



university of  
 groningen

faculty of science  
and engineering

UNIVERSITY OF GRONINGEN

COMPUTING SCIENCE

---

# Building a dependency graph from Java source code files

Bachelor thesis

---

*Author*

Patrick Beuks

*Primary supervisor*

Prof. dr. ir. P. Avgeriou

*Secondary supervisor*

Darius Sas, MSc

July 10, 2019

## **Abstract**

Source code reuse is not as simple as it sounds. Just copy-pasting code that works in one class to another is the recipe for creating software components that are not maintainable. One of the key aspects of creating reusable software components is properly managing their dependencies on other classes and packages.

There are programs that allow for finding these dependencies, but none of them work on the source files of the software components. This thesis tries to remedy this.

In here background information on Java is given, together what source code and compiled code means. There is also an explanation on what graphs and dependencies are and what the program Arcan is.

The design of the program is handled where each part is explained on how it works. This includes how the dependency graph is formed and how this is parsed from the source files. It is also explained how this program implements and uses Git and how the program can be run from the command line.

The results of the program are mentioned and explained. The results are compared with output that is known and what the performance of the program is like.

A conclusion is provided with a look on how well it performed and how the complexity is managed.

This thesis ends with mentions of what work can be done in the future to improve on this thesis. How it can be extended by being able to include methods in the graph. How the parser can be changed to be able to increase the performance of the program and how improvements to the CLI can increase the usability.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Java . . . . .	3
2.2	Source code vs. compiled code . . . . .	3
2.3	Graphs . . . . .	3
2.4	Dependencies . . . . .	4
2.5	Arcan . . . . .	4
<b>3</b>	<b>Design</b>	<b>5</b>
3.1	Graph . . . . .	5
3.2	Code parsing . . . . .	6
3.3	Git . . . . .	8
3.4	CLI . . . . .	10
<b>4</b>	<b>Results</b>	<b>12</b>
4.1	Reference results . . . . .	12
4.2	Performance . . . . .	13
<b>5</b>	<b>Conclusion</b>	<b>14</b>
<b>6</b>	<b>Future work</b>	<b>15</b>
6.1	Method parsing . . . . .	15
6.2	Parser . . . . .	17
6.3	CLI improvements . . . . .	17
	<b>Bibliography</b>	<b>18</b>

# Chapter 1

## Introduction

Since the early days of programming, maintenance has always been a primary concern for engineers. Java is a high level Object Oriented (OO) programming language that allows you to create objects, making it possible to have a more abstract structure [Jan17]. These objects make it easy to structure your code and change components with each other for the purpose of adapting to new functionality or fixing bugs.

It is possible to measure the quality of Object oriented design (OOD) by looking at its robustness, maintainability and the re-usability. This is done by finding the dependencies that the objects have and which objects depend on them [Mar94]. To calculate metrics on these properties the program has to find these dependencies and label them. This can be done with a compiled java program (JAR file) and already exists [Fon+17][Tes01].

Having to depend on a compiled program brings a few problems. First it means that you need to have a compiled program or compile it yourself, but this is not always possible. Compiling it yourself can also take a lot of time. This is on top of the fact that for finding dependencies the program does not need to run. This means that the program already has all dependency information at the moment of compiling.

This thesis tries to solve this problem by proposing a tool to mine dependencies from the source files of a program. Source files are plenty available, one of the largest being Github with over 100 million repositories [Git19].

This thesis starts by providing some background information on the technologies used later on. It is continued by explaining how the program works, this is separated into four sections, each going into detail how it is implemented. Next are the results of the program and what they mean, followed by the conclusion. Lastly this paper will go into future work that can be done to improve the program.

# Chapter 2

## Background

As mentioned in the introduction, it is currently possible to create a dependency graph from a compiled Java program. But what is Java and what are the differences between a compiled program and source files. Also what are graphs and dependencies. In this chapter a program called Arcan will be presented, its purpose explained and what it means for this project.

### 2.1 Java

“T[he] Java programming language is a general-purpose, concurrent, class-based, object-oriented language.”[Gos+19, p. 1]. Java is the most popular programming language in the world [TIO19]. It is a programming language that allows someone to make a program, compile it and it will run on any computer with Java installed. This is different from most other programming languages where for each computer system you need to compile a version for it.

### 2.2 Source code vs. compiled code

When programmers write code, they do this generally in a programming language, in this case, Java. This code is called the source code of a program, but this code is not what the computer runs. For this the source code first needs to be converted to code that the computer can understand. This converting is called compiling and the result is machine-runnable code.

### 2.3 Graphs

Graphs are networks of vertexes, connected by edges. The graphs used in this paper are all directed graphs, meaning that each connecting edge has a specific direction from one vertex to another. In figure 2.1 you can see an example of a graph. Each circle represents a node and each arrow an edge.

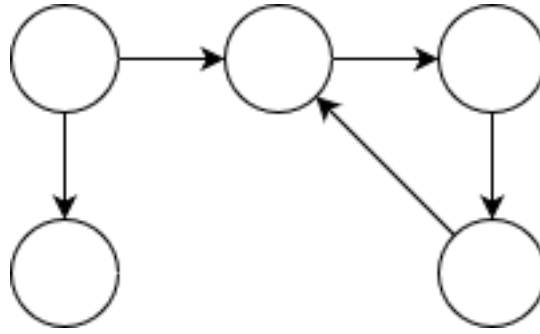


Figure 2.1: Example of a graph

## 2.4 Dependencies

Programs that are created do not often stand on their own, they use code from other libraries and from other parts of the program. This is so that the programmer does not have to write the code again. This usage of other code is called a dependency and this thesis focuses on mapping out these dependencies into a graph. Dependencies allow the programmer to not have to write code again, but rather reuse it. Nonetheless, it also means that if the dependency changes, the program also has to change.

## 2.5 Arcan

In this thesis project the program Arcan will be mentioned multiple times. Arcan can use compiled Java code to generate a dependency graph and calculate metrics on them. This project can be considered an addition to Arcan, where instead of having to use compiled code, it can use source code to create the dependency graph. Which will then be used in Arcan to compute the metrics and detect architectural issues. Because Arcan needs to understand the graphs created by this program, the tool created in this projects needs to output a dependency graph as close as possible to the one generated by Arcan starting from JAR files

# Chapter 3

## Design

This project had a few requirements as well as an extended goal. The program created for this thesis should be able to create a graph from Java source files. The extension would be that it is able to create multiple graphs from the history from Git.

### 3.1 Graph

The output of the program is very strict, as a requirement is, that it is the same output as that of Arcan. The technology behind the graph also needed to be the same, namely Apache TinkerPop. This is a graphing framework that allows for a lot of flexibility in its usages.

Apache TinkerPop can be used in many different programming languages, but therefore also does not use some of the features that Java offers, this being the ability to use object oriented design. Because of this, the project uses an additional library called Ferma. This additional library is an abstraction layer between TinkerPop and Java. Which gives the ability to use Java objects as vertexes and edges in the TinkerPop graph. This also allows for adding functions on those vertexes to manipulate the data before putting it on the vertexes and edges of the TinkerPop graph.

The structure of the code is setup in such a way that every different type of vertex and edge is its own object in the project. The goal of the program is to build a graph where classes and packages from the source files are the vertexes in the graph. Because classes and packages have different purposes, they are put in there own Java object class. Then for each class and package the program finds in the source code that it is parsing, the program creates a new instance of a vertex, either a `VertexClass` or `VertexPackage` respectively.

### 3.2 Code parsing

With a way to create vertexes and edges in the graph, the second part of the requirements is that the program can parse Java source files so it can find the different Java classes and packages. Because Java is a complicated language, it is out of scope to write something from scratch. Spoon [Paw+15] was recommended to be used in the program. Spoon is an open source library useful analyzing and transforming Java source code.

With Spoon the program is able to parse the Java source code given to the program. The program implements processors that denote what we are looking for in the source code, in this instance they are the classes of the source code. Every time Spoon finds a class in the source code it will send the class to the processor where the program can use and manipulate it.

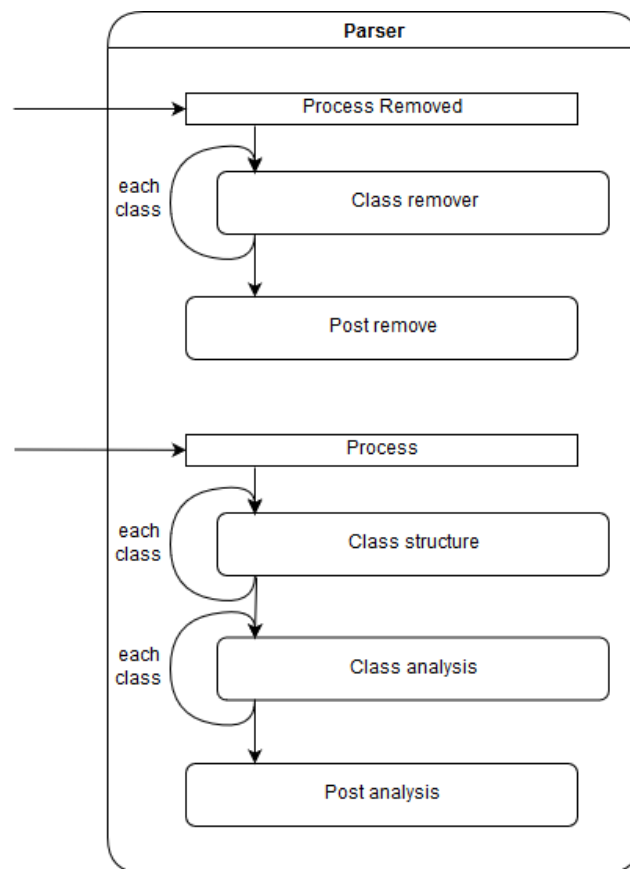


Figure 3.1: Flow of the parser

The program uses two or three steps to process the source code, depending on whether it needs to take a remove step. The other steps are the structure step and the analysis step. To set all information up and set the right processors the program uses a **Parser** class. This class remembers the settings for a project and holds and sets the right information. After all classes have been processed by Spoon an additional post process step is taken to add more data on the vertexes or to clean information up.

The remove step is ignored in this section and will be explained in section 3.3. The



structure step is used to map out all classes and the packages they belong to. The classes created in this way get a special property on their vertex denoting that they are a `SystemClass`. This is according to the Arcan specifications.

Now that the structure is known and the classes are mapped out, the next step is executed. This is the analysis step. This step follows loosely the steps that Arcan executes. For each class given by Spoon it adds an edge for being a `childOf` a super class if the current class extends from it. It then checks if the current class implements interfaces. For each of these an edge `implementationOf` is created. Lastly it finds all references to other classes the current class has. For these `dependOn` edges are created.

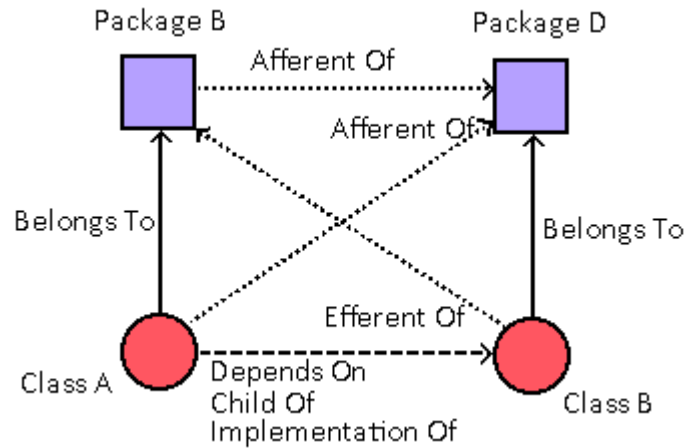


Figure 3.2: Structure of the dependency graph

After the analysis a post process step is executed. It first checks if there are class vertexes with no edges. This is possible when the remove step removed class vertexes that had pointed to the given class before but was removed. After this `afferentOf` and `efferentOf` edges are added for each edge found in the analysis step. In figure 3.2 you can see the relationship between the edges. Every class belongs to a package, this is the solid line. A class can then depend on another class, be a child of or be an implementation of a class as seen by the dashed line. When such a relationship exists between classes, the afferent and efferent edges are created in the post analysis step, seen as the dotted line.

Some of the logic that Arcan had in their implementation of analysis has been moved to the class vertex itself. In particular two actions have been moved. The first one is that when adding an `afferentOf` edge the class checks if it also needs to add an `afferentOf` edge between package vertexes if this does not yet exist. The second addition is that when a `dependOn` edge is created with a given class, it is checked if this edge already exists. If so the weight of the found edge is increased instead.

### 3.3 Git

Git is a way for programmers to keep track of the history of changes that are made to the source code of a program. This gives the ability to track what has changed and when needed, the programmer can go back to a previous state.

For this thesis, the program needs to be able to use Git. It has to handle a Git repository and step through the history of the given repository, after which it gives an output graph. Git is an open source version control system that is widely used in programming projects in order to share code, work together in a team and keep track of history. It starts by opening a git repository, called cloning. After this it is possible to edit the files of the project.

Git keeps track of changes made to files in the repository. When changes are made, the files can be saved into what is called a commit. In here it is described what the changes are that were made. These commits get special IDs from Git to be able to reference it later, these IDs are used by this program. When the commits are made they can be pushed to the original repository. There are more possibilities with git, but these are not in the scope of the project.

What this feature provides is a way to not have to download an entire project for each graph it needs to create. With this it can calculate the differences and only parse the graph for those differences.

To implement Git in Java this program uses Jgit, a Java implementation of the open source version of git by the Eclipse foundation [Soh+]. This gives the ability to use Git within the program itself. The program does not directly use the history, instead it uses commit IDs from specific commits you can input in it.

The implementation is first given an URI. This is comparable to URLs you would see in browsers, but URLs can only point towards web addresses, while URIs can also point towards files and remote servers. This way it is possible to either open a repository from the internet or one from a local system. This repository is then cloned to a temporary directory on the computer.

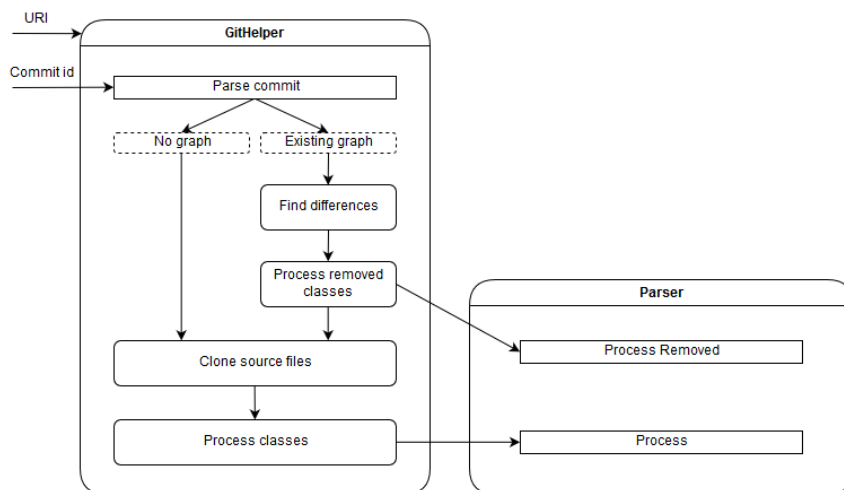


Figure 3.3: Flow of the git helper

After the clone is made the program is given a commit ID that needs to be parsed. Git will then set the repository up to have the same files as when that commit was made. At this point a complete parse of the graph will occur as described by the previous two sections. At this point no speedup has yet occurred.

It is now possible to give it another commit ID. A list of differences will be calculated between the given commit ID and the previous commit ID. This list is split in three lists for later use. A list with all removed files, a list with modified files and a list of added files.

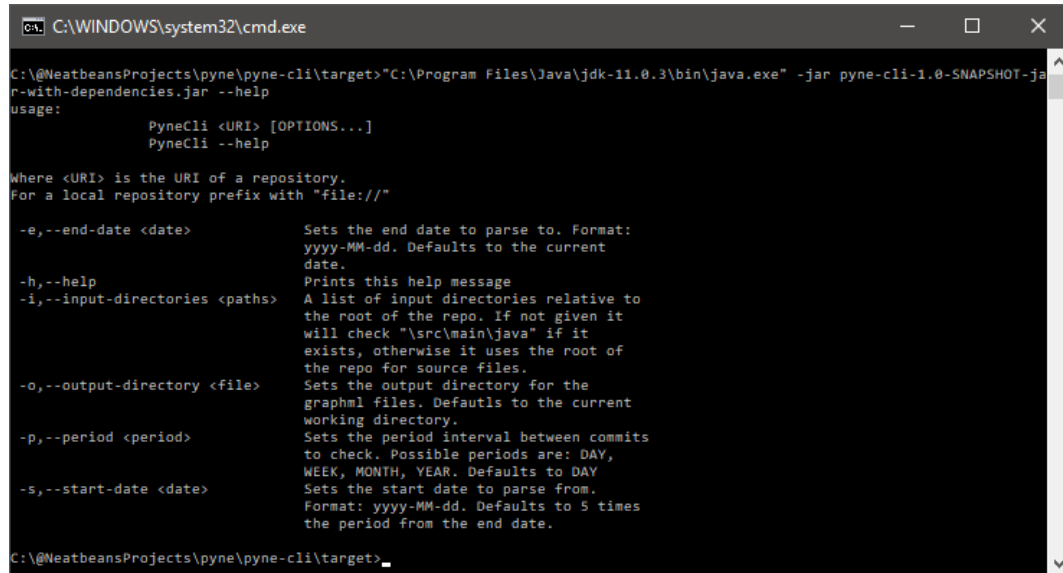
Before Git does anything else the remove step from the code processing is called, the step that was previously skipped. With the lists of removed files it goes over all classes of the program. Since these are still the old files it will include the files that were removed. When a class is found from a file that was removed it will remove that vertex and all edges from the graph. The same is done with all modified files, but instead of removing the vertex only the edges are removed. This is so that the program is able to use the vertexes again when parsing the files to be added.

A post remove process will occur at this point to remove all efferent and afferent edges as these are recalculated at the post analysis step again. It is done in this way because it is not trivial to know if a afferent or efferent edge still exists. This is because only one efferent and afferent edge are created between an class vertex and a package vertex.

After the classes are parsed again the program uses Git to check out the files of the given commit ID. Git only changes files that are changed, so this step is often quick. The code parsing is called again and will check for each class if the class being parsed is one that was added or modified. Only if this is true it will be parsed by the program. This drastically improves speed as often between commits only a small portion of the files are changed.

### 3.4 CLI

The program is now able to generate a dependency graph from a Git repository. In order for the end user to be able to select which repository to parse it is important that they are able to enter the information. For this the program uses what is called a CLI. A CLI or command line interface, is a way to interact with the program from a special text window available on almost all computers.



```
C:\WINDOWS\system32\cmd.exe
C:\@NeatbeansProjects\pyne\pyne-cli\target>"C:\Program Files\Java\jdk-11.0.3\bin\java.exe" -jar pyne-cli-1.0-SNAPSHOT-jar-with-dependencies.jar --help
usage:
    PyneCli <URI> [OPTIONS...]
    PyneCli --help

Where <URI> is the URI of a repository.
For a local repository prefix with "file://"

-e,--end-date <date>      Sets the end date to parse to. Format:
                           yyyy-MM-dd. Defaults to the current
                           date.
-h,--help                Prints this help message
-i,--input-directories <paths> A list of input directories relative to
                           the root of the repo. If not given it
                           will check "\src\main\java" if it
                           exists, otherwise it uses the root of
                           the repo for source files.
-o,--output-directory <file> Sets the output directory for the
                           graphml files. Defaults to the current
                           working directory.
-p,--period <period>      Sets the period interval between commits
                           to check. Possible periods are: DAY,
                           WEEK, MONTH, YEAR. Defaults to DAY
-s,--start-date <date>    Sets the start date to parse from.
                           Format: yyyy-MM-dd. Defaults to 5 times
                           the period from the end date.

C:\@NeatbeansProjects\pyne\pyne-cli\target>
```

Figure 3.4: The help interface of the CLI program

In the CLI it is possible to enter arguments that are used at the same time as the program is executed. When the program is executed it needs to know what repository the program needs to use. So when executing the program, it is mandatory to supply the program with an URI to a Git repository.

It is also possible to give a few optional arguments when the program is ran. The first ones being the date range of Git history that needs to be put into a graph, by supplying a start-date and end-date. It is also possible to supply an period, as it is often not needed to parse all commits between each date since they are often not that different. A third option is to give an output directory for where to save the graphs in. The last option to give is a list of input directories from the Git repository that contain the source files. As it often spread out over different folders and difficult for a program to determine what are actually source code files and what files are not.

As the program is implemented to this point it cannot handle the history of a Git repository, it can only have commit IDs as input. So the CLI needs to be able to convert the history from the repository into commit IDs so that the program can handle the input. It will first get a list of commits between the given start-date and end-date. This will be ordered into a list by date. The program will then take the first date and parse it. After this it will check if the next commit date is smaller then the given period and skip it until it finds a date that is after the given period and parses that commit. This will continue until the end date is reached.

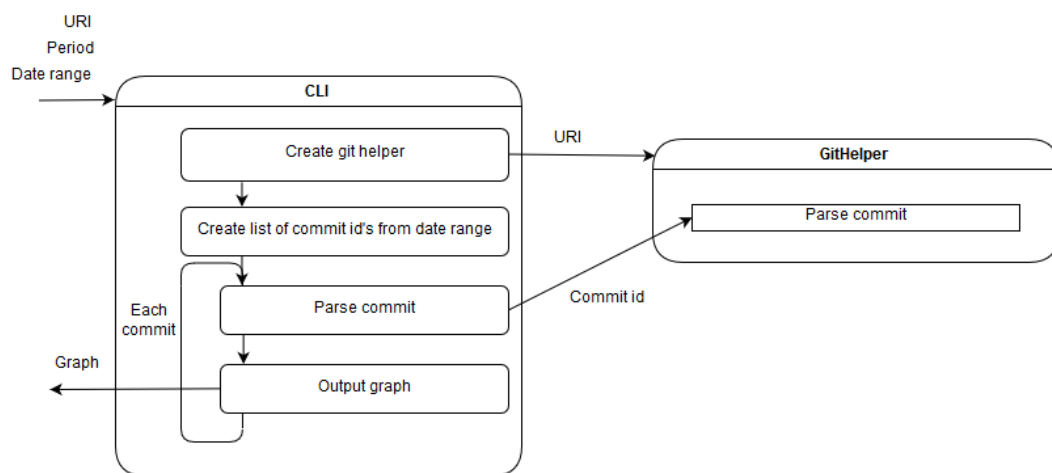


Figure 3.5: The flow of the CLI program

# Chapter 4

## Results

With the design of the program explained and created it is necessary to be able to say something about the results that the program generates. In this chapter two types of results are measured. It is measured against an reference graph and the performances are measured to see how fast the program is.

### 4.1 Reference results

To be able to test the results of the program a graph created by Arcan was given to compare. The program needed to be able to replicate this. It was soon discovered that it is very difficult to reverse engineer these graphs, as they were large enough that it is difficult to compare by hand. The first results of the program could almost not be compared with the reverence files and the differences that where found could not give information on what went wrong. After investigating why this was, it was discovered that Arcan only uses specific references to classes that it uses as dependencies.

After inspecting the source code for an older version of Arcan, that is publicly available, it was possible to implement a reference parser. In this reference parser the only changes that were made is that it uses Spoon objects instead of the class object Arcan uses. This gave a good baseline for what the program had to aim for.

The reason for not using the reference file but still implementing the project separately where two fold. The first is that Spoon while able to mimic the structure Arcan used it was not optimal for it. Secondly was that since the program uses a structured approach to vertexes and edges, the logic for what happened when an edge or vertex is added could be implemented in the vertexes and edges themselves.

The program was changed until the results where the same as the reference file for a multitude of projects<sup>1234</sup>. After this the file was removed as this contained code that was copied from Arcan. While express permission was given to use the code, it did not seem right to keep this file as it was used for the program itself only as a point of comparison.

---

<sup>1</sup><https://www.antlr.org/download/index.html> (antlr-4.0.tar.gz)

<sup>2</sup><https://github.com/antlr/website-antlr3/tree/gh-pages/download> (antlr-3.0.tar.gz)

<sup>3</sup><https://github.com/killje/XPStorage/commits/master> (commit ba27ddc)

<sup>4</sup><https://github.com/apache/commons-io/commits/master> (commit e921bc6)

## 4.2 Performance

A goal for this project was to be able to parse Java programs without having a compiled version or having to compile them before parsing. As a consequence of this, it was possible to do the parsing from a git repository. An extension for this program then was to be able to create graphs for different versions of a program using the history from Git.

An additional benefit of doing it this way is that commits are done way more often then releases of a program. This gives a chance to have more insight of dependency changes between commits, giving the ability to spot problems early on.

In order to do these graphs for multiple commits it is important that it happens fast. The program was adapted to be able handle Git commits and only update the parts that had changed between commits. This greatly increases performance as changes often only happen to a very few files, compared to the total amount of files there are in a repository.

Method	Time run	Commits parsed	Time to parse a commit
<b>Git</b>	3 minutes 46 seconds	517	~ 0.44 seconds
<b>Reparse</b>	15 minutes 41 seconds	101	~ 9.32 seconds
<b>Build</b>	31 minutes 58 seconds	21	~ 1 minute

Table 4.1: Run times for different methods of parsing

Table 4.1 shows the results of a demonstration program that is included with the project. The demo program uses a large repository [ComIO] and takes a given amount of commits to parse. The demo that runs is the Git implementation to only update the dependency graph for the changed files. The second one is having the program parse the complete graph from scratch each time and the last demo builds the given project to simulate what Arcan would need to do, to be able to create the graph. A note on the last demo is that the graph is still created from the source files, but changes in time that are visible here, are because of building. The parsing time is negligible compared with the other results. As can be seen the demo with the Git implementation, it is considerable faster then creating a dependency graph from scratch.

## Chapter 5

# Conclusion

The goal for the project was to create a Java library that is able to generate a dependency graph from Java source code. It should also have a command line tool to interact with the library. It has also an optional requirement to be able to use Git and the history from git to be able to have dependency graphs over a time period.

To this end the project has succeeded in doing this. The program can make the graphs needed as a library and can be both extended or incorporated as is. It has a CLI that allows you to generate multiple dependency graphs given a time period. For that end it uses git to be able to get the history of a program.

With the help of external libraries like TinkerPop[TinkPop] and Ferma[Ferma], it turned out that creating the graph was not difficult. Being able to use the strength of Java objects with Ferma to create the vertexes and edges was a good thing to have. Allowing to separate code that parses the source code and the part that generates the graph, reducing the overall complexity.

Using Spoon[Paw+15] also allowed to be able parse the source code. Without it this project would not have been possible to create. Having JGit[Soh+] was also a huge help, while it possible to use git console within the program it would have been cumbersome to implement it.

The project is available on Github<sup>1</sup>. This includes the API that parses and creates the graphs. A CLI that implements the API and offers the ability to generate a program that creates graphs and can be executed from the command line. Lastly it includes a demo program that implements the API in order to run the three demos. The results of these demos can be found in section 4.2.

---

<sup>1</sup><https://github.com/darius-sas/pyne>



# Chapter 6

## Future work

While the project, together with the Git implementation, is very fast. There are still possibilities to increase the speed and expand the usage of the program.

There are three areas in the program that can be extended in the future to increase the speed and usability. In this chapter these areas are explained, why they are not in the program yet and how they can be implemented. The areas for improvement that have been identified are the ability to parse methods, using a different parser and CLI improvements.

### 6.1 Method parsing

A possible optional feature for this program was to add methods to the graph and including corresponding edges. This was at the start of the project also planned to be included in the program, but was later removed. The reason that it was removed was that the Arcan version available online did not have method parsing included yet. This online version was instrumental in checking the validity of the output of the graph.

When the first graph could be parsed from source code it was discussed and concluded that using Git was at that point more important to implement then to implement methods in the graph. The Git implementation is a large chunk of the program and took time to research and implement into the program and did not leave time to also implement method parsing.

That said, the program is made so that it is easy to implement the parsing of methods. All changes that need to be done are inside the API part of the project. The first step would be to add a method vertex to the structure together with all edges that are wanted. The second step would be adding adding method parsing. This would probably be done inside `ClassAnalysis` as methods are already parsed here. Thirdly would be adding required logic to post processors to add and remove edges that are only known after the parsing. See also figure 6.1.

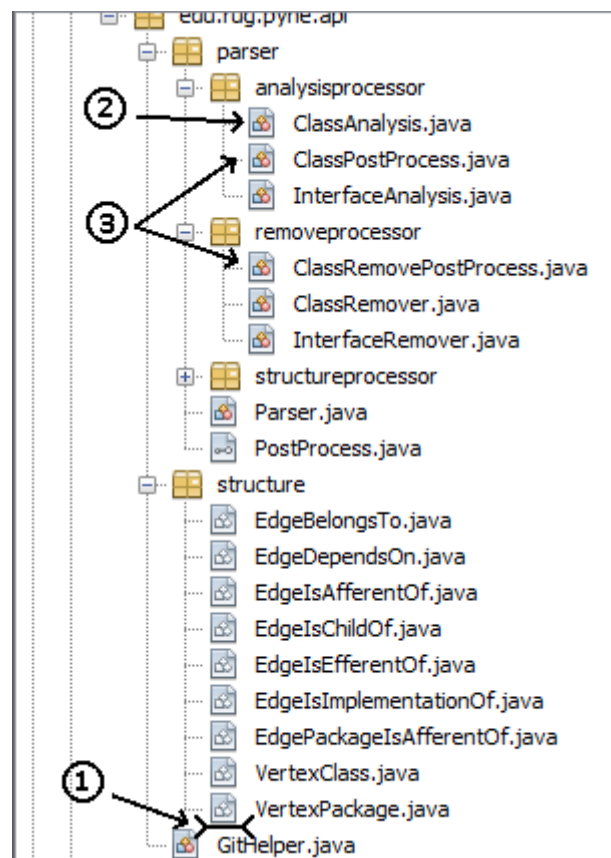


Figure 6.1: Steps to parse methods

## 6.2 Parser

When the project started out spoon seemed like a very good library to use. It is a very up to date library with lots of features. One of which was to get all references to other classes from a given class. This seemed to be perfect for getting the dependencies. Other libraries have been considered as well, but none had the ability to give all references so it was decided to use that one. As the project continued it became clear that it needed to be very similar to the Arcan project and the output should be the same. This threw a wrench in the ability to use the references from Spoon as they were ever so slightly different from Arcan. So the references finding from Spoon was removed, so the program could be more similar to Arcan.

Now that it is not needed to get all references from Spoon, it is possible to use a different parser. The reason to do this is that Spoon implements a full compiler. This gives a lot of overhead. There are libraries that can do the parsing way quicker as they do not include this compiler. In the research for the project one such library was found that could possibly fit the project called JavaParser.

To be able to implement a different parser into the project, one would need to replace Spoon in the parser package from the API. As most logic for edges has been moved to the `ClassVertex` the actual code that needs to be replaced is minimal. Most of the parsing comes down to getting the class name, class package and class file. The biggest task would be to get the references that are calculated in `ClassAnalysis`.

## 6.3 CLI improvements

The last area for improvement would be the command line interface (CLI) of the program. The CLI was made so that it was able to be given a start date, end date and a period. This would then calculate all commits between those dates, it would then take the first commit and parse it. It would then skip commits until the commit date is a period away from the previous commit.

This way of doing periods is not optimal. For example, if you have a period of one day and it parses a commit that has been made at 13:01 and there is a commit the next day at 12:59 it would not parse it, since not a complete day, that means exactly 24 hours, has passed.

There are two ways of solving this problem. One would be easier to implement and another would be more difficult but has nicer results. The first solution would be to make a special case for the day period and set the time for the next commit to 00:00 of that day, so the first commit in that day will be parsed. A second way of doing things is to take the last commit in a period time frame. This would split the commits in distinct time frame. Let's take as an example a period of a week. If you have a commit on Friday one week and only a commit on Tuesday the next week, it would still be a good thing to parse that commit for that week.

A second idea for the CLI is to be able to load in a configuration file. This file would include all relevant information to be able to generate the dependency graphs. The reason for implementing such a feature is that for a given project the arguments needed to run the CLI are often the same. So the ability to save this configuration would give the ability to run the program on a later date with the same configuration without having to enter them in again.

# Bibliography

- [Jan17] Thorben Janssen. *OOP Concept for Beginners: What is Abstraction?* 2017. URL: <https://stackify.com/oop-concept-abstraction/> (visited on 07/05/2019).
- [Mar94] Robert Martin. “OO Design Quality Metrics. An Analysis of Dependencies”. In: (1994). URL: <https://www.cin.ufpe.br/~alt/mestrado/oodmetrc.pdf> (visited on 07/05/2019).
- [Fon+17] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. “Arcan. A Tool for Architectural Smells Detection”. In: (2017). DOI: [10.1109/ICSAW.2017.16](https://doi.org/10.1109/ICSAW.2017.16). URL: <https://boa.unimib.it/retrieve/handle/10281/155470/221227/PID4705339.pdf> (visited on 07/05/2019).
- [Tes01] Jean Tessier. *Dependency Finder*. 2001. URL: <http://depfind.sourceforge.net/> (visited on 07/10/2019).
- [Git19] Github. *About*. 2019. URL: <https://github.com/about> (visited on 07/05/2019).
- [Gos+19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java<sup>®</sup> Language Specification*. Java SE 12 Edition. 2019. URL: <https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf>.
- [TIO19] TIOBE - The Software Quality Company. *TIOBE Index*. 2019. URL: <https://www.tiobe.com/tiobe-index/> (visited on 07/03/2019).
- [Paw+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”. In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179. DOI: [10.1002/spe.2346](https://doi.org/10.1002/spe.2346). URL: <https://hal.archives-ouvertes.fr/hal-01078532/document>.
- [Soh+] Matthias Sohn, Andrey Loskutov, Christian Halstrick, Dave Borowitz, David Pursehouse, Gunnar Wagenknecht, Jonathan Nieder, Jonathan Tan, Matthias Sohn, Sasa Zivkov, Terry Parker, and Thomas Wolf. *Jgit*. URL: <https://projects.eclipse.org/projects/technology.jgit> (visited on 07/05/2019).
- [ComIO] The Apache Software Foundation. *Apache Commons IO*. 2019. URL: <https://github.com/apache/commons-io> (visited on 07/05/2019).
- [TinkPop] The Apache Software Foundation. *Apache TinkerPop*. 2015. URL: <https://github.com/apache/tinkerpop> (visited on 07/05/2019).

[Ferma] Jeffrey Phillips Freeman. *Ferma*. Syncleus. 2004. URL: <https://github.com/Syncleus/Ferma> (visited on 07/05/2019).