CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

# Clustering of software modules using Jaya algorithm

Jakub Pavlát

August 2020

# Acknowledgement / Declaration

There are many people I would like to thank, because while the thesis has my name on it I could not have finished it without the support of people around me. In no particular order I would like to thank my supervisor Doc. Ing. Miroslav Bureš, Ph.D. who took a big risk and took this project, and me with it, on. A special thanks also belongs to Bestoun S. Ahmed, Ph.D. who is the co-author of MS-Jaya and thanks to whom I was able to find a genuinely interesting topic for my thesis. A big thanks to my family, related or no, who made sure I had all the support to research, work and write.

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Diploma Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

# Abstrakt / Abstract

Tato diplomová práce se zaměřuje na použití Jaya algoritmu k řešení problému clusterování softwarových modulů. Tento problém se dá popsat jako hledání clusterování modulů tak, aby uvnitř clusteru byly vazby co nejsilnější a směrem ven z daného clusteru co nejslabší. Cílem této práce je vývoj nástroje, který bude pracovat s existujícím Java kódem, ve kterém identifikuje problém clusterování a bude ho řešit. Řešení problému clusterování softwarových modulů je možné využít především v oblasti kontroly kvality kódu a managementu. Proto by takový nástroj měl vytvářet lidsky i strojově čitelný pohled na řešení.

This diploma thesis focuses on the usage of Jaya algorithm for solving the software module clustering problem. The problem can be described as searching for a clustering of modules such that links between modules inside a cluster are many and strong and any links going outside a cluster are sparse and weak.The goal of this work is developing a tool that will work with existing Java code, in which it will identify the clustering problem and will attempt to solve it. Solving such a problem can be beneficial predominantly in the field of code quality assurance and team management. As such the tool should offer a human and machine readable view of the solution.

# Contents /

# Chapter 1
# Introduction

## 1.1 Overview

Software development can be described as a set of activities with the aim of creating, designing, deploying and supporting software [1].

When the software in question is of considerable size - meaning bigger than a hobby project you work on occasionally at home - usually more than one person is involved, even if we are not talking about a trending project on GitHub or BitBucket with thousands of contributors.

Different people have different opinions about the ideal number of developers working in a team on a project. There are even companies that specialize in forms of crisis management for software development and take a more scientific and mathematical approach to analyzing the ideal number of developers [2] than the usual rule-of-thumb advice you can find in online forums.

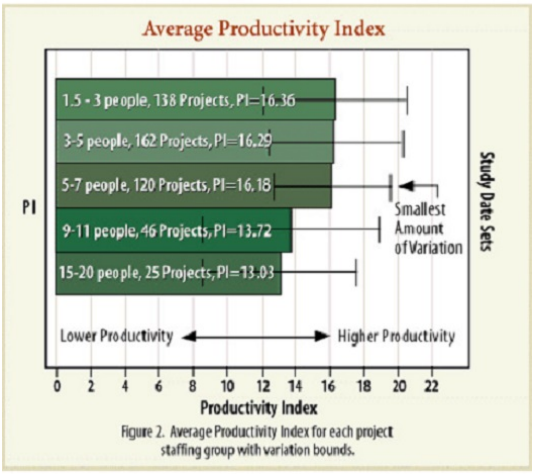


**Figure 1.1.** Average productivity matrix describing ideal team sizes for software development. [2]

In my short career as a software developer as well as my academic career I have been a part of projects that had multiple developers with the same focus - be it backend, frontend, UX and so on. Just as I have, I am sure most software developers have been witness to a situation where one task has been unknowingly worked on by two people. Without any real data - as far as I can tell no one keeps public records of how often this happens - I do not want to presume that this causes a real issue and is a reason for falling behind schedule.

I do think however that it points to a fairly important underlying issue. Take for example the RACI matrix - a common tool in project management - which connects people or teams with tasks through different relations - one of which is „Responsible“[3].

Usually the relation „Responsible“ in software projects is *new application feature* and *development team* [4][5].

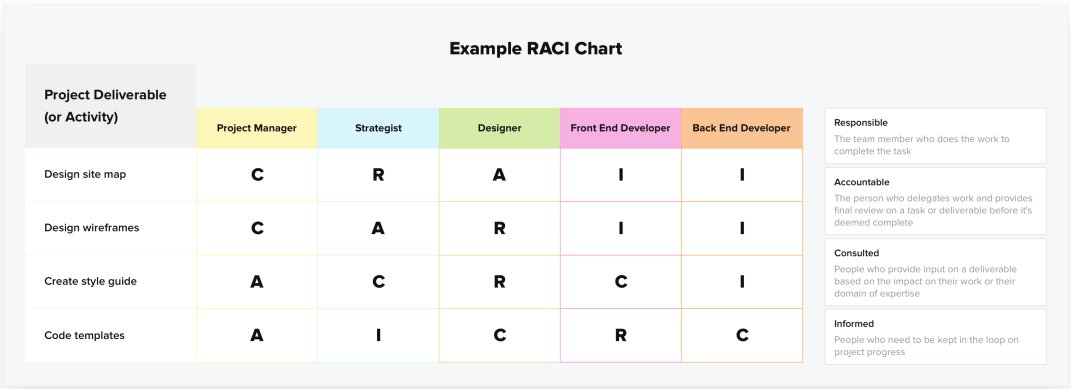


**Figure 1.2.** RACI matrix example [5]

The problem I see here are the roles. Often the roles in which developers are hidden are titled along the lines of „BE developer“, „FE developer“, „SCRUM team“ and so on. What I see as a big challenge however comes next - assigning the task within the role or team. Assuming a large enough project there will be more than one backend developer. Therefore there is the problem of deciding which programmer deals with the specific task.

This is not supposed to take away from the importance or efficiency of the RACI matrix, but rather point to a problem that emerges after using it.

I believe that looking at issues for popular open-source projects also shows a part of this issue. Whenever an issue is opened it does not get assigned right away. What you can often see is a contributer comment and tag another contributor or simply assign the issue. This is done manually or with the help of bots. The process is however error prone as you can usually see multiple reassignments for one issue.

This gets me to the topic of this thesis. The goal is to be able to analyze software and suggest how to divide it in a way that would benefit development. Beneficial for development would be if individual developers were responsible for code that is functionally related. That way developers can decouple even within one codebase. Also based on how the ideal division one could imply things about the quality of code - if the code becomes harder to divide it may be a sign of it becoming „spaghetti“.

I will be creating a tool that will operate on a Java codebase. It will read Java code and based on the particular code structure, such as method calls, object instantiations, field accesses and imports, will create an instance of a software module clustering problem. Using Multi-Start Multi-Objective Jaya the tool will then attempt to solve the problem - separating different modules into clusters. While the criteria for partitioning these modules can vary, I will be focusing on the software related coupling and cohesion. Simply put I will be trying to find clusters that have strong links with modules in the same cluster and weak links with modules outside it.

The pipeline of tasks the tool should do is as follows:

- read/scan Java code (bytecode)
- create a module dependency graph (MDG)
- render the MDG

- run Multi-Start Multi-Objective Jaya to approximate the solution
- render the MDG now with clusters

Based on these tasks I will divide the implementation into three main modules which correspond to chapters in this thesis. User interface for displaying the MDGs, Java code scanner to create the MDGs and a Jaya algorithm implementation that will take a cluster-less MDG as input and produce an MDG with clustering.

I would like this solution to be made available for anyone interested as well, which means all parts of the solution will be taken from open-source or similarly licensed projects. I am mentioning this now because there are alternatives to approaches I describe later which could be substituted by paid software. I would like to say however that this does not mean in any way that I do not see value in paid software, just that this particular project should not rely on such software.

## 1.2 Technology and standards

Before diving into different parts of the implementation I will describe the technology stack and standards that apply to it. I will be talking about the programming language and its APIs, used libraries, build tools and document formats for persisting graphs. This section is not supposed to go in-depth on any of the technologies/standards. It is supposed to give the reader a minimal working knowledge that I feel is necessary to understand the rest of this work.

### 1.2.1 Java

When querying popular search engines „what is java“ about half of results on the first page provide only half of the actual answer. The incorrect, or more precisely incomplete, answer is that Java is an Object-Oriented Programming Language. Java however also refers to the Java platform which the language runs on. The official documentation, provided by Oracle (Oracle is currently the owner of the official reference implementation - OpenJDK) states that: „Java technology is both a programming language and a platform.“ [6] for this reason. This means that when people talk about Java they may be talking about the language, its specification, standard library APIs or the Java Virtual Machine.

While the problem of software clustering is very general and would generally apply to most procedural languages and could be possibly extended beyond the scope of a single component codebase - meaning it could span multiple languages - I will be working with Java. The reason being that Java allows very elegant separation of source code into packages. The really useful feature being that Java allows multiple levels of packages and that way creates a tree-like structure. This means that there are more ways to create modules in a single codebase. Modules can then be anything from methods, classes, low-tier packages to high-tier packages [7].

Since Java 9 (GA 2017) there is a native way to separate Java into what is called `Modules`. In short they are packages of Java packages. They aim at bringing even more encapsulation. What is the point of this thesis then if modules are already done? The problem is that someone or something (an IDE or a bot) has to create them. They are not the solution to the problem but rather the goal.

Modules define things like [8]

- Dependencies
- Public packages

- Services offered
- Services consumed
- Reflection permissions

Some items on this list directly translate to what we will be looking for in the code to create dependencies in chapter 4. In relation to these modules I would imagine this tool being the aid that helps a developer decide how to create these modules.

### 1.2.2 Maven

Maven is an Apache project that aims to simplify the build process for Java projects. It provides a way to define what is a part of a project, an easier way to publish project information and a way to share libraries packed in JARs across multiple projects [9].

This project is not really on such a large scale that I can leverage that many of Maven's features such as writing my own plugins for more complex builds, generating documentation in form of a website or release management. What I can leverage however is the automated build and dependency management.

*Automated builds* are a feature I enjoy whenever I clone a project from GitHub, BitBucket etc. If it is a Maven project usually all you need is a machine with Maven installed and you can easily produce all artifacts required to run the project just by running Maven in the project directory. This is done by describing the process in a POM file. POM is an XML file describing the project and its configuration [10]. Regarding builds POM can for example include the `jar` plugin. That can in turn specify all artifacts (class files, images, schema files, etc.) that are to be packaged and set a main class to make the built `jar` executable.

*Dependency management* is also a feature that is useful even for the smallest of projects. By adding just a couple of lines to the POM you can add `jar`s from the Maven repository in a specific version to your project. Maven then handles downloading these packages and issues like adding them to the classpath for proper execution without any further configuration.

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.9</version>
</dependency>
```

Just using this short snippet adds the *Apache Commons* library to your project.

Maven has many other configuration options and plugins, however as I am not leveraging them there is no need to dive deeper into the workings of Maven for now.

### 1.2.3 XML

Since the runtime of our clustering algorithm may be substantial, I will have to be persisting partial solutions - at the very least it would be a good idea to save the MDGs between the steps of scanning Java code and finding optimal clustering. Good machine and human readability is desirable and either JSON or XML provide a reasonable amount of both. Any persistence (in our case mostly to a filesystem) will be done by serializing graphs into XML or JSON. Their respective schemas will be described later on.

Extensible Markup Language is a text format originally developed for large-scale electronic publishing. It is widely used on the Web and in enterprise applications [11].

Accurately described as plastic boxes that can be used to store anything, even each other, have different colors and have different labels [12].

Human readability is achieved by the .xml files being still text files meaning any notepad-like text editor will make them readable, provided the character encoding matches[12].

The format is also machine readable via the use of markup. Markup serves as a way to partition information or enhance it. Both markup and text is human readable in XML [12].

An example XML document [13]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<graph>
  <descr>My favorite graph</descr>
  <nodes total="2">
    <node label="node A"/>
    <node label="node B"/>
  </nodes>
</graph>
```

XML documents start with an XML declaration

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

which is optional, but if present must be the first construct. There are a maximum of 3 name/value pairs (with the same format as attributes). *version* is required and must have the value `1.0`. *encoding* and *standalone* are optional. The order is important, which is different from regular XML attributes.

After the declaration are the actual contents of the document. In this case the opening tag `<graph>` denotes the start of a root element (also called the document element because it encapsulates the whole document) referred to as *graph element*. There is only one root element. The element ends on the last line with the closing tag `</graph>`. Inside the graph element are elements `descr` and `nodes` which in turn contains two `node` elements. Other than beginning with an opening tag `<graph/>` and ending with a closing tag `</graph>`, empty elements can be denoted with a slash at the end of the graph like `<graph/>` [12]. Also element `nodes` has an attribute `total`.

The formatting is purely aesthetic to demonstrate the tree-like structure. In theory an XML document can be a single line. Editors such as notepad++ or Visual Code offer native (or plugin) tools for formatting XML documents as shown above.

The opening and closing of elements works the same as with parentheses. You can only close the last opened tag therefore a document like this is not correct

```
<graph>
  <descr>
</graph>
  </descr>
```

Part of the XML standard is also XSD - XML Schema Definition. It is used to describe the structure of an XML document. When a document is syntactically correct it is called *well formed*. If it also conforms to the specified schema it is both *well formed* and *valid*. XSD can also define the document structure using DTD, but since I am using XSD describing DTD in more detail is out of the scope of this work. I picked XSD over DTD because I believe it is more commonly used, is more powerful (in terms of rules applied to documents) and also because the schema document is still XML, whereas DTD uses a custom format [14].

If you want an XML document to adhere to a schema you only need to reference the schema and have a parser enforce the rules described. This is done by adding

attributes to the element that should be validated against the schema. This can be the root element in which case the whole document is validated against the schema.

Since XML and XSD are not the main focus of this work I will not go into much detail about the specifics of XSD namespaces or multiple schema references. I will only show an example schema which would successfully validate the above XML 1.2.4.

```
1   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2       <xs:element name="graph">
3           <xs:complexType>
4               <xs:sequence>
5                   <xs:element name="descr" type="xs:string"/>
6                   <xs:element name="nodes" type="nodes"/>
7               </xs:sequence>
8           </xs:complexType>
9       </xs:element>
10      <xs:complexType name="node">
11          <xs:attribute name="label" type="xs:string"/>
12      </xs:complexType>
13      <xs:complexType name="nodes">
14          <xs:sequence maxOccurs="unbounded">
15              <xs:element name="node" type="node"/>
16          </xs:sequence>
17      <xs:attribute name="total" type="xs:integer"/>
18      </xs:complexType>
19  </xs:schema>
```

If you look back at the example XML 1.2.4 you will be able correlate the schema. Notable lines iclude

- (1) XML root element denoting this is an XSD document
- (2) definition of a potential root element `graph`
- (5,6) adding child elements `descr` and `nodes` to the `graph` element
- (10) definition of type `node`
- (11) adding attribute `label` to the `node` type
- (13) definition of type `nodes`
- (14) adding any number of child `node` elements to the `nodes` element

If you are interested in the part of this work that describes the serialization formats and do not have experience with XML and XSD, as stated earlier XML and XSD are both W3C standards and have lots of learning resources. List of helpful links can be found at https://www.w3.org/XML/.

The actual XSD I will be using to store MDGs can be found in the apendix ??.

### 1.2.4 JSON

JavaScript Object Notation is a text-based format for representing structured data based on JavaScript object syntax. Commonly used in transmitting data in web applications [15]. While it is closely tied to JavaScript object syntax it is not really dependent on it. It is text based and therefore most programming languages can easily read it, which is proved by there being many libraries for Java, C++, C, Python and so on to parse JSON.

Here is an example a similar structure as the XML 1.2.4 example

```
{
  "nodes": [
```

```
    {
      "label": "node1"
    },
    {
      "label": "node2"
    }
  ],
  "total-nodes": 2
}
```

JSON can be aptly described by four rules [16]

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

As with XML the indents and new lines are purely cosmetic and this JSON could be written as a single line.

While arguably much less popular than XSD, JSON also has a standard for defining the structure of documents. Appropriately called JSON Schema it is a specification for defining structure of JSON data [17].

Just like XSDs are XML documents, so JSON Schemas are JSON documents. An example that would validate the JSON example above:

```
1  {
2    "$schema": "http://json-schema.org/draft-07/schema#",
3    "$id": "http://example.com/product.schema.json",
4    "title": "graph",
5    "type": "object",
6    "properties": {
7       "nodes": {
8        "type": "array",
9        "items": {
10         "type": "object",
11         "properties": {
12           "label": {
13             "type": "string"
14           }
...
19 }
```

Unlike XSD you can only specify one root schema object - in this case `graph`. Other notable lines include

- (4) schema title
- (5) schema root object type
- (7) definition of `nodes` child object

The actual JSON schema I will use to store MDGs can be found in the appendix C.

# Chapter 2
# Existing work and tools

Before talking about the actual implementation I would like to mention projects, tools and other works that try to deal with the same issue as this thesis. As I have mentioned this work can be neatly arranged into three main modules - graph rendering, code analysis and module clustering. In this chapter I will be talking about documents and projects that deal even just with one of these modules. Also I will be explaining the benefits of using them or reasons why not to.

## 2.1 Code analysis

### 2.1.1 IntelliJ Idea UML

As one of the leading vendors for Java IDEs [18] ItelliJ Idea has tools that deal with code analysis - specifically the Module dependency diagram functionality of the UML plugin.

This part of the UML plugin is designed to create UML diagrams from projects. It is based on a combination of modularity that came as part of `Project Jigsaw` with Java 9 and Idea's own module system [19].

This is an example of code analysis done by the plugin.

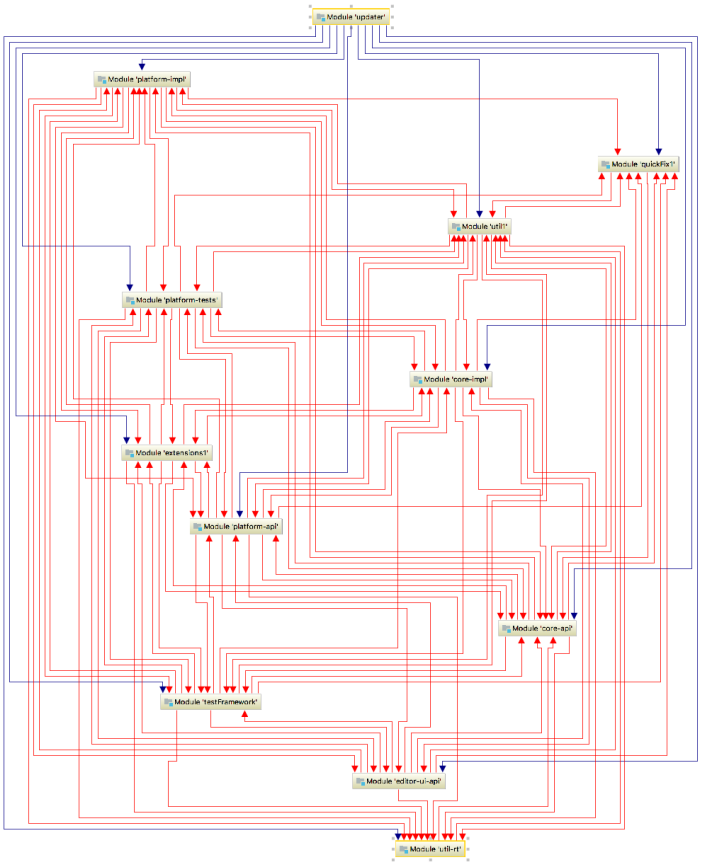


**Figure 2.1.** IntelliJ Idea UML code analysis diagram

While the visual result is very good there are issues with being able to use this plugin for our work.

The first reason being that Idea places restrictions on how modules can be created. Also it implies that either modules have to already exist in the Java code, or they have to manually created by the user [19]. Manual creation is just not realistic if I want to be able to create modules as small as a single class. If modules already exist in the Java code our original task of dividing the code into modules does not make sense.

The second, equally important, reason is that this plugin is part if the Idea's Ultimate edition [19] which is not free. I would like our solution to not require other paid software.

In conclusion I will not be using the features of the UML plugin for Idea.

### 2.1.2 JavaParser

This opensource project allows developers to programatically interact with Java code. JavaParser builds Java objects from Java code and organizes them in a structure referred to as Abstract Syntax Tree. Developers can analyze the code, change it or generate new code. [20]

Currently the parser can work with Java up to version 13.

JavaParser is crucial for this work. It enables me to analyze a large Java codebase quickly and then traverse the syntax tree to identify patters and relationships between modules.

While it is still being actively worked on the project development is not as reactive as I would need. Later on I do bring attention to some issues that have been reported but are still waiting for a fix or a workaround.

Overall the library is well maintained and reasonably popular. On GitHub there are usually new commits to the master branch every couple of days and the repository has been starred more than three thousand times.

Unfortunately as of this time there is a known bug in relation to generic types. Building an AST from a class that uses generics causes a cyclical method call and ultimately a stack overflow.

### 2.1.3 Other „Java to graph“ tools

There are other tools that can produce dependency graphs from Java code. Some of them are under an acceptable license. Often they are offered as extensions to an existing IDE such as Eclipse or NetBeans.

Usually they are limited just to a class diagram that does not go to enough detail on method invocations or field accesses etc.

Not enough flexibility, not enough detail or IDE requirements are some of the reasons that lead me to believe building a code analysis tool is the best way to move forward.

### 2.1.4 Apache BCEL

Part of the Apache Commons, the Byte Code Engineering Library aims to provide an easy way to analyze, create and manipulate Java class files [23].

BCEL does not analyze Java code. It reads class files and therefore is useful when you need to trace a call or field access from Java code to an already compiled Java resource - most likely a class file from the standard library or an external dependency.

I will describe how exactly I am using BCEL in the Code Analysis chapter 4 later on.

## 2.2 Graph rendering

Deciding whether to use an existing project/library for rendering the graph was difficult. As with all other external tools I do not want to depend on paid software.

Also standalone programs for graph plotting do not really come into consideration for this project. Just from the workflow perspective this does not make sense as rendering the graph should be available both before and after you apply the clustering algorithm. That way the user would have to go back and forth between this tool and the standalone graph drawing software. Another con is that I would be creating a strong dependency - unlike for example libraries in the Maven repository it is not always possible to download older versions of such software.

### 2.2.1 D3.js

There are a lot of JavaScript libraries that work client side in a browser. The most prominent at the moment seems to be the D3.js. D3 is a massive library. It contains dozens of templates for many types of graphs and charts.

I go into this issue more in-depth in the chapter about UI but this option would require making the whole UI browser based, which is not something I want.

Another issue for this project with D3 is that for network data (graphs) it offers limited out-of-box node labeling. This would in turn require working with D3.js library as a whole - going into the source code - instead of just using the documented APIs.

While it is common for new programmers to interchange Java and JavaScript I am aware of this important distinction and if it can be helped I do not wish to bring another language into the mix

To sum up - because it is written in JavaScript, does not provide all the render capabilities I need and requires a browser to use I will refrain from using D3.

### 2.2.2 JGraphT

An open source Java library for working with anything graph related. It offers a lot in terms of algorithms you can run against the graph like cycle detection, strongly connected subgraph detection or shortest paths.

I tried working with JGraphT because it even has a rendering capability. However it can only produce a serialized image of the graph which makes inspecting large graph (which would require zooming in for example) difficult.

However I can definitely see the added value JGraphT can have and could prove useful for adding additional functionality.

### 2.2.3 Conclusion

There are many other libraries written in Java and other languages and while I would like to not spend time making something that already exists, I simply cannot find the right combination of features.

Hence I will not be using any existing library or software for working with graphs.

## 2.3 Module clustering

The module clustering problem is not overlooked. In the past year there have been many works published about this topic.

In the scope of this thesis I will not be able to go into each individual research of this topic especially since most published works choose a unique approach to solving it [24] [25] [26].

Unfortunately most of the work I have found focuses more on the partitioning problem and does not go into detail how to properly formulate the module clustering problem in software. Only the Multi-Start Jaya Algorithm for Software Module Clustering Problem [36] really deals with that.

# Chapter 3
# User interface

## 3.1 Motivation

A part of this thesis is the development of a software tool with the ability to visualize the graph representing Java codebase and dependencies and links inside it. While graphs like these may be usable in automated systems, since I will be persisting these graphs as XML or JSON documents (both machine readable formats), tools that could take advantage of such data are not a part of this thesis. Therefore it is important to have a layer that is capable of visualizing these graphs so that a person may extract relevant information from them.

The user interface of this tool has one primary function and that is to offer a human friendly view of the software structure (the MDG). Editing the graph is not really relevant and therefore functionality in this area will be fairly minimal.

## 3.2 Design

One of the benefits, or rather properties, of Java is certainly portability [27] with a *write once, run anywhere* principle [28]. Java also has its own toolkit for creating graphical user interfaces (GUI) called Swing, which was created to extend the capabilities of the older AWT and to be more platform independent [29].

While I considered the option for this toolkit to have a web-based GUI accessible using a browser, I think that using Swing, an all-Java framework that is part of the standard library, is more portable and platform agnostic.

Architecturally I will be coding a model-view-controller pattern as is supported by Swing. Swing operates on an MVC model but does not necessarily call for creating a Controller class since view and controller responsibilities are being handled by component classes (such as JButton, JTree etc.)[29].

Java projects can have quite a lot of classes. This is apparent just by looking at some of popular Java open-source projects. For instance Google's Guava turned up with roughly 3500 Java classes (including test classes). Selenium has over 1400. While I have no ambition that this tool would be actually used in production for projects like these (I also talked about why open-source projects with thousands of contributors are not necessarily the ideal target) I should take into account that projects with hundreds of classes do exist and it is important that the UI can work with numbers like these. It is necessary that the graph layouts and UI controls take this into account.

## 3.3 Controls

As frontend and UX is not my strongest suit I feel the need to keep the UI as clean as possible. Minimal controls and minimal visual flair with comprehensibility and readability being most important.

### 3.3.1 Horizontal Menu Bar

Taking a look at existing IDEs, one of the control components will be a horizontal menu bar.

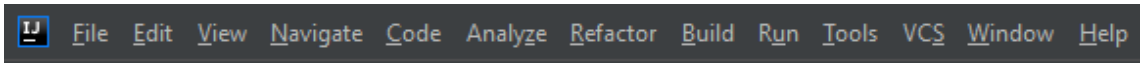


**Figure 3.1.** IntelliJ Idea menu bar

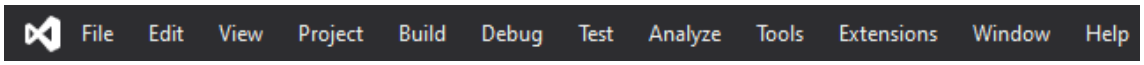


**Figure 3.2.** Visual Studio menu bar

While using the horizontal menu bar as a concept I only require a sized-down version of it. All I need are three menu items (with submenus)

- `File`
- `Edit`
- `Help`

The contents of each option are straightforward. `File` has items

- `Open`
- `Save as`
- `Create dependency graph`

`Open` loads an MDG graph from files of which structure will be described later on in chapter 4. The files are either in JSON or XML format. Based on the file extension a parser is selected.

`Save as` is used to serialize the loaded graph and save the file to the filesystem. `Swing`'s `JFileChooser` does not offer the common option for selecting the file extension when selecting a file to save to. For this reason `Save as` has a sub-menu that makes the user choose between saving as XML or JSON.

`Create dependency graph` is for creating the MDG, not loading it from a file. This option also renders a `JFileChooser` frame that makes the user select the root of the Java project to be scanned.

`Edit` is composed of

- `Undo`
- `Redo`
- `Apply random layout`
- `Apply force-directed layout`

`Undo` and `Redo` are exactly what any user could expect - reverts the last change, brings back last reverted change respectively. They can also be called by their usual keyboard shortcuts.

`Apply [layout]` are used to update the positions of nodes based on the selected layout.

The `Help` item only offers information about the build and usage of either the GUI or CLI version of this tool.

### 3.3.2 Render panel

The rest of the UI is just what I call a `Render panel`. It is a component of type `JPanel` and we render the MDG onto it.

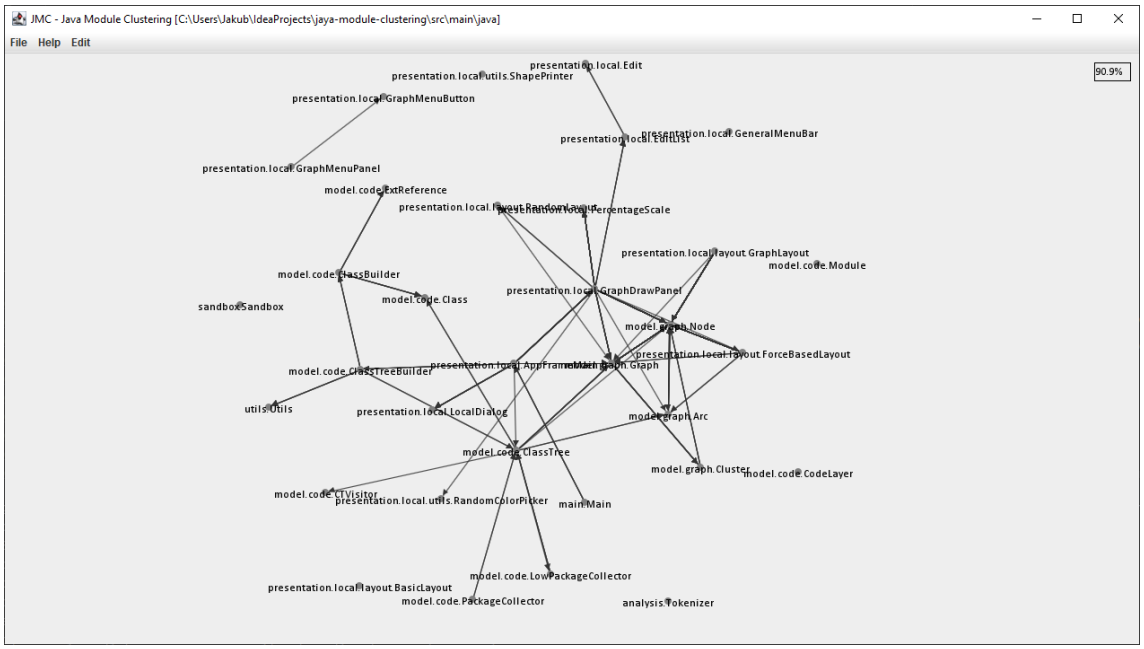


**Figure 3.3.** UI with loaded MDG

The whole area is dedicated to rendering the MDG. As mentioned before this is because the number of modules can be very high therefore we want to maximize the drawing area.

The upper right hand corner has a percentage scale indicating the zoom level of the current render.

For the control scheme I went with the following:

- Clicking on a node selects that node
- Clinking on a node while holding the `ctrl` key adds that node to selection
- While having a non-empty selection, click and drag moves the selection
- Clicking into empty space discards any selection
- Click and drag without a non-empty selection moves the view
- Holding `ctrl` and using the scroll wheel changes the zoom level

## 3.4 Graph Layout

While there is a click & drag feature, that allows the selected nodes in the graph to be moved around, along with two commonly used controls `ctrl + click` to add a node to current selection and right-click & drag to select all nodes within an area, this always means work for the user. With large graphs this can become tedious and does not always produce good results. Also if there are any edge crossings the problem becomes harder as well.

Drawing graphs is about communicating complex data through visuals. Drawing graphs that are easy to read is a complicated process. You need to combine knowledge from graph theory, mathematics, computer science and information visualization. Some

parts of the process can be quantified like determining time complexity or number of edges that cross. Other parts are more subjective as different individuals react differently to forms of visualization. It is also hard to quantify answers to a questions like „Does this graph look good? Is it esthetically pleasing?“ [30].

The research paper „Validating graph drawing aesthetics“ [31] hypothesizes that:

- More edge bends decrease understandability
- More edge crossings decrease understandability
- Local symmetry increases graph understandability

the paper then goes on to test these by making subjects search for and extrapolate data from the graph and answer test questions. The conclusion of the experiment was that edge bends and edge crossings contribute negatively to overall readability and understandability. The test of the effect of local symmetry is described as inconclusive [31], although some publications claim it is actually important as well [30].

Arguably, drawing non-linear arcs is programatically harder. This combined with the evidence that more curved arcs actually make understandability worse I will refrain from using those and only use linear arcs.

More recent research also considers other quantitative qualities for drawing graphs. Findings show that it is desirable to [32] [33]

- Maximize the minimal angle between edges from/to one node
- Maximize edge orthogonality (how well edges follow an imaginary Cartesian grid)
- Maximize node orthogonality (how close are nodes placed on grid points in an imaginary Cartesian grid).
- Minimizing change in direction in multiedge paths

Edge orthogonality is something I will not be putting much focus on. The reason is fairly simple - my graphs need to contain text information (like fully qualified class names). Since I do not want to print any text at an angle as it compromises readability, the text will be printed horizontally. If I was to try to maximize edge orthogonality it would only make reading it harder. If a line runs through a text at an angle and crosses only a few letters the text remains fairly readable. If a line runs in parallel with the text it can, based on text font size and line thickness, make the text virtually illegible.

Multiedge paths are not really relevant. I went ahead and generated graphs from code and it seems that several nodes being connect by one *line* (multiedge path) are quite rare. This may be simply due to patterns like this not emerging in code naturally. Also this metric is mostly discussed in papers about flows inside acyclic graphs, whereas MDGs are very likely (though not guaranteed) to have cycles.

At this point the ideal layout seems to be one that

- only has straight edges
- minimizes edge crossings
- promotes local symmetry
- maximizes minimal angle between edges on one node
- maximizes node orthogonality

There is a lot of research regarding drawing *perfect* graphs and there is no consensus or evidence pointing to the best way to do it. Considering the criteria I have chosen *force-directed layout* as the one to implement.

Using straight edges poses no problem in a force-directed layout. I would even argue that the overwhelming majority of examples of force-directed layouts have straight edges.

By their nature force-directed layouts minimize edge crossings and exhibit symmetries [34].

Not many research papers talk about node orthogonality. Looking at the D3 source code I have not found any mention of trying to fit nodes onto a Cartesian grid. Therefore I will only concern myself with maximizing orthogonality of nodes if I find the produced layout inadequate.

### 3.4.1 Force-directed layout

This layout is usable for directed-weighed-graphs and should produce „visually pleasing“ graph layouts.

As the name suggest the core idea is setting the node positions based on acting forces in the graph. Every node is moved based on the sum of forces acting on it. Nodes are trying to push each other away, while edges pull nodes together. [34]

The force-directed layout was not originally meant for general graphs. The design of Tutte was limited to only work with polyhedral graphs and was able to reach a globally minimal solution [34]. The following work on the subject done by Eades introduced the combination of attractive forces between adjacent nodes and repulsive forces between all other nodes. Even later Fruchterman and Reingold modified the existing algorithm by Eades with applied Hooke's law and the concept of ideal edge length [35].

The algorithm I am using is certainly closest to Fruchterman and Rienglod's with minor changes. Simply put the principle of the algorithm is as follows:

```
G := (V, E);
for iterations or some change in positions
  for (v,u) in (V,V) where u != v
    calculate and store repulsive force for v
  for e in E
    calculate and store attractive force for e.from and e.to
  for v in V
    move v based on repulsive and attractive forces
```

The original algorithm in more detail [35]:

```
area := W * L; { W and L are the width and length of the frame }
G := (V, E); { the vertices are assigned random initial positions }
k := sqrt(area / |V|)
function fa(z) := begin return z^2/k end;
function fr(z) := begin return k^2/z end;
for i := 1 to iterations do begin
  { calculate repulsive forces}
  for v in V do begin
    { each vertex has two vectors: .pos and .disp }
    v.disp := 0;
    for u in V do
      if (u != v) then begin
        { D is short hand for the difference}
        { vector between the positions of the two vertices )
        D := v.pos - u.pos;
        v.disp := v.disp + ( D /| D |) * fr (| D |)
      end
  end
  { calculate attractive forces }
```

```
    for e in E do begin
      { each edge is an ordered pair of vertices .v and .u }
      D := e.v.pos { e.u.pos
      e.v.disp := e.v.disp { ( D/| D |) * fa (| D |);
      e.u. disp := e.u.disp + ( D /| D |) * fa (| D |)
    end
    { limit the maximum displacement to the temperature t }
    { and then prevent from being displaced outside frame}
    for v in V do begin
      v.pos := v.pos + ( v. disp/ |v.disp|) * min ( v.disp, t );
      v.pos.x := min(W/2, max(-W/2, v.pos.x));
      v.pos.y := min(L/2, max({L/2, v.pos.y))
    end
    { reduce temperature as layout approaches a better configuration }
    t := cool(t)
  end
```

Please note the distinction between the nodes attributes `.pos` (position) and `.disp` (displacement). Position are the current node coordinates as one might expect. Displacement is the vector by which the node will be moved in the last loop. For that reason displacement has to be reset to a zero-vector every iteration.

Some of the libraries I discussed when talking about existing tools for graph drawing have nearly one-to-one implementations of this very algorithm. I implemented the algorithm from this pseudocode.

I however ran into the following issue. Before repositioning each node based on its displacement, the algorithm corrects the possibly too large values for displacement. In case the node would move more than is allowed each step or would move outside the frame it is assigned the maximum (or minimum) allowed value.

```
v.pos.x := min(W/2, max(-W/2, v.pos.x))
```

Normally when two nodes are moved to the same position, they are artificially moved by an extremely short distance which slingshots them very far away. This is a great way to deal with randomly overlapping nodes.

Unfortunately this mechanic combined with the issue I described above meant that my nodes were stuck in a *infinite loop* of

- being moved into one of the corners of the frame
- clashing with another node which was moved here
- being slingshot away

If the slingshot did not immediately place the node into a corner, it still meant that the node was not primarily being moved by the attractive and repulsive forces.

To be precise they were moved by an overly large repulsive force, which was designed to disrupt the current graph to allow for finding a different equilibrium of forces.

The process is meant to find a local minimum. Unfortunately after many iterations this did not produce any reasonably good results because the graph kept getting disrupted by slingshot nodes.

I had to take a look at other implementations and found the `gephi` library.

The `gephi` implementation keeps the core of the algorithm the same. The difference is in how it handles placement of nodes outside the frame. Instead of letting nodes move

out of frame and then reeling them in it reels them preemtively by adding another force labeled as „gravity“.

I will note however that this algorithm still employs the use of a „maximal displacement“ concept. Meaning I still limit the distance a node can be moved during a single iteration. Although it is not used to stop nodes from being moved outside of the frame. This distance was found experimentally.

The algorithm then looks like this:

```
G := (V, E);
for iterations or some change in positions
  for (v,u) in (V,V) where u != v
    calculate and store repulsive force for v
  for e in E
    calculate and store attractive force for e.from and e.to
  for v in V
    calculate and store gravitational force for v
  for v in V
    move v based on repulsive, attractive and gravitational forces
```

While the name would suggest that we will have to calculating another N-body simulation this is not the case. The gravity force component does not take into consideration other nodes and their weights. It is closer to just assuming that the 0,0 coordinate of the Cartesian grid is massive and pulls everything towards itself.

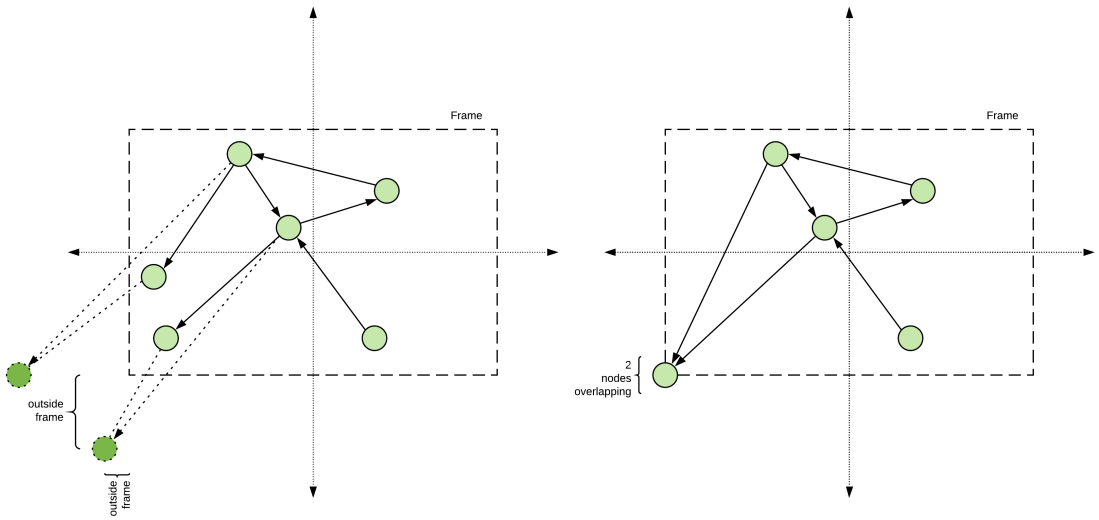


**Figure 3.4.** Force-directed layout without gravity

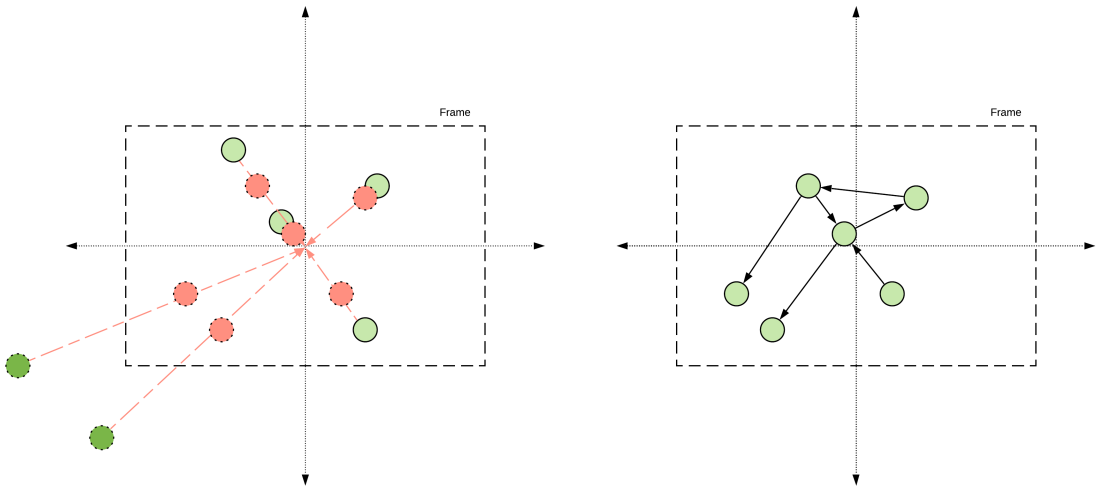


**Figure 3.5.** Force-directed layout with gravity

By adding the gravitational force to calculate total displacement for the node in the current iteration I can avoid the scenario of a node being placed outside the frame.

I calculate the force as such:

```
v is node from V
d := |v, {0,0} //distance from the origin {0,0}
gf := k * gravity * d
v.disp := v.disp gf * v.pos / d
```

`gravity` is an experimentally found constant.

I also had to add scaling constants for the attractive forces, because they were smaller than the repulsive forces by a large factor. The final displacement every iteration is dependent on the total number of the graph and edges. This means that the ratio of attractive/repulsive forces depends on how the specific graph looks. Average arcs per node, arcs per node mean, total arcs vs total nodes ratio all play a role. Ultimately the scaling constants would have to be changed for a force-directed layout that dealt with inherently different graphs than those generated from source code.

One more difference between the `gephi` force-directed variant and the original is that it does not use cooling. Temperature `t` does not only serve as a „countdown“ variable that just gets closer to 0 and then stops the loop. The original algorithm uses it to determine the size of the displacement vector.

```
v.pos := v.pos + ( v. disp/ |v.disp|) * min ( v.disp, t );
```

As such its actual value plays an important role. Because `t` converges to 0 the distance a node can move every iteration also converges to 0. In other words, the longer the algorithm runs the smaller are the changes to node positions.

While there are „rules of thumb“ for setting the initial temperature as well as the cooling function they did not work very well for me. This could also be cause by the „slingshot“ issue described above.

Since the `gephi` algorithm does not need the temperature to scale the displacement, the convergence of `t` to 0 change is swapped with a simple counter of iterations. I have therefore simply set the number of iterations for the most outer for loop to four thousand.

I only update the rendered view after all iterations have finished. Therefore this a good way of avoiding potential edge cases (similar to the „slingshot“) that could cause infinite loops or generally very high running time as force-directed layouts have fairly high time complexity.

```
G := (V, E);
for iterations or some change in positions                    \\O(c)
  for (v,u) in (V,V) where u != v                             \\O(|V|^2)
    calculate and store repulsive force for v
  for e in E                                                  \\O(|E|)
    calculate and store attractive force for e.from and e.to
  for v in V
    calculate and store gravitational force for v             \\(|V|)
  for v in V                                                  \\(|V|)
    move v based on repulsive, attractive and gravitational forces
```

In total the algorithm has the time complexity of `O(|E| + |V|^2)`.

### 3.4.2 Cluster-aware random layout

This is the default starting layout. In this layout not every coordinate is random. This layout first assigns random coordinates to clusters and calculates an optimal size for the cluster. It then places the nodes in said cluster so that they are only in the area adjacent to the cluster coordinates.

This layout helps with preventing the Force-directed layout from having nodes from one cluster spread too far away in case they are not connected by lots of arcs with other nodes from the same cluster.

### 3.4.3 Random layout

This is a custom schema that can be used to jump-start any other layouts that require non-zero starting positions for nodes. If for example the force-based layout assumes that the nodes have some non-zero non-identical (with respect to other nodes) coordinates on a 2D plane I can use this layout.

As the name suggests positions for nodes are picked at random. There are no checks for whether any arcs are crossing or nodes are overlapping. It is important to note that when I state that the positioning is random, I refer to pseudo-random generation using the Random class.

This layout has practically zero benefits and is used purely as a starting point for other layouts.

# Chapter 4
# Code analysis

In this chapter I will be describing how to use existing Java resources - source code and bytecode, to create a representative MDG.

## 4.1 Class structure

The central component for generating the graph are classes called Class and ClassTree. ClassTree holds information about the hierarchy of classes and packages. It is structured as a tree. Its children can be of both of type ClassTree and Class. The UML class object for the class is (trivial methods such as getters and setters omitted)

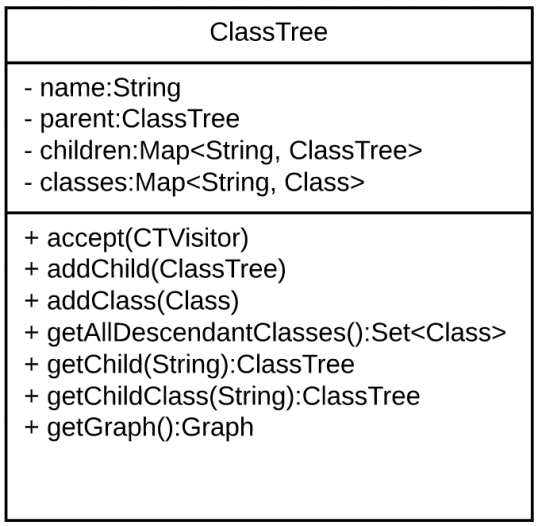


**Figure 4.1.** UML representation of the ClassTree class

You may notice that it differs from a standard tree structure by having two types of child nodes - ClassTree and Class. From a strictly programmers point of view it is possible that leaves are both of type ClassTree and Class. However in Java, empty packages are called non-observable and for that reason I do not add them into the ClassTree structure. This means that leaves are only of type Class.

To demonstrate how a Java source code translates to a ClassTree structure consider the following example Java source files

```
package example;
public class Main{...}
```

```
package example.algo;
import org.apache.commons.math.estimation.*;
public class Fourier{...}
```

```
package example.swing;
public class SwingFrame{...}
```

```
package example.awt;
public class AWTFrame{...}
```

with the following directory structure

```
<project-path>/src
|--example
   |-algo
   | |-Fourier.java
   |-gui
   | |-AWTFrame.java
   | |-SwingFrame.java
   |-Main.java
```

and a JAR file containing the `org.apache.commons.math.estimation` classes somewhere on the filesystem.

You then need to pass the Java source code root directory (`<project-path>/src`) and the JAR file directory to the ClassTree. This can be done using the ClassTreeBuilder class.

When you then call `build()` on the ClassTreeBuilder class a top level ClassTree is returned. Here is a reduced version of how the ClassTree would look. The extended structure it produces can be found in the appendix A.

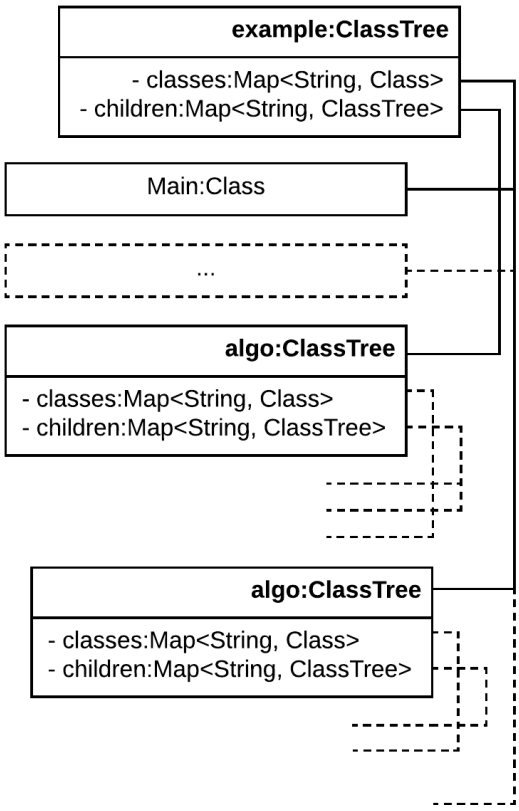


**Figure 4.2.** Reduced UML representation of the ClassTree class

Every ClassTree holds references to ClassTrees that are underneath in the hierarchy (`example.algo` is a child of `example`) as well as its direct child classes (`example.algo.Fourier` is a child of `example.algo`).

The extended version also shows that the ClassTree also follows packages and classes from the imported JAR.

Creating the ClassTree items from source code is done using the JavaParser, while creating items from the JARs and class files is done using BCEL.

## 4.2 Parsing Java code

The object structure to create and fill with data is now defined. I will now describe how to programatically get all relevant information from the source files.

For that I will use the opensource project JavaParser. It is capable of building syntax trees from Java source code and resolve field accesses or method calls.

### 4.2.1 Abstract syntax tree

An Abstract syntax tree differs from a Concrete syntax tree(or Parse tree). While the latter stores artifacts such as whitespace and parenthesis which could be inferred, the former omits those [20]. Another way of looking at the difference is what you can generate from the tree. In both cases you should be able to recreate code with the same functionality. Only the Concrete Syntax Tree will be able to retain comments or formatting.

Historically JavaParser only generated abstract syntax trees and was recommended to be used in tandem with JavaSymbolSolver (another opensource project with some of the same contributors). However the JavaSymbolSolver development has been discontinued on the main repository and the solver has been integrated into the parser.

How JavaParser builds the AST can be describe with this example [20]

```
package com.github.javaparser;

import java.time.LocalDateTime;

public class TimePrinter {
  public static void main(String args[]){
    System.out.print(LocalDateTime.now());
  }
}
```

which the parser resolves to this tree

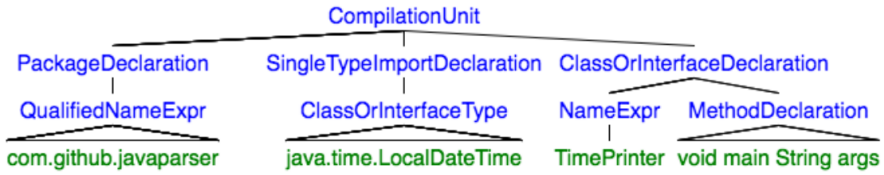


**Figure 4.3.** Abstract Syntax Tree

The tree has three children of different types - package declaration, import and class declaration. Please note that the triangle shapes mean that a part of the tree has been left out. The rightmost triangle would be expanded as such:

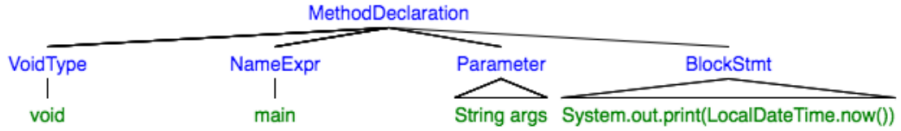


**Figure 4.4.** Abstract Syntax Tree method declaration

Expanding the rightmost triangle yet again shows the detail of the block statement

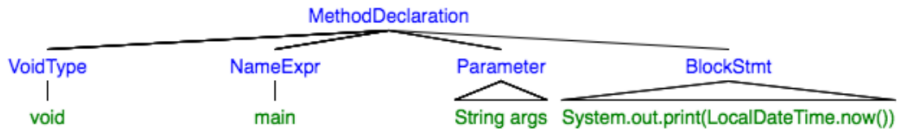


**Figure 4.5.** Abstract Syntax Tree block statement

If all you need is an AST the API is very simple. You can get an instance of `CompilationUnit` returned by a static method of `StaticJavaParser`.

```
CompilationUnit cu = StaticJavaParser.parse(<.java file>);
```

You can then access fields of the `CompilationUnit` object and traverse the AST for any data you require. You can use a Visitor to traverse the tree or you can write your

own logic. Also if you do not care about the structure, you can call the `findAll()` method which returns a list of all matches - however it returns a list which of course does not reflect the tree structure [20].

### 4.2.2 Symbol solver

It may seems that the parser is all you need to create an MDG. However there are things the parser cannot do. Let us take a look at two examples:

```
package com.example.p1;
public class MyClass { }
```

```
package com.example;
public class MyClass { }
```

```
package com.example;
import com.example.p1.*;
public class Main {
   private MyClass m;
}
```

Interestingly enough the declaration `private MyClass m` is not ambiguous - although one might think that the compiler would not know which `MyClass` is referred to. Even more interesting is that the declaration becomes ambiguous if I was to move `com.example.MyClass` to `com.example.p2`. The ambiguity, or lack thereof, comes from Java's specification regarding obscuring [21] [22].

Another interesting mechanic is shadowing.

```
import java.util.*;
class Vector {
    int val[] = { 1 };
}

class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

This program can compile and will output `1`. That is because the class `Vector` declared here has precedence over `java.util.Vector` [21].

To avoid having to deal with all the rules of compilation JavaParser also contains a symbol solver which mimics the work of a Java compiler and is therefore able to correctly identify fully qualified names for classes.

## 4.3 Parsing class files

While I currently do not display all connected Java classes from imports (simply because there are too many), I do include them in the `ClassTree` structure.

JavaParser is able to add any .class file in the symbol solver, read the bytecode and solve symbols that are referenced in the classfile as well as symbols from other sources, that reference symbols inside the classfile.

Of course most imports are not actual classfiles but rather JARs with classfiles inside. Fortunately Java provides a class `java.util.JarFile` which can be used to obtain an input stream for entries in a JAR.

## 4.4 Creating relations between classes

All that I need now is to actually create classes. That means coming back the `ClassTree` structure and the `Class` and `ClassBuilder` classes.

For every source file I create a `ClassBuilder` class and pass it the symbol solver. The symbol solver at this point has loaded all classes from the standard library, classpath and all source files. I then retrieve the `CompilationUnit` object for the given source file and search the AST for things of interest (method calls, field accesses, etc.). Those get stored in the `Class` object, which is then in turn stored in the class tree under its package.

Since I have all the references solved I know exactly to which class in which package they point to. For every reference I search the class tree for the referenced class (or create it if not present) and link the classes together.

The strength of the link depends on where in code the reference is. Consider

```
Integer.parseInt("0");
if(Math.random > .5){
  Long.parseLong("0");
  try{
    Float.parseFloat("0");
  } catch (Exception e){
      Double.parseDouble("0");
  }
}
```

I propose that the connection of this class to `Integer` should be stronger than its connection to `Long`, `Float` and `Double` as the likeliness of those being called is smaller. This is due to the fact that additional conditions have to be met for them to be executed. This is however where static code analysis falls a bit short. It is very well possible that an `if` condition is always true, and exception is never thrown or is always thrown.

I would theorize that the strength and weakening of every statement or code block the call is inside could be found during the program execution. This would however require some form of dynamic code analysis and at this point is outside the scope.

# Chapter 5
# Clustering problem

This chapter focuses on implementing the Multi-Start Multi-Objective Jaya algorithm and using it on MDGs generated from source code.

## 5.1 Problem specification

„Software module clustering problem can be defined as the problem of partitioning modules into clusters based on some predefined quality criterion. Typically, the quality criterion for software module clustering problem relates to the concept of coupling and cohesion. Coupling is a measurement of dependency between module clusters whilst cohesion is the measurement of the internal strength of a module cluster. Thus, a good cluster distribution aids in functionality-cluster-module traceability provides easier navigation between sub-systems and enhances source code comprehension.“ [36]

In the earlier chapters I often mentioned that the structure of code will be represented by a module dependency graph. An MDG is essentially just a directed weighed graph but that is ideal for this task. Every module is a node and every dependency is an arc connecting two nodes. Creating partitions in the graph means creating clusters. Essentially this becomes a graph partitioning problem with a specific objective function.

In regards to the original statement of the problem I will be looking for a clustering that maximizes cohesion and minimizes coupling. Zamli et al. [36] also uses a modularization metric called MQ and extends these goals. If translated to fit the partitioning problem they are as follows:

- maximize sum of edges inside a cluster
- minimize sum of edges between clusters
- maximizing number of clusters
- minimizing size difference between largest and smallest cluster
- minimizing modularization quality metric

Now the problem is specified as a multi-objective partitioning problem and can be solved using Jaya.

## 5.2 Jaya

Jaya is a very simple yet efficient optimization algorithm. Its simplicity is twofold. It is relatively easy to implement and is parameter-free, i.e. requires no control parameter to run.

The often described down side is that it has problems balancing roaming the whole search space and exploring neighboring solutions. This can result in premature convergence and being stuck in a local optimum. Also it is primarily designed for single objective problems. These issues are the reason behind the creation of Multi-Start Multi-Objective Jaya. [36]

The original and simplest version of Jaya can be described using this diagram [37]

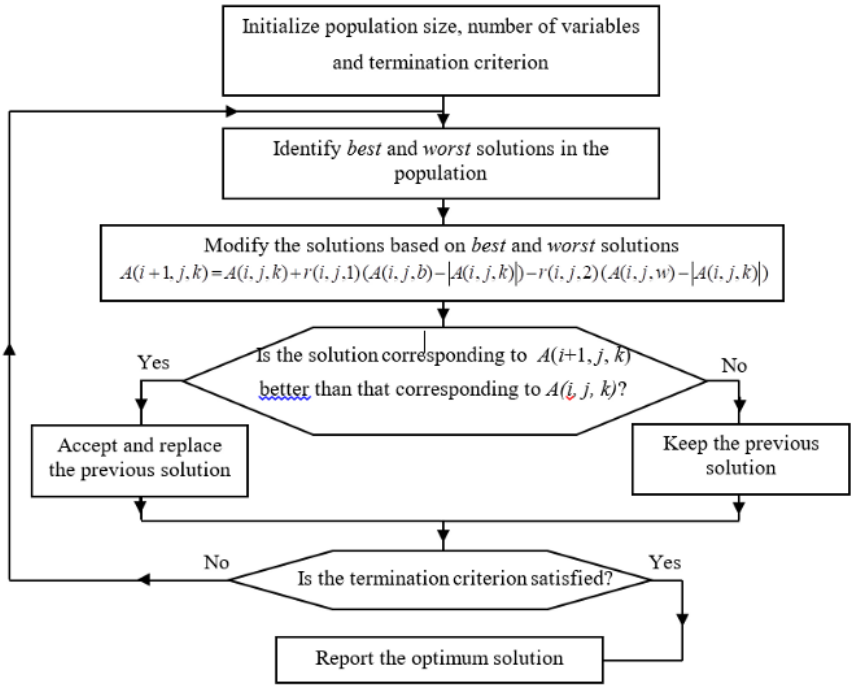


**Figure 5.1.** Jaya algorithm flowchart

One issue you may notice is that Jaya expects to be able to compare two solutions directly and pick the better one. This is not possible with multiple objectives. Rather than looking for a better solution it looks for a dominating one. It is said that solution `a` dominates solution `b` if it is not worse in all objectives and strictly better in at least one[36]. This means the algorithm cannot keep just one best solution, but rather a list. This list consists of only pareto-optimal solutions - no solution is better in all objectives than another solution [38].

This version also suffers from the issues mentioned above which is why Zamli et al. [36] propose changes.

Firstly, enhancing the formula for updating a single member of the population from

$$X_i^{(t+1)} = X_i^{(t)} + r_1\left(X_{best} - |X_i^{(t)}|\right) - r_2\left(X_{worst} - |X_i^{(t)}|\right)$$

**Figure 5.2.** Original Jaya update

to

$$X_i^{(t+1)} = X_i^{(t)} + \alpha\left[r_1\left(X_{best} - |X_i^{(t)}|\right) - r_2\left(X_{worst} - |X_i^{(t)}|\right)\right]$$

**Figure 5.3.** Proposed Jaya update

$$\alpha = M\left(1 - \frac{t}{T}\right)$$

**Figure 5.4.** Scaling factor

Notice that the whole second term which modifies the member is scaled. `M` is an upper bound constant for the problem, `t` is the current iteration and `T` is the total number of iterations [36].

Also every iteration the worst member of the population is discarded and replaced with a new randomized member.

## 5.3 Software clustering formulation

The question remains „how to represent a solution to the partitioning problem as a population member?“.

To fit the update function it would be ideal that a potential solution be an array of numbers. This array would however need to represent a partitioning. I propose that the array be of length equal to the number of nodes (modules) in the graph. The value at index `i` in the array then represents a partition (cluster) that node `i` is a member of.

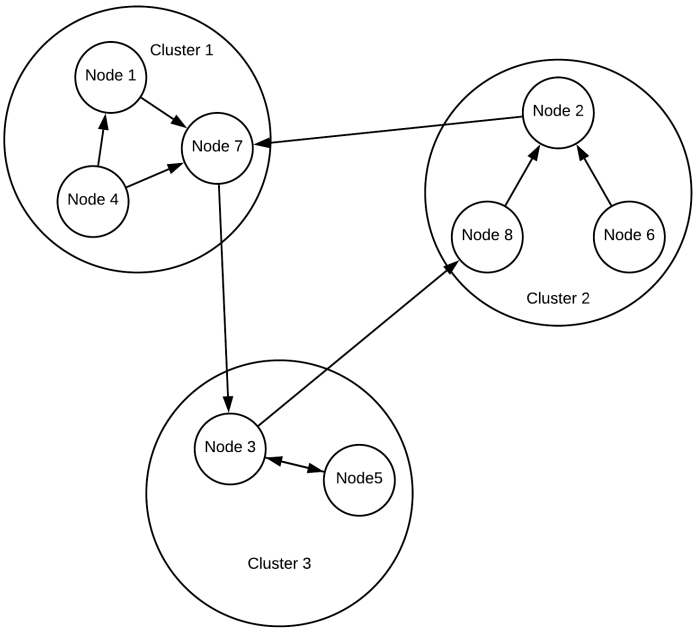


**Figure 5.5.** Clutering example

The above diagram representing a clustering of 3 clusters and 8 nodes would be represented as

```
[1, 2, 3, 1, 3, 2, 1, 2]
```

Node 1 (position 1 in array) belongs to Cluster 1 (value at position 1), Node 2 belongs to Cluster 2 and so on.

Because Jaya operates on real numbers and not just integers it is necessary to round the values.

Jaya can then be applied without any other significant alterations.

# Chapter 6
# Testing

## 6.1 Unit testing

As stated earlier I am using Maven for this project. Aside from dependency management Maven can be elegantly used to run unit tests. One way to do this is to add the Surefire plugin to the `<build>` portion of the POM. You also need to have a correct directory structure in which the test packages mirror the structure of non-test code.

The plugin record in POM:

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.0</version>
    </plugin>
  </plugins>
</build>
```

An example of the directory structure:

```
<project-path>/src
|--main
|  |-java
|    |-package1
|    |-package2
|    ...
|--test
   |-java
     |-package1
     |-package2
     ...
```

Maven can then simply be run with `mvn test` to compile the sources and run unit tests. For this project I have also included a `package` path (`mvn package`) which runs tests and if successful creates an executable JAR.

## 6.2 Usability testing

While unit tests are good for examining how the smaller parts of code behave, it is also important to know how well can users actually use the software.

„In a usability-testing session, a researcher [...] asks a participant to perform tasks, usually using one or more specific user interfaces. While the participant completes each task, the researcher observes the participant's behavior and listens for feedback.“ [39]

There are only so many tasks a user can accomplish by using this tool. Let us set two tasks

```
* create an MDG from a Java project on the filesystem
* create a clustered MDG from a Java project on the filesystem
```

Since both the CLI and GUI can be used to accomplish both of these tasks I will have users complete the first task using the CLI and the second using the GUI.

Introduction:

- clustering problem explanation
- MDG explanation
- tool has CLI and GUI
- tool can create MDG from Java project
- tool can create clustering for MDG
- tool can save graphs on filesystem

Tasks:

- using the CLI save to filesystem an MDG from this Java project (this project)
- using the GUI save to filesystem an MDG with clustering from this Java project (using GUI)

Additional info:

- path to project directory (as cloned from GitHub)
- java is configured on this machine
- maven is configured on this machine
- there is a README.md file in the project root

### 6.2.1 Tools

For testing I used my own machine. Using Zoom, a video conference software, I went through four tests with different users. The tests were conducted remotely using Zoom.

Zoom's use cases include conferencing, video calls, voice calls, text chat and others. For testing I was using the remote control feature during a video call. Essentially the user asks for remote control and if you accept can remotely control your mouse and keyboard. This allowed me to set up the environment specifically for the outlined tests.

It is also worth mentioning that this feature comes in handy when you need sensitive data on the machine where tests are conducted. Naturally you do not want to share these and the easiest way around this is handing over control of you machine for the duration of the tests.

### 6.2.2 Results

The testing starts by establishing the remote control. Then verbally explaining the items listed in the Introduction section, listing the Tasks (as close to the description as possible) and explaining the Additional info items.

After that there should be no communication until the end of the test. However the tasks I selected are dependent, which means that if a user failed the first task I would show them how to complete it and then again relieve control.

First of all let me acknowledge that usability testing with four users is not representative and if this were a production release a lot more testing would have to be done.

| User | Is familiar with Java | Is familiar with Maven | Task 1 | Task 2 |
|---|---|---|---|---|
| user A | yes | no | fail | pass |
| user B | yes | yes | fail | pass |
| user C | yes | yes | pass | pass |
| user D | no | no | fail | pass |

**Table 6.1.** Result table of user testing.

Secondly, while not all testers had experience with Java or Maven, all of them were IT professionals, although with different specializations. Unfortunately I do not have any form of consent (although I am sure they would not mind) to share information like the exact position or years of experience in the field.

The table above does contain some information, however thanks to the fact that the testing was on a very small scale I interviewed each tester afterwards.

All participants agreed that the `classpath` argument for code scanning is confusing. This was expected as classpath is in my opinion one of the more difficult subjects in Java.

Clusterings should be more apparent from the MDG render. This issue might require a rethinking of the layout, scaling of the nodes or the color scheme.

GUI portability, which was the reason why I built the GUI using Swing, is not as important and it might be better to build a more familiar browser based UI.

One last non-critical issue or suggestion was that the GUI should be made responsive when tasks like finding a clustering or applying a layout are running.

# Chapter 7
# Conclusion

The goal of this thesis was to develop a tool that can help solve the clustering problem in Java code. I believe that this goal has been achieved fully. The application sources, build and usage information are available to everyone at https://gitlab.fel.cvut.cz/pavlajak/jaya-module-clustering and there are no requirements for software that is not freely available.

At the very beginning I have described the issue I want to solve, what are the reasons that this problem even exists and why I believe it is worth investing time into solving it. I hope the arguments were compelling and will make others aware of this interesting problem and ideally will make them read on. I also give a quick tour of the technologies that are going to be discussed and used throughout this thesis to either give the reader a recap or the necessary minimum knowledge to continue.

The second chapter is about tools and research connected to this topic. It is apparent that while there are considerable number of papers and even doctorate theses there is no complete tool that would start at the source code and end with a clustered graph. This should show that this thesis and the program are not redundant and could actually prove useful.

User interface is one of the three implementation components and therefore chapter 3 is reserved for UI only. I argue why the UI components are necessary and provide the reasoning for choosing the Java-native Swing toolkit for that. As usability testing showed however the UI does have some issues that need to addressed. This is not surprising - as I admit in the chapter UI is not my strong suit. However the pure graph displaying capabilities seem adequate.

Chapter 4 describes the beginning of the process - how to create a graph from code. Using only open-source projects the program is capable of analyzing most of Java code and identifying relationships between components. There still is an issue with analyzing classes that use generic types, however this is not critical and has been reported as a bug already.

The fifth chapter briefly describes the algorithm I use to approximate the solution of the clustering problem. I propose a different way of looking at the candidate solution, but other than the actual implementation most of the work has been done by researches before me.

Lastly I describe the testing procedures. The results of usability testing helped a lot in determining which parts of the application need improving.

Although I work with Java quite often I have to admit that working on this thesis has taught me a lot. Especially to appreciate code completion features in IDEs. Working on the GUI has really shown how important it is not to overlook any part of development.

*Future work* I mentioned that the work does not end with the generated clustering. This data should then be incorporated into other team management tools. That was

the whole point of making everything human and machine readable. Researching which tools could adapt the graphs and leverage the data inside is something that could benefit developers anywhere.

Although my work on this particular thesis is done I hope time will allow me to keep improving this software. Also I would like to get more feedback - not just about the product but the idea of static code analysis based on module clustering. To see if it would usable in an enterprise environment. Hopefully I will get to work on a project where I could introduce Jaya Module Clustering.

# References


[1] What is software development? https://www.ibm.com/topics/software-development [online, cit. 15.4.2020]

[2] Team Size Can Be the Key to a Successful Software Project https://www.qsm.com/process$_{i}mproveme$

[3] Responsibility assignment matrix https://en.wikipedia.org/wiki/Responsibility$_{a}ssignment_{m}atrix[onl$

[4] Understanding Responsibility Assignment Matrix (RACI Matrix) https://project-management.com/understanding-responsibility-assignment-matrix-raci-matrix [online, cit. 1.8.2020]

[5] How to Clear Project Confusion with a RACI Chart https://www.teamgantt.com/blog/raci-chart-definition-tips-and-example [online, cit. 1.8.2020]

[6] About the Java Technology https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html [online, cit. 9.8.2020]

[7] What is a package? https://docs.oracle.com/javase/tutorial/java/package/packages.html [online, cit. 1.4.2020]

[8] A Guide to Java 9 Modularity. https://www.baeldung.com/java-9-modularity [online, cit. 8.8.2020]

[9] Maven. https://maven.apache.org/what-is-maven.html [online, cit. 12.2.2020]

[10] POM. https://maven.apache.org/guides/introduction/introduction-to-the-pom.html [online, cit. 12.2.2020]

[11] Extensible Markup Language https://www.w3.org/XML/ [online, cit. 1.8.2020]

[12] Ray, E. T. (2003). *Learning XML*. O'reilly

[13] XML Examples. https://www.w3schools.com/xml/xml$_{e}xamples.asp[online, cit.$1.8.2020]

[14] XML Schema. https://www.w3schools.com/xml/xml$_{s}chema.asp[online, cit.$1.8.2020]

[15] Working with JSON. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON [online, cit. 10.8.2020]

[16] JSON Syntax. https://www.w3schools.com/js/js$_{j}son_{s}yntax.asp[online, cit.$22.7.2020]

[17] JSON - Schema. https://www.tutorialspoint.com/json/json$_{s}chema.htm[online, cit.$22.7.2020]

[18] The Arrival of Java 14! https://blogs.oracle.com/java-platform-group/the-arrival-of-java-14 [online, cit. 1.4.2020]

[19] Module dependency diagrams. https://www.jetbrains.com/help/idea/project-module-dependencies-diagram.html [online, cit. 18.5.2020]

[20] Smith, N., Bruggen, van D. & Tomassetti, F. (2019). *JavaParser: Visited*. Lean Publishing [digital]

[21] Chapter 6. Names. https://docs.oracle.com/javase/specs/jls/se14/html/jls-6.html [online, cit. 3.8.2020]

[22] Chapter 7. Packages. https://docs.oracle.com/javase/specs/jls/se14/html/jls-7.html [online, cit. 3.8.2020]

[23] Commons BCEL. https://commons.apache.org/proper/commons-bcel/ [online, cit. 22.7.2020]

[24] Mitchell, B. (2002). *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD. diss., Drexel University

[25] Mahdavi, K. (2005). *A Clustering Genetic Algorithm for Software Modularisation with a Multiple Hill Climbing Approach. PhD. diss.*. Brunel University London

[26] Praditwong, K., Harman, M. & Yao, X. (2009). *Software Module Clustering as a Multi–Objective Search Problem.* Software Engineering, IEEE Transactions on. 37. 264 - 282

[27] Introduction to Java TM Technology https://www.oracle.com/java/technologies/introduction-to-Java.html [online, cit. 12.5.2020]

[28] Write once, run anywhere? https://www.computerweekly.com/feature/Write-once-run-anywhere [online, cit. 12.5.2020]

[29] A Swing Architecture Overview https://www.oracle.com/java/technologies/a-swing-architecture.html [online, cit. 12.5.2020]

[30] Xu, T., Yang, J. & Gou, G. (2018). *A Force-Directed Algorithm for Drawing Directed Graphs Symmetrically.* Hindawi, Mathematical Problems in Engineering, vol. 2018

[31] Bhanji, S., Purchase, H. C., Cohen, R. F. & James, M. (1995). *Validating graph drawing aesthetics: A pilot study.* Technical Report, University of Queensland Department of Computer Science

[32] Purchase, H. C. (2002). *Metrics for graph drawing aesthetics.* Journal of Visual Languages and Computing, vol. 13, no. 5, pp. 501–516

[33] Ware, C., Purchase, H., Colpoys, L., McGill, M. (2002) *Cognitive Measurements of Graph Aesthetics.* Journal of Visual Languages & Computing 13(5):501-516

[34] Kobourov, Stephen. (2013). Force-Directed Algorithms.

[35] Fruchterman, T., M., J. & Reingold, E., M. (1991) *Graph Drawing by Force-directed Placement.* SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(1 1), 1129-1164

[36] Zamli, K. Z., Alsewari, A., & Ahmed, B. S. (2018). *Multi-Start Jaya Algorithm for Software Module Clustering Problem.* Azerbaijan Journal of High Performance Computing, 1(1), 87-112

[37] Rao, R., V. (2019). Jaya: *An Advanced Optimization Algorithm and its Engineering Applications.* Springer International Publishing

[38] Pareto Efficiency. https://en.wikipedia.org/wiki/Pareto$_e ff iciency[online, cit.$19.5.2020]

[39] Usability Testing 101. https://www.nngroup.com/articles/usability-testing-101/ [online, cit. 7.7.2020]

[40] Zoom Meetings & Chat. https://zoom.us/meetings [online cit. 5.8.2020]

[41] XML Declarations. https://www.w3resource.com/xml/declarations.php [online, cit. 1.8.2020]

# Appendix A
## ClassTree example structure

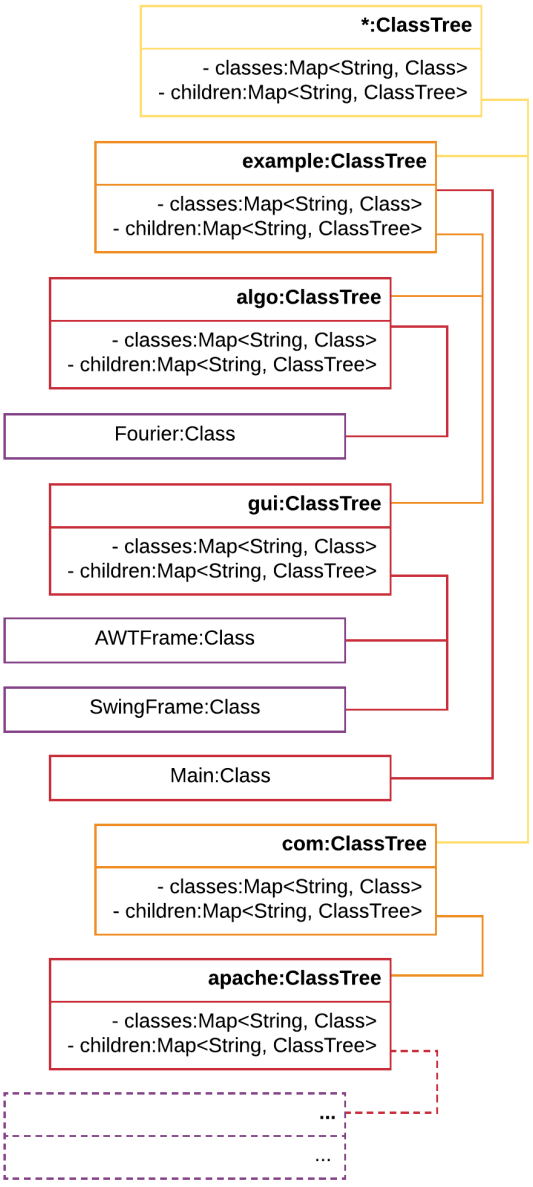


**Figure A.1.** Example of the ClassTreeStructure

# Appendix B
## MDG XSD

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="graph">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="nodes" type="nodes"/>
                <xs:element name="clusters" type="clusters"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="arc">
        <xs:attribute name="to" type="xs:string" use="required"/>
        <xs:attribute name="weight" type="xs:double"/>
    </xs:complexType>
    <xs:complexType name="arcs">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="arc" type="arc"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="node">
        <xs:sequence>
            <xs:element name="label" type="xs:string"/>
            <xs:element name="size" type="xs:double"/>
            <xs:element name="arcs" type="arcs"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="nodes">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="node" type="node"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="cluster">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="nodeLabel" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="label" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="clusters">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="cluster" type="cluster"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

# Appendix C
## JSON MDG schema

```
{
  "$schema": "jmc.schema",
  "$id": "http://example.com/product.schema.json",
  "title": "graph",
  "description": "A weighed graph.",
  "type": "object",
  "properties": {
    "nodes": {
      "description": "List of nodes in graph.",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "label": {
            "type": "string"
          },
          "size": {
            "type": "number"
          },
          "arcs": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                  "to": {
                    "type": "string"
                  },
                  "weight": {
                    "type": "number"
                  }
                }
            }
          }
        }
      }
    },
    "clusters": {
      "description": "List of clusters in graph.",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "label": {
            "type": "string"
```

```
            },
            "nodes": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "label": {
                    "type": "string"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
```

# Appendix D
# Abbreviations

| | |
|---|---|
| MDG | Module Dependency Graph. A directed graph representing dependencies of software modules. |
| GUI | Graphical User Interface. A form of user interface using icons for interaction rather than text. |
| IDE | Integrated Development Environment. Software which provides tools that aid with software development. |
| UML | Unified Modeling Language. General-purpose modeling language used to visualize system design. |
| JVM | Java Virtual Machine. Virtual machine that runs Java programs. |
| UX | User Experience. Person's attitude while using a system/product. |
| XML | Extensible Markup Language. Markup language for human and machine readable documents. |
| XSD | XML Schema Definition. Language used to define structure of XML documents. |
| W3C | World Wide Web Consortium. A community maintaining standards for the World Wide Web. |
| JSON | JavaScript Object Notation. Open standard, human redable file format. |
| JAR | Java Archive. Package format typically used to aggregate Java class files and other resources. |
| POM | Project Object Model. |
| GA | General Availability. |
| API | Application Programming Interface. [AST] Abstract Syntax Tree. |

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Pavlát** Jméno: **Jakub** Osobní číslo: **435009**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Clusterování softwarových modulů použitím Jaya algoritmu**

Název diplomové práce anglicky:

**Clustering of software modules using Jaya algorithm**

Pokyny pro vypracování:

Navrhněte a implementujte nástroj pro clusterování softwarových modulů. Nástroj bude přes příkazovou řádku/terminál a GUI umožňovat vygenerování MDG (Module Dependency Graph) z Java zdrojového kódu a aplikovat na něj Jaya algoritmus pro clusterování modulů. Pomocí GUI bude umožňovat také vizualizovat graf před a po clusterování. Nástroj ověřte sadou unit testů a usability testů

Seznam doporučené literatury:

[1] Zamli, K. Z., Alsewari, A., & Ahmed, B. S. (2018). Multi-Start Jaya Algorithm for Software Module Clustering Problem. Azerbaijan Journal of High Performance Computing, 1(1), 87-112
[2] Mitchell, B. (2002). A Heuristic Search Approach to Solving the Software Clustering Problem. PhD. diss., Drexel University
[3] Mahdavi, K. (2005). A Clustering Genetic Algorithm for Software Modularisation with a Multiple Hill Climbing Approach. PhD. diss., Brunel University London

Jméno a pracoviště vedoucí(ho) diplomové práce:

**doc. Ing. Miroslav Bureš, Ph.D., laboratoř inteligentního testování softwaru FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.07.2019** Termín odevzdání diplomové práce: **14.08.2020**

Platnost zadání diplomové práce: **19.02.2021**

| doc. Ing. Miroslav Bureš, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Podpis studenta

Datum převzetí zadání