A tool for the reverse engineering of Java object-oriented source code into UML diagrams

Dimitrios Anyfantakis

Diploma Thesis

Supervisor: Prof. Panos Vassiliadis

Ioannina, October 2022

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Panos Vassiliadis, who guided me through this Diploma thesis. His insightful comments and suggestions were essential to the completion of this Diploma thesis. I would also like to thank my family for their support.

October 2022

Dimitrios Anyfantakis

Abstract

The goal of this Diploma thesis was to develop a tool that reverse engineers Java object-oriented source code, in order to create a Unified Modelling Language (UML) diagram. Our tool gives the designer the ability to load a project and view its folder hierarchy. When the designer chooses to create a UML diagram, our system parses the source code of the selected files and creates a graph, representing the project's structure, using the files as nodes of the graph and the relationships among them as the edges of the graph. After visualizing the created diagram, from within our tool's integrated canvas, the designer can choose to export the diagram in GraphML, text and image format, or create a new diagram from the same project by choosing different files.

Keywords: UML, diagram, reverse-engineer, object-oriented, Java, JavaFX, JDT, GraphML

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας ήταν η σχεδίαση και ανάπτυξη ενός εργαλείου ανάστροφης μηχανικής πηγαίου αντικειμενοστρεφούς κώδικα Java σε διαγράμματα UML. Το εργαλείο το οποίο αναπτύχθηκε δίνει στον σχεδιαστή την δυνατότητα να φορτώσει ένα πρότζεκτ και να δει την ιεραρχία των φακέλων καθώς και τα περιεχόμενα τους. Αφού ο σχεδιαστής επιλέξει τα αρχεία τα οποία θέλει να συμπεριλάβει στο διάγραμμα, το σύστημα αναλύει τον πηγαίο κώδικα των επιλεγμένων αρχείων. Χρησιμοποιώντας την πληροφορία την οποία έχει συλλέξει το σύστημα μας, το σύστημα δημιουργεί ένα γράφημα όπου τα επιλεγμένα αρχεία αποτελούν τους κόμβους του γραφήματος και οι σχέσεις μεταξύ των αρχείων αποτελούν τις ακμές του γραφήματος. Αφού το εργαλείο έχει οπτικοποιήσει το διάγραμμα στον ενσωματωμένο του καμβά, ο σχεδιαστής έχει την δυνατότητα να εξάγει το δημιουργημένο διάγραμμα σε GraphML φορμάτ, κείμενο ή και εικόνα καθώς και να δημιουργήσει ένα νέο διάγραμμα του ιδίου πρότζεκτ επιλέγοντας διαφορετικά αρχεία.

Λέξεις Κλειδιά: UML, diagram, reverse-engineer, object-oriented, Java, JavaFX, JDT, GraphML

Table of Contents

Chapter 1. Introduction

1.1 Scope of the Diploma Thesis

In Object-Oriented-Programming (OOP), structural Unified Modelling Language (UML) diagrams, such as class and package diagrams, are used to depict a static view or structure of the system. UML diagrams have become very popular in software engineering and for a good reason. The UML structural diagrams are used in the documentation of a software's architecture and provide a visual representation of the system's modeling. Class diagrams are the most used diagrams because OOP is based on classes and the relationships among them. This makes class diagrams very beneficial when it comes to documenting software. The class diagrams provide information regarding the classes' attributes and methods along with the relationships among different classes. Package diagrams are used to illustrate the organization of the packages and show the relationships among different large components of the system.

Even though there are a lot of tools that give the designer the ability to hand-draw a UML diagram, or others that use their unique language syntax in order to create a UML diagram, only few exist that automate this process. When this process is automated using a designated tool, relationships among classes or packages may appear, which the designer had not taken under consideration. This is due to the fact that our system will parse the source code of the project in order to identify the relationships, in which case the designer might observe some unwanted behavior in his software or take into consideration something that he would have otherwise missed. Reverse engineering the source code is the most efficient way when it comes to creating a UML diagram and that is what our system is designed to do.

This tool is a standalone software, meaning the designer does not need to install any third-party software in order to use it. It provides a user-friendly user interface that makes the creation of a diagram a straightforward process. The designer will have the option to export a created diagram or choose different classes or packages to create a new diagram from the same project.

1.2 Thesis Contribution

The contribution of this Diploma thesis is the design and implementation of a tool that reverse engineers Java object-oriented projects into UML class and package diagrams. The tool performs the following tasks concisely

- parses a Java project at a specified path, the path to the project's root folder, and represents the structure of every Java file's source code as an AST tree.
- converts the produced AST tree to a graph, with the nodes of the graph being the structural components, i.e., packages, interfaces and classes, of the Java project and the edges being the relationships among them, i.e., dependencies, associations, aggregations, extensions, and implementations.
- visualizes the produced graph inherently, in its own canvas.
- exports the produced graph in various formats, i.e., GraphML, text and image.

1.3 Organization of the volume

This diploma thesis consists of five chapters that will be further discussed in our next points.

In the second chapter we further analyze the aim of the thesis, compare and evaluate different tools that provide similar functionality to our tool and provide the system's requirements in the form of use cases.

The third chapter illustrates the system's design and software architecture using class and package diagrams as well as a brief definition of the packages and the classes' roles and responsibilities in our system.

In the fourth chapter we provide the evaluation of our system as well as its results by providing screenshots of diagrams visualized and exported by our tool using object-oriented projects as input.

The fifth chapter is the synopsis of this diploma thesis along with the possible future work that can be added to improve the functionality and performance of our tool.

Chapter 2. Thesis Description & Background

2.1 Aim of the Thesis

The aim of this thesis is the development of a tool that produces UML diagrams by reverse engineering a Java object-oriented project. The source code of the project will be parsed using the Eclipse's Java Development Tools (JDT) APIs and every Java source file will be represented as an Abstract Syntax Tree (AST). Using the information provided by the processing of the AST, a graph of the project will be created. The nodes of the graph will represent the classes, interfaces and the edges will represent the different relationships among the classes and interfaces. The graph of the whole project, or a subset of it, will then be able to be exported in a GraphML format in order to be loaded in graph visualization tools that accept GraphML input, such as yEd by yWorks, as well as to be visualized by our system via a JavaFX canvas.

2.2 Related work and tools

There are several tools for hand drawing UML diagrams, but only a few that reverse engineer Java source code and create UML diagrams for a given project. Most of these tools are plugins for IDEs such as Eclipse or IntelliJ IDEA, however, standalone software tools exist as well.

2.2.1 Eclipse plugins

The ObjectAid UML Explorer [Obje19] is a lightweight code visualization tool for the Eclipse IDE that generates Class and package diagrams from Java source code. Some of its features include the dynamic update of the diagram if the source code changes, the ability to save diagrams in image formats as well as export them as PDF and a handy drag and drop canvas that makes it very easy to drag the desired classes or packages for which the diagram can be created.

Eclipse Papyrus Software Designer [Papy21] is an extension of the Papyrus editor that, since Eclipse Neon, is as a separate component of Papyrus. Although Papyrus is not as easy to install and set up as ObjectAid, Papyrus Designer provides code generation from

UML models along with the reverse engineering of Java files and packages. This comes with a few handy features including the addition of stereotypes that act as tags, for managing elements the designer does not want to generate and the Java library which provides the ability to use Java primitives for attributes and parameters.

2.2.2 IntelliJ IDEA plugins

UML Generator [Umlg22] is a plugin for IntelliJ that is very easy to install via the marketplace and is integrated automatically in the IDE, thus making it very fast to create a class diagram. The designer has the option to choose which classes you can include in your diagram and it comes with an erase tool to erase entities once the diagram has been created. There are a few cons though, when using this tool, one of which is that the designer cannot dynamically add classes to the diagram. Another problem with UML Generator is that the UML models cannot be resized to fit the canvas and one can only export the diagram as an image of the whole canvas making it very hard to read.

UML Design Tool [Umld21] is another plugin to IntelliJ similar to UML Generator. This plugin can be characterized as a simplified, or rather a lacking version of the tool mentioned above. This plugin does not give the designer the option to choose the classes or even the packages for which he wants to create the diagram and instead creates a diagram for all the classes of the project. Another missing feature is the inability to modify the layout of the UML models and export the diagram.

2.2.3 Standalone Software

StarUML [Star21] is a cross-platform software modeling tool that supports UML notation. In StarUML's extension manager you can find open-source extensions, one of which is the Java extension, which supports the reverse engineering of Java code to UML diagrams along with the generation of Java code from UML models. Using the reverse code tool, the user can load the desired model into the model explorer and drag the desired package or class to the diagram area where he can edit them using the style editor. The software also supports high pixel density displays when exporting a diagram to an image. The downside of using StarUML for the reverse engineering of Java source code would be that the extensions documentation states that it is a temporal feature that doesn't provide perfect reverse engineering.

Visual Paradigm [Visu22] is an application designed for software development and project management. Along with the reverse engineering of Java code to UML diagrams, Visual Paradigm offers several features including enterprise architecture, business analysis & design and several project-management tools. The most useful feature though

would be the ability to integrate the visual modeling environment within IDEs such as Eclipse, IntelliJ IDEA, Visual Studio and NetBeans, thus making the creation of UML diagrams simpler and faster.

2.2.4 **Comparison Matrix**

At this point we should point out that ObjectAid is not included in our comparison matrix because the plugin is no longer supported. UML Design Tools is also not included because there have not been any commits to the project's repository since the release of the alpha version several years ago, hence the issues mentioned are not likely to be resolved and new features are not likely to be added. Visual Paradigm is excluded because the community edition, which is available for free, does not include the generation of UML diagrams from source code which is the main feature we are interested in.

	Papyrus	UML Generator	StarUML
Setup & Installation		✓	✓
Easy-To-Use		✓	✓
Features			✓
Support	✓		✓
Documentation	✓		✓
Nonfunctional properties			✓
Community	✓		✓
Popularity			✓

Matrix 2.1 Comparative table of existing tools with respect to a set of evaluation criteria.

Based on the comparison matrix we can easily conclude that StarUML is far superior software than the other plugins we examined. StarUML has a free version available for personal use with the only limitation being that there are watermarks in the exported diagram images. When it comes to the user interface (UI), StarUML's UI looks a lot cleaner and seems more friendly to the user. Nonetheless, the ability to use Papyrus and UML Generator within an IDE would be as simple as it gets when it comes to simply creating UML diagrams fast. To summarize, StarUML is well documented with quality guides of usage, supports a lot of unique features, has a community where you can resolve common issues and is available cross-platform. When it comes to an external software, StarUML

would be the best choice. However, if the designer wants to make a UML diagram quickly within the IDE they are using, Papyrus and UML Generator, for Eclipse and IntelliJ IDEA respectively, would be the only choices.

Regarding the most common and useful features included in the software tools we examined, we can conclude that it is essential for the software tool to give the designer the ability to select for which classes or packages the UML diagram will be generated. That can be achieved either by either using a drag and drop feature or via a file explorer menu embedded in the software. Another helpful feature would be the ability to modify the diagram once it has been created by erasing UML models the designer does not want to include in the diagram and by repositioning the UML models inside the canvas. Finally, the most common and vital feature would be the ability to export the created diagram as an image.

2.3 Requirements Analysis

In this subsection, we summarize the desired functionality of the system, and we express the requirements as a set of use cases.

Use Case: Load Project

ID: UC01

Description and Goal

The system loads a project so that a graph can be created.

Actors

User

Preconditions

Basic Flow

1. This use case starts when the user chooses to load a project.
2. The system displays the system files.
3. The user selects the desired project.
4. The system presents the project's source files in a tree format.

Extensions/Variations

Post Conditions

The projects source files are displayed in a tree format.

Use Case: Create Class Diagram

ID: UC02

Description and Goal

The system creates a class diagram for the whole project or a subset of it.

Actors

User

Preconditions

The user has loaded a project.

Basic Flow

1. This use case starts when the user chooses to create a class diagram.
2. The system parses the loaded project and creates its tree.
3. The user selects for which files he wants to create the diagram.
4. The system processes the information regarding the selected files.
5. The system creates a diagram by converting the tree created in the previous step.

Extensions/Variations

Post Conditions

A class diagram has been created for the selected files and plotted in the applications canvas.

Use Case: Create Package Diagram

ID: UC03

Description and Goal

The system creates a package diagram for the whole project or a subset of its packages.

Actors

User

Preconditions

The user has loaded a project.

Basic Flow

1. This use case starts when the user chooses to create a package diagram.
2. The system parses the loaded project and creates its tree.
3. The user selects for which source packages he wants to create the diagram.
4. The system processes the information regarding the selected packages.
5. The system creates a diagram by converting the tree created in the previous step.

Extensions/Variations

Post Conditions

A package diagram has been created for the desired packages and plotted in the applications canvas.

Use Case: View Diagram

ID: UC04

Description and Goal

The system plots the diagram in a JavaFX canvas.

Actors

User

Preconditions

A class or a package diagram has been created.

Basic Flow

1. This use case starts when the user chooses to view the created diagram.
2. The system uses the created diagram to populate the graph that will be plotted in the JavaFX canvas.
3. The system displays the diagram in a JavaFX canvas.

Extensions/Variations

Post Conditions

This use case ends when the diagram has been plotted in a JavaFX canvas.

Use Case: Export Diagram

ID: UC05

Description and Goal

The system exports the diagram in a GraphML format.

Actors

User

Preconditions

A class or package diagram has been created.

Basic Flow

1. This use case starts when the user chooses to export the created diagram.
2. The system displays the file explorer window.
3. The user chooses the file path location and types the file name.
4. The system converts the created diagram to GraphML modelling language.
5. The system writes the GraphML file to the disk.

Extensions/Variations

Post Conditions

This use case ends when a GraphML file from the created diagram has been created and saved to the disk.

Possible extensions with desirable features

There are several possible features that could be added as an extension to the systems current functionality. One desirable feature would be to give the designer the ability to dynamically add classes/packages via dragging and dropping classes/packages, respectively to the created diagrams canvas. Another useful feature would be the ability to modify the created diagram by deleting or moving the UML models inside the canvas in order to make it more readable to the designer. Furthermore, in addition to the ability to export the graph in a GML/GraphML format, the designer should have the option to save the created diagram as an image or even export it in a pdf format.

Chapter 3. Design & Implementation

3.1 Problem definition and outline of the solution

The first step towards the design of the tool, is to design the architecture, that will facilitate the reverse engineering of the source code of a given Java object-oriented project.

Model. First, we will discuss the domain of the infrastructure's modeling that will be populated in our system. Specifically, in our case, this domain is the main entities that make up a Java project. In order to achieve this, we have designed and implemented a tree-structured software architecture for the interpretation of the Java project. The software architecture tree is constructed based upon the following elements.

- Package

  The Java source packages are going to be represented by the nodes of the tree.

- Class/Interface

  The classes and interfaces of the project are going to be represented by the leaves of the tree.

- Class/Interface relationships

  The relationships between the classes and interfaces of the project are going to be represented by the branches of the tree.

Parser. Second, we will discuss the domain of the source code's parsing. In order to implement this tree-structured software architecture, we first need to parse the source code of the given Java object-oriented project. In order to achieve this, we are using the ASTNode API from the JDT library. The ASTNode provides us with the ability to iterate through a class's fields, methods and the methods' parameters. Unfortunately, the ASTNode doesn't provide any information regarding the classes'/interfaces' extensions and implementations. However, the ASTNode can give us access to the classes'/interfaces' declaration line and utilizing this we extract the information regarding the classes'/interfaces' inheritance.

Diagram. Third, we will discuss the domain of the conversion of the tree-structured software architecture to a diagram, in GraphML format. In order to achieve this, we are converting the nodes and the branches of the tree by utilizing the GraphML syntax used by yEd in order to visualize a UML diagram. Then we need to arrange the position of the created diagram's components and to do that we are using the SpringLayout algorithm of the JUNG framework [Jung16].

3.2 Design and software architecture

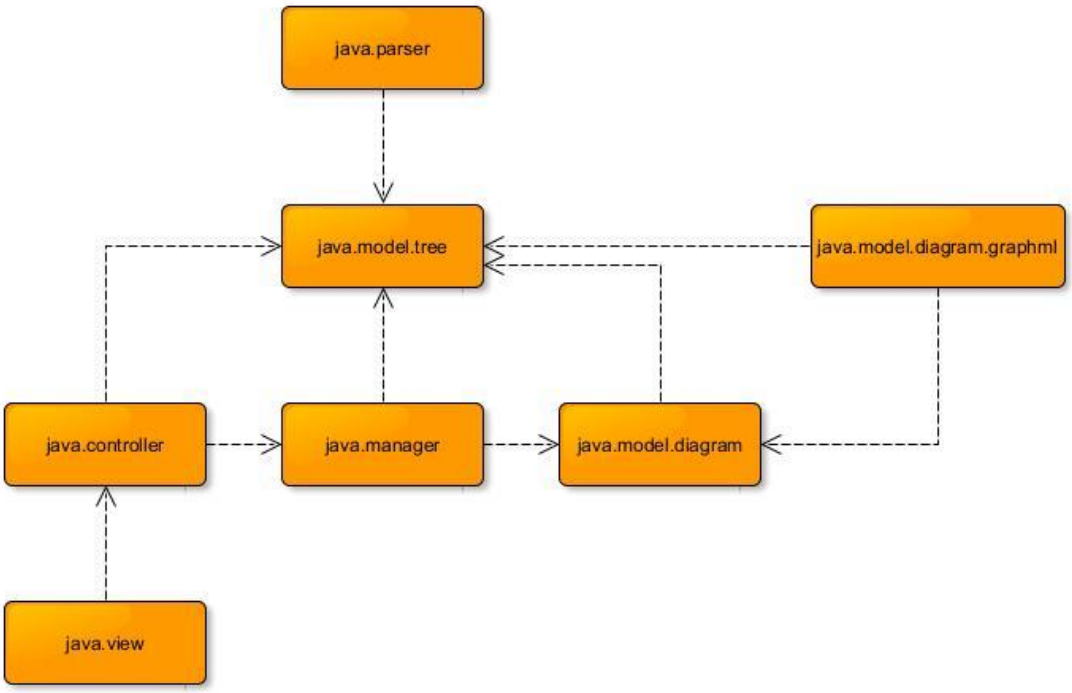

Figure 3.1 Package diagram

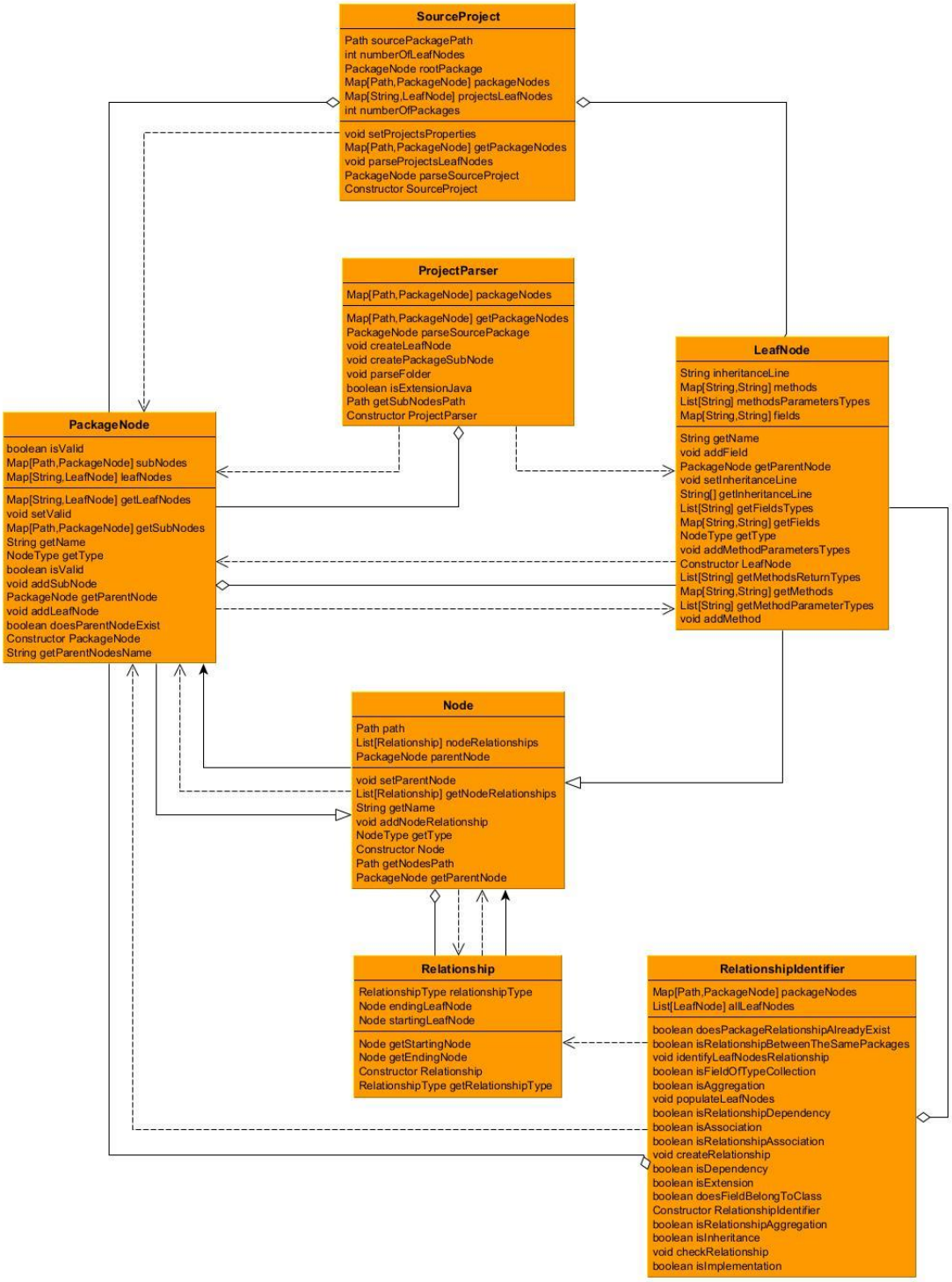

Figure 3.2 Class diagram of 'model.tree' and 'parser' packages

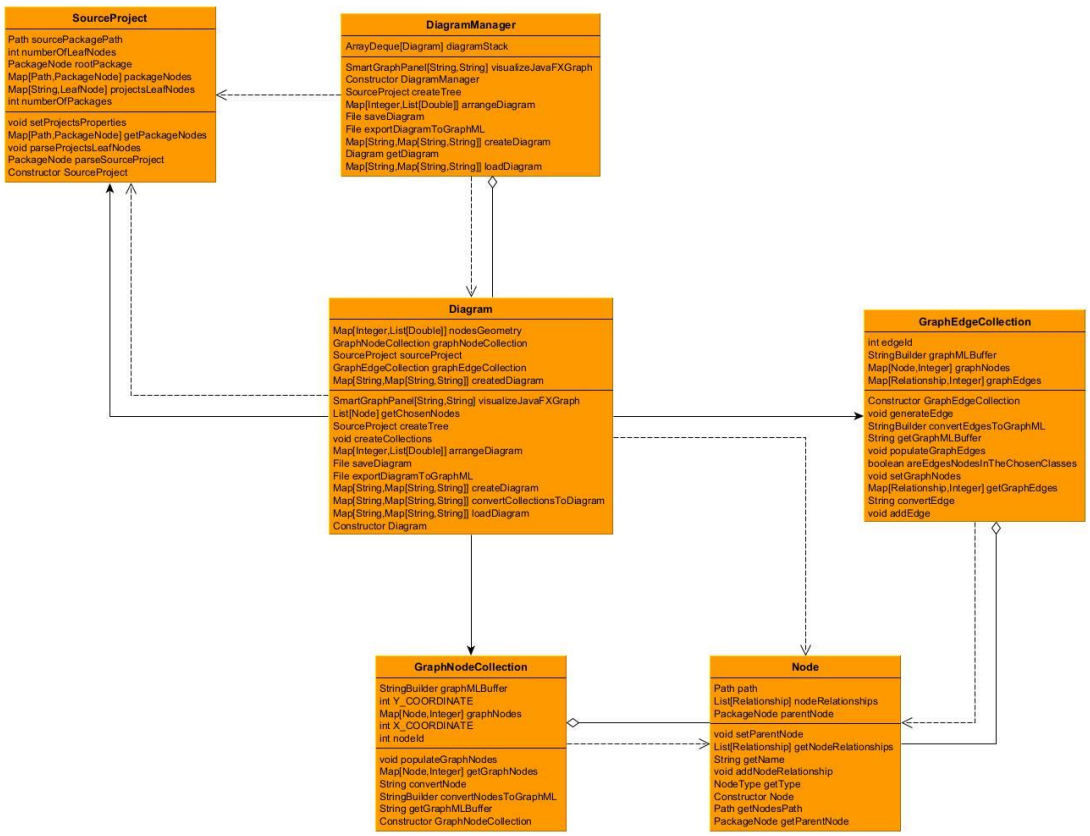

Figure 3.3 Class diagram of the Diagram

3.2.1 Package 'model.tree'

This package is responsible for the implementation of the tree structure. The classes in this package interpret the tree's elements.

- **Node**

  This class is responsible for the implementation of a node in the structure. The Node class has functionalities used by both the PackageNode and the LeafNode, which will be defined in our next two points.

- **PackageNode**

  This class is responsible for the implementation of a package node in the tree structure. Each package node represents a source package. The package node retains information regarding its parent node, the path of the package, a map of the nodes children using their names as keys, a map of the nodes children nodes, using their names as keys if the source package is valid, i.e., a package that contains java source files and a flag in order to identify if a package is valid or not.

- **LeafNode**

  This class is responsible for the implementation of a leaf node. Each leaf node represents a java source file. The leaf node retains information about its parent node, the path of the java source file and a String containing the classes it extends

and/or interfaces it implements. The leaf node also contains collections with the classes/interfaces fields names and types, the methods parameter types, the methods names and return types and the relationships, we will define the relationship in our next point, that start from this node.

- **Relationship**
  This class is responsible for the implementation of a UML relationship. Each relationship object represents the relationship between two Java source files. The relationship holds information regarding its starting, ending point and the type of the relationship. The available types of the relationship are association, dependency, aggregation, extension and implementation.
- **RelationshipIdentifier**
  This class is responsible for creating the relationships of the tree by identifying the relationships between two Java source files.
- **SourceProject**
  This class represents the Java source project, that the designer has selected. It holds information regarding the source project and is responsible for the creation of the tree structure by using the ProjectParser class that will be defined in our next point.

3.2.2 Package 'parser'

This package is responsible for the parsing of the project's Java source code by recursively visiting all the project's folders and files and the construction of the tree representing the project.

- **ProjectParser**
  This class is responsible for the parsing of a Java source project by recursively visiting all the folders of the given project. While parsing the project, the parser implements the project's tree structure by utilizing the model's classes.
- **FileVisitor**
  This class is responsible for the parsing of a Java source file by using the ASTNode API. The FileVisitor iterates through the Java source file's methods and field declarations and then the FileVisitor stores information regarding the source file.

3.2.3 Package 'controller'

This package is responsible for defining our tool's functionalities, i.e., the creation of the tree, the conversion of the tree to a diagram the arrangement of the diagram, the

exportation of the diagram to GraphML format, the saving and loading of a JavaFX diagram.

- **DiagramController**
  The DiagramController is a base class, responsible for the implementation of our Controller Interface with the functionalities we mentioned above. It is responsible for the implementation of the creation of the tree, the arrangement, the exportation, the saving of the diagram and the visualization of the diagram. It is extended by the PackageDiagramController and the ClassDiagramController, which will be further explained in our next point.
- **PackageDiagramController, ClassDiagramController**
  These two classes are responsible for implementing the conversion of the tree to a package/class diagram respectively and the loading of a package/class diagram respectively. The PackageDiagramController and the ClassDiagramController are also responsible for the instantiation of the corresponding DiagramManager, i.e., the PackageDiagramManager and the ClassDiagramManager, which will be further discussed in our next point.

3.2.4 Package 'manager'

This package is responsible for converting the tree structure created by the parser to a diagram using GraphML language.

- **DiagramManager**
  The DiagramManager is a base class, responsible for the implementation of the DiagramController's functionalities. The DiagramManager uses objects of the Diagram class, we will define the Diagram class in another point, in order to call their methods for the corresponding functionalities that are define by the DiagramController. The DiagramManager is extended by the PackageDiagramManager and the ClassDiagramManager.
- **PackageDiagramManager, ClassDiagramManager**
  These classes are responsible for the creation of the PackageDiagram and ClassDiagram respectively.

3.2.5 Package 'model.diagram'

This package is responsible for converting the tree's elements to UML entities, using GraphML syntax and features provided by yEd, as well as the implementation of the arrangement of the diagrams.

- **Diagram**

  The Diagram class is responsible for the implementation of a diagram created by the conversion of the tree, that has already been created by the SourceProject class. Furthermore, the Diagram class uses classes from the 'model.diagram' package and its sub packages to implement the functionalities of a diagram. We will further discuss these classes in our next points.
- **GraphNodeCollection, GraphEdgeCollection**

  The GraphNodeCollection and GraphEdgeCollection classes, represent the collection of nodes and edges respectively of the graph.
- **DiagramArrangement**

  The DiagramArrangement class is responsible for the arrangement of the graph. This is accomplished by creating a Graph object of the [Jung16] framework, that is populated with the nodes and edges collection, provided by the GraphNodeCollection and GraphEdgeCollection classes. Then, the SpringLayout algorithm provided by [Jung16] is used and we retrieve and store the layout's information regarding the nodes' geometry.

3.2.6 Package 'model.diagram.graphml'

This package is responsible for representing the Diagram using GraphML syntax and its conversion to GraphML.

- **GraphMLExporter, GraphMLFile**

  These classes are responsible for the creation of the GraphML file and the exportation of the file to the location selected by the designer.
- **GraphMLLeafEdge, GraphMLPackageEdge, GraphMLLeafNode, GraphMLPackage, GraphMLSyntax**

  These classes are responsible for converting objects of the Relationship class to graph edges and objects of the LeafNode and PackageNode classes to graph nodes, by retrieving information from the 'model.tree' package, in order to be used for the visualization of the graph by yEd.

3.2.7 Package 'model.diagram.javafx'

This package is responsible for the JavaFX's diagram representation along with its exportation and loading.

- **JavaFXExporter**

The JavaFXExporter class is responsible for exporting the created diagram to a text file.

- **JavaFXLoader**
  The JavaFXLoader class is responsible for loading an exported diagram text file.
- **JavaFXVisualization**
  The JavaFXVisualization class is responsible for creating the SmartGraphPanel of the [Smar21] library, populated with the contents of a diagram. The SmartGraphPanel is created using a directed graph, a placement strategy and is responsible for the visualization of the graph.

3.3 Algorithms used in the solution of the problem

Parser. In order to parse the source code of the Java project, we used a Depth-first search (DFS) algorithm, that visits every file under the root directory that was passed as an argument. For every directory we create a PackageNode and for every Java source file we create a LeafNode. When we create a LeafNode from a Java source file, we then create the AST of this file. The AST is a detailed tree representation of the abstract syntactic structure of the Java source code that defines an API to modify, create, read and delete source code. Every Java source code construct, e.g., name, type, expression, statement or declaration, is represented as a subclass of the abstract superclass of all AST node types, AST node.

Procedure: Euclidean algorithm

Input: natural number A, natural number B

Output: the greatest common divisor (GCD) of A, B

Begin

WHILE B is not 0

IF A greater then B

SET A to A – B

ELSE

SET B to B-a

Result = A

End

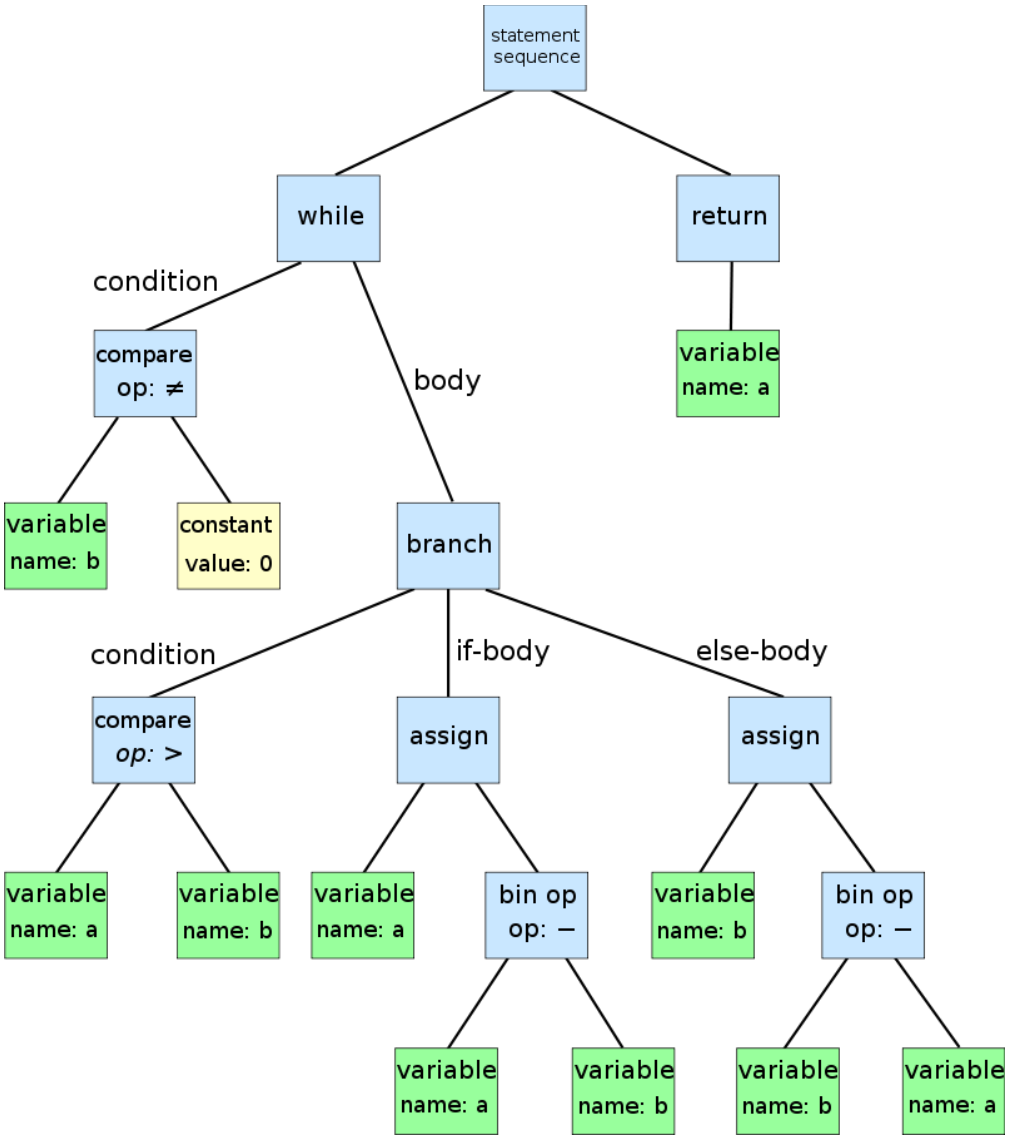

Figure 3.4 The AST of the Euclidean algorithm

Procedure: parseFolder

Input: The path to a directory D

Output: An object of the PackageNode class representing the root package of the project

Begin

CREATE PackageNode object P, representing the parent package, with D's path

FOR every file F in D

IF F is a directory

CREATE PackageNode object C, representing the child, with F's path

SET C's parent to be P

ADD C to the collection holding the project's PackageNode objects

ADD C to the collection of P's child package nodes

CALL parseFolder with C's directory path

ELSE

CREATE LeafNode object L with F's path

SET D to be a valid package

ADD L to the collection of D's child leaf nodes

CREATE the AST for L

Result: All files under the source directory haven been parsed

End

Diagram. In order to visualize a diagram, we must first convert the tree implemented by our model to a diagram.

Procedure: convertTreeToDiagram

Input: A list of the file names that will be included in the diagram

Output: A Map M that holds the nodes as keys and a Map, with its keys being the edges starting from this node and their values being the relationship type of the edge, as their values

Begin

GET the Node objects, from our tree, that correspond to the input files

CREATE a GraphNodeCollection that will hold the Node objects, along with their unique node identifier

CREATE a GraphEdgeCollection, using the GraphNodeCollection, that will hold the Relationship objects along withtheira unique edge identifier

CREATE the result Map, using the GraphNodeCollection and GraphEdgeCollection

Result = M

End

In order to visualize the created diagram in our canvas, we must create a JavaFX Pane Node where the Graph object from the JavaFXSmartGraph library is drawn.

Procedure: visualizeJavaFXGraph

Input: The diagram D that was created by converting the tree

Output: A Pane Node

Begin

CREATE a Digraph G of JavaFXSmartGraph library

INSERT D's nodes to G

INSERT D's edges to G

CREATE a Pane Node P using G

Result = P

End

To export the diagram to GraphML format, we must first arrange the diagram using a Layout Algorithm. We chose to use the SpringLayoutAlgorithm of the Jung framework, as we have already mentioned.

Procedure: arrangeDiagram

Input: The path to the file that will be exported

Output: A Map M, with the keys being the identifiers of the nodes and the values being the geometry of the nodes in X, Y coordinates

Begin

CREATE a Jung Graph object G

INSERT the nodes from GraphNodeCollection to G and the edges from GraphEdgeCollection

INSERT the edges from GraphEdgeCollection to G

CREATE a SpringLayout object L using G

CREATE M using L and the node's identifiers from the GraphNodeCollection

Result = M

End

The second step to exporting the diagram to GraphML format, is to convert the GraphNodeCollection and GraphEdgeCollection using GraphML syntax and save it to a file.

Procedure: exportDiagramToGraphML

Input: The path to the file that will be exported

Output: Exported File F

Begin

FOR every node in the GraphNodeCollection

    CALL convertNode with the Node object, the node's id and the node's geometry

FOR every edge in the GraphEdgeCollection

    CALL convertEdge with the Relationship object and the edge's id

WRITE the result from convertNode and convertEdge to F

Result = F

End

In order to do this, we use the GraphMLNodeCollection and GraphMLEdgeCollection that extend the GraphNodeCollection and GraphEdgeCollection respectively.

3.4 Graph visualization libraries comparison

A critical step to the design of our tool, is the visualization of the created diagram using a JavaFX graph. In order to achieve that, it is important to choose a library that not only is able to visualize a graph, but it is also open for extension. The reason it is important for the library to be open for extension, is because there are no free libraries that support the visualization of UML diagrams using JavaFX. That means we must extend the library in order to visualize our graph as a UML diagram, rather than a common graph. To be more specific, our graph must have dashed and non-dashed edges, closed and open arrows, that can be filled or be transparent, as well as support of custom nodes, in order to differentiate classes from interfaces when drawing a class diagram.

FXGraph [Fxgr21] is a simple library that was created from a stackoverflow post, as an answer to a question regarding JavaFX graph visualization libraries. The FXGraph supports the resizing and moving of the nodes in the JavaFX canvas. It also supports a tree layout algorithm as well the ability to use custom nodes and edges.

The Jung Framework [Jung16] also supports the visualization of a graph, along with the layout algorithm that we used for the layout of the diagram. The most significant pro of using the VisualizationViewer of Jung is the fact that we have already created a graph from the Jung Framework in order to apply the SpingLayout algorithm that we use for our diagram. Although, this saves us time, because we have already set up the library and already have created the graph to be visualized. The visualization of Jung has some critical cons. Even though, Jung provides the functionality to transform a vertex, in our case a node, to have custom color, shape and size, it doesn't provide the same functionality when it comes to edges and arrows. This means we cannot use Jung's visualization to create a UML diagram, as dashed and non-dashed edges, as well as custom arrows are critical to the creation of a UML diagram.

JavaFXSmartGraph [Smar21] is a graph visualization library with features that are useful for visualizing a UML diagram. These features include, a force directed algorithm for the placement of the nodes as well as a static circular placement algorithm. The support of custom stylesheets that can be applied to nodes and edges, with the ability to change one's style at runtime and use different styles for different types of nodes, in our case classes, interfaces and packages. These custom styles also provide us with the ability to customize an edge in order to be dashed and non-dashed, which is vital for our tool. Another useful

feature is the ability to move nodes inside the canvas as well runtime functionalities, such as removing a node and all of its edges from the graph when its double clicked or provide the user with information regarding an edge's starting and ending node, as well as the type of the edge, when its double clicked. Furthermore, JavaFXSmartGraph's design makes it open for extension, as it was not that difficult to read through the source code and extend it.

	JavaFXSmartGraph	FXGraph	VisualizationViewer
Custom node support			
Custom edges	✓		
Custom arrows			
Layout algorithm	✓	✓	✓
Movable nodes	✓	✓	
Open for extension	✓	✓	

Matrix 3.5 Comparative table of existing tools with respect to a set of evaluation criteria

In conclusion, we have discussed over the options we had when it comes to choosing a visualization library, in order to visualize the UML diagrams and we opted in favor of JavaFXSmartGraph. VisualizationViewer is only capable of visualizing a simple graph and with it not being open for extension we neither use the different arrows and edges needed for a UML diagram nor change the shape of the nodes from circle to rectangle. Even though FXGraph is open for extension and thus we can implement all these essential components for creating UML diagram. Nevertheless, FXGraph is so simplistic we would end up implementing a visualization library from scratch.

In JavaFXSmartGraph, the extensions we had to make were straightforward. First, we had to implement a closed arrow. This was done by extracting a base class for the Arrow class and created two classes, one implementing the open and one the closed arrow, that extend it. Second, we created a third class that extends the Arrow class that implements the diamond arrow, for the aggregation relationships. Third, we had to modify the Node class accordingly in order to extend the Rectangle class instead of the Circle class.

Chapter 4. System Validation

4.1 Validation Methodology

In this section, we will discuss the validity of our system. The purpose of the testing was to validate the creation of UML diagrams for their validity. The validity is defined by comparing the diagrams produced by our tool, with diagrams that were produced using a similar tool. The tool we chose to compare its diagrams to our own, is ObjectAid. As we have already mentioned in chapter 2, ObjectAid is a tool that reverse engineers Java source code to produce UML diagrams. The data we used for the testing consisted of Java object-oriented projects, hosted on GitHub that varied in size.

4.2 Detailed presentation of the results

The process we pursued was the following. First, we produced the package diagram, consisting of all source packages, using ObjectAid and then we produced the same package diagram using our tool. Second, we chose one of the project's packages and produced a class diagram, consisting of all the classes inside that package. We followed that process using ObjectAid first and then our tool. The project we used for testing our system's validity first was the DelianCubeEngine [Deli21].

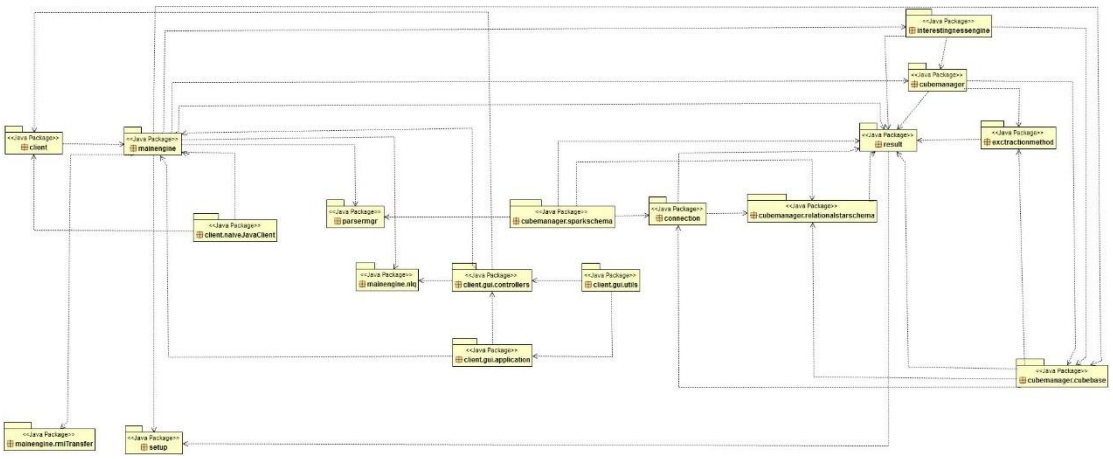

Figure 4.1 Package diagram of DelianCubeEngine's source folder using ObjectAid

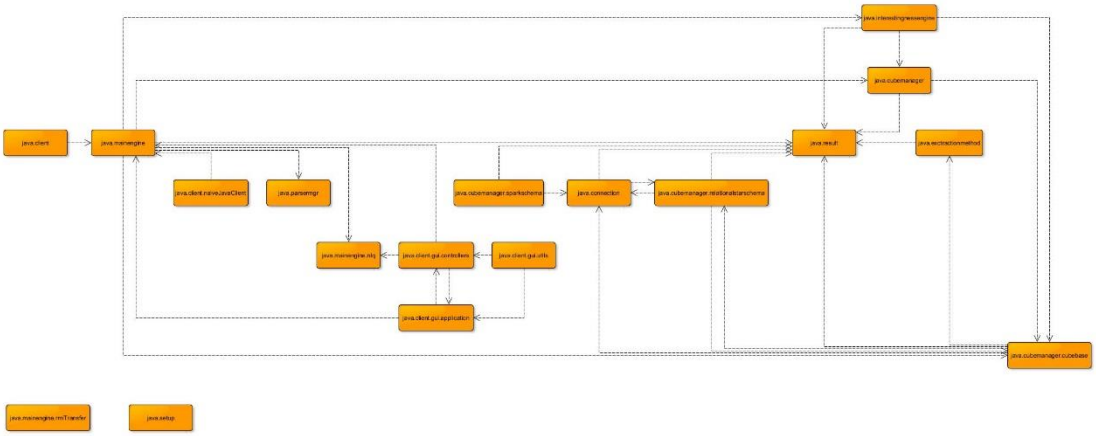

Figure 4.2 Package diagram of DelianCubeEngine's source folder using our tool

Viewing the package diagram, we can observe that although all packages exist in both diagrams, some edges are missing from the diagram produced by our tool. We will further examine this, when viewing the class diagrams, as it will be easier to identify exactly why that is.

The package we chose to create our class diagram, was the 'relationalstarchema' package, located in the 'cubemanager' package.

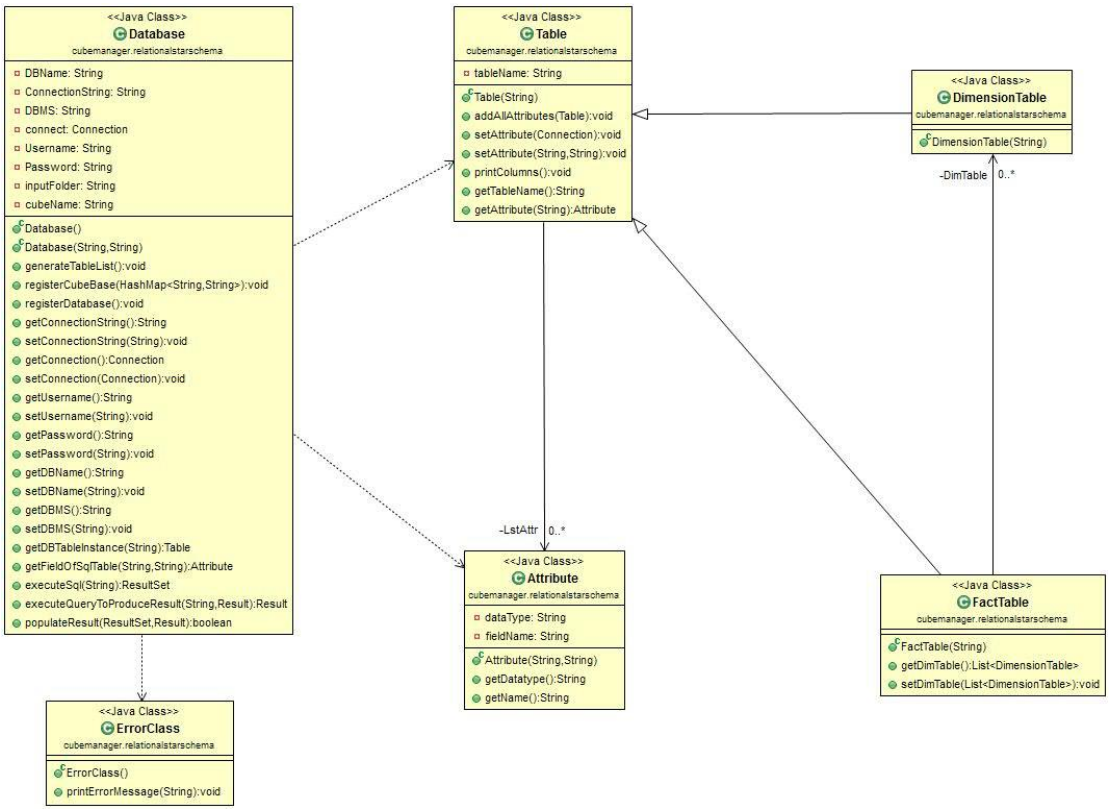

Figure 4.3 Class diagram of the DelianCubeProject's 'relationalstarschema' package using ObjectAid

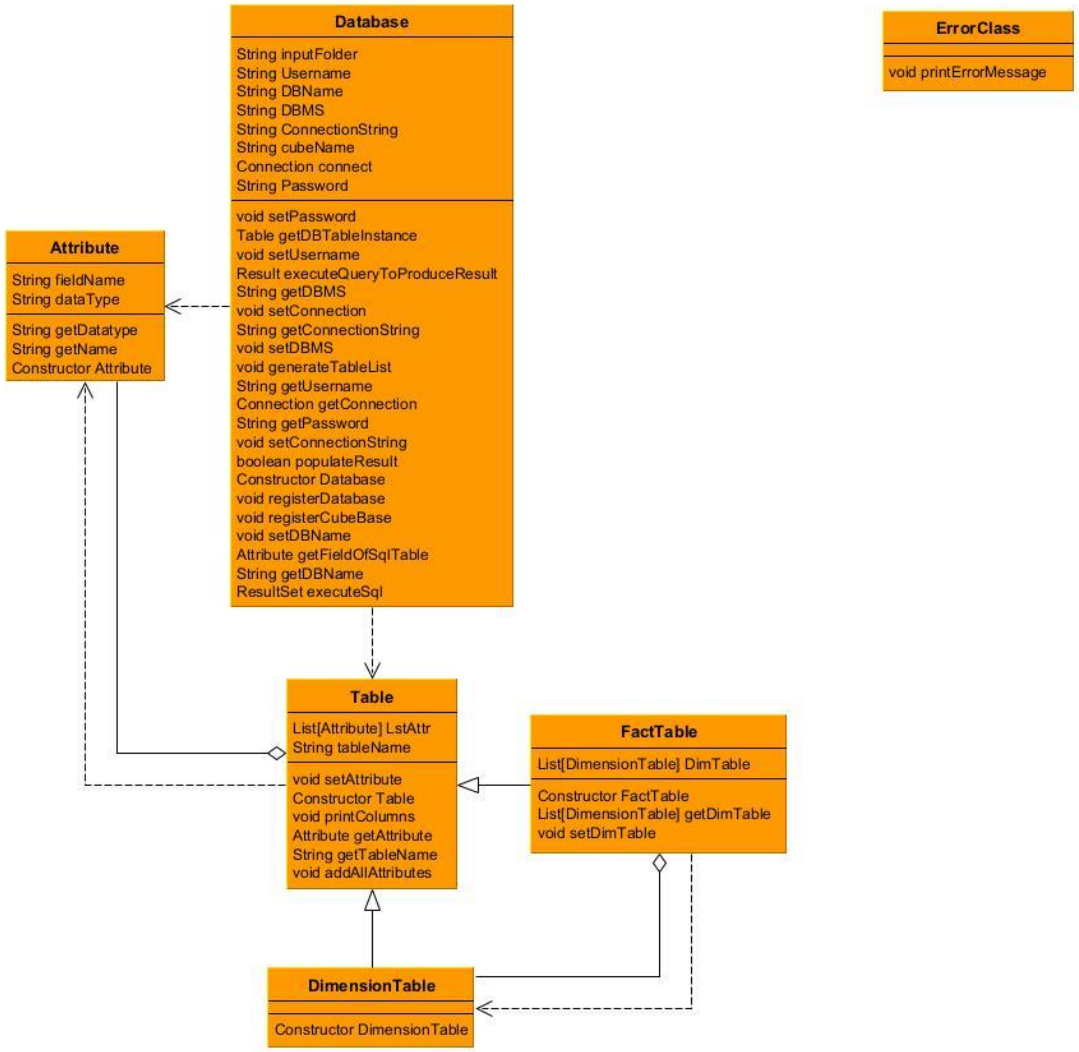

Figure 4.4 Class diagram of the DelianCubeProject's 'relationalstarschema' package using our tool

Viewing the class diagram, of the 'relationalstarschema' package, we can observe that a dependency arrow is missing, when comparing the diagram produced by our tool with the one produced by ObjectAid. This dependency arrow starts from the Database class and ends in the ErrorClass class.

```
87          public void generateTableList() {
88                  try {
89                          DatabaseMetaData Metadata = connect.getMetaData();
90                          ResultSet rs = Metadata.getTables(null, null, "%", null);
91                          ;
92                          while (rs.next()) {
93                                  Table tmp = new Table(rs.getString(3));
94                                  tmp.setAttribute(connect);
95                                  this.Tbl.add(tmp);
96
97                          }
98                  } catch (SQLException ex) {
99                          ex.printStackTrace();
100                         (new ErrorClass()).printErrorMessage(ex.getMessage());
101                 }
102         }
```

Figure 4.5 Database's generateTableList method instantiating the ErrorClass

Looking at the source code of the Database class, we can locate where the issue occurs. At line 100, we can see that the ErrorClass class is being instantiated without the use of a field. The AST does not provide us with information regarding the local fields of a class's methods. This is a limitation of the JDT's AST API that we cannot overcome. In chapter 5, we further discuss how to change the parsing method of our tool, using an alternative parser software tool.

However, apart from the missing dependency arrow, we can also observe that the diagram produced by our tool has other edge differences with the diagram produced by ObjectAid. The most significant difference is that the diagram produced by ObjectAid does not have any aggregation arrows and only uses association arrows, whereas our tool uses aggregation arrow when an aggregation relationship is present. For example, the Table class has the field LstAttr, which is a List of Attribute objects. This indicates that there is an aggregation relationship between the Table class and the Attribute class. From this we can conclude that similar tools do not have the capability to identify aggregation relationships.

One final edge difference is the extra edge that the diagram produced by our tool has. This edge is the dependency arrow starting from the FactTable and ending at the DimensionTable. The DimensionTable has two methods, one that takes as parameter an Object of the FactTable class and one that returns an Object of the FactTable class.

The second project that we used as input for the validity testing, was the Hecate [Heca20].

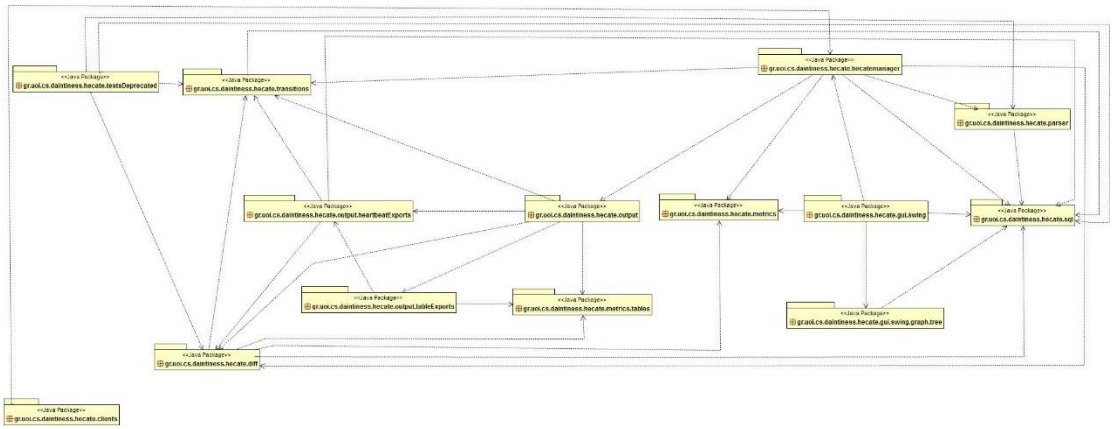

Figure 3.6 Package diagram of Hecate's source folder using ObjectAid

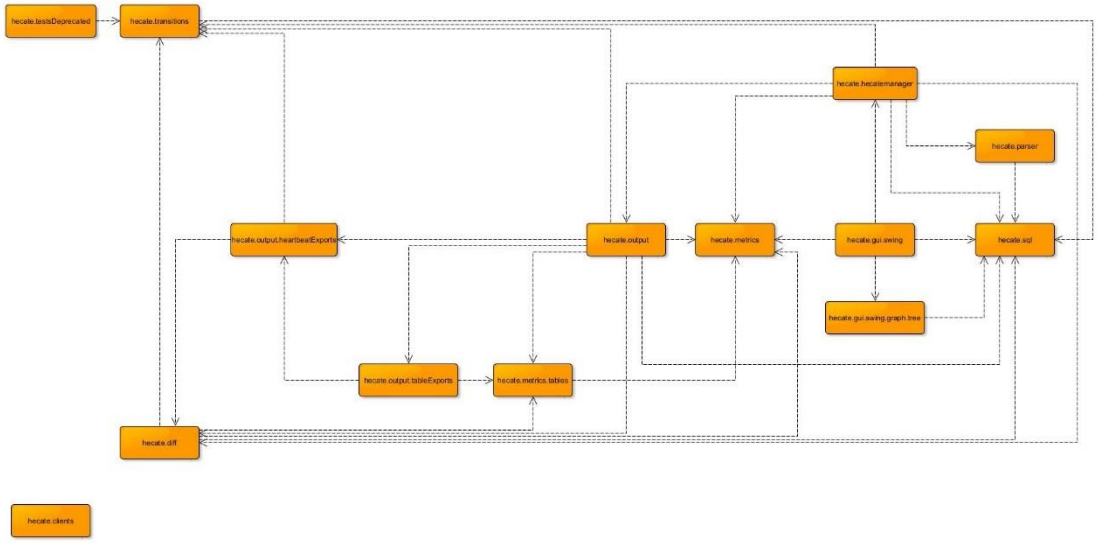

Figure 4.7 Package diagram of Hecate's source folder using our tool

When viewing the package diagram produced by our tool, we can again observe missing package relationships. Let us investigate further the missing relationship between the

'clients' package and the 'hecatemanager' package. The 'clients' package consists of the SingleFolderProcessingClient class.

```
19  public class SingleFolderProcessingClient {
20
21          /**
22           * A simple client to process a folder.
23           * Assume you have the project <code>PRJ</code> its should contain a folder <code>PRJ/schemata</code> with the schema history.
24           * You should pass the <code>PRJ/schemata</code> as the parameter needed at args[0]
25           *
26           * @param args the first parameter given should be the path of the schema history.
27           *
28           */
29          public static void main(String[] args) {
30                  String folderOfSchemaHistory = args[0];
31                  File folderToProcess = new File(folderOfSchemaHistory);
32
33                  HecateBackEndEngineFactory factory = new HecateBackEndEngineFactory();
34                  IHecateBackEndEngine engine = factory.createApiExecutioner(folderOfSchemaHistory);
35
36                  System.out.println("Working with " + folderOfSchemaHistory + "\n");
37
38                  engine.handleFolderWithSchemaHistory(folderToProcess);
39
40                  System.out.println("\nSUCCESSFUL END: Done with " + folderOfSchemaHistory + "\n" +
41                          "********************************************************************************* \n\n");
42
43
44          }
45
46  }
```

Figure 4.8 SingleFolderProcessingClient class

Reading through the main method of SingleFolderProcessingClient, we can locate the issue at line 33. The HecateBackEndEngine is being instantiated using a local field. As we have already mentioned JDT's API does not provide any information regarding local field types and we thus cannot identify this dependency relationship.

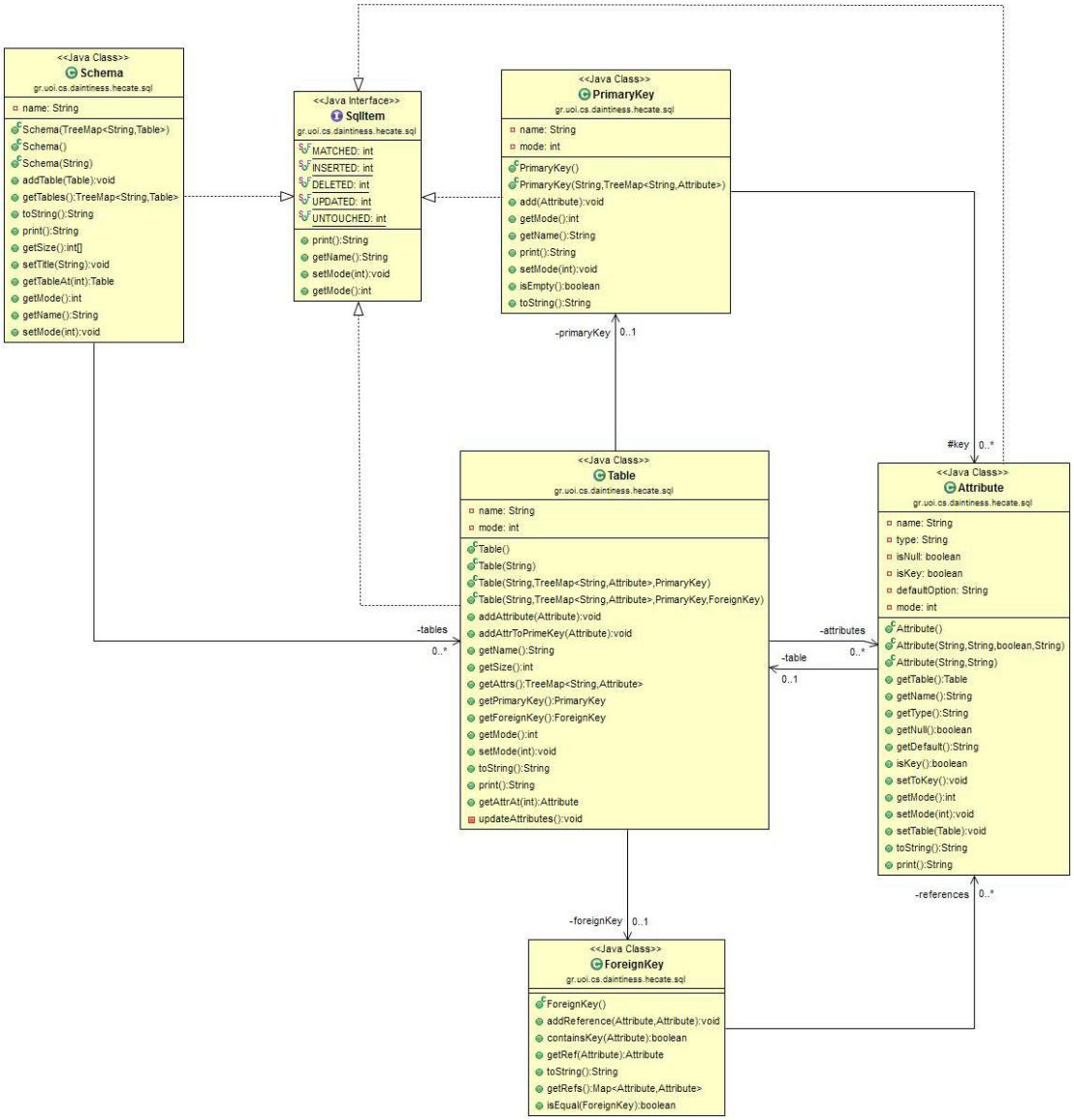

Figure 4.9 Class diagram of the Hecate's 'sql' package using ObjectAid

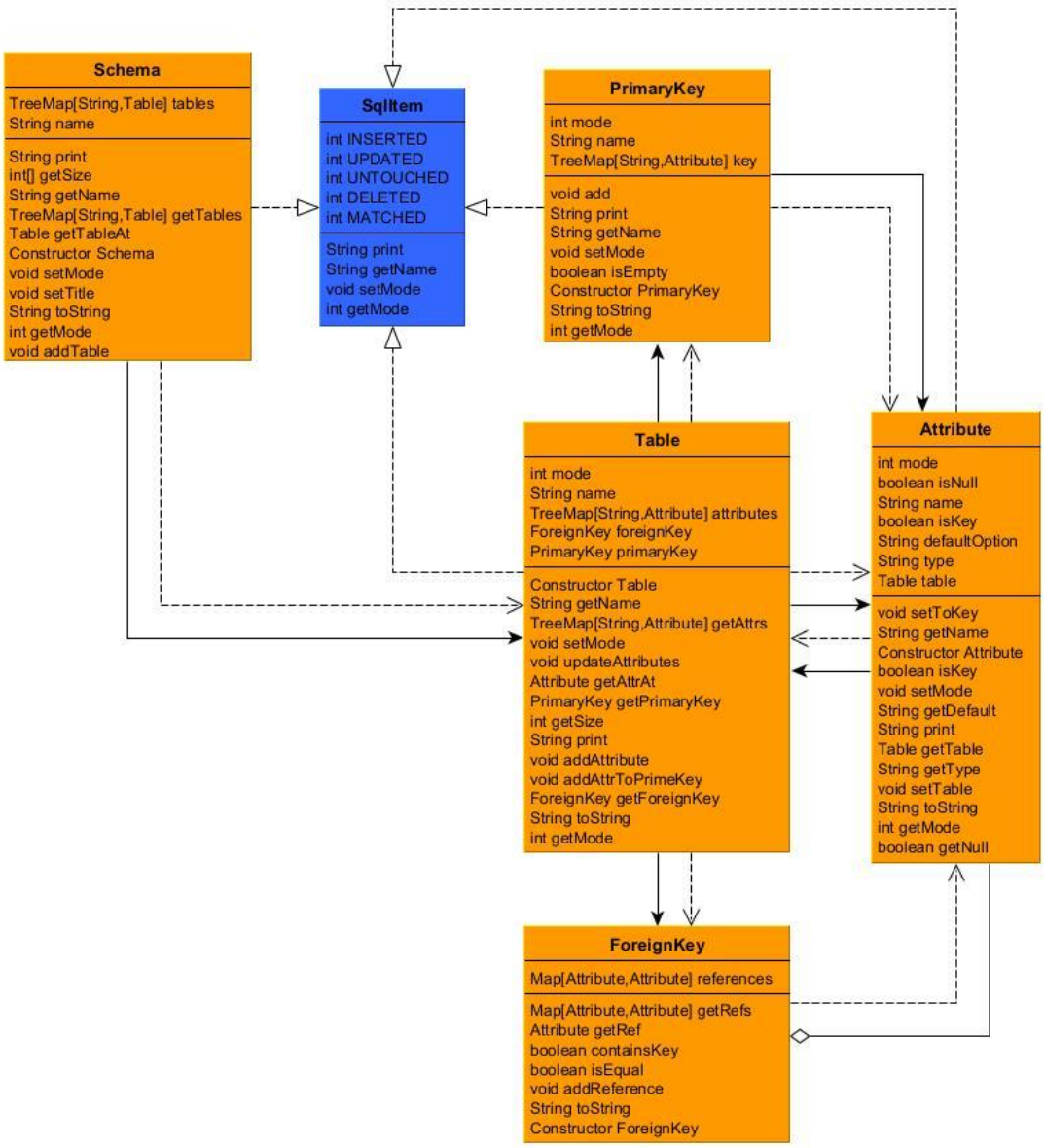

Figure 4.10 Class diagram of the Hecate's 'sql' package using our tool

In the class diagram of the 'sql' package located in Hecate's source package, we can observe that in the diagram produced by our tool, all nodes are present, and no edge is missing when compared to the diagram produced by ObjectAid. However, our diagram has more edges. To be more specific, the Table class has dependency arrow towards the Attribute, ForeignKey and PrimaryKey classes. The dependency relationship between Table and Attribute exists because the Table class has the addAttribute method, which takes an Attribute object as parameter. Moreover, the dependency relationship between Table and ForeignKey as well as PrimaryKey exists because Table's constructor takes an ForeignKey and a PrimaryKey objects as parameters. The same case stands with the dependency arrow starting from the Schema class that ends at the Table class and the dependency arrow starting from the Attribute class that ends at the Table class.

Chapter 5. Epilogue

5.1 Summary and conclusions

In this Diploma thesis, we have presented the design, implementation and validation of a system that reverse engineers the source code of a Java project and depicts its structure visually, in the form of class and package diagrams.

Our system was designed to parse the source code of an object-oriented Java project and identify the project's classes, interfaces, packages and the relationships among them. The Graphical User Interface (GUI) of our system gives the designer the ability to load his desired project and select the classes, interfaces or packages that will be included in the diagram. The designer can export the diagram that has been visualized or select different classes/packages to create and visualize a new diagram.

5.2 Future work

There are a lot that can be added to our system in order to improve the clarity of the visualized and exported diagrams but also the performance of the system.

In order to improve the performance of our system, the main addition, or rather modification, is to improve the parsing of the source code. Our system used the Eclipse's JDT APIs, specifically the AST, to get access to the Java source code of the project. This comes with its flaws since the AST does not provide any information when it comes to Enums. Another downside to using the AST is that it does not provide information regarding a class's local objects. This affects the validity of the created diagrams because relationships among classes that use local objects are not being taken under consideration.

The only solution to this problem is to change the way our system parses the source code. One possible way alternative to parsing the source code, is using the Tree-sitter [Trees22]. The Tree-sitter is a parser tool that is very fast and can also parse any programming language.

To improve the clarity of the diagrams there are two things that need to change in our system. To improve the clarity of the visualized diagrams a library must be developed, that will have the ability to visualize UML diagrams. As we have mentioned in chapter three, there are no libraries that we can use and extend when it comes to creating UML

diagrams. Therefore, a library must be created that will support, the visualization of UML diagrams, real-time addition or subtraction of models from the diagram and the moving of the models inside the canvas. The library must also implement a layout algorithm.

To improve the clarity of the exported diagrams, the spring layout algorithm should be replaced with an orthogonal layout that uses bend minimization to minimize the number of bends on the edges in the diagram.

One further extension that can be added to our tool is the ability to export a diagram in other formats, aside from GraphML. PlantUML [Plan22] is an open source tool that uses simple textual descriptions to draw UML diagrams. Using the information from the parser, we can define a diagram using PlantUML's language, that will be exported to a text file. We can then call PlantUML using the text file as input. The output will be an image that can either be viewed inside our tool or saved as an image file to the disk. The prerequisites in order to run PlantUML are Java and GraphViz [Gravi22], an open source graph visualization software.

References

[Ast06] Abstract Syntax Tree. Available at https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, last retrieved: 2022-10-08, last updated: 2006-11-20.

[Astn22] Abstract Syntax Tree Node. Available at https://www.ibm.com/docs/en/rsar/9.5?topic=SS5JSH_9.5.0/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html, last retrieved: 2022-10-08.

[Deli21] DelianCubeEngine. Available at https://github.com/DAINTINESS-Group/DelianCubeEngine, last retrieved: 2022-10-08, last updated: 2021-01.

[Fxgr21] FXGraph. Available at https://github.com/sirolf2009/fxgraph, last retrieved 2022-08-04, last updated 2021-03-08.

[Grvi22] Graphviz. Available at https://graphviz.org/, last retrieved: 2022-10-07.

[Heca20] Hecate. Available at https://github.com/DAINTINESS-Group/Hecate/, last retrieved: 2022-10-08, last updated 2020-07.

[Jung16] JUNG. Available at https://github.com/jrtom/jung.last, last retrieved: 2022-07-10, last updated 2016-09-07.

[Obje19] ObjectAid. Available at https://marketplace.eclipse.org/content/objectaid-uml-explorer, last retrieved: 2022-03-22, last updated 2019-02-04.

[Papy21] Papyrus Software Designer. Available at https://wiki.eclipse.org/Papyrus_Software_Designer, last retrieved: 2022-03-22, last updated 2021-01-06.

[Plan22] PlanUML. Available at https://plantuml.com/, last retrieved: 2022-10-07.

[Smar21] SmartGraph. Available at https://github.com/brunomnsilva/JavaFXSmartGraph, last retrieved: 2022-08-04, last updated 2021-10-27.

[Star22] StarUML. Available at https://staruml.io/, last retrieved: 2022-03-22.

[Trsi22] Tree-sitter. Available at https://tree-sitter.github.io/tree-sitter/, last retrieved: 2022-10-04.

[Umld22] UML Design Tool Plugin. Available at https://plugins.jetbrains.com/plugin/8394-uml-design-tool-plugin, last retrieved: 2022-03-22.

[Umlg22] UML Generator. Available at https://plugins.jetbrains.com/plugin/15124-uml-generator, last retrieved: 2022-03-22.

[Visu22] Visual Paradigm. Available at https://www.visual-paradigm.com/, last retrieved: 2022-03-22.