IŞIK UNIVERSITY
COMPUTER SCIENCE AND ENGINEERING

GRAPHGEN

Bachelor's Thesis

Gülden GÜLLÜ

216CS2014

Supervised by

Asst. Prof. Emine EKİN

June 2020

GRAPHGEN

Abstract

When designing a software project, the aim is to create a cost and time efficient design structure with compatible components. For the same reason, analysis of a software architecture is also important in the development process. There are many solutions provided for analysis of algorithms. Call graphs are one of them. Call graphs are basically directed graphs where the nodes represent the methods in project and the edges of the graph represents the method calls. Call graphs can be dynamically or statically created.

In this project we aimed to implement a static call graph using the JavaParser, language parser library. JavaParser and SymbolSolver has been used for analyzing the method calls of projects. Then, graph is created with Vis.js visualization library. As a result, we built an application which can analyze and create a call graph for Java projects of most levels, with external library support. Assumptions has been made for creating a reliable call graph.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Asst. Prof. Emine EKİN for providing me this interesting subject, guiding me through my thesis and for her continuous support and motivation since my first year at the university.

I wish to thank my parents, brothers and my friends for supporting me spiritually throughout my university and thesis journey. Lastly, I would like to thank Işık University Computer Engineering department for providing an opportunity to study there.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

1. Purpose of the System

The project is aiming to build a desktop application that creates a method call graph when a Java project is given as source code. Application will create a readable and easy to scan method call graph for the given Java project or Java source code files.

2. Scope of the System

Application can be used by Java developers of all levels. Understanding the structure of a project is very important, mainly in the cases where building a relatively complex project and it is harder to follow up the structure development or find out if structural errors has been made in the development process or when already built project is needed to be understood or analyzed by a developer. Therefore, the application is aimed to be used by mostly in these cases. The call graph to be generated, is aimed to be understood by a developer easily regardless of the aim of usage.

3. Objectives and Success Criteria of the Project

Application is developed to create a method call graph with a simple design, including only the sufficient function details to prevent extra work and confusion on the analysis. Application will be developed covering the possible requirements of the users. Project is aiming to be robust application, compromising most of the cases which could make the program error prone and lower the reliability of the application.

CHAPTER TWO

LITERATURE REVIEW

The analysis of a software helps to understand the complex structures of it by dividing into subsystems and helping to detect conflicts and errors in the system. In addition to helping development process, analysis of a project may help to understand workflow in a complete project. Because a complete software structure has many subsystems and code pieces and it may be hard to follow the dependencies between them.

1. Current Applications

There are current systems, which are creating call graphs or dependency analyses. They have all been developed individually and have different type of specifications. They work on different platforms, taking different inputs or giving different outputs. For example ‘java-callgraph’[1] generates a table of calling relations from a jar file source or ‘CallGraph’[2] works on Eclipse Java development tools platform. There are also other examples. GraphGen does not aim to specifically replace one of these systems. Therefore, there is no information available for the current software architecture. GraphGen will provide a call graph generator which will provide easier setup, simple architecture and interface design and robust algorithm.

2. The Algorithms

Regarding analysis needs of software applications, call graph construction is a popular topic in field of software analysis, because it is important to be able to analyze the code and architecture of software. The analysis may help to follow up with the build of software architecture, detect code inconsistencies, possible attack points and find bugs, or after the development of project simply to understand how the flow in the program works. But the construction process of a call graph is a complex problem. There is an ambiguity of how the software project to be analyzed implemented exactly. Because the source code consists many points in its details; there are many syntax rules and possible applications when a programming language is considered.

There are mainly two kind of call graph construction techniques: static and dynamic call graphs. Dynamic call graphs follow the path of program execution and determine the actively called method calls, but the static call graphs check the method calls of a passive code. For the object-oriented languages, the target of a call often depends on the runtime behavior of the program, therefore, a static call graph builder has to make assumptions about what methods could be called, resulting in possible imprecisions.[3] Therefore, dynamic call graphs provide a more precise and exact analysis, whereas static call graphs produce a more general result, including many possible executions of the program and includes approximations of the analysis.

In this project, the aim was the static analysis of Java language, therefore the document will be towards the subject after this point.

Because static analysis requires an approximation, there are many approaches or ideas how this approximation should be performed for a more precise analysis. There are some widely accepted popular algorithms as well as lots of individually developed algorithms, implemented for different language fundamentals and implementation platforms, that are always aiming to improve precision and complexity. The existing call graph construction algorithms that will be recognized in this paper are Reachability Analysis (RA), Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA).

Reachability analysis is the simplest algorithm among others, only considers name of methods. RA does not include the method’s signature in the analysis. Method’s signature consists of method’s type, parameter types and return type. In the sense of precision, RA provides the least precise result, with the largest result set.[4] If we consider the following source code, Figure 1 will be the result when analyzed with Reachability Analysis.

```
Class Example{
  Class A {
     method a() {
     }
  }
  Class B extends A {
       method b() {
       {
       @Override
       method a() {
       }
  }
```

```
Class C {
   method a() {
   }
}
Class D extends C
{
   method b() {
   }
}
```

```
Main{
   A a = new B;
   D d = new D;
   a.a();
   d.b();
}
}
```

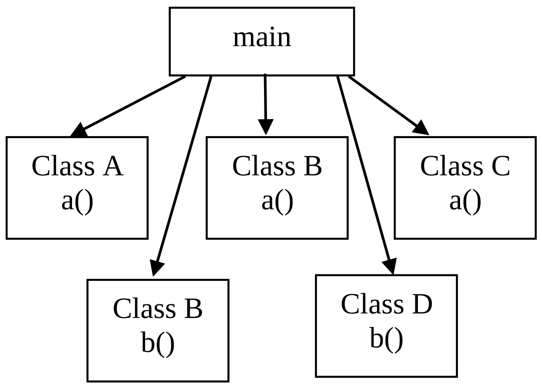

Figure 1. Call graph example with RA

Next algorithm is the Class Hierarchy Analysis. CHA considers the method's signature and also the class hierarchy besides the method's name. Class Hierarchy Analysis takes the idea that, by examining the complete inheritance graph of a program, including where methods are defined, information of possible target method of each method call, therefore quality of static class information obtained can be improved.[5] If we consider a method call named "a.foo()", then by analyzing with CHA, we understand that foo method call can be placed in either class of "a" or subclasses of it. [4] Therefore, we can increase the precision of our analysis.

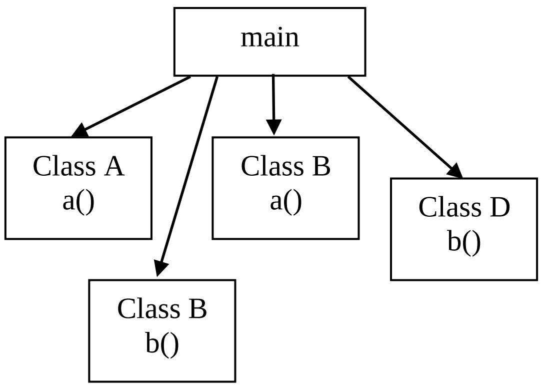

Figure 2. Call graph example with CHA

But for little more precision we consider the Rapid Type Analysis. On top of CHA, RTA takes into consideration only methods of the instantiated classes. Because in the class hierarchy some of the classes may never been instantiated, therefore we can eliminate them in the analysis, there is no way to reach them. RTA algorithm is built on CHA algorithm and only difference between them is that RTA's result set only includes the instantiated classes.

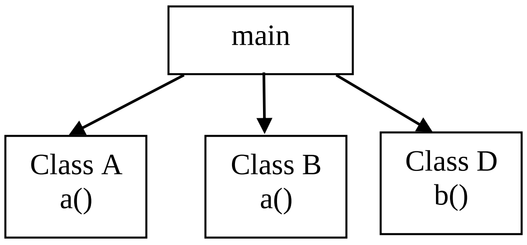

Figure 3. Call graph example with RTA

CHAPTER THREE

PROPOSED SYSTEM

1. Proposed System

GraphGen application will create a static call graph for Java source code. Program will be available open source on Github and also available for download as application. Aimed system offers easily understood and used interface design and possible foreseen requested functions to manipulate the generated graph. The algorithm to be implemented will be highly reliable and efficient. The system will be easily understood by anyone who may need this application. Generated output graph will be in a tree structure to show a similar pattern of how the code would execute, in cases of level and order, therefore, it will be manageable even with more complex projects.

2. Overview

GraphGen will get a Java project or Java files as input and will generate a static method call graph as output. Application will create a static call graph; therefore, it is aiming to detect all possible run paths of the Java projects with an efficient algorithm. The application will generate the graph depending on the request of which files the user want to choose to generate a graph. Generated graph will be available for manipulation such as getting sources of the graph nodes (Which are method calls), getting from which source code the nodes has been generated or performing search on the nodes.

3. Functional Requirements

The application is free to use and can be used by any level of developer as a result of being simple and having an easy to use interface. Using the application, the user is able to select a Java project or a single Java file for analyzing. The application will create a static method call graph of the given source code. Working on the graph, method calls determined from the source code can be analyzed. From the created graph, some desired parts can be highlighted, search for a specific method can be executed and graph can be zoomed for detailed observation in cases that graph occurs too small to read and observe in detail.

4. Nonfunctional Requirements

Usability

To use the application no previous knowledge of a similar tool is needed. The application is simply designed and easy to use. To begin using the application minimum introduction will be required.

Reliability

The design has high reliability requirements because the main function of the system is to analyze the code depending on the graph generated. Every inaccuracy on the graph which may be showing incorrect bound between methods, duplicate implementations or mistakes on the graphical implementation errors etc. will highly affect the usage of the application negatively. Only by assuring a highly reliable system, the application will achieve its goals and will be usable. Since the application is for individual usage and will not save any sources into the application, there are no important safety or security issues to consider.

Performance

In cases of analyzing a relatively big and complex project, the system functions may take more time to execute than usual. To keep this in minimum level, the application will be written with algorithms which has low complexity both for space and time complexities.

Supportability

The possibility of getting in touch with the developers for reporting an error or suggesting a development for the application will always be available through the application or through the source of application.

Implementation

The application will be implemented with Java language. Also, in the development process, libraries for parsing the code (JavaParser) and for creating the graphs will be used.

Interface

The project is a desktop application. It requires external Java files to be able to function. The project is aimed to be used by software developers. Therefore, user would be familiar with the graphical interface of the application.

Packaging

After the release of the project, it can be downloaded to the computers. Users are predicted to be in the areas of software development.

Legal

Basic functions of the application will be done using libraries of Java. Mainly, libraries will be used for parsing the code or drawing the graphs.

5. Object model

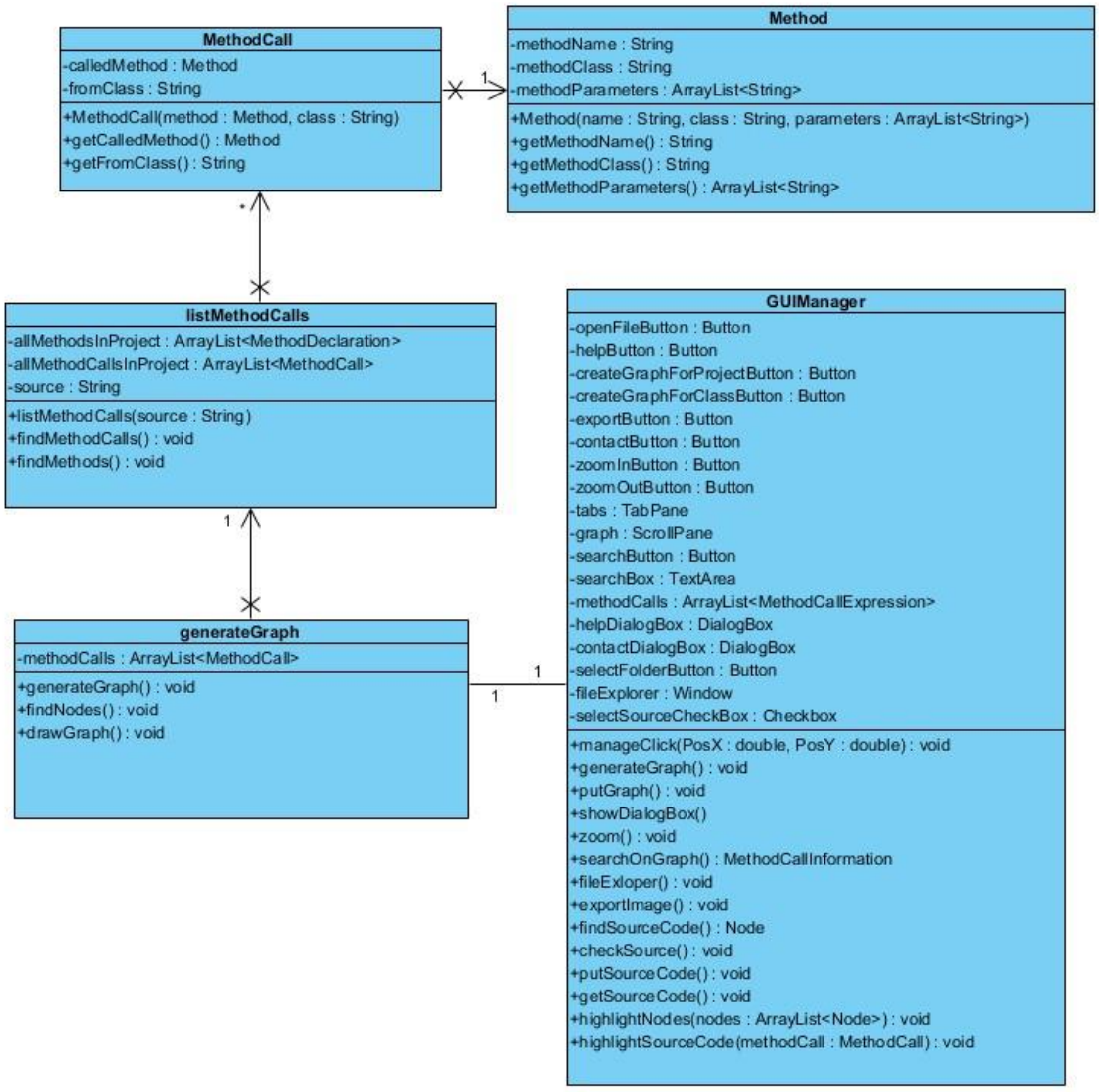
MethodCall
-calledMethod : Method
-fromClass : String
+MethodCall(method : Method, class : String)
+getCalledMethod() : Method
+getFromClass() : String
1
Method
-methodName : String
-methodClass : String
-methodParameters : ArrayList<String>
+Method(name : String, class : String, parameters : ArrayList<String>)
+getMethodName() : String
+getMethodClass() : String
+getMethodParameters() : ArrayList<String>
*
listMethodCalls
-allMethodsInProject : ArrayList<MethodDeclaration>
-allMethodCallsInProject : ArrayList<MethodCall>
-source : String
+listMethodCalls(source : String)
+findMethodCalls() : void
+findMethods() : void
1
generateGraph
-methodCalls : ArrayList<MethodCall>
+generateGraph() : void
+findNodes() : void
+drawGraph() : void
1
1
GUIManager
-openFileButton : Button
-helpButton : Button
-createGraphForProjectButton : Button
-createGraphForClassButton : Button
-exportButton : Button
-contactButton : Button
-zoomInButton : Button
-zoomOutButton : Button
-tabs : TabPane
-graph : ScrollPane
-searchButton : Button
-searchBox : TextArea
-methodCalls : ArrayList<MethodCallExpression>
-helpDialogBox : DialogBox
-contactDialogBox : DialogBox
-selectFolderButton : Button
-fileExplorer : Window
-selectSourceCheckBox : Checkbox
+manageClick(PosX : double, PosY : double) : void
+generateGraph() : void
+putGraph() : void
+showDialogBox()
+zoom() : void
+searchOnGraph() : MethodCallInformation
+fileExloper() : void
+exportImage() : void
+findSourceCode() : Node
+checkSource() : void
+putSourceCode() : void
+getSourceCode() : void
+highlightNodes(nodes : ArrayList<Node>) : void
+highlightSourceCode(methodCall : MethodCall) : void

Figure 4. Class Diagram

6. Use Case Model

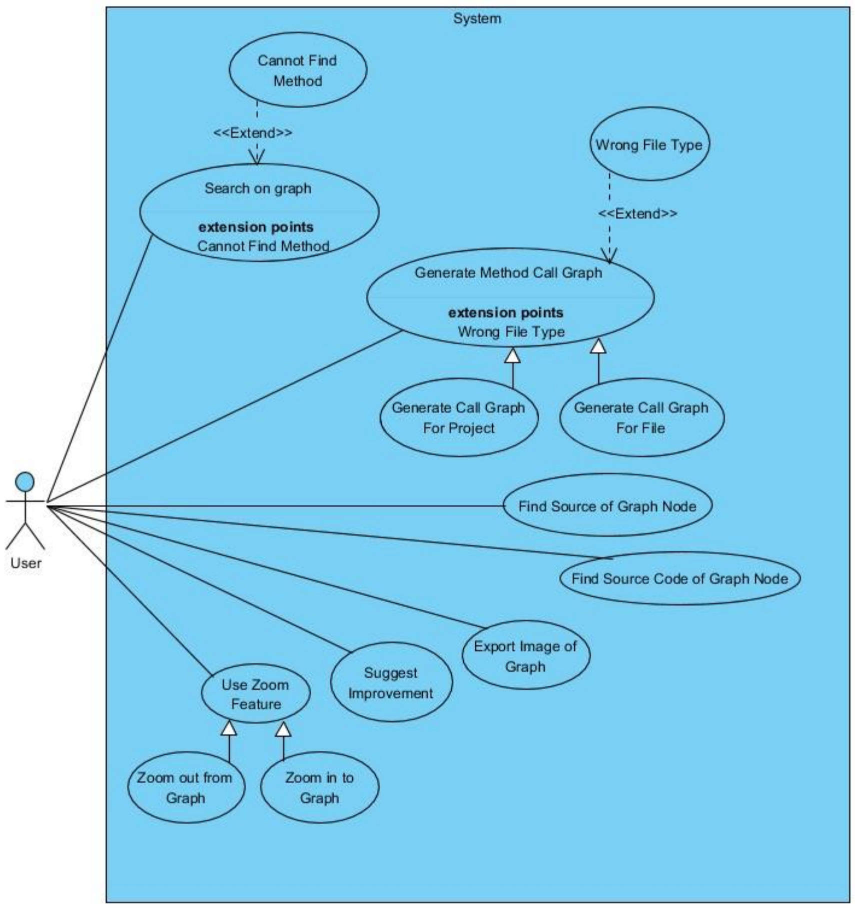

Figure 5. Use Case Diagram

7. Component Diagram

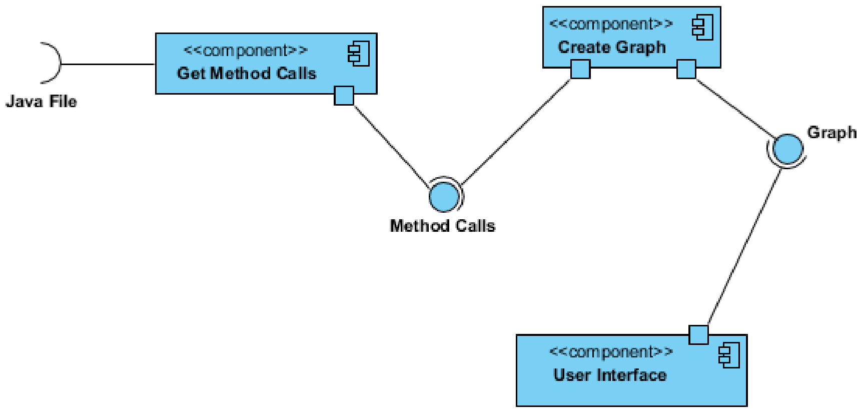

Figure 6. Component Diagram

CHAPTER FOUR

IMPLEMENTATION DETAILS

Overview

To build our own call graph, we started by investigating structure of method call, method call definitions, how does a call graph look and what should we expect from it. Possible call graph examples have been created and analyzed so we could understand what kind of call graph we plan to implement.

I needed to analyze the source code, therefore, JavaParser [6] and SymbolSolver [6] libraries have been used. JavaParser is a source code parser library for Java, as the name implies. So, we can obtain the structure of a project. SymbolSolver library solves the referenced objects by its name. With these two libraries used, method calls could be extracted from a project.

Second step, I needed to create a graph with the obtained data. For this purpose, a graph library, Vis.js has been used. Therefore, the data obtained could be rendered in the graphical user interface. GUI designed to be simple and easily understandable.

Implementation Details

To build the call graph I started with the idea of implementing Reachability Analysis algorithm. First working version of GraphGen analysis subsystem, only used JavaParser. JavaParser works by parsing a given java class and creates an abstract syntax tree which is called CompilationUnit. An abstract syntax tree represents the structure of a source code, a root node is the parsed class (CompilationUnit), then each children node represents each import, object creation, method, method call etc. So, when we obtain a parsed class, we can reach to every element in the class using this syntax tree. JavaParser provides visitor classes to get specific elements from the AST. As an example, the following class represents a visitor. Using the visit method any operation with the method call expression can be done. Therefore, I used these visitor classes to get required method and method call statements from project.

```
private class MethodCallFinder extends VoidVisitorAdapter<Void> {
     @Override
     public void visit(MethodCallExpr methodCallExpr, Void arg) {
     super.visit(methodCallExpr, arg);
     //Operations with method call expression
     }
}
```

But the parsed classes, namely the abstract syntax tree, has the elements only by their name and not by their references. For example, for a method call expression object, the object includes the name, parameters etc. of the called method but it does not know the actual method called. Therefore, even though we get the method calls, we cannot match them directly to the called methods to create a graph. For this reason, I thought of an algorithm which would match the method calls to the actual method declarations. I needed to get name, parameter list and class of each method and by mapping this information to methods, I could understand which method call statement refers to which method.

But mapping the methods was easier said than done. I investigated the AST components to get information of class and parameters of methods. I could get the class of method calls and methods, but the parameters were hard to analyze. Because parameters of a method call could be primitive types, math operations, null, object declarations, another method calls, method calls to Java classes and many more. I could not analyze every probability of a parameter.

At this point, my algorithm could implement the Reachability Analysis, but not the fully Class Hierarchy Analysis. Because CHA needs the method signature to be analyzed but I could not obtain the whole method signature.

I decided to use SymbolSolver, which is part of JavaParser library, to be able to resolve parameters. SymbolSolver creates a relation between the AST nodes, by resolving references. What resolving means can be explained with a simple example:

When we consider the expression:

```
container.element
```

We do not know what ‘container’ is. Perhaps container is a variable and element is a field of that variable. Or maybe container is a class name and element is a static field of that class. [7] SymbolSolver examines the syntax tree and resolves what ‘container’ is.

To resolve the method parameters, I decided to use SymbolSolver. My needs were fit with SymbolSolver. But I wanted to make more use of it, that I could solve the method calls with SymbolSolver.

Therefore, project structure was changed to use JavaParser and SymbolSolver for resolving method calls only, obtaining parent method and called method pairs and to use them for creating the graph.

To create the graph, I considered to use JavaFX drawings, but considering a call graph may become very complex, my drawing would not suffice my needs. I wanted to use a visualization library. Examining many options, my application needed a graph which is manipulatable in some levels. The JavaScript library vis.js [8] met my needs. I found the application called VisFX which provides access to vis.js network graph through JavaFX and uses JavaFX WebView to plot the results. [9] Consequently vis.js library was integrated to my project. Using the method calls obtained, I determined and declared the graph nodes and edges. The basic graph created by vis.js needed modifications to be able to fit GraphGen application requirements. Thus, graph settings were altered and graphical user interface were designed.

User interface of the application needed extra functions to increase usability. Because even though the graph was built to fit the needs, a relatively big project creates a complex graph which can be hard to examine. There added a search function for the nodes. Also, when a node on the graph is selected, the selected node and the connected nodes to it is highlighted to increase visibility.

CHAPTER 5
TESTS AND EXPERIMENTS

For testing the GraphGen application, following project analyzes has been done. Also, some other larger project tests were done, and the results will be included in the conclusion.

1. Sample code with simple structure

```
public class CleanEx {

    public static void main(String[] args) {
        a();
        e();
    }

    public static void a(){
        b();
    }

    public static void b(){
        d();
    }

    public static void c(){
        d();
    }

    public static void d(){
        c();
    }

    public static void e(){
        a();
    }
```

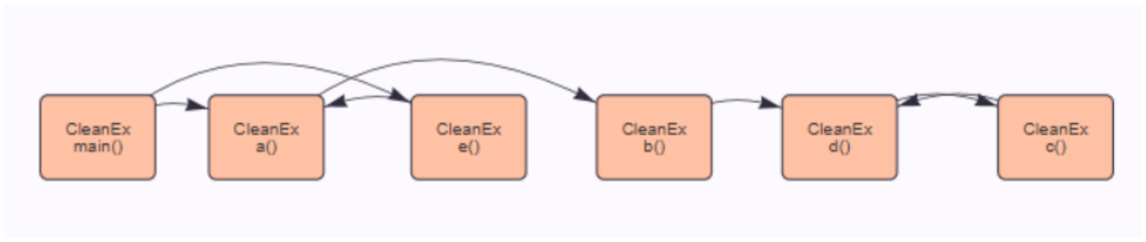

Figure 7. Call Graph for sample code with simple structure

2. <u>Sample code with class hierarchy</u>

```
public class ExampleClass {
    private static int i = Math.abs(25);
    public static void main(String[] args) {
       ExampleClass e = new ExampleClass();
       ChildExampleClass ch = new ChildExampleClass();
       e = ch;
       System.out.println(e.toString());
       System.out.println("Sum result:" + squareCalculation(i) + 5);
    }
    public static int squareCalculation(int i){
        mathFunction();
        mathFunction(5);
        return i*i;
    }
    public static int
mathFunction(){
        squareCalculation(6);
        return 5 + 3 ;
    }
    public static int
mathFunction(int i){
        return i + 3 ;
    }
    public String toString() {
        return "This is toString
method of example class";
    }
}

public class ChildExampleClass
extends ExampleClass {
    @Override
    public String toString() {
        return "This is child
example class";
    }
}
```

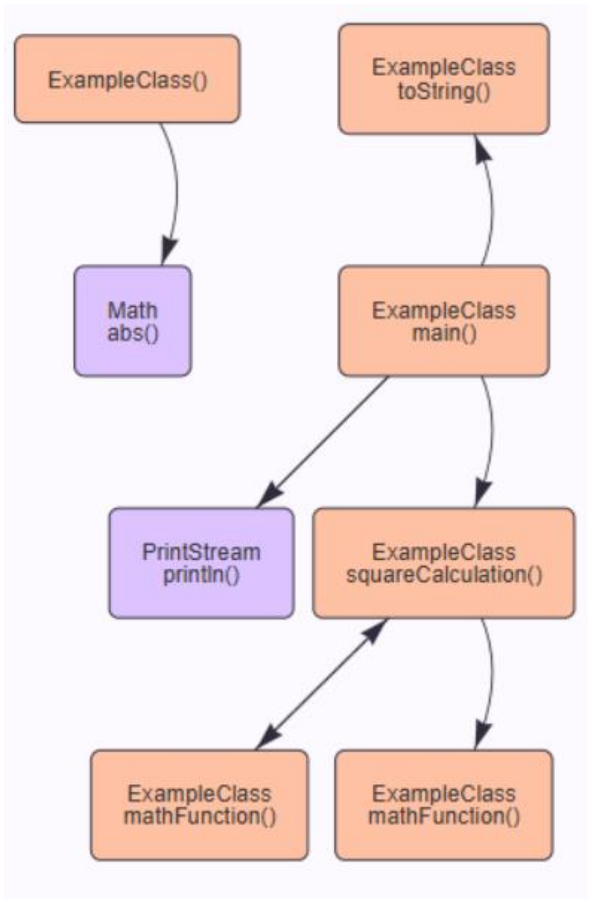

Figure 8. Call Graph for sample code with class hierarchy

3. Sample code with external library usage

```
import com.github.javaparser.StaticJavaParser;
import com.github.javaparser.ast.CompilationUnit;

import java.io.File;

public class VoidVisitorStarter {

    private static final String FILE_PATH = "…";

    public static void main(String[] args) throws Exception {

        CompilationUnit cu = StaticJavaParser.parse(new
  File(FILE_PATH));
    }
}
```

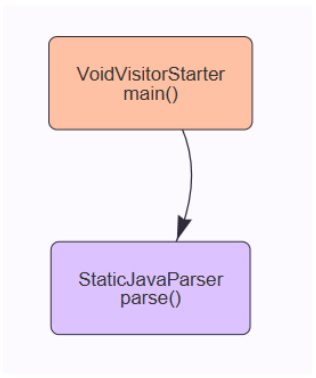

Figure 9. Call Graph for sample code with external library usage

The tests for the GraphGen application showed that, the application analyze covers the common method implementations and library methods. The JavaParser can analyze most language features up to Java 14 (work-in-progress). But still some method calls remain unsolved with the application, but they cover small number of calls in a properly analyzed project.

For the application to be able to analyze the project, project should have an expected type of project structure. It can be hard to analyze the whole project when there is a complex project structure. Larger projects may fail to analyze, depending on the design of the project.

Even though GraphGen can detect the override and overload methods, method calls for these methods may not be analyzed correctly as can be seen on Table 1. The execution time of the application is dependent on the project size and structure. On average the simpler projects with no libraries take less than a second to implement, where the larger applications take more time to resolve dependencies and execute. Following table shows average execution time of tested projects.

Project	**External Library?**	**# of Method Calls**	**# of Edges**	**# of Nodes**	**# of Unsolved Method Calls**	**Average Execution Time (milliseconds)**
Project 1	Yes	230	168	102	6	2386
Project 2	No	262	111	47	0	463
Project 3	Yes	402	324	221	2	11130

Table 1. Comparison of test results

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

The aim of GraphGen was to build a static call graph generator with the usage of JavaParser library. The application successfully can create a call graph which is easily understood, usable and modifiable. Application, when provided the required information can detect method calls with a high rate. However, the application design was not guaranteed to cover all cases of method call implementations. Therefore, the covered implementations were limited with the external sources, in most of the cases. Such as overload and override functions may not be analyzed correctly. These kind of exceptionally language features needed to be handled separately from the main flow of application. These exceptions can be included in the future versions of the application.

There was a couple of points created a problem in the implementation phase. First issue was that, some language features which cannot be covered and not being able to solve the ambiguity problem which static analyze of a source code creates. The design built for the application was not sufficient to solve the problem. For this reason, assumptions has been made for evaluation process.

Second issue was that the ambiguity of the graph to be created. Because we do not know the exact node and edge numbers or the connections between them for a graph, there cannot be a design rule for graphs. Thus, after some point complexity cannot be controlled. To solve this problem, the functions to manipulate the graph has been added and usability has been increased.

The success criteria and design goals for the application was mostly satisfied. User interface is simple and modern. The modifiable graph provides a user-friendly platform.

Application flow is easily understood, also user is provided with the information that may be needed. Even though some exceptions exist as explained in above paragraph application is considered to be reliable because user will be informed about these cases. Therefore, user will not be mis leaded.

Response time of the application is variable. Depending on the input, the application may have high response time, which lowers the usability of the program. Delay of response can be resolved with another solution for the algorithm in the next versions.

REFERENCES

1. Gousios, Georgios. “Gousiosg/Java-Callgraph.” *GitHub*, 24 Oct. 2018, github.com/gousiosg/java-callgraph.
2. Rosenberg, Gerald. “CallGraph.” *Certiv Analytics*, www.certiv.net/projects/callgraph.html.
3. Jász, Judit & Siket, István & Pengo, Edit & Ságodi, Zoltán & Ferenc, Rudolf. (2019). Systematic Comparison of Six Open-source Java Call Graph Construction Tools. 117-128. 10.5220/0007929201170128.
4. Honar, Elnaz, and Seyed AmirHossein Mortazavi Jahromi. A Framework for Call Graph Construction. 2010, http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-6629.
5. Dean J., Grove D., Chambers C. (1995) Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: Tokoro M., Pareschi R. (eds) ECOOP’95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995. ECOOP 1995. Lecture Notes in Computer Science, vol 952. Springer, Berlin, Heidelberg
6. “Tools for Your Java Code.” *JavaParser*, javaparser.org/.
7. Smith, Van Bruggen, Tomassetti, et al. *JavaParser: Visited*. Leanpub, 2019.
8. “A Dynamic, Browser Based Visualization Library.” *Vis.js*, visjs.org/.
9. Arocketman. “Arocketman/VisFX.” *GitHub*, github.com/arocketman/VisFX.