PROJECT AND REPORT - 1 SOEN 6971, SUMMER 2024

PROJECT REPORT
ON
ENHANCING THE PTIDEJ TOOL SUITE - PATTERN TRACE IDENTIFICATION DETECTION AND ENHANCEMENT IN JAVA

SUBMITTED TO

Dr. YANN-GAËL GUÉHÉNEUC, Ph.D., eng.

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

IN PARTIAL FULFILMENT OF THE AWARD OF THE DEGREE

OF

MASTER OF ENGINEERING

IN

SOFTWARE ENGINEERING

BY

VISHNU RAMESHBABU

CONCORDIA UNIVERSITY

BONAFIDE CERTIFICATE

Certificate that this project report on "**Pattern Trace Identification Detection and Enhancement in Java",** is the bonafide work of "**VISHNU RAMESHBABU" (STUDENT ID : 40233562),** who carried out the project work under my supervision.

ABSTRACT

This summer project was aimed to integrating PlantUML into the Ptidej Tool Suite Swing GUI (Pattern Trace Identification Detection and Enhancement in Java), such that its libraries can be used to enhance the visualisation and comprehension of the design and implementation of software programs, in particular the different types of relationships among classes, such as aggregation, association, composition, etc., and design decisions, such as design patterns. The project also included converting some existing Java projects composing the Ptidej Tool Suite to the Maven build system for easy compilation, testing, and deployment and to provide consistent builds and a centralised dependency management. The project finally included integrating additional logger features to the existing logging mechanism of the application, to ease the analysis of debugging information wherever required.

TABLE OF CONTENTS

	TITLE	
	ABSTRACT	**3**
	LIST OF FIGURES	**7**
	LIST OF SYMBOLS, ABBREVIATIONS AND FILE EXTENSIONS	**8**
1.0	**INTRODUCTION**	**9**
	1.1 GENERAL	9
	1.2 TECHNOLOGY STACK	10
	1.3 DEPENDENCIES	11
2.0	**PROJECT GOAL**	12
	2.1 OBJECTIVES	12
	2.2 EXISTING SYSTEM	12
	2.3 PROPOSED METHOD	12
	2.4 SYSTEM FLOW	13
	2.5 SYSTEM DESIGN	13
3.0	**PROJECT DESCRIPTION**	**15**
	3.1 INTEGRATION OF PLANTUML LIBRARY	15
	3.1.1 PLANTUML STRING GENERATOR	15

3.1.2 TESTING PLANTUML GENERATOR 17
3.2 ENHANCEMENT OF PTIDEJ UI 17
3.3 CONVERSION OF EXISTING PROJECTS TO MAVEN 17
3.4 INTEGRATING LOG4J2 18

4.0 IMPLEMENTATION 19

4.1 SETTING UP PTIDEJ 19
4.2 IMPLEMENTING PLANTUML GENERATOR 19
4.2.1 PLANTUML VISITOR CLASS 19
4.2.2 DESIGNING THE PLANTUML VISITOR CLASS 20
4.3 USER INTERFACE FOR PLANTUML VISUALIZATION 27
4.3.1 TOOLBAR INTEGRATION 27
4.3.2 PLANTUML DIAGRAM DISPLAY WINDOW 30
4.4 CONVERTING PROJECTS TO MAVEN 40
4.4.1 INITIAL STEPS 41
4.4.2 BUILD AND DEPENDENCIES 43
4.5 INTEGRATING LOG4J2 46
4.5.1 LOGGING PROPERTIES 47
4.5.2 LOGGER WRAPPER 48

5.0 RESULTS, TESTING AND VALIDATION 55

5.1 RESULTS 55
5.1.1 PLANTUML CODE AND IMAGE 55

5.1.2 PTIDEJ LOGGING OUTPUT 60

5.2 TESTING AND VALIDATION 62

6.0 CONCLUSION AND FUTURE SCOPE 64

6.1 CONCLUSION 64

6.2 FUTURE SCOPE 64

REFERENCES 65

LIST OF FIGURES

FIGURE NO	NAME OF THE FIGURE	PAGE NO
2.1	SYSTEM FLOW FOR PLANTUML IMAGE GENERATION	13
2.2	PROJECTS USING MAVEN SYSTEM	14
2.3	LOG4J2 IMPLEMENTATION	14
3.1	VISITOR INTERFACE HIERARCHY	16
3.2	SEQUENCE DIAGRAM OF LOGGER	18
4.1	SEQUENCE DIAGRAM OF PLANTUML GENERATION	27
4.2	FILE CHOOSER WINDOW	29
4.3	WINDOW FRAME	33
4.4	MAVEN DIALOG BOX	41
5.1	FACADE2 CLASS FILE CONTENTS	55
5.2	SELECTING LOAD PLANTUML PROJECT	56
5.3	SELECTING DEMIMA FACADE.PTIDEJ	56
5.4	PLANTUML WINDOW IMAGE VISUALISATION	57
5.5	PLANTUML IMAGE OF FACADE2	60
5.6	LOGGER OUTPUT TO CONSOLE	61
5.7	LOG FILE OUTPUT	61
5.8	JUNIT TESTING RESULTS - PLANTUML CODE	63
5.9	JUNIT TESTING RESULTS - PLANTUML IMAGE	63

LIST OF SYMBOLS, ABBREVIATIONS AND FILE EXTENSIONS

- -- association
- O-- aggregation
- *-- composition
- --^ classes
- ..^ interfaces
- Ptidej Pattern Trace Identification Detection and Enhancement in Java
- UI User Interface
- UML Unified Modelling Language
- API Application Programming Interface
- JAR Java Archive Resource file
- AST Abstract Syntax Tree
- .ptidej File extension that contains Name and JavaCode as variables.
- JDK Java Development Kit
- IDE Interactive Development Environment
- JSON Javascript Object Notation
- YAML YAML Ain't Markup Language

CHAPTER - 1

INTRODUCTION

1.1 GENERAL

The **Ptidej Tool Suite** is a software tool built in Java to identify, detect, and analyse design patterns in object-oriented programs, in Java, C++, etc. It provides PADL, Pattern and Abstract-level Description Language, which is a meta model to describe and analyse programs at different abstraction levels. When a package or collection of packages is selected as a source input, parser instantiate PADL models that can then be visualised to show the classes and interface entities present in the project/package and establish relationships between each entity.

The main task was set to develop a Visitor that visits each entity in an abstract model that contains package, classes, interfaces hierarchically and checks if the relationship between classes or interfaces is an aggregation, association, or a composition relationship. If it is available then, accordingly, the generator should build a string that conforms to the PlantUML domain-specific language (DSL) of representing classes and relationships along with integrating the PlantUML library itself into Ptidej and creating a separate feature that enhances the current visualization capabilities.

FEATURES

The features to be built in Ptidej are

1) Visualizing PADL models using PlantUML Library.
2) Converting non-Maven projects, like Caffeine, to Maven.
3) Implementing Log4j2 alongside existing custom loggers.

1.2 TECHNOLOGY STACK

This section contains the technology stack that is used to run Ptidej and dependencies that were used to implement the required features.

JDK 21

Java Development Kit 21 is a minimum requirement to run Ptidej. The JDK provides a development environment used for developing Java applications and provides a vast set of libraries. It also provides a Java Runtime Environment, a Java Virtual Machine, along with the compiler and other tools.

MAVEN

For the development of these features, Maven 3.9.6 is required as a build automation tool. It is primarily used for managing dependencies, configuring goals for during the build process and provides a defined life cycle, like clean, test, verify, and build. It also fetches external dependencies from the central Maven repository and also from the local repository and keeps it up to date through an XML file called `pom.xml`.

ECLIPSE IDE

This project is mainly developed using Eclipse IDE and it is a versatile tool specifically for developing Java applications. Eclipse IDE provides a rich GUI which consists of project explorer to view the various packages, folders of a workspace, debugger that uses breakpoints to pause the execution at a desired point in the program. It also provides easy integration with external tools like gradle, maven, git features, etc.

1.3 DEPENDENCIES

PLANTUML LIBRARY

PlantUML Library is a versatile tool that helps in generating UML diagrams like sequence diagram, usecase diagram, object diagram, class diagram etc and also supports generating uml diagrams using JSON data convention, YAML data, etc. The PlantUML Library can be downloaded from its official website https://PlantUML.com/download This library can be downloaded as a compiled JAR, to ease its integration in Ptidej.

JUNIT

JUNIT is a standard testing framework used for the verification of Java applications. It supports a test-driven development approach by enabling test automation, as it provides the means to build and run test suites.

It provides annotations like `@Before` and `@After`, which specifies whether a method should be run before or after test methods as the former annotation can be used to set any necessary resources required for the test methods and the latter annotation can be used to clean up or terminate any open connections.

Likewise there are many such annotations, which are useful. Also, assertions like `assertEqual`, `assertFalse`, etc. can be used to check methods if the values compared are equal or if it evaluates to false, respectively.

CHAPTER - 2
PROJECT GOAL

2.1 OBJECTIVE

Major milestones of the Ptidej project includes:

1. Integrating PlantUML library with existing system to enhance visualization.
2. Implementing the standard directory structure in existing projects to use Maven.
3.Additional logging features using log4j2 to existing logging capabilities in the system.

2.2 EXISTING SYSTEM

The existing GUI uses Java AWT and Java Swing to display boxes and lines to represent classes, interfaces, and their relationships. It does not have advanced visualisation capabilities and could benefit from a library like PlantUML. Existing logging mechanism uses a custom class that extends `PrintWriter` class to write to both `File` and `console` but using an advanced logger system like Log4j2 would enhance the debugging process and help to search a specific execution trace much quicker and offers a centralised point to configure the logger. Finally, a few projects in Ptidej do not use Maven yet as dependency management and build tools, which would make them easier to compile, test, and deploy.

2.3 PROPOSED METHOD

Regarding the implementation of PlantUML, the proposed method is to create a separate Visitor class to visit the Abstract model that is obtained after reading the input class files. Then, it would add the PlantUML library as part of the project and use the methods to visualize the elements as a separate feature within the application.

Then, the approach is to convert projects into Maven and configuring their dependencies through `pom.xml` files and arrange the class files and the resources in a standard project structure expected by Maven.

Finally, to add log4j2 as a dependency to Ptidej and configure it to output a specific pattern for easy analysis of traces, the approach is to create wrapper classes to channel both ProxyConsole and logger output through a single wrapper.

2.4 SYSTEM FLOW

For PlantUML, a visitor class is designed to generate the appropriate PlantUML DSL This PlantUML DSL is then used to generate the respective PlantUML image by sending this as an input to the PlantUML library. The system flow is presented in the flowchart in Fig 2.1.

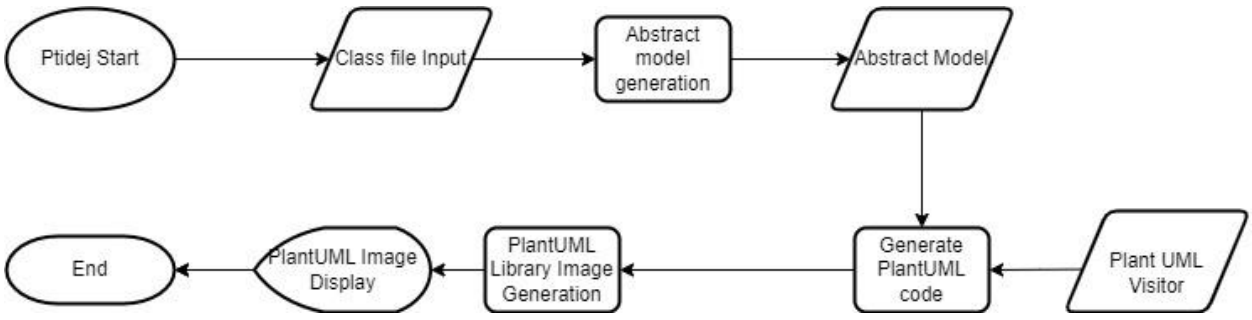


Fig 2.1 System Flow for PlantUML Image Generation

2.5 SYSTEM DESIGN

Four projects in Ptidej must be converted into Maven projects, namely `Caffeine`, Caffeine Analyses, Caffeine Examples, and Caffeine Tests. It is designed to work as depicted in Fig 2.2

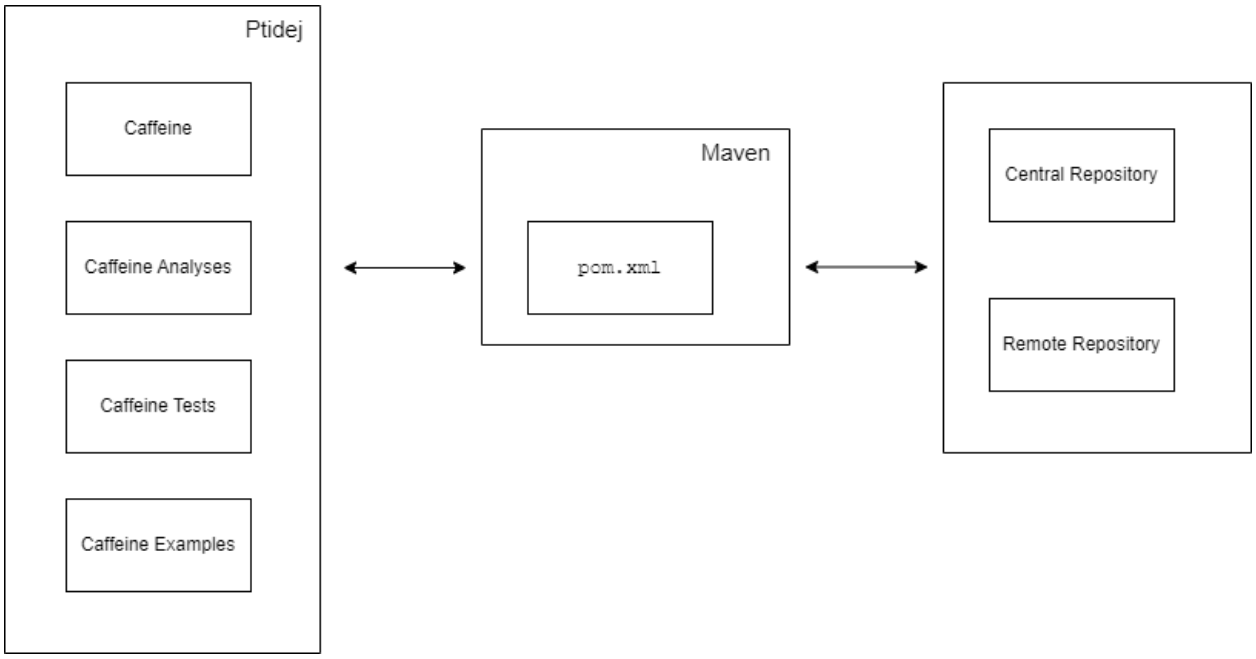


Fig 2.2 Projects using Maven System

For integrating Log4J2, we write two wrapper classes, LoggerWriter wrapper class, for generating logger instance based on Log4J2 and another wrapper to combine both print Writer methods and loggerWriter methods which can be used to nest similar writer classes.

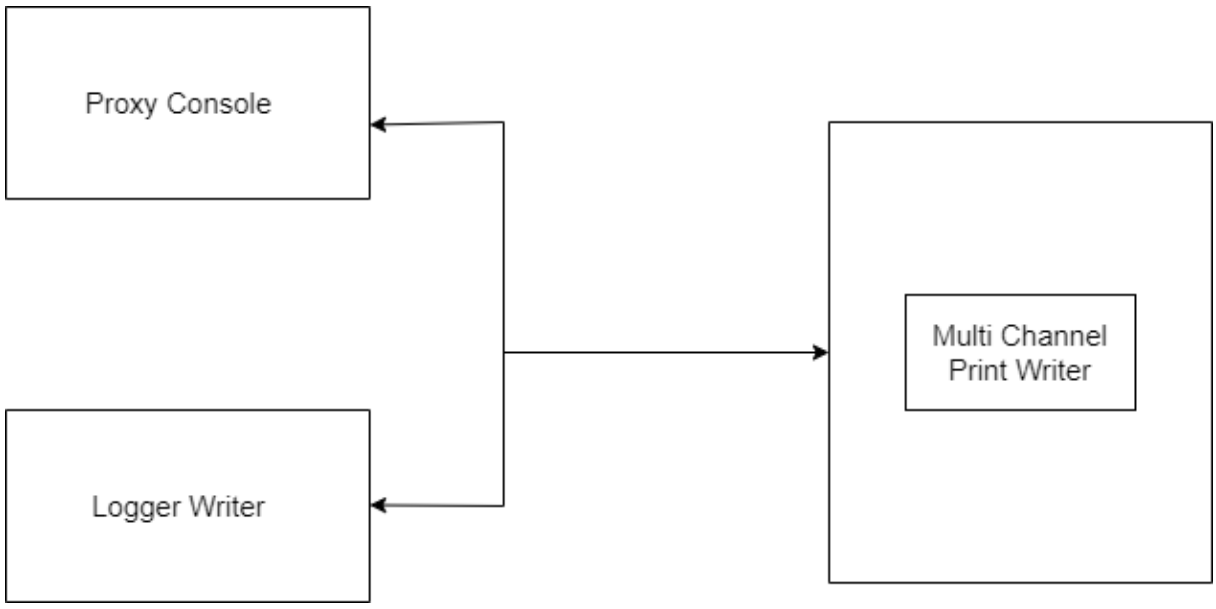


Fig 2.3 Log4j2 Implementation

CHAPTER 3
PROJECT DESCRIPTION

In this chapter we can review the various modules/processes developed for Ptidej for the duration of the summer project.

- Integration of PlantUML Library.
- Conversion of Existing Projects to Maven.
- Implementing Log4j2 feature.

3.1 INTEGRATION OF THE PlantUML LIBRARY

To achieve offline visualization of PlantUML diagrams, we must add the PlantUML library JAR file. First we need to visit https://PlantUML.com/download to download the compiled JAR under any appropriate licence listed. Then, we add the JAR to the resources directory of the `PADL Analyses` project. This project is where the code related to generating the PlantUML code will be implemented. This library should also be added as a dependency to Maven which ensures the project's build.

3.1.1 PLANTUML INPUT GENERATOR

Before we integrate the PlantUML library, we first require a Visitor class to visit each package, class, method, and field of a PADL model, which is obtained from the `AbstractRepresentationWindow` when a .ptidej file is opened as an input. The Visitor class consists of a `StringBuilder`, which is used to create an instance of the PlantUML DSL as the visitor visits each entity.

A .ptidej file contains 'Name' that contains the title of the UI window that lists the checkboxes and visualises the classes and interfaces. 'Java code' contains the directory/class path of the packages separated by semicolon (;)

The visitor class implements the IGenerator interface, which implements a IVisitor interface. The visitor class can be defined as a design pattern that isolates the algorithm used on the object structure which is an Abstract Syntax Tree (AST). Instead of modifying the AST every time when we need to add a new algorithm to perform a set of operations on the elements.

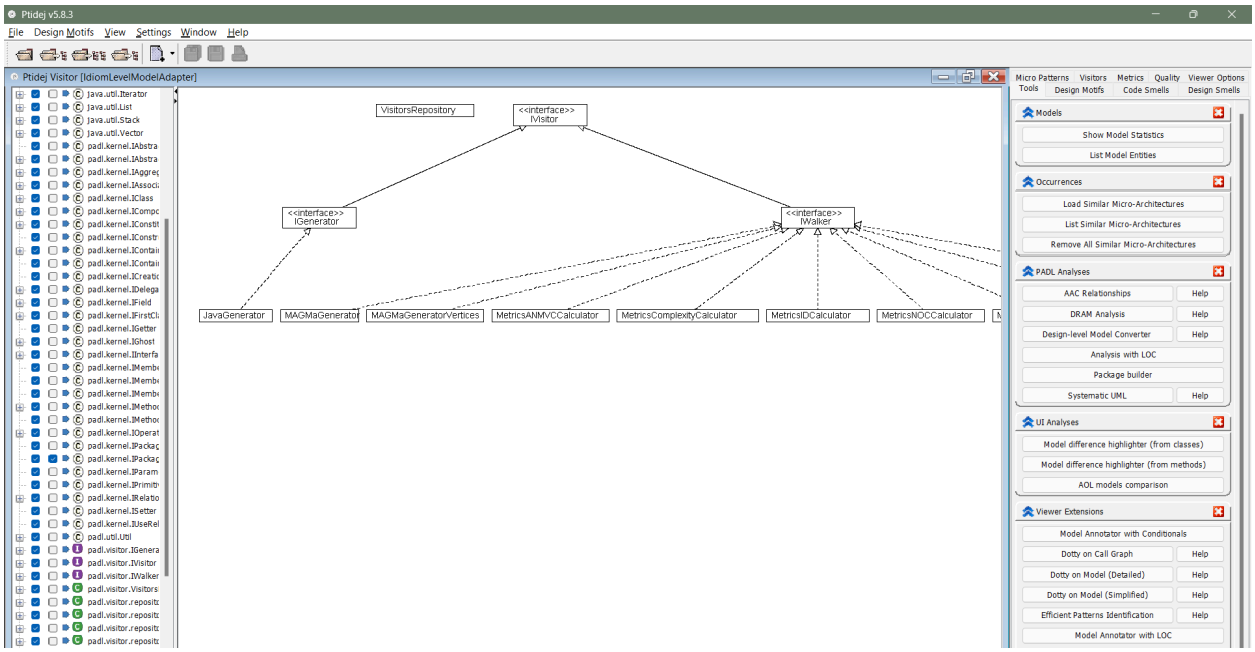


Fig 3.1 Visitor Interface Hierarchy

To create an instance of the PlantUML DSL, we create a separate Visitor class that has specific operations: open, close, and visit methods. Open methods are used to find the implemented interfaces, inherited/inheriting classes by running an iterator on the constituents. The close methods are usually to reset the current entity and perform any required operations before it exits the entity. The visit methods in this context are used to apply the logic, which is to construct relationships between identified entities. As for each visit method, the StringBuilder will be updated with appropriate code as specified by the PlantUML DSL.

3.1.2 TESTING PLANTUML GENERATOR

Test cases are mandatory whenever a feature is built because they validate the expected behaviour and ensures that the feature does not break when changed. There are also two types of testing done for this feature, which is JUnit testing (standalone tests) and integration testing (testing the feature on the entire application).

JUnit testing involved verifying the methods by comparing the generated PlantUML code against a pre-generated code stored in a text file and also against an incorrect PlantUML code. The reason being that, if the implementation of the feature would change in the future, this test can serve as a validation point.

3.2 ENHANCEMENT OF THE PTIDEJ UI

The Existing user interface of Ptidej must be modified to accommodate the visualization of PlantUML DSL within the application. There are two major steps to output the visualization, one is to create an option in the context menu and an icon in the toolbar for the user to select an input file from the file explorer. The next step would then be creating a separate window frame to output the generated image along with the available classes and interfaces listed as dropdown checkboxes.

3.3 CONVERSION OF EXISTING PROJECTS TO MAVEN

In Ptidej, some existing projects like Caffeine, Caffeine Analyses, Caffeine Examples and Caffeine tests do not follow the directory structure and build environment of Maven. This was a pending issue because there is a need for a centralized dependency management using a single `pom.xml`, segregating resources, Java files, libraries into separate subdirectories, which gives flexible project management and ease building and deployment in any environment. This integration with Maven helps large applications like Ptidej, which has many projects and many dependencies among projects.

3.4 INTEGRATING LOG4J2

Log4J2 is the logging tool by Apache logging services that has significant improvements over its initial version and the '2' in Log4j2 represents the 2nd version. It has six different types of severity levels which are trace, debug, info, warn, error, fatal in ascending order. It is part of the Apache family of libraries that can be included in the project as a dependency in Maven.

Log4J2 has a set of default configurations but can be modified by including a properties file. It can be configured to output to console and file and also define the pattern of the log message, specifically for each output type. It can also be configured to set which severity level to output and to which directory the log file should be written and if it is an append type or generating a new log file for each day/hour etc.

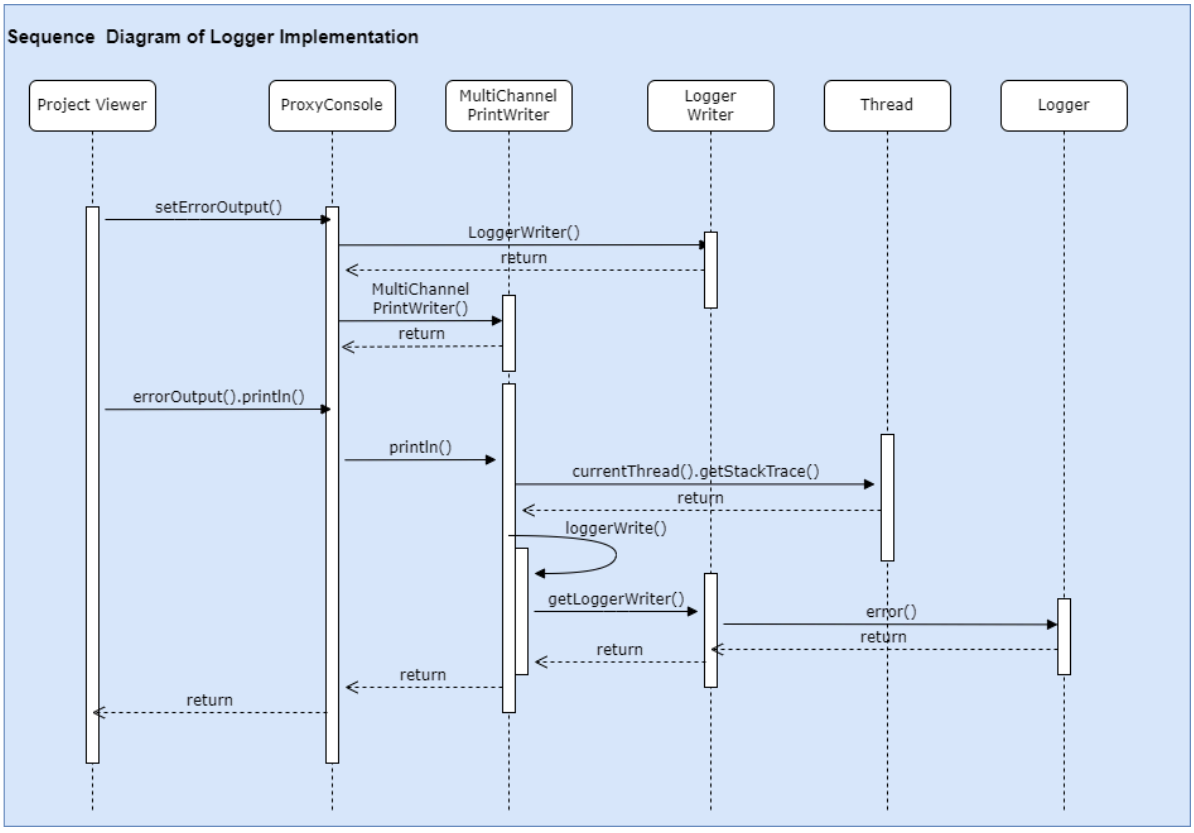


Fig 3.2 Sequence Diagram of Logger

CHAPTER 4
IMPLEMENTATION

4.1 SETTING UP PTIDEJ

First, we must ensure JDK 21 is installed in the work environment, and if not, it can be downloaded from Oracle official website, and based on the OS Environment and architecture, the respective installer can be download to require Eclipse IDE 2024-03 to be installed and Maven 3.9.6 should be installed within Eclipse using Eclipse marketplace.

Then, the project must be cloned from `https://github.com/ptidejteam/ptidej-Ptidej`. After that we need to import ptidej into Eclipse as a maven project. Once the project is built, we can select the projectViewer class located at `DeMIMA UI Standalone Swing project/ptidej/viewer/` and then run as a Java application. We could also run the entire ptidej application from the JAR file of the project using a single line command,

```
java -jar "DeMIMA UI Viewer Standalone Swing/target/demima-ui-viewer-swing-1.0.0-jar-with-dependencies.jar"
```

4.2 IMPLEMENTING PLANTUML GENERATOR

4.2.1 PLANTUML VISITOR CLASS

The Ptidej application has an object structure implemented in the IAbstractModel interface and is a composite model. This IAbstractModel serves as abstract class, which is further implemented by IIdiomModel, which describes high level design patterns in a

given set of class files and ICodeModel, which describes the code structure of classes. Once an input is given, the model generator generates an intermediate ICodeModel, which is then further analysed to give an IIdiomModel.

The IIdiomModel implements IAbstractModel, it has the provision to accept an instance of a IVisitor. The `generate(final IGenerator aBuilder)` can be used to send in the instance of the PlantUML visitor class that can return a string type.

4.2.2 DESIGNING THE PLANTUML VISITOR CLASS

As the Visitor interface is implemented in the PlantUML visitor class, we can implement `open/close` methods for entities and visit methods for entity relationships and entity fields, method invocation and parameters. For the purpose of PlantUML, we implement open and close only for the class and interface types and visit methods for aggregation, association and composition types only.

In this code snippet, we can see IGenerator being implemented and two StringBuilder instances are created. The `PlantUMLBuilder` to append the general PlantUML Code that creates the structure and identifies classes and interfaces. `PlantUMLBuilderRelationship` is used to generate intended relationships between entities. We then create `currentEntity`**,** a `IFirstClassEntity type` variable that holds the currentEntity being referenced.

PlantUMLGenerator() - The constructor appends the first line of the output with `@startuml` PlantUML convention, which indicates the start of the PlantUML code.

getCode() - This is used to return the final output of the string builder.

```
public class PlantUMLGenerator implements IGenerator {
protected final StringBuilder PlantUMLBuilder =new StringBuilder();
protected final StringBuilder PlantUMLBuilderRelationship = new
StringBuilder();
     private IFirstClassEntity currentEntity;
     public PlantUMLGenerator() {
          this.PlantUMLBuilder.append("\n@startuml\n");
     }
     public String getCode() {
          return this.PlantUMLBuilder.toString();
     }
     public void reset() {
          PlantUMLBuilder.setLength(0);
     }
```

OPEN METHODS

open(IClass cls) - In this code snippet, an overloaded method accepts the IClass abstract type. The currentEntity is set with the cls variable. Here we obtain the className and check if the entity is an abstract class and append the string to the stringBuilder instance. An iterator is obtained on the inherited entities.The FirstClassEntity types create an iterator for both inherited classes and interface and we can use this iterator because the IClass extends IFirstClassEntity. As per the PlantUML DSL, '--^' symbol is used to indicate the currentEntity inherits from the class entity. For interfaces, the '..^' is used to indicates the current entity implements the interface entity.

```
public void open(IClass cls) {
          currentEntity = cls;
          String className = String.valueOf(cls.getName());
```

```
            PlantUMLBuilder.append("\n");
            if (cls.isAbstract()) {
                  PlantUMLBuilder.append("abstract ");
            }
            this.PlantUMLBuilder.append("class " + className + " {");
            PlantUMLBuilder.append("\n");
            Iterator  iterator = cls.getIteratorOnInheritedEntities();
            if (iterator.hasNext()) {
                  while (iterator.hasNext()) {
                        IFirstClassEntity  entity  =  (IFirstClassEntity)
iterator.next();
                        if
(String.valueOf(entity.getName()).equals("Object")) {
                              continue;
                        }
                        this.PlantUMLBuilderRelationship.append("\n");

this.PlantUMLBuilderRelationship.append(entity.getName());
                        this.PlantUMLBuilderRelationship.append("    --^
");

this.PlantUMLBuilderRelationship.append(className);
                        if (iterator.hasNext()) {

this.PlantUMLBuilderRelationship.append("\n");
                        }
                  }
            }
            iterator = cls.getIteratorOnImplementedInterfaces();
            if (iterator.hasNext()) {
                  while (iterator.hasNext()) {
                        this.PlantUMLBuilderRelationship.append("\n");
```

```
this.PlantUMLBuilderRelationship.append(((IFirstClassEntity)
(iterator.next())).getName());
this.PlantUMLBuilderRelationship.append(" ..^ ");
this.PlantUMLBuilderRelationship.append(className);
                if (iterator.hasNext()) {
this.PlantUMLBuilderRelationship.append("\n");
                      }
                }
           }
     }
```

open(IInterface iInterface) - For interface entities, it is similar to the `open(IClass cls)` method, except that the name 'interface' is added to the stringBuilder and we use the '--^' to represent the inheritance of interface entities with the current interface entity

```
          this.PlantUMLBuilder.append("interface " +
interfaceName + " {\n");
          iterator =
iInterface.getIteratorOnInheritingEntities();
          if (iterator.hasNext()) {
               while (iterator.hasNext()) {
                    IFirstClassEntity entity =
(IFirstClassEntity) iterator.next();

this.PlantUMLBuilderRelationship.append("\n");

this.PlantUMLBuilderRelationship.append(entity.getName() );
                    this.PlantUMLBuilderRelationship.append("
--^ ");
```

```
this.PlantUMLBuilderRelationship.append(interfaceName);
                    if (iterator.hasNext()) {

this.PlantUMLBuilderRelationship.append("\n");
                    }
                }
```

CLOSE METHODS

Close methods ensure that the entities are closed for modification, meaning that its visit is finished. It is used to properly close the elements when the elements are processed and ensure the results are properly appended.

close(IAbstractModel model) - This method adds newlines and appends the value of the relationship string builder and finally appends '@enduml' which indicates the end of the PlantUML code.

```
    public void close(IAbstractModel model) {
        this.PlantUMLBuilder.append("\n"                    +
this.PlantUMLBuilderRelationship.toString() + "\n");
        this.PlantUMLBuilder.append("@enduml");
    }
```

close(IClass cls) & **close(IInterface iface) -** Both these methods append a closing braces and a new line to indicate the closure of a class or an interface.

```
    public void close(IInterface iface) {
                currentEntity = null;
          this.PlantUMLBuilder.append("\n}\n");
    }
```

```
    public void close(IClass cls) {
          currentEntity = null;
          this.PlantUMLBuilder.append("\n}\n");
    }
```

VISIT METHODS

Visit methods handles the relationship type between different identified entities:

visit(final IAggregation aggregation) - This method handles the entities that has aggregation style relationship by adding ‘o--’ between the entity names of the current and the target entities and appends ‘: aggregation’ at the end of the string as per the PlantUML DSL. One of the outputs of this method is given in the example:

Example: `ModelGraph o-- Constituent: aggregation`

visit(final IComposition composition) - This method handles the entities that has composition style relationship by adding ‘*--’ between the entity names of the current and the target entities and appends ‘: composition’.

Example: `Implementation-- IPrimitiveFactory: composition`

visit(final IAssociation association) - This method handles the entities that has association style relationship by adding ‘--’ between the entity names of the current and

the target entities and appends ‘: association’ at the end of the string as per the PlantUML DSL.

Example: `Specialisation -- Point : association`

```
public void visit(final IAggregation aggregation) {
     if (currentEntity != null) {
this.PlantUMLBuilderRelationship.append(currentEntity.getName())
.append(" o--
").append(aggregation.getTargetEntity().getName()).append('
').append(": aggregation\n");
          }
     }
public void visit(IAssociation association) {
     if (currentEntity != null) {
PlantUMLBuilderRelationship.append("\n").append(currentEntity.ge
tName()).append(" --
").append(association.getTargetEntity().getName()).append('
').append(": association\n");
          }
     }
public void visit(IComposition composition) {
          if (currentEntity != null) {
PlantUMLBuilderRelationship.append("\n").append(currentEntity.ge
tName()).append(" *--
").append(composition.getTargetEntity().getName()).append('
').append(": composition\n");
          }
}
```

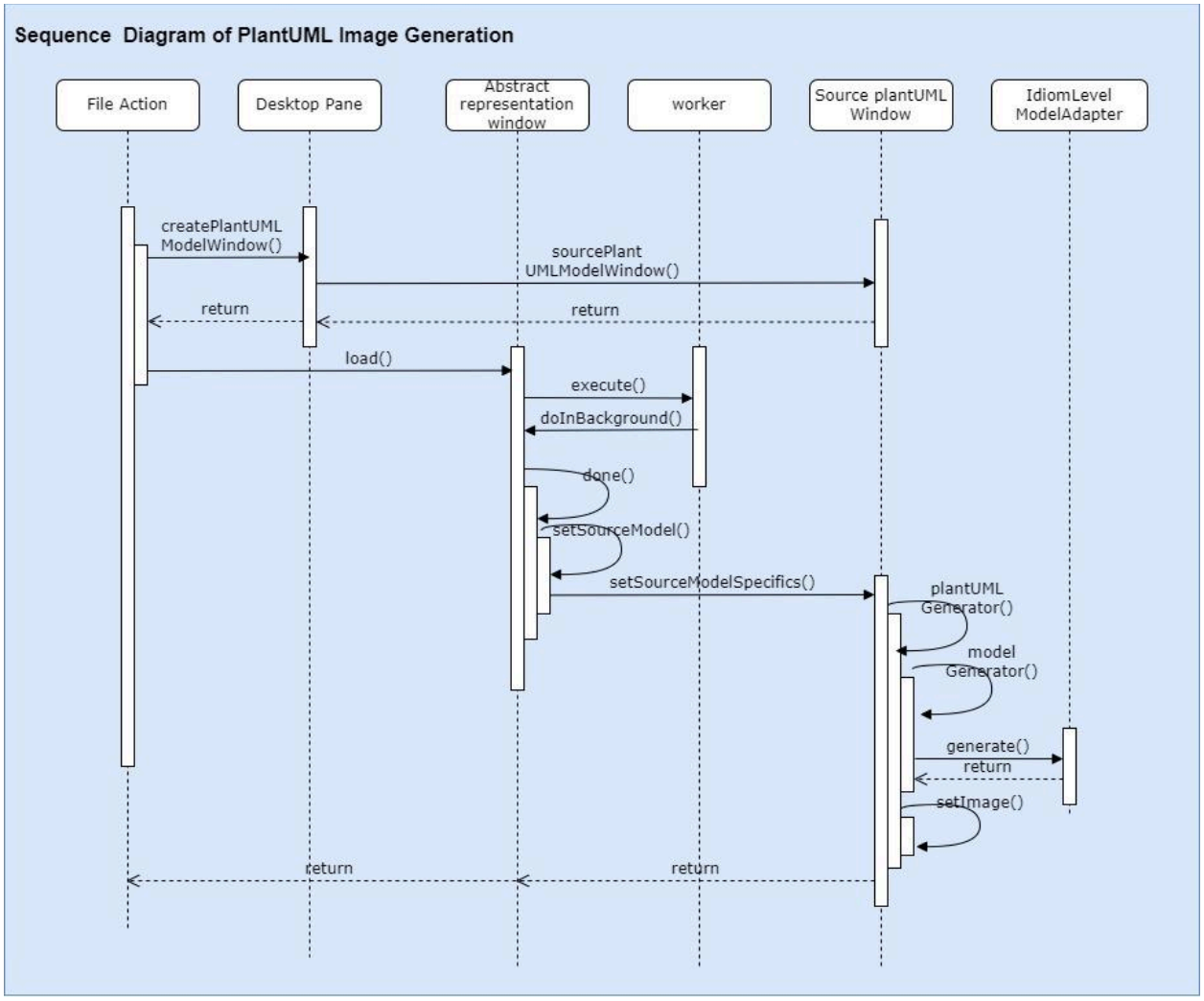


Fig 4.1 Sequence Diagram of PlantUML Generation

4.3 USER INTERFACE FOR PLANTUML VISUALIZATION

This user interface change comprises the visualisation of PlantUML diagrams as an additional feature to Ptidej.

4.3.1 TOOLBAR INTEGRATION

An icon is to be added to the toolbar and for this, two entries need to be added in the `PtidejResourceBundle.java` class file. The variable reference to load PlantUML based window is added in the toolbar by using ptidej.viewer.ui.ToolBar. An icon is defined in a similar fashion.

```
{ "ptidej.viewer.ui.ToolBar::CMD_LOAD_PlantUML_PROJECT",
                                "Load PlantUML Project" },
{"ptidej.viewer.ui.ToolBar::CMD_LOAD_PlantUML_PROJECT_ICON",
                                "OpenHierarchical24.gif" },
```

In addition, a static final String variable containing the value must be created in `Resource.java`, because this variable will then be referenced to add in toolbar and load the respective actions. It is used for internationalisation.

```
    public static final String LOAD_PlantUML_PROJECT =
"LOAD_PlantUML_PROJECT";
```

In `Toolbar.java`, the `load_PlantUML_project` variable is referenced to add it as a button, groupName and to enable the button.

```
      this.addToolbarButton(Resources.LOAD_PlantUML_PROJECT,
                    Resources.FILE,true);
```

In `FileAction.java`, we need to define the action to be performed when the icon is clicked. Once it is checked, we can call a method `loadPlantUMLProject()`.

```
    public void actionPerformed(final ActionEvent anActionEvent)
{final String action = anActionEvent .getActionCommand();
     if (action.equals(Resources.NEW_GRAPHICAL_PROJECT)) {
this.createGraphicalProject();}
else if (action.equals(Resources.LOAD_PlantUML_PROJECT)) {
this.loadPlantUMLProject(); }
```

`loadPlantUMLProject()` - This method is used to define the file chooser using Utils.loadFile that takes the unique instance of DesktopFrame, multiselection boolean false, set the title of dialog window, the file type to accept and filter description name.

```
private void loadPlantUMLProject() {
     final File file =Utils.loadFile(DesktopFrame.getInstance(),
false,"Choose Ptidej project file", "ptidej", "Ptidej project
files");
     if (file == null) { return; }
     final Properties properties = new Properties();
     try {
          properties.load(new FileInputStream(file)); }
     catch (final IOException e) { e.printStackTrace
(ProxyConsole.getInstance().errorOutput()); return;}
     DesktopPane.getInstance().createPlantUMLModelWindow();
          this.processSelectedFile(file);}
```

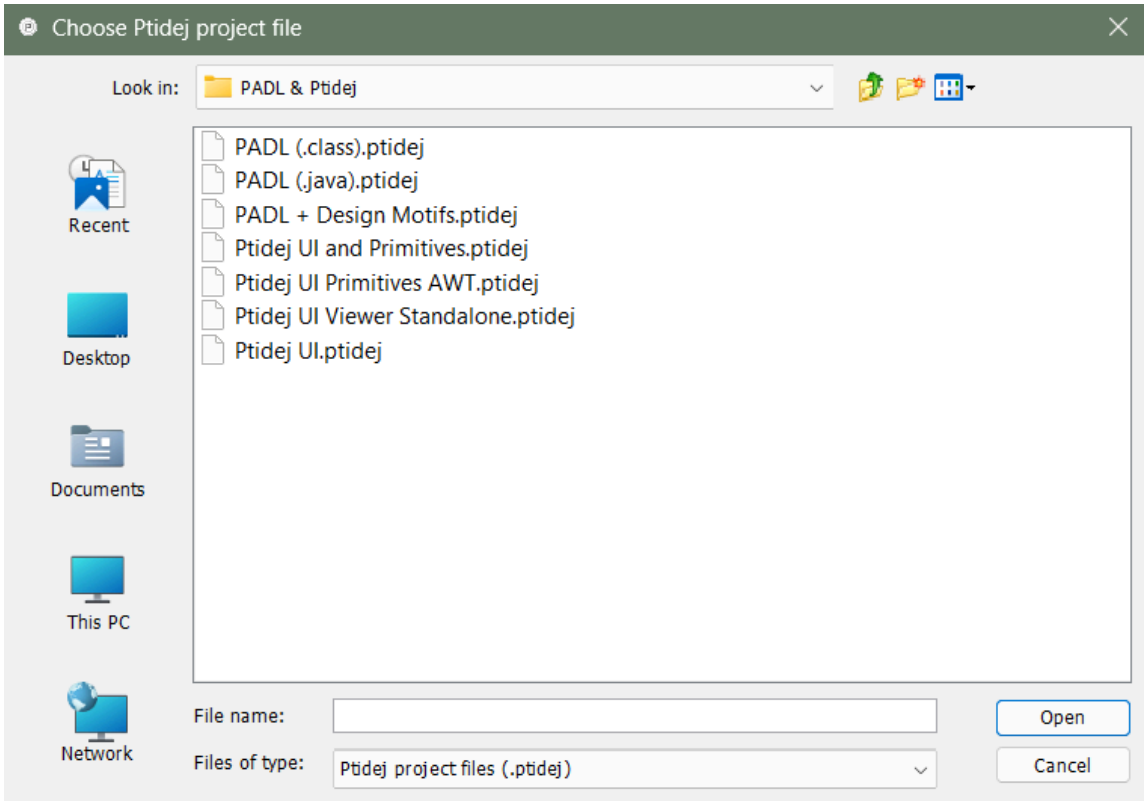


Fig 4.2 File Chooser Window

Then, an instance of properties is created and the file is loaded into an instance of FileInputStream. After that, an unique instance of `DesktopPane` is created and `createPlantUMLWindow()` is called. Then, `processSelectedfile(final File file)` is called, sending the file as input and with this, the multiple file paths in the .ptidej file input that contain all the classes will be processed iteratively and a respective AbstractModel will be created.

In `DesktopPane.java`,we create an instance of `SourcePlantUMLModelWindow` and set it as the current desktop window.

```
    public void createPlantUMLModelWindow() {
        final AbstractRepresentationWindow window =
            new SourcePlantUMLModelWindow();
        this.currentDesktopWindow = window;
        this.setWindowProperties(window);
        this.currentDesktopWindow = window;
    }
```

4.3.2 PLANTUML DIAGRAM DISPLAY WINDOW

After we have created an instance of sourcePlantUMLWindow, the constructor sets outputImage path and the modelStatistics. Then, the content pane is set with the border layout and the dimensions. The tree pane is added with listeners and to list the elements. A canvas panel is then created to display the image and add it to the scroll pane and the background viewport is set accordingly along with the vertical and horizontal scrollbar listeners and renderers. The entire window is split into two using JSplitPane with the tree pane on the left and the canvas pane on the right.

```
public SourcePlantUMLModelWindow() {
```

```
            this.setImagePath("../OutputUML.png");
            this.modelStatistics = new SilentModelStatistics();
            this.getContentPane().setLayout(new BorderLayout());
            this.treeRoot = new DefaultMutableTreeNode();
            this.treeRoot.setUserObject(new JLabel(""));
            this.tree = new JTree(this.treeRoot);
            this.tree.addTreeWillExpandListener(new
TreeWillExpandListener() {
                  public void treeWillCollapse(final
TreeExpansionEvent aTreeExpansionEvent) throws
ExpandVetoException {
  if (aTreeExpansionEvent.getPath().getPathCount() < 2) {
     throw new ExpandVetoException(new
TreeExpansionEvent(SourcePlantUMLModelWindow.this.tree, null));
     } }
public void treeWillExpand(TreeExpansionEvent event) throws
ExpandVetoException {
                  }
            });
 final TreeCellRenderer renderer = new HierarchicalTreeCell
Renderer();
     this.tree.setCellRenderer(renderer);
     this.tree.setCellEditor(new HierarchicalTreeCellEditor());
     this.tree.setEditable(true);
     this.canvasPanel = new CanvasPanel();
     final ScrollPane scrollPane = new ScrollPane
(this.canvasPanel);
   scrollPane.getViewport().setBackground(Color.WHITE);
   scrollPane.getHorizontalScrollBar().addAdjustmentListener(new
AdjustmentListener() {
   public void adjustmentValueChanged(final AdjustmentEvent e) {
     SourcePlantUMLModelWindow.this.canvasPanel.repaint(); } });
```

```
     scrollPane.getVerticalScrollBar().addAdjustmentListener(new
AdjustmentListener() {  public void adjustmentValueChanged(final
AdjustmentEvent e) {
     SourcePlantUMLModelWindow.this.canvasPanel.repaint();}});
     this.imageLabel = new JLabel();
     scrollPane.setViewportView(imageLabel);
   final JSplitPane treeAndGraphSplitPane = new JSplitPane
(JSplitPane.HORIZONTAL_SPLIT, new ScrollPane(this.tree),
scrollPane);
     treeAndGraphSplitPane.setOneTouchExpandable(true);
     treeAndGraphSplitPane.setDividerLocation(200);
     this.getContentPane().add(treeAndGraphSplitPane,
BorderLayout.CENTER);     }
```

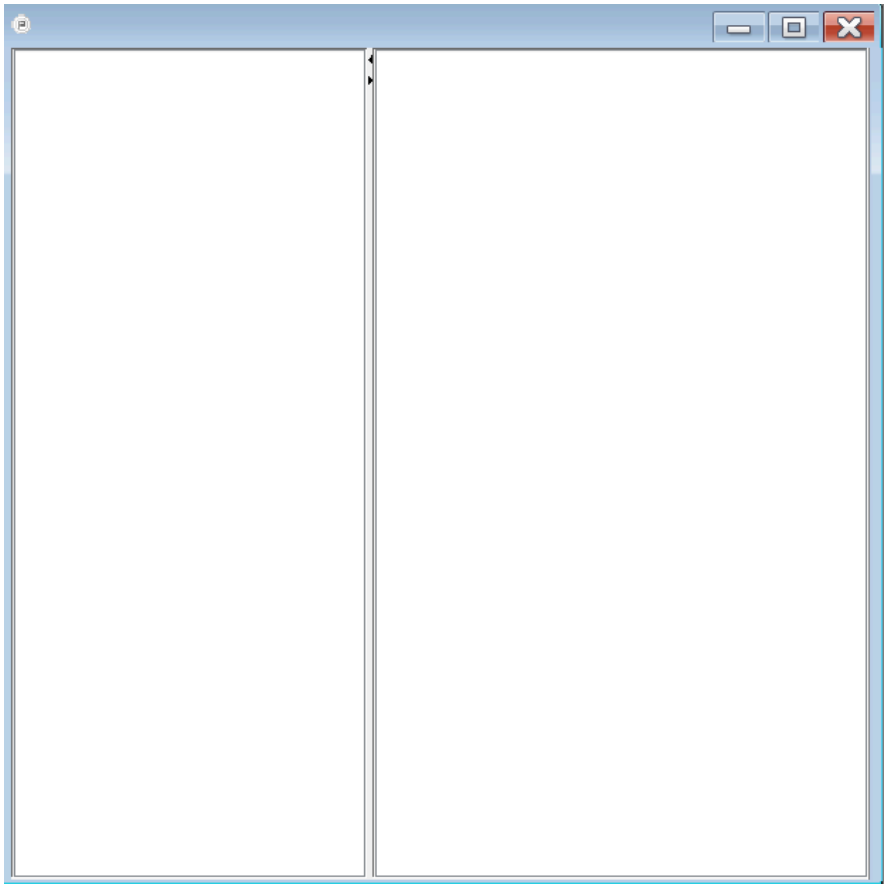


Fig 4.3 Window Frame

SETTING SPECIFICS OF SOURCE MODEL

After the constructor is called, the sourceModelSpecifics() method will be called by the SwingWorker when the done() method is executed. This ensures the tree is built with the tree node elements by calling the **getIteratorOnTopLevelEntities()** from the sourceModel variable. This gives an iterator on the entities that can be added to the root tree node which is done by calling the **processSourceModel()**. Also at the beginning of the method call, the **PlantUMLGenerator()** method is called followed by **setImage().** The former method generates the PlantUML code and subsequently generates the image by referencing the source model and sending an instance of the PlantUML visitor class. The latter sets the image to the imageLabel by reading the image.

```
@Override
protected void setSourceModelSpecifics() {
PlantUMLImageGenerator();
setImage();
this.treeRoot.setUserObject(new HierarchicalTreeCell
(this.getBuilder(), this.sourceModel,this.DISPLAY_ALL_LISTENER,
this.SELECTION_ALL_LISTENER));
this.treeRoot.removeAllChildren();
final Iterator iterator =
this.sourceModel.getIteratorOnTopLevelEntities ();
while (iterator.hasNext()) {
final IFirstClassEntity firstClassEntity = (IFirstClassEntity)
iterator.next();
     this.processSourceModel(this.treeRoot, this.awtCanvas,
this.canvas, this.sourceGraph, firstClassEntity,
firstClassEntity);}
     this.tree.expandRow(0);
     final Iterator iteratorOnGraphModelListeners =
DesktopPane.getInstance().getIteratorOnGraphModelListeners();
     while (iteratorOnGraphModelListeners.hasNext()) {
     final IGraphModelListener graphModelListener =
(IGraphModelListener) iteratorOnGraphModelListeners.next();
     graphModelListener.graphModelAvailable(new
SourceAndGraphModelEvent(DesktopPane.getInstance().getAbstractRe
presentationWindow())); }     }
```

GENERATING PLANTUML CODE AND IMAGE

Once the **PlantUMLImageGenerator()** is called**,** necessary flags and catch blocks are set to ensure that the image is generated and possible errors are handled. The imagePath is retrieved from the getImagePath() and set to the Outputstream, then the **modelGenerator()** is called which returns the uml code. Then, the resultant umlCode is sent to a new instance of **SourceStringReader(String),** a PlantUML library specific class, which sets the **reader** variable. This variable is used to call the outputImage() method where the Outputstream **png** variable is sent to write the image in the outputstream. Then **getDescription()** is used to retrieve the status of the generation and is used for error handling.

The **ModelGenerator()** method creates an instance of the PlantUMLGenerator Visitor class and sends it to the **generate()** method of the sourceModel, which accepts this visitor to traverse the model and returns the string of the PlantUML code using **getCode()** on the visitor.

The **setImage()** method sets the image to the image label and sets the horizontal and vertical alignments, by reading the image from the **imagePath.**

```
public boolean PlantUMLImageGenerator() {
     OutputStream png = null;
     boolean imageGeneratedFlag = false;
     try {
          png = new FileOutputStream(getImagePath());
     } catch (FileNotFoundException e) {
          e.printStackTrace();
          imageGeneratedFlag = false;
     }
     try {
```

```
          String umlCode = this.modelGenerator();
          SourceStringReader reader = new SourceStringReader
(umlCode);
          String desc =reader.outputImage(png).getDescription();
          if (desc.equals("Error")) { String errorMessage =
"Unable to process UMLCode"; throw new FileSystemException
(errorMessage);
          }
ProxyConsole.getInstance().normalOutput().println("PlantUML
Image generated successfully.");
          imageGeneratedFlag = true;
          } catch (UnsupportedSourceModelException e) {
               imageGeneratedFlag = false;
               e.printStackTrace();
          } catch (IOException e) {
               imageGeneratedFlag = false;
               e.printStackTrace();
          }
          return imageGeneratedFlag;
     }
```

```
public String modelGenerator() throws
UnsupportedSourceModelException {
          String finUMLContent = new String();
     try {
          PlantUMLGenerator PlantUMLGeneratorNew = new
PlantUMLGenerator();
this.getSourceModel().generate(PlantUMLGeneratorNew);

System.out.println(PlantUMLGeneratorNew.getCode());
```

```
     String umlContent = (String)PlantUMLGeneratorNew.getCode();
     String timeStamp = new Timestamp(System.currentTimeMillis()
).toString().split(" ")[0];
     System.out.println(timeStamp);
     finUMLContent = umlContent;
     } catch (Exception e) {
          e.printStackTrace();
     }
     return finUMLContent;
     }
```

```
     public void setImage() {
          File imageFile = new File(getImagePath());
          Image image = null;
          try {
               image = ImageIO.read(imageFile);
               this.imageLabel.setIcon(new ImageIcon(image));
          this.imageLabel.setHorizontalAlignment(JLabel.CENTER);
          this.imageLabel.setVerticalAlignment(JLabel.CENTER);
          } catch (IOException e) {
               e.printStackTrace();
          }
          this.revalidate();
          this.repaint();
     }
```

SELECTION LISTENERS

There are two listeners, **SELECTION_LISTENER** one to select the checkbox of an individual tree node and **SELECTION_ALL_LISTENER** to iteratively select all the checkboxes of the tree nodes present in the tree pane.

```
private final ItemListener SELECTION_ALL_LISTENER = new
ItemListener() {
     public void itemStateChanged(final ItemEvent anEvent) {
final DefaultMutableTreeNode root =
SourcePlantUMLModelWindow.this.treeRoot;
final Enumeration enumeration = root.depthFirstEnumeration();
    while (enumeration.hasMoreElements()) {
    final DefaultMutableTreeNode node = (DefaultMutableTreeNode)
enumeration.nextElement();
    final HierarchicalTreeCell cell = (HierarchicalTreeCell)
node.getUserObject();
    final IConstituent sourceConstituent =cell.getConstituent();
    if (sourceConstituent instanceof IFirstClassEntity) {
          final Constituent graphConstituent =
SourcePlantUMLModelWindow.this.sourceGraph.getEntity(sourceConst
ituent.getDisplayID());
          if (graphConstituent != null) {
                    if (anEvent.getStateChange() ==
ItemEvent.SELECTED) {
cell.setSelectedWithoutNotification(true);
graphConstituent.isSelected(true);
                    }
                  else if (anEvent.getStateChange() ==
ItemEvent.DESELECTED) {

cell.setSelectedWithoutNotification(false);
graphConstituent.isSelected(false);
                         }
                    }
               }
     }     }};
```

DISPLAY LISTENERS

Similar to Selection listeners, there are also two display listeners, **DISPLAY_LISTENER** to select an individual tree node and display that particular constituent on the Image Pane. **DISPLAY_ALL_LISTENER** is implemented to display all the constituents on the Image pane. Here, based on the selection, it adds the constituents to the **setOfEntitiesToDisplay** variable present in the **window** variable. This window variable holds the current window which is then retrieved from the unique instance of the DesktopPane using **getAbstractRepresentationWindow()**.

```
private final ItemListener DISPLAY_ALL_LISTENER = new
ItemListener()
public void itemStateChanged(final ItemEvent anEvent) {
                final SourcePlantUMLModelWindow window =
(SourcePlantUMLModelWindow) DesktopPane.getInstance()
                            .getAbstractRepresentationWindow();
                final DefaultMutableTreeNode root =
SourcePlantUMLModelWindow.this.treeRoot;
                if (anEvent.getStateChange() ==
ItemEvent.DESELECTED) {
                        ((HierarchicalTreeCell)
root.getUserObject()).setSelectedWithoutNotification(false);
                        ((HierarchicalTreeCell)
root.getUserObject()).setSpecialedWithoutNotification(false);
                }
final Enumeration enumeration = root.depthFirstEnumeration();
     while (enumeration.hasMoreElements()) {
     final DefaultMutableTreeNode node =
(DefaultMutableTreeNode) enumeration.nextElement();
     final HierarchicalTreeCell cell = (HierarchicalTreeCell)
```

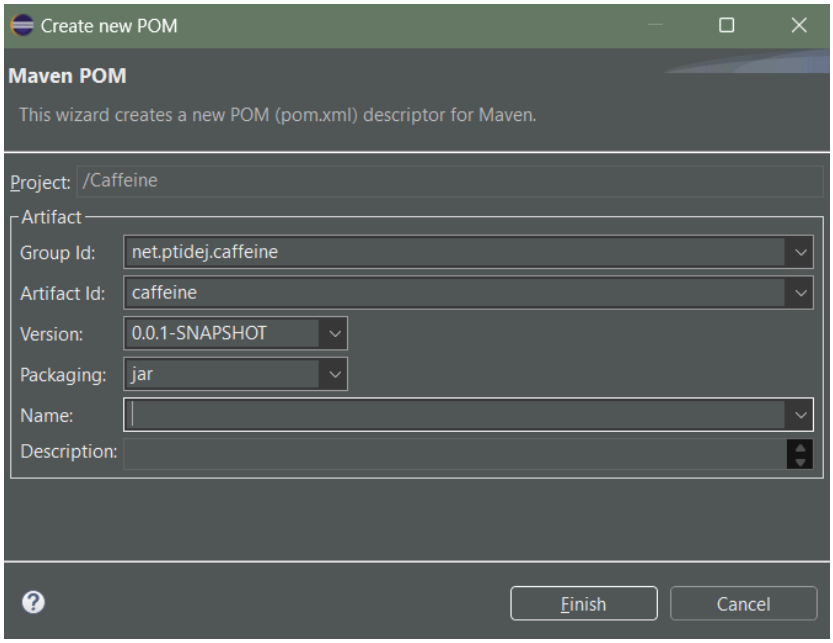
```
node.getUserObject();     final IConstituent sourceConstituent =
cell.getConstituent();
     if (sourceConstituent instanceof IFirstClassEntity) {
     if (anEvent.getStateChange() == ItemEvent.SELECTED) {
     cell.setDisplayedWithoutNotification(true);
     window.setOfEntitiesToDisplay.add(sourceConstituent);
     } else if (anEvent.getStateChange() ==
ItemEvent.DESELECTED) {
     cell.setDisplayedWithoutNotification(false);
     window.setOfEntitiesToDisplay.remove(sourceConstituent);
     cell.setSelectedWithoutNotification(false);
     window.setOfEntitiesToSelect.remove(sourceConstituent);
                              }
                         }
                    }
SourcePlantUMLModelWindow.this.updateWindowDisplay();
               }
     };
```

4.4 CONVERTING PROJECTS TO MAVEN

There are four existing projects in Ptidej that require Maven integration: **Caffeine, Caffeine Tests, Caffeine Analyses**, and **Caffeine Examples**. **Caffeine** is a tool that analyses the execution trace of Java applications dynamically instead of a static analysis. It uses the Java platform debug architecture to generate the execution trace and Prolog coroutine to perform queries over the traces. The reason is that, if a maintainer must understand the behaviour of a program, they perform static analyses and documentation to arrive at a conclusion but it might be error prone.

4.4.1 INITIAL STEPS

First we can start the process with the Caffeine project, by right clicking and selecting **Configure → Convert to Maven project**. A dialog window is opened with fields Group Id, Artifact Id, version, packaging, which has jar, war, and pom options, name and description. We can give **net.ptidej.caffeine** in groupId and leave the rest of the options as such and select finish. Also, the Group Id and Artificat Id cannot have white spaces in them and should be separated by the '-' convention.



4.4 Maven Dialog box

This dialog will generate a `pom.xml` file which holds the build information of the specified project, a target folder which is used to contain the temporary files that is created during the project build and a bin folder that contains the compiled classes.

INITIAL POM.XML STRUCTURE

The `pom.xml` is generated and it provides some default and input values provided to it. It includes a src directory and resource but for the project, it isn't necessary and it can be removed. Instead, we can create a sub-directory `/main` that has a nested directory `/java`, which translates to `src/main/java`. Similarly, we create src/main/resources to hold the resources that can be referred and `src/main/test` that contains the test packages.

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>net.ptidej.caffeine</groupId>
 <artifactId>Caffeine</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <build>
     <plugins>
     <plugin>
       <artifactId>maven-compiler-plugin</artifactId>
       <version>3.8.1</version>
       <configuration>
         <release>21</release>
       </configuration>
     </plugin>
   </plugins>
 </build>
</project>
```

As we have four projects and they reference one another, they can be grouped into net.ptidej. We create a new <parent> tag and add net.ptidej to the new <groupId>, all-ptidej in the new <artificatId> tag and 1.0.0 version to the new <version> tag and enclose it with the closing tag </parent>. This will ensure that this project will belong to the net.ptidej group and under net.ptidej.caffeine.

4.4.2 BUILD AND DEPENDENCIES

Apart from the default maven-compiler-plugin, we add maven-install-plugin. This install plugin will have certain goals configured, which is to build the external JAR resources from the src/main/resources folder. There are three jars to build, cfparse.jar, javaassist.jar, and JIProlog.jar and the intended goal command is install-file which is enclosed within <goal> tag. In configuration, we specify the path, groupId, packaging, artificatId, version of the JAR file to be built.

```
<executions>
     <execution>
          <id>install-cfparse</id>
               <phase>validate</phase>
               <goals>
               <goal>install-file</goal>
               </goals>
               <configuration>
<file>src/main/resources/cfparse.jar</file>
     <groupId>com.ibm.toad</groupId>
          <artifactId>cfparse</artifactId>
          <version>1.0</version>
          <packaging>jar</packaging>
<generatePom>true</generatePom>
     </configuration>
```

```
    </execution>
    <execution>
    <id>install-javassist</id>
         <phase>validate</phase>
         <goals>
              <goal>install-file</goal>
         </goals>
         <configuration>
         <file>src/main/resources/javassist.jar</file>
         <groupId>javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <generatePom>true</generatePom>
         </configuration>
    </execution>
    <execution>
         <id>install-JIProlog</id>
         <phase>validate</phase>
         <goals>
              <goal>install-file</goal>
         </goals>
              <configuration>
         <file>src/main/resources/JIProlog.jar</file>
         <groupId>JIProlog </groupId>
         <artifactId>JIProlog</artifactId>
         <version>1.0</version>
         <packaging>jar</packaging>
         <generatePom>true</generatePom>
         </configuration>
    </execution>
</executions>
```

For dependencies we can create a <dependency> tag for each dependency along with their Artifact Id, Group Id, and version. Other projects can also be added as dependencies in addition to the external JAR files or external standard libraries. This addition will download any external library if it is available from the Maven repository.

```
    <dependencies>
        <dependency>
            <groupId>net.ptidej.cpl</groupId>
            <artifactId>cpl-core</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>com.ibm.toad</groupId>
            <artifactId>cfparse</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>JIProlog</groupId>
            <artifactId>JIProlog</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
```

The same can be performed for Caffeine Analyses and Caffeine Examples except that it need not refer to all the dependencies or the build goals and instead it can just add

Caffeine just to their dependencies and need not build Caffeine as it is already built.

For Caffeine Tests, we can add maven-jar-plugin to define MANIFEST.mf files and allow the capability to build jar files. Another plugin is the maven-surefire-plugin which is used during the test phase of the build lifecycle to execute the unit tests of an application.

Once this is set, we can use maven clean, validate and install commands to build maven projects.

4.5 INTEGRATING LOG4J2

Logging capabilities are required to print and store the execution traces, error messages or any non-trivial messages for debugging purposes. Ptidej currently has a custom logger called `ProxyConsole`, which has four `PrintWriter` output types, debug, error, normal, and warning outputs and `PrintWriter` is used to write to an output stream.

There are more existing classes which are used in proxyconsole. `AutoFlushPrintWriter` class extends the `PrintWriter` class and overrides the `write()` method with buffer array, position and length, to write to the outputstream each time any bytes are added to the buffer. `UnclosablePrintWriter` class also extends the PrintWriter class to prevent the printwriter from being closed by another program, like Eclipse. It overrides the **`close()`** method to perform nothing except to print an empty string.

We will integrate Log4J2 with ProxyConsole and provide an abstraction, such that the users who use proxyConsole APIs will not know that they also use Log4J2: it will write using the PrintWriter methods and Log4J2 methods in parallel.

4.5.1 LOGGING PROPERTIES

The logger needs to be configured with a `rootLogger` and the logger level can be set to DEBUG and appender names can be assigned for printing and writing to console and file respectively.

For printing to console, `STDOUT` is the appender name and console.type is Console. In addition, we can specify the layout type as PatternLayout and define a custom pattern to follow.

The same can be done when writing to a log file, by defining LOGFILE as appender name, file type as File, directory path of the log file, threshold level to debug, and the pattern layout and pattern which is similar to the console type.

```
rootLogger=DEBUG, STDOUT, LOGFILE
appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd
HH:mm:ss.SSS} [%t] [%X{className}] -  %msg%n

appender.file.type = File
appender.file.name = LOGFILE
appender.file.fileName= logs/log4j.log
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd
HH:mm:ss.SSS} [%t] [%X{className}] -  %msg%n
appender.file.filter.threshold.type = ThresholdFilter
appender.file.filter.threshold.level = debug
```

4.5.2 LOGGER WRAPPER

In ProxyConsole, we can define constants for each threshold level. For each respective output type, we will set a new instance of the `MultiChannelPrintWriter`. An example would be:

```
this.debugOutput = new MultiChannelPrintWriter(new
PrintWriter(new LoggerWriter(WARN)), messageWriter);
```

The `LoggerWriter` class extends the `Writer` Class and this instance is casted to a `PrintWriter` since, `PrintWriter` also extends `Writer` class

First, we create a separate class called `LoggerWriter`, which extends `Writer` for creating logger instances and to receive threshold log level. The constructor receives the threshold level which is used to create the logger instance. It has getters and setters to return the logger and threshold level.

The `loggerWrite()` method takes a StringBuilder. This method has a switch case that checks for the threshold level Type by retrieving it from **logger.getLoggerLevelType()** for the specific loggerWriter instance. Based on the type, it will match the threshold level type and retrieve the logger using **getLoggerWriter()** and outputs the message using **debug()**, **info()**, **trace()**, **error()**, or **warn()**.

We then override the abstract methods of the Writer class like flush(), write() and close(). For flush and close methods we don't implement any functionality but for write(), it accepts char[] , offset and length parameters. We receive the messages to be logged in the char[] and we convert it to a String builder by traversing the array using the offset and length. Offset param specifies from where the message starts and length gives the total length of the messages

```
public class LoggerWriter extends Writer {
     private Logger logger;
     private String loggerLevelType;
     private static final String DEBUG = "debug";
     private static final String INFO = "info";
     private static final String WARN = "warn";
     private static final String TRACE = "trace";
     private static final String ERROR = "error";
     public LoggerWriter(final String loggerType) {
          this.loggerLevelType = loggerType;
          this.logger = LogManager.getLogger();
     }
     public Logger getLoggerWriter() {
          return this.logger;
     }
     public void setLoggerWriter(Logger logger) {
          this.logger = logger;
     }
     public void setLoggerLevelType(String loggerLevelType) {
          this.loggerLevelType = loggerLevelType;
     }
     public String getLoggerLevelType() {
          return this.loggerLevelType;
     }
     private void loggerWrite(final StringBuilder message) {
          switch (this.getLoggerLevelType()) {
          case DEBUG:
               this.getLoggerWriter().debug(message);
               break;
```

```
            case INFO:
                this.getLoggerWriter().info(message);
                break;
            case TRACE:
                this.getLoggerWriter().trace(message);
                break;
            case ERROR:
                this.getLoggerWriter().error(message);
                break;
            case WARN:
                this.getLoggerWriter().warn(message);
                break;
            default:
            }
        }
        @Override
        public void write(char[] cbuf, int off, int len) throws
IOException {
            StringBuilder messageBuilder = new StringBuilder();
            boolean carraigeAndNewLineSkip = false;
            if (len == 2 && (cbuf[0] == '\r') && cbuf[1] == '\n')
{
                carraigeAndNewLineSkip = true;
            }
            for (int i = 0; i < len; i++) {
                messageBuilder.append(cbuf[i]);
            }
            if (!carraigeAndNewLineSkip) {
                loggerWrite(messageBuilder);
            }
        }
        @Override
```

```
    public void flush() throws IOException {
        // Nothing to do for Log4J
    }
    @Override
    public void close() throws IOException {
        // Nothing to do for Log4J
    }
}
```

Secondly, we will have to create a class called **MultiChannelPrintWriter** that extends PrintWriter to handle instances of the PrintWriter and loggerWriter methods. It has two constructors to receive both PrintWriter instances.

It overrides the **write(final char[] buf, final int pos, final int len)** method to flush the contents in the buffer and overrides four other methods from **PrintWriterClass**, **print(final char charc)**, **print(final String message)**, **println(final char charc)**, and **println(final String message)**. Print methods `print` char or a string does not provide a newline while printing and `println` does the same except it prints in a newline.

All these methods make use of the **Thread.currentThread().getStackTrace()** method. This method returns an array that contains the stack trace entities that are executed in the current Thread. Furthermore, we can use the **getClassName()** on the 4th entry in the **stacktrace**, and add the obtained class name value on the map **ThreadContext** as **'className'**. This is **significant** because in the thread properties we refer to the className to print it along with the time stamp and method name. We can instruct the logger to use this context by calling the respective `print/println` method which by using reflecting calls the `write()` method of the loggerWriter class which then calls the

loggerWrite() method and once it is done, we can clear the ThreadContext map. After this we can call the print/println method of the printWriter along with the message.

Even though this class accepts two PrintWriter class instances, during initialization of **MultiChannelPrintWriter** class in **ProxyConsole** class, we create an instance of LoggerWriter, which extends a Writer class and implements the parent class methods in own class. At run time, due to reflection only, the `write()` method of LoggerWriter will be executed and not the Writer class methods itself.

```
package util.io;
import java.io.IOException;
/*
*
* @author Vishnu Rameshbabu
* @since 2024/07/11
*/
import java.io.PrintWriter;
import java.io.Writer;
import org.apache.logging.log4j.ThreadContext;
public class MultiChannelPrintWriter extends PrintWriter {
     private PrintWriter printWriter1;
     private PrintWriter printWriter2;
     public MultiChannelPrintWriter(final Writer writer) {
          super(writer, true);
     }
     public  MultiChannelPrintWriter(final  PrintWriter  writer1,
final PrintWriter writer2) {
          super(writer2, true);
          this.printWriter1 = writer1;
          this.printWriter2 = writer2;
     }
```

```
    public void write(final char[] buf, final int pos, final
int len) {
        super.write(buf, pos, len);
        this.flush();
    }
    public void print(final char charc) {
    StackTraceElement[]  stackTrace  =  Thread.currentThread()
.getStackTrace();
        String className = stackTrace[3].getClassName();
        ThreadContext.put("className", className);
        ThreadContext.clearAll();
        this.printWriter1.print(charc);
        this.printWriter2.print(charc);
    }
    public void print(final String message) {
  StackTraceElement[]  stackTrace  =  Thread.  currentThread()
.getStackTrace();
        String className = stackTrace[3].getClassName();
        ThreadContext.put("className", className);
        this.printWriter1.print(message);
        this.printWriter2.print(message);
    }
    public void println(final char charc) {
    StackTraceElement[]  stackTrace  =  Thread.  currentThread()
.getStackTrace();
        String className = stackTrace[3].getClassName();
        ThreadContext.put("className", className);
        this.printWriter1.println(charc);
        this.printWriter2.println(charc);
    }
    public void println(final String message) {
```

```
     StackTraceElement[]   stackTrace   =   Thread.currentThread().
getStackTrace();
          String className = stackTrace[3].getClassName();
          ThreadContext.put("className", className);
          this.printWriter1.println(message);
          this.printWriter2.println(message);
     }
}
```

CHAPTER - 5
RESULTS, TESTING AND VALIDATION

5.1 RESULTS

5.1.1 PLANTUML CODE AND IMAGE

We will select a target folder that contains .class files called Facade 2 present in the **DeMIMA project** and write the directory path terminated by a semicolon (;). We can save the file as a DeMIMA Facade.ptidej file in any directory path.

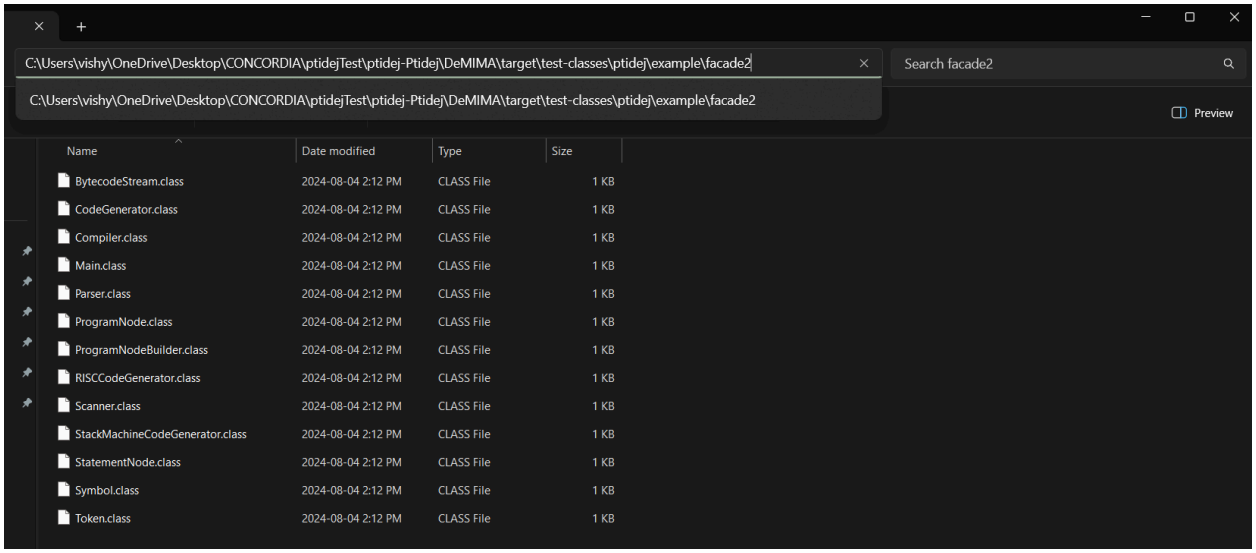


Fig - 5.1 Facade2 class File contents

`DeMIMA Facacde.ptidej` contents:

```
[Ptidej Project]
Name = DeMIMA Facade2
JavaCode = ../DeMIMA/target/test-classes/ptidej/example/facade2/;
```

In Eclipse, we run `/DeMIMA UI Viewer Standalone Swing/src/main/java/ptidej/viewer/ ProjectViewer.java` as a Java Application. This starts the Ptidej application and once started we can select the .ptidej file, which was created earlier. We can select the 4th folder icon from the left, which shows Load PlantUML Project help text.

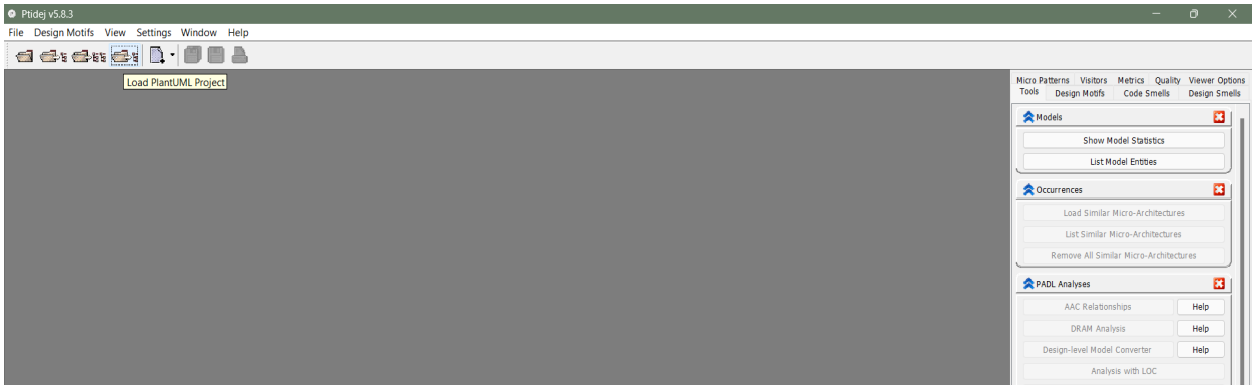


Fig - 5.2 Selecting Load PlantUML Project

Then, we can select `DeMIMA Facade.ptidej` from the file chooser.

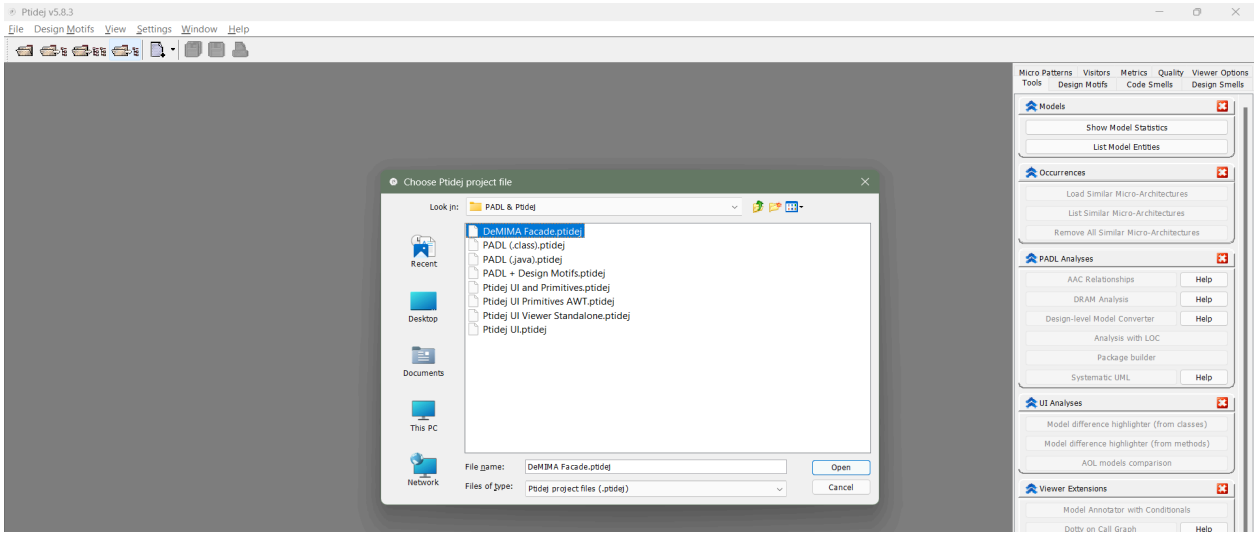


Fig - 5.3 Selecting DeMIMA Facade.ptidej

After selecting the required .ptidej file, a separate window frame is generated with the collapsible tree nodes on the left pane and the PlantUML image on the right pane. If we drag the console window from the button represented by an up and down arrow key, we can see the corresponding PlantUML code being generated.

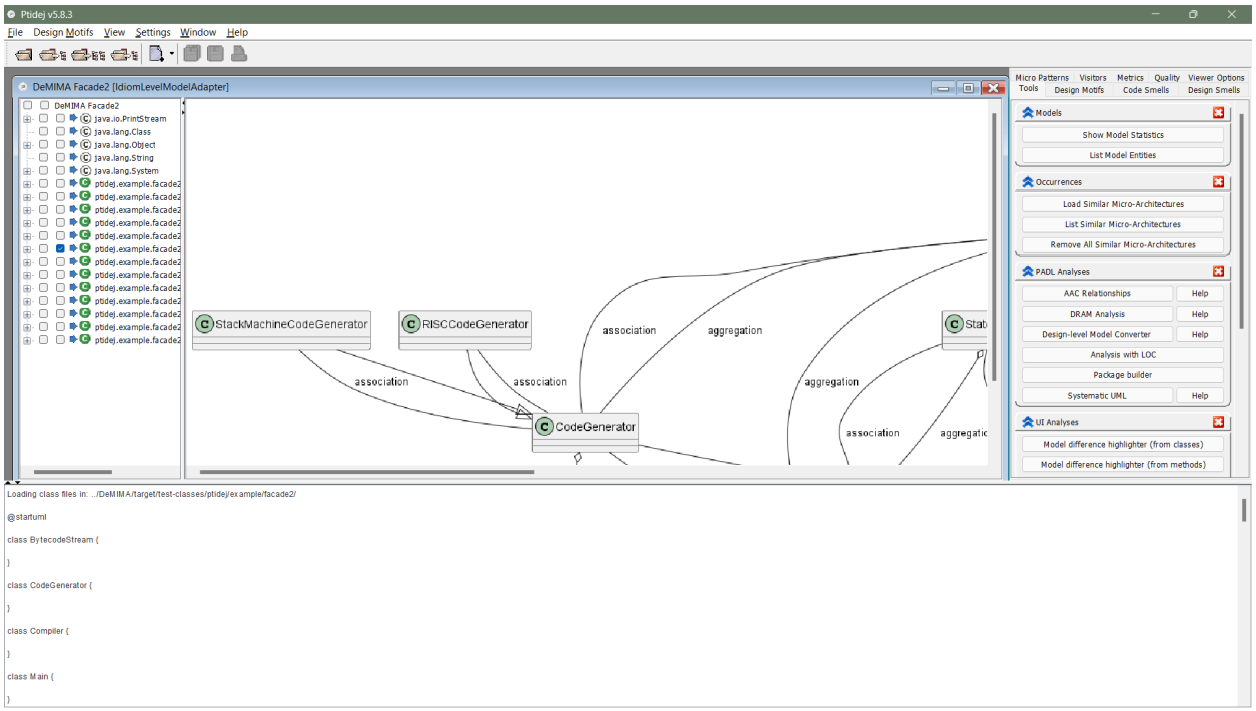


Fig - 5.4 PlantUML Window Image Visualization

GENERATED PLANTUML CODE

The code below is the code generated by the PlantUML visitor:

```
Loading class files in: ../DeMIMA/target/ test-classes/ ptidej/
example/facade2/
     @startuml
     class BytecodeStream {
     }
     class CodeGenerator {
```

```
    }
    class Compiler {
    }
    class Main {
    }
    class Parser {
    }
    abstract class ProgramNode {
    }
    class ProgramNodeBuilder {
    }
    class RISCCodeGenerator {
    }
    class Scanner {
    }
    class StackMachineCodeGenerator {
    }
    class StatementNode {
    }
    class Symbol {
    }
    class Token {
    }
    BytecodeStream -- Object : association
    CodeGenerator -- Object : association
    CodeGenerator -- PrintStream : association
    CodeGenerator o-- PrintStream : aggregation
    Compiler -- Object : association
    Compiler o-- Parser : aggregation
    Compiler o-- CodeGenerator : aggregation
    Compiler -- Parser : association
    Compiler -- CodeGenerator : association
```

```
    Compiler o-- PrintStream : aggregation
    Main -- Object : association
    Main -- Compiler : association
    Parser -- Object : association
    Parser -- Scanner : association
    Parser o-- Scanner : aggregation
    ProgramNode -- Object : association
    ProgramNodeBuilder -- Object : association
    RISCCodeGenerator --^ CodeGenerator
    RISCCodeGenerator -- CodeGenerator : association
    Scanner -- Object : association
    Scanner o-- PrintStream : aggregation
    StackMachineCodeGenerator --^ CodeGenerator
    StackMachineCodeGenerator -- CodeGenerator : association
    StatementNode --^ ProgramNode
    StatementNode -- ProgramNode : association
    StatementNode -- PrintStream : association
    StatementNode -- Object : association
    StatementNode o-- PrintStream : aggregation
    Symbol -- Object : association
    Token -- Object : association
    @enduml
    2024-08-04
    PlantUML Image generated successfully.
```

OUTPUT

The image below is the image generated by PlantUML and shown in the GUI of Ptidej:

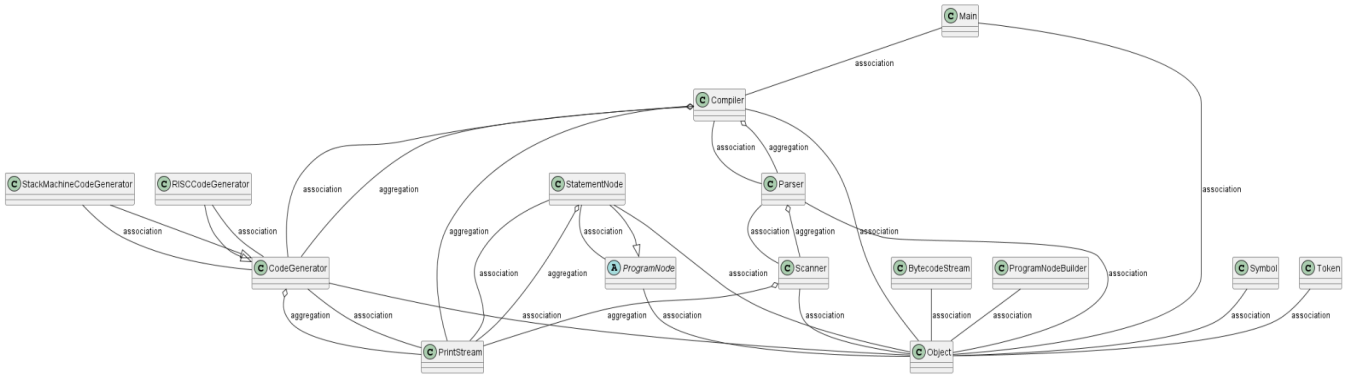


Fig - 5.5 PlantUML Image of Facade2

5.1.2 PTIDEJ LOGGING OUTPUT

As we launched the **`projectViewer.java`** which is the entry point for the ptidej application, the console prints out [WARN] messages, which is done by Log4J2 logger and the warning messages in red are printed by the **proxyConsole.java.** This log contains the threshold level, timestamp, thread name, the class that outputs the log, and the actual message, as example:

```
[WARN] 2024-08-04 18:48:50.627 [AWT-EventQueue-0]
[util.lang.ConcreteReceiverGuard] -
     Please do not instantiate metrics directly to allow
efficient caching, use the methods of
"pom.metrics.MetricsRepository" to obtain metric instances.
```

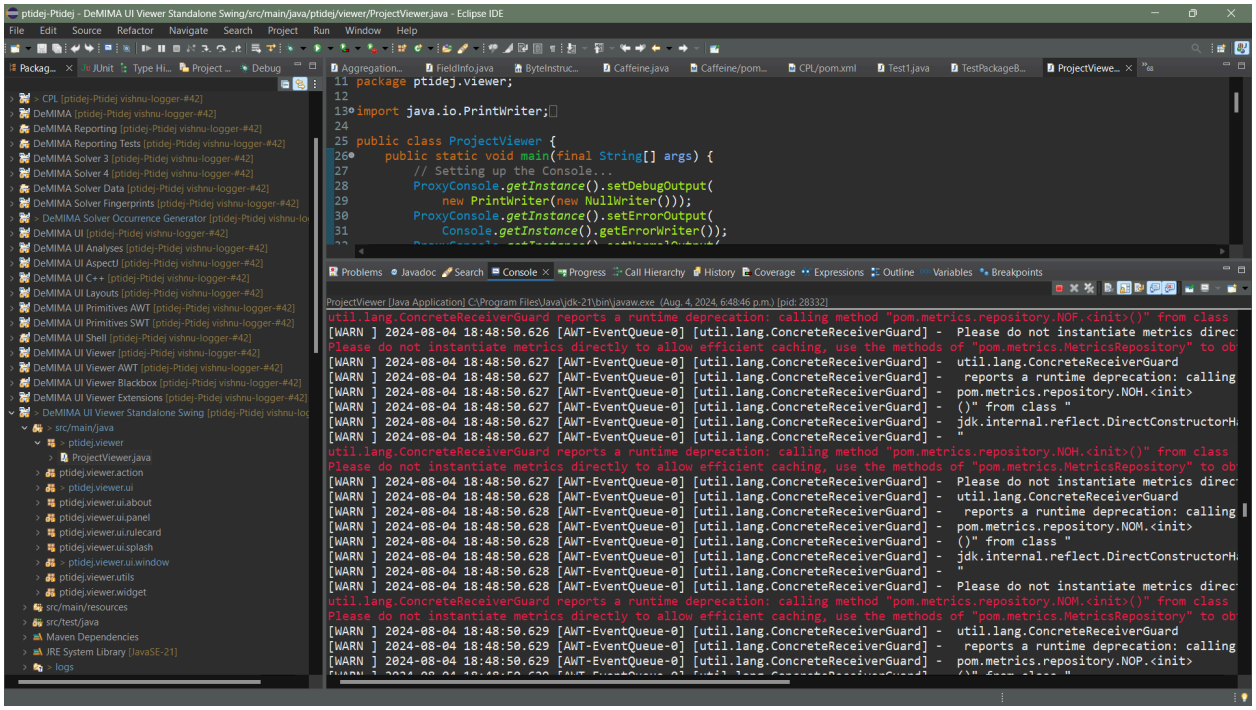


Fig - 5.6 Logger output to Console

For the Logger file, in this image, we could see the log file generated in the logs folder in **DeMIMA UI Viewer Standalone Swing** project.

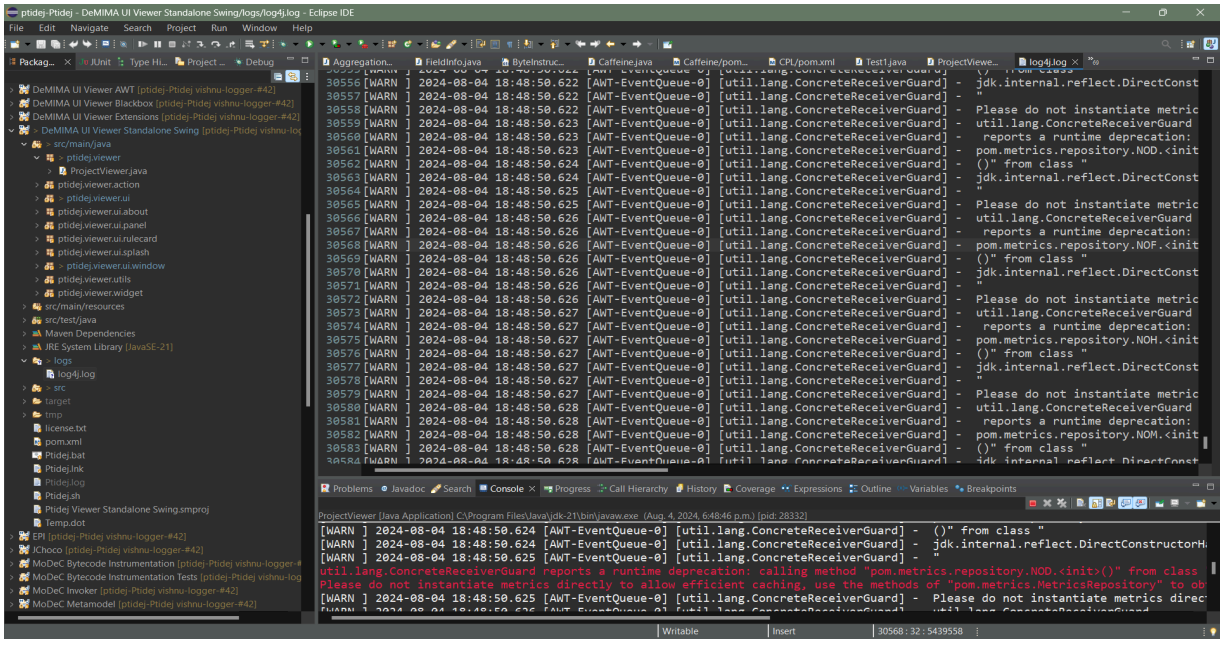


Fig - 5.7 Log File Output

5.2 TESTING AND VALIDATION

For PlantUML code testing must be done to validate if the generated output is correct when checked against the intended output for the same. JUnit4 can be used to write the test cases and the test suite for generating PlantUML Code.

A new package is created in the src/test/java as **padl.analysis.PlantUMLGenerator.test** and another package **padl.analysis.PlantUMLGenerator.test.exampleFile.** is created to store the correct PlantUML code of **../DeMIMA/target/test-classes/ptidej/example/composite2** in a .txt file. This txt file is then tested against the actual generation of composite2 target classes to ensure the code works properly.

Other test cases include the confirmation that the generated image is tested against the correct output of the image by comparing the byte stream. Both the images are read using `ImageIO.read()` and the respective results are assigned to an instance of `BufferedImage` and the individual data array size is obtained. First, both the sizes are compared and if it is the same, we will then proceed to compare the individual image's data array element with one another. A threshold of 99.8% is set to ensure both the generated image and the test image are the one and the same. A 0.2% error margin is set, just in case, if the PlantUML JAR gets an update and the pixels change.

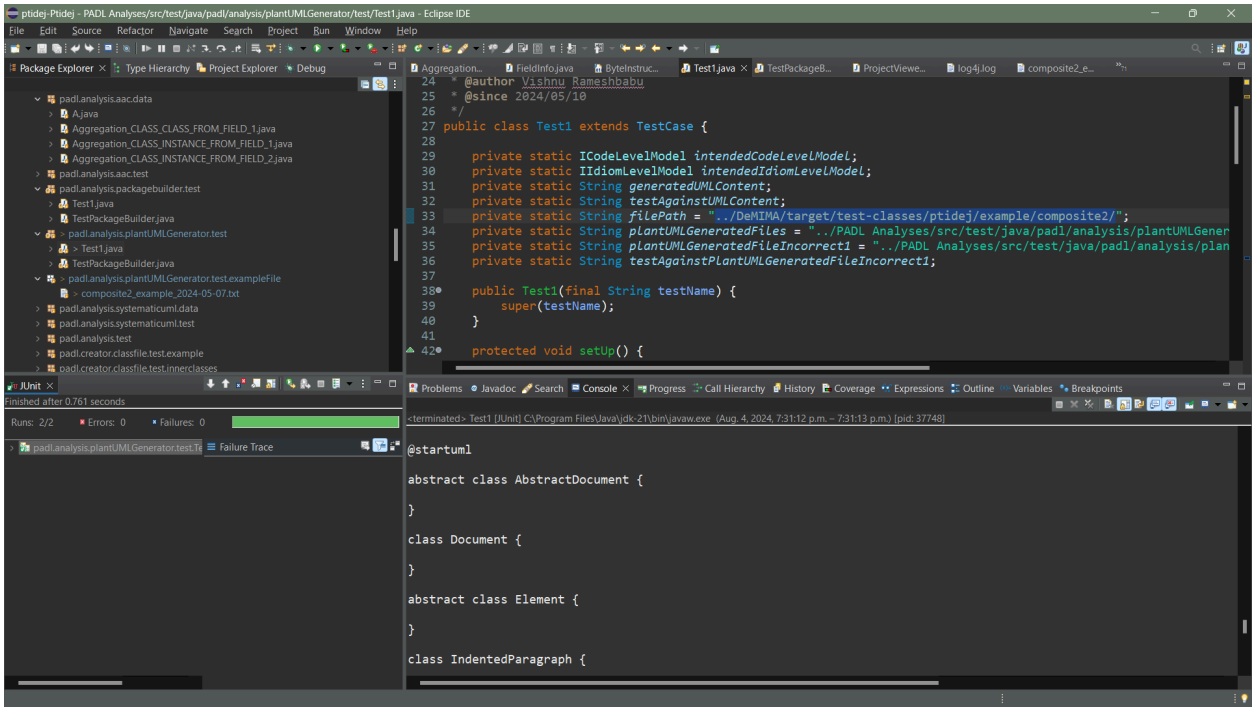


Fig - 5.8 JUnit Testing Results of PlantUML Code

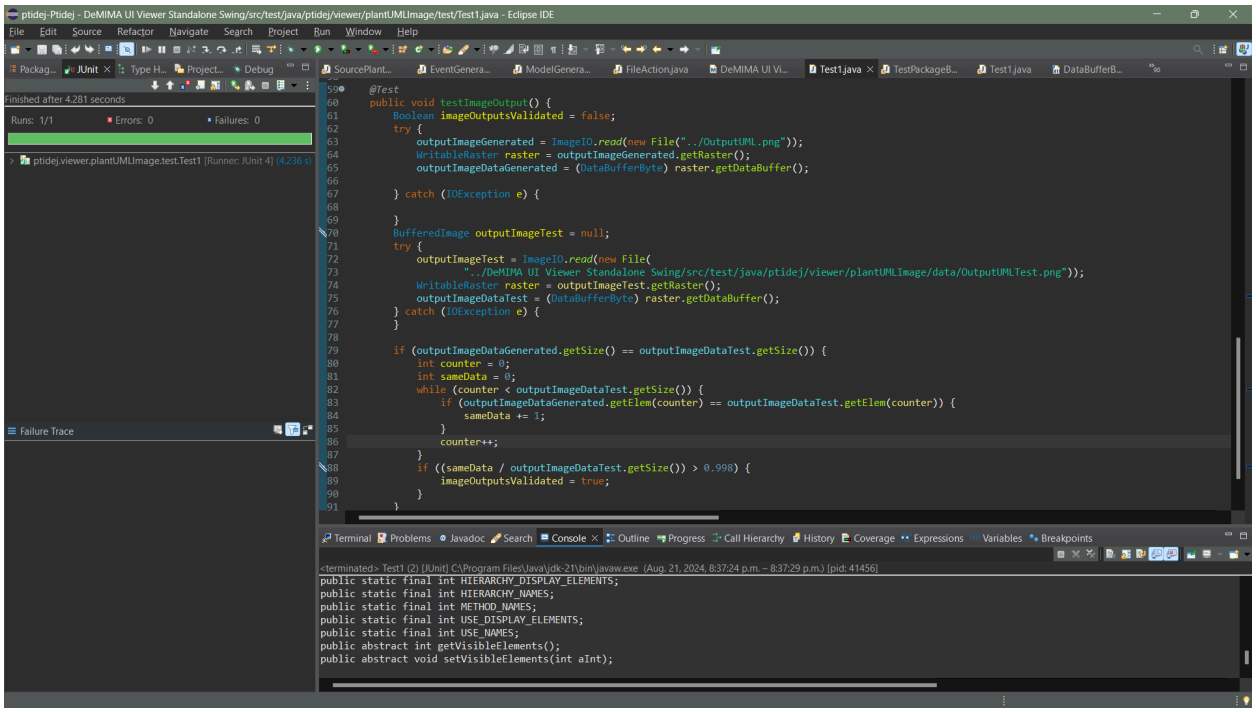


Fig - 5.9 JUnit Testing Results of PlantUML Output Image

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

6.1 CONCLUSION

The proposed method has highlighted the features to be added and integrated and standard convention to project structures. This project has satisfied the requirements stated with proven testing and validation methods to ensure the added feature work.

6.2 FUTURE WORK

Ptidej can be enhanced, by adding a feature that can be added to dynamically manipulate PlantUML code which helps to generate PlantUML diagrams when an entity is selected in the Tree node.

REFERENCES

[1] https://PlantUML.com/

[2] https://logging.apache.org/log4j/2.x/manual/configuration.html

[3] Y.-G. Guéhéneuc, R. Douence, and N. Jussien, "No Java without caffeine: A tool for dynamic analysis of Java programs," *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, Edinburgh, UK, 2002, pp. 117-126, doi: 10.1109/ASE.2002.1115000.

[4] https://maven.apache.org/surefire/maven-surefire-plugin/