctionListenerPack stores:

- $ArrayList < String >$ containing the names of the coupled variables.
- $HashMap < String, ArrayList < String >>$ containing the local variable names per type.

We now need to consider again the computeEfferentCoupledTypes() method in figure **??**.

```
@Override
public void computeEfferentCoupledTypes(){
    extractHalfCoupledTypes();
    unifyCouplingData();
    for(String type : this.AllVariablesType.keySet()){
        for(String var : this.AllVariablesType.get(type)){
            if(this.coupledVariables.contains(var)){
                if(!(this.efferentCoupledTypes.contains(type))){
                    this.efferentCoupledTypes.add(type);
                }
            }
        }
    }
}
```

Figure 32: Code to extract coupled types from .jar file.

As we can see, after the extractHalfCoupledTypes() method described before, another private method is called, unifyCouplingData(). This method unifies all coupled variables names found in every FunctionListenerPack and also all the local variable names per type of every FunctionListenerPack with all field names per type. After this operation the ExtractorPack contains:

- $ArrayList < String >$ containing the names of all the coupled variables in the Class.
- $HashMap < String, ArrayList < String >>$ containing the variable names presents in Class per type.

At this point, with a simple match between these data structures we can extract all the efferentCoupledTypes of the Class. After all this process, during the MetricsObserver thread life cycle, the computeCBO() method is invoked on the GeneralsPack, in which all the afferent types of the classes are computed using the efferent types, after that thanks to collected data CBO value is computed for every Class.

5.12 Depth of Inheritance Tree - DIT

The depth of inheritance tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1.

Collecting data and computation: DIT value is computed during the MetricsExtractor thread invoking the computeDIT() method in figure **??** on the AllPack object. Thanks to the Class<? > object of the analysed class extracted from the .jar file the value is calculated with a simple recursive method:

```
@Override
public void computeDIT() { computeRecursiveDIT(this.classObject); }

private void computeRecursiveDIT(Class<?> c){
    Class<?> superC = c.getSuperclass();
    if(superC != null ){
        if(superC.getName().endsWith("Object")){
            this.DIT = DIT + 1;
        }
        else{
            this.DIT = DIT + 1;
            computeRecursiveDIT(superC);
        }
    }
}
```

Figure 33: Code to extract DIT from .jar file.

5.13 Number Of Children - NOC

A class's number of children metric simply measures the number of immediate descendants of the class.

Collecting data and computation: NOC is computed thanks to Class<? > objects of the analysed classes, from which the super classes names are extracted. During the MetricsObserver life cycle the computeNOC() method in figure **??** is called on the GeneralsPack:

```
@Override
public void computeNOC() {
    for (ExtractorPack ep : this.dataPacks) {
        AllPack examinedPack = (AllPack) ep;
        for (ExtractorPack p : this.dataPacks) {
            AllPack pack = (AllPack) p;
            if(examinedPack.getSuperClassName() != null){
                if (examinedPack.getSuperClassName().equals(pack.getFileName())) {
                    pack.increaseNOC();
                }
            }
        }
    }
}
```

Figure 34: Code to extract NOC from .jar file.

6 Computation Outcome

6.1 Printing .csv Files

To create easily readable results, it was decided to print them in .csv format. Printed data are distributed on 3 different levels of inspection: total values in figure **??**, class values in figure **??** and function values in figure **??**. A directory where to print the files and a name are required when the user wants to export the data.

	A
1	LOC_Comments_Blank_NOM_NPM_DIT
2	991_52_329_153_116_2
3	

Figure 35: File with total values.

	A	B
1	Name_Type_Comments_Blank_LOC_NOM_NPM_WMC_CC_AMC_DAM_MOA_RFC_CAM_LCOM3_DIT_NOC_CBO	
2	MessageViewCtrl_Class_2_5_3_1_0_1_1.0_1_1.0_0_2_1.0_0.0_1_0_0	
3	ControlReportSceneCtrl_Class_0_9_28_4_4_4_1.0_17_1.0_2_31_1.0_1.3333334_1_0_0	
4	InsertValueSceneCtrl_Class_0_20_75_7_2_7_1.0_55_1.0_17_66_1.0_1.1666666_1_0_0	
5	ConsumptionSceneCtrl_Class_0_15_50_5_3_6_1.2_35_1.0_10_43_1.0_1.25_1_0_0	
6	EditingBottomCtrl_Class_0_8_9_3_0_3_1.0_3_1.0_0_6_1.0_1.5_1_0_0	
7	SpendingSceneCtrl_Class_0_6_15_3_3_3_1.0_6_1.0_3_12_1.0_1.5_1_0_0	
8	ManagerBottomCtrl_Class_0_7_15_5_0_5_1.0_5_1.0_0_10_1.0_1.25_1_0_0	
9	OperatorBottomCtrl_Class_0_8_9_3_0_3_1.0_3_1.0_0_6_1.0_1.5_1_0_0	
10	LoginSceneCtrl_Class_0_5_19_1_0_6_6.0_15_1.0_1_12_1.0_0.0_1_0_0	

Figure 36: File with metrics per class.

	A
1	ClassName_FunctionName_LOC_CC
2	MessageViewCtrl_initialize()_1_1
3	ControlReportSceneCtrl_initialize()_1_1
4	ControlReportSceneCtrl_showReport()_4_1
5	ControlReportSceneCtrl_clearDetails()_4_1

Figure 37: File with metrics per function.

6.2 Program Size Chart

Figure **??** shows a simple bar chart that was created to offer a direct response to the user about software size. In few seconds, an idea of the code appears comparing total values of lines of code, comments and blank lines. The number of public methods can be used to understand the dimension of a possible API compared to the total number of methods in the libraries.

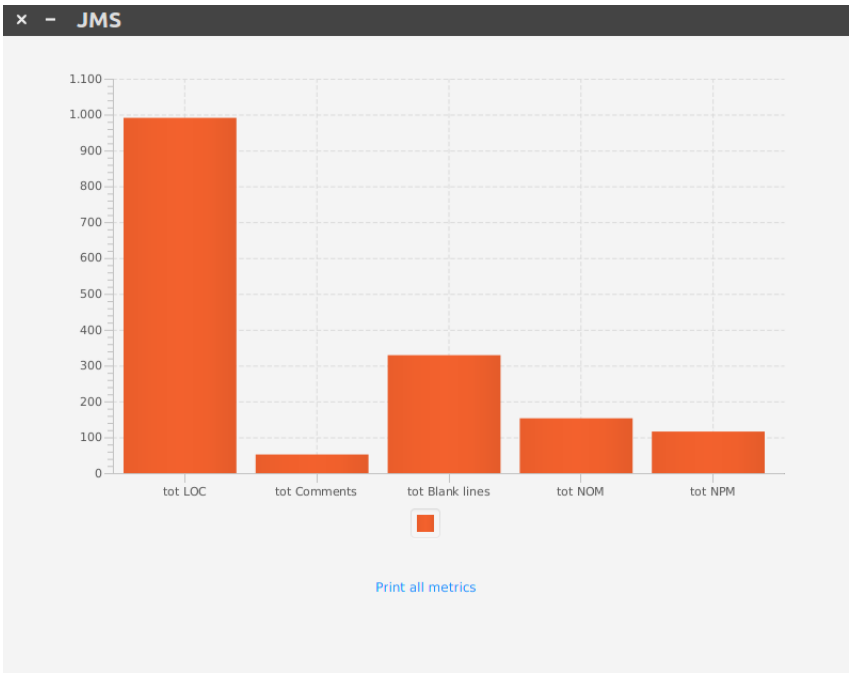


Figure 38: Chart scene

7 Conclusions

7.1 Comments

The main goal if this work was to understand how a software able to calculate engineering metrics works and to develop a tool to do the same at least with basic metrics. The research started studying the compiler's structure. As can be expected the first result I found is that building a software trying to recreate a compiler with specific functionalities is a difficult and hard task. Once I came across ANTLR4, I suddenly decided to use it to reach the objective.
The theory of automata and formal languages is the root of the work. Without its understanding the Lexer and the Parser jobs would be really difficult. For example, it is possible to understand how a parse tree listener works but it is hard to write code in order to extract data during a compilation process if you do not know the structure of the parsing tree. In this case the relationship between a program statement and a parsing tree explained in subsection 2.2 is fundamental.
Another very important piece of the puzzle is the theory behind a programming language grammar. When the Lexer and the Parser were generated from the java8 grammar the first impact with the classes was disarming. All the Java8Listener methods were corresponding possible nodes of java8 parsing tree. I had to decide which methods needed to be overridden in order to extract the data I was searching for. Moreover metrics like coupling between objects had required a lot of data extracted not only by different methods but also in various moments during the execution. Initially ANTLR4 grammar file was not so helpful because it was difficult to understand. Thanks to the grammar concept explained in subsection 2.1 and a little bit of patience reading all symbols it was possible to individuate the methods to be overridden and implemented.
Simple metrics computation like the count of lines of code, number of methods but also number of children or depth of Inheritance tree did not require a scrutinized study of the metrics itself. More effort was spent studying the theory behind more complex metrics instead. The deep description of the cyclomatic complexity made in subsection 5.4 is the best example.

7.2 Future Work

In order to implement a metric calculation that can be considered valid all variables involved have to be considered. To do this a deep knowledge of the desired metrics and probably more experience using them is required. All the metrics computations were tested on "home made" java code for which the metrics values can be calculated personally. Even if all of this tests gave positive results a future work could try to image more stressing tests to find lacks in the computation and consequently enhancing it. New ways to show the results could be created. For example, significant charts able to offer the user a direct response more than the software dimension. Otherwise new format for file printing can surely make the data readable by a great number of software products. The java 8 grammar used to generate the Lexer and the Parser was found in the open source code on Github. The project from which it was extracted offers grammars corresponding a lot of programming languages like c, c++, html, ruby, php, phyton3, scala, swift, xml and others. The software developed in this work could be emulated or extended to analyse not only the java programming language.

References

[1] Terence Parr, *The definitive ANTLR4 reference*, The pragmatic programmers.

[2] Bill Campbel, Swami Iyer and Bahar Akbal-Delibas, *Introducrtion to Compiler Construction in a Java World*, CRC press Taylor & Francis Group.

[3] John E. Hopcroft, Rajeev Motwani and Jeffrey D Ullman, *Automi linguaggi e calcolabilità*, Third edition.

[4] Thomas J. McCabe, *A complexity measure*, 1976.

[5] Sushil Kumar Bagi , Mr. Sumit Sharma and Dr. Sanjeev Bansal, *Analysis of software Metrics Tools (A Survey Approach)*, 2013.

[6] Rüdiger Lincke, Jonas Lundberg and Welf Löwe, *Comparing Software Metrics Tools*, School of Mathematics and Systems Engineering Växjö University, Sweden.