Software Architecture Checker

Yasin Bahtiyar
Abstract
By the increasing needs of software industry, software systems became more complex constructions than ever before. As a result of increasing complexity in software systems, functional decomposition of these systems gains the status of the most important aspect in the software development process. Dividing problems to sub-problems and producing specific solutions for divided parts makes it easier to solve the main problem.

Component Based Software Engineering is a way of developing software systems that consists of logically or functionally decomposed components which integrated to each other by the help of well-defined interfaces. CBSE relies on architectural design of a software system.

Planning phase and implementation of a software project may differ time to time. Because of the complexity of software systems, solving specific problems may affect the architecture of the whole system.

In spite of sophisticated software engineering processes and CASE tools there is still a large gap between the planned and implemented architecture of software systems. Finding deviations from architecture in source code is a non-trivial task requiring tool support.

Since, matching operation of designed software architecture and implemented software architecture needs to check design documents against implementation code. This manual checking operation is nearly impossible for major software systems. Software Architecture Checker provides a great approach to check the architecture of any software system.

This bachelor thesis examines the approach behind the Software Architecture Checker.

Keywords: software, architecture, java, regex, vizzanalyzer
# Table of Contents

Abstract ........................................................................................................................ ii  
List of Tables ............................................................................................................... v  
List of Figures ............................................................................................................. vi  
Glossary ..................................................................................................................... vii  

1 Introduction .......................................................................................................... 1  
  1.1 Problem .......................................................................................................... 1  
  1.2 Goals and Criteria ...................................................................................... 2  
  1.3 Motivation ...................................................................................................... 2  
  1.4 Outline ............................................................................................................ 2  

2 Background ........................................................................................................... 3  
  2.1 VizzAnalyzer .................................................................................................. 3  
  2.2 Eclipse ............................................................................................................ 4  
  2.3 Regular Expressions ....................................................................................... 4  

3 Requirements ........................................................................................................ 6  
  3.1 Actors ............................................................................................................. 6  
  3.2 Features .......................................................................................................... 6  
  3.3 Use-Cases ....................................................................................................... 7  
    3.3.1 Use-Case Model ...................................................................................... 7  
    3.3.2 Use-Case Descriptions ............................................................................ 7  
  3.4 Requirements ................................................................................................ 10  

4 Architecture ........................................................................................................ 12  
  4.1 Architecture of Eclipse ................................................................................. 12  
  4.2 Plug-in Model of Eclipse ............................................................................. 12  
  4.3 Architecture of Software Architecture Checker ........................................... 13  
    4.3.1 Eclipse Plug-In UI .................................................................................. 14  
    4.3.2 Configuration ......................................................................................... 14  
    4.3.3 Analyzer Core ....................................................................................... 14  

5 Design and Implementation ................................................................................ 15  
  5.1 Class Diagram .............................................................................................. 15  
    5.1.1 Configuration Class ............................................................................... 17  
    5.1.2 ComponentsModel Class ...................................................................... 17  
    5.1.3 RulesModel Class .................................................................................. 17  
    5.1.4 Component Class .................................................................................. 18  
    5.1.5 Rules Class ............................................................................................ 18  
    5.1.6 TypeValues Class .................................................................................. 18  
    5.1.7 AnalyzerCore Class .............................................................................. 19  
    5.1.8 ArchitectureCheckerHandler Class ...................................................... 19  
    5.1.9 ArchitectureCheckerWizard class ....................................................... 19  
    5.1.10 SelectGraphFileWizardPage Class ..................................................... 20  
    5.1.11 ComponentDefinitionWizardPage Class ........................................... 20  
    5.1.12 RuleDefinitionWizardPage Class ....................................................... 20  
    5.1.13 ComponentsContentProvider Class .................................................. 21  
    5.1.14 ComponentsLabelProvider Class ...................................................... 21  
    5.1.15 RulesContentProvider Class ............................................................... 21  
    5.1.16 RulesLabelProvider Class .................................................................... 21
5.2 Implementation of Software Architecture Checker ........................................ 22
5.2.1 Plugin.xml ................................................................................................ 22
5.2.2 Package Structure of Software Architecture Checker .......................... 24

6 Conclusion and Future Work............................................................................. 25
6.1 Conclusion ..................................................................................................... 25
6.2 Future Work ................................................................................................. 27

7 References ........................................................................................................ 28

8 Appendices ....................................................................................................... 29
Appendix A - Package Structure of Software Architecture Checker .............. 29
A.1 Package se.vxu.msi.vizzanalyzer.architecturechecker .................. 29
A.2 Package se.vxu.msi.vizzanalyzer.architecturechecker.configuration ... 29
A.3 Package se.vxu.msi.vizzanalyzer.architecturechecker.core .............. 29
A.4 Package se.vxu.msi.vizzanalyzer.architecturechecker.handlers ....... 29
A.5 Package se.vxu.msi.vizzanalyzer.architecturechecker.wizards ......... 29
A.5.1 ArchitectureCheckerWizard Class ..................................................... 30
A.5.1 ComponentDefinitionWizardPage Class ........................................... 31
A.5.2 ComponentsContentProvider Class .................................................... 31
A.5.3 ComponentsLabelProvider Class ....................................................... 31
A.5.4 DataChangedEvent Class .................................................................. 31
A.5.5 DataChangedEventListener Class ........................................................ 32
A.5.6 RuleDefinitionWizardPage Class ....................................................... 32
A.5.1 RulesContentProvider Class ............................................................... 33
A.5.2 RulesLabelProvider Class ................................................................. 33
A.5.3 SelectGraphFileWizardPage Class ..................................................... 33


List of Tables

Table 3-1 Features of Software Architecture Checker ................................................ 6
Table 3-1 Requirements Analysis for Software Architecture Checker .......................... 11
List of Figures

Figure 2.1 The VizzAnalyzer Framework's Architecture(2) ........................................ 3
Figure 2.2 Simplified Eclipse Platform architecture (3) .............................................. 4
Figure 3.1 Use-Case Model for Software Architecture Checker ............................. 7
Figure 4.1 The Eclipse Architecture (7) .................................................................. 12
Figure 4.2 Basic plugin.xml file .............................................................................. 13
Figure 4.3 Component Diagram of Software Architecture Checker ........................ 13
Figure 5.1 Class Diagram of Software Architecture Checker .................................. 16
Figure 5.2 Configuration Class ............................................................................... 17
Figure 5.3 ComponentsModel Class ....................................................................... 17
Figure 5.4 RulesModel Class .................................................................................. 17
Figure 5.5 Component Class .................................................................................. 18
Figure 5.6 Rules Class ......................................................................................... 18
Figure 5.7 AnalyzerCore Class ................................................................................ 19
Figure 5.8 ArchitectureCheckerWizard Class ......................................................... 19
Figure 5.9 SelectGraphFileWizardPage Class .......................................................... 20
Figure 5.10 ComponentDefinitionWizardPage Class ............................................... 20
Figure 5.11 RuleDefinitionWizardPage Class ......................................................... 20
Figure 5.12 plugin.xml of Software Architecture Checker ...................................... 23
Figure 5.13 Eclipse Extensions of Wostware Architecture Checker ........................ 23
Figure 5.14 List of Packages for Software Architecture Checker ............................. 24
Figure 6.1 XML Report of Software Architecture Checker ..................................... 26
Figure 6.2 Text File Report of Software Architecture Checker .............................. 27
Figure 8.1 List of classes for configuration ............................................................... 29
Figure 8.2 List of Classes for wizard package .......................................................... 29
Figure 8.3 performFinish() method of ArchitectureCheckerWizard ....................... 30
Figure 8.4 Component Definition Wizard Page ..................................................... 31
Figure 8.5 Example Usage of DatachangeEventListener ....................................... 32
Figure 8.6 Rule Definition Wizard Page ................................................................. 32
Figure 8.7 Select Files Wizard Page ....................................................................... 33
<table>
<thead>
<tr>
<th>Glossary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBSE</td>
<td>Component Based Software Engineering</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>RCP</td>
<td>Rich Client Platform</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
</tr>
<tr>
<td>PDE</td>
<td>Plug-In Development Environment</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>SAC</td>
<td>Software Architecture Checker</td>
</tr>
</tbody>
</table>
1 Introduction
By the increasing needs of software industry, software systems became more complex constructions than ever before. As a result of increasing complexity in software systems, functional decomposition of these systems gains the status of the most important aspect in the software development process. Dividing problems to sub-problems and producing specific solutions for divided parts makes it easier to solve the main problem.

Since 1970’s, Software architecture has came up as a discipline to understand structures of complex software systems. Still there is more than one definition the most suitable one is pointed by Bass, et al.:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them."(1)

Component Based Software Engineering (CBSE) is a way of developing software systems that consists of logically or functionally decomposed components which integrated to each other by the help of well-defined interfaces.

Component Based Software Engineering has higher-level abstraction than Object Orientation and the idea behind CBSE is to be able to define software systems like in blue prints.

CBSE uses best practices for specific problems. There are many types of software architectures developed over years, these well-defined components communicates over well-defined interfaces.

Despite sophisticated software engineering processes and CASE tools there is nowadays still a large gap between the architecture of software system as planned and the architecture of software system as implemented. Finding deviations from architecture in source code is a non-trivial task requiring tool support. The VizzAnalyzer tool allows to read software systems from source and to represent them as graphs. Yet it lacks an editor for specifying architecture and for comparing it with the software as is, to identify deviations, which can be used to close the gap.

1.1 Problem
Planning phase and implementation of a software project may differ time to time. Because of the complexity of software systems, solving specific problems may affect the architecture of the whole system.

Normally, well-defined components must communicate over well-defined interfaces. This is planned to increase reusability of the components. But, for instance, to solve a specific problem, a software developer may directly access to another class inside some other component. In fact such an operation was not planned because it makes the components dependent to each other. As a result, not planned dependency decreases the quality of implementation.

On the other hand, checking the architecture of a software system whether as planned or not is a non-trivial job without tool support. Generally, Architectural design of a software system uses UML Component Diagrams and it is hard to compare these diagrams with implementation.

The objective of this thesis is to implement a plug-in for Eclipse platform, which extends VizzAnalyzer’s functionality to check the architecture of the software system.

The problem could be addressed as design and implementation of an Eclipse plug-in to define and check Software Architecture.
This is a non-trivial task, since most programming languages, including Java, do not provide a mechanism to define software components. Therefore it is often not clear which classes or packages define a component, and which parts of them are interfaces.

1.2 Goals and Criteria
This section explains the goals to solve the problem and criteria to validate the solution.

The first goal of this thesis is defining the components of software system. Users of the Software Architecture Checker plug-in should be able to define the components that are already designed. This goal has been reached when Software Architecture Checker has a user interface to define components.

The second goal is defining sub-components. Components should be able to define several sub-components. Because implemented software components may consist of several sub-components. This goal has been met when component definitions can be made in a hierarchical way.

The third goal is mapping defined component to implementation of software system. The regular expressions help to fulfill this goal by relating components with classes and packages of the implemented software system.

The fourth goal is defining rules between already defined components. User should be able to specify the interaction between components. This goal is met when Software Architecture Checker has a user interface to define the rules between components. These rules should match the rules supplied by the VizzAnalyzer framework.

The fifth goal of the thesis is reporting violations of the given software system through given component and rule definition. This goal has been reached when Software Architecture Checker is able to compare given component definitions and relations with implementation of the system. Test results should be reported in various XML and text format.

The sixth goal of this project is grouping the defined components in the graph file supported by VizzAnalyzer. This goal has been reached when given Graph file modified through defined rules, components and test results.

1.3 Motivation
In spite of sophisticated software engineering processes and CASE tools there is nowadays still a large gap between the architecture of software system as planned and the architecture of software system as implemented. Finding deviations from architecture in source code is a non-trivial task requiring tool support.

Since, matching operation of designed software architecture and implemented software architecture needs to check design documents against implementation code. This manual checking operation is nearly impossible for major software systems. Software Architecture Checker provides a great approach to check the architecture of any software system.

1.4 Outline
The structure of this paper is addressed as follows, Chapter 2 provides some background information about the problem, goal domain. And also gives information Eclipse and VizzAnalyzer. Chapter 3 clarifies the actors and features for Software Architecture Checker. Also Use-Case Model and Descriptions are provided in this chapter. Chapter 4 mainly gives information about architecture of Software Architecture Checker and Eclipse. Chapter 5 describes design and implementation of Software Architecture Checker. Also this chapter provides detailed information about software components, packages and classes. Chapter 6 gives information about results of the project and future work.
2 Background
This section describes the technologies that are related to the problem domain. We will give short summary about the VizzAnalyzer and the Eclipse platform.

2.1 VizzAnalyzer
The VizzAnalyzer Framework is a reusable framework for a rapid composition of reverse engineering tools. It is extensible for new programming languages, analyses, and visualization tools.(2)

The architecture of VizzAnalyzer is depicted in Figure 2.1:

“The framework consists of the framework-core, frozen-spots and hotspots. The framework-core is responsible for communicating information between the different reverse engineering components connected to the framework. It has the functionality of a controller and information converter. The frozen-spots are in-house and externally developed reusable components supporting the framework-core with main functionalities. For instance, configurations necessary for reverse engineering tool compositions reuse our tiny-xml editor, a tool reused by the framework-core. Hot-spots are technically realized as directories and allow the simple and fast connection of arbitrary reverse engineering components with the framework.”(2)

Figure 2.1 The VizzAnalyzer Framework's Architecture(2)

More information about VizzAnalyzer is available at (2)
2.2 Eclipse

Eclipse is defined as “Java-Based, extensible open source development platform”. By itself, it is simply a framework and a set of services for building applications from plug-in components.(3)


On the other hand, Eclipse gives many opportunities to developers to integrate their tools into Eclipse by the help of Eclipse RCP’s extensible architecture. Plug-in development tools are also included in a standard Eclipse installation. Plug-in Development Environment (PDE) consists of a set of specialized plug-ins to develop new plug-ins for the platform.

A simplified architecture of Eclipse RCP could be seen in Figure 2.2.

![Figure 2.2 Simplified Eclipse Platform architecture (3)](image)

“The dark blue pieces signify components that are the core part of Eclipse's Rich Client Platform (RCP). The light blue pieces are optional (but recommended) pieces to be included in RCP-based applications. And the gray pieces are completely optional.”(3)

Eclipse is open source and well-documented platform. It also uses operating systems default user interface. Furthermore VizzAnalyzer already uses Eclipse’s plug-in technology to supply graphical user interface support.

More information about the relation of Eclipse RCP and VizzAnalyzer could be found on “VizzAnalyzer Goes Eclipse”. (4)

2.3 Regular Expressions

Regular expressions are used to define the software components and rules between them. Classes and packages that which consist a component can easily be expressed by regular expressions.

“A regular expression (or regex) is a specified text string to define a search pattern. For example regex “ba*” matches all strings that starts with one ‘b’ character and continues with zero or more ‘a’ characters.

Java is supporting regular expressions by the help of java.util.regex API. There is a basic example that shows usage of this API:

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a Matcher object that can match arbitrary
character sequences against the regular expression. All of the state involved in performing a match resides in the matcher; so many matchers can share the same pattern.

A typical invocation sequence is thus

```java
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

(5)
The code example given above shows how to check a string matches to regex pattern or not.

More information about regular expressions is discussed in (5) and (6)
3 Requirements
This section presents information about Software Architecture Checker’s requirements.

First of all we will give information about the actors that will use the planned software system. Later on we will discuss Use-Case Model and present Use-Case. We will conclude this section with a presentation of the functional requirements of the Software Architecture Checker.

3.1 Actors
Because of the Software Architecture Checker is a sub-module for VizzAnalyzer, actors of the Software Architecture Checker and VizzAnalyzer exactly match each other.

As already defined by (4) there are two different actors for this system:

The first actor of this system will be end-user who is intended to be experienced in analysis and design of a software project. Generally Software Architects, Software Analysts, Software Designer or Project Managers could fit in end-user actor group. End-user is the actor who wants to check the architecture of implementation against architecture of designed software system by the help of Software Architecture Checker.

The second actor for this system is developers who want to extend the functionality of Software Architecture Checker or give maintenance for the system.

3.2 Features
This section describes the features of the Software Architecture Checker. These features that can be seen in Table 3-1 are derived from the problem domain analysis and will help to define the requirements of the software system.

<table>
<thead>
<tr>
<th>Feature 1</th>
<th>Capability to Define Components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architecture Checker has the capability to define Software components for given Software System. A software Component may consist of several different sub-components so component definition must be in a hierarchical way. Components should be defined with the help of regular expressions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 2</th>
<th>Capability to Define relations of components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architecture Checker has the capability to define relations of already defined components. These relations are supplied by VizzAnalyzer framework and each different relation type helps to define a rule of communication between components.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 3</th>
<th>Capability to check defined components against implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architecture Checker has the capability of checking the architecture of implemented system against defined components, sub-components and relations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 4</th>
<th>Capability to report the violations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architecture Checker has the capability to report the violations of the implemented Software System through the test results that derived from defined components and rules. Report should be in xml format and/or text format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 5</th>
<th>Capability to edit graph file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architecture Checker has the capability of updating the graph file that supplied by VizzAnalyzer to visualize the defined components and violations between them.</td>
</tr>
</tbody>
</table>

Table 3-1 Features of Software Architecture Checker
3.3 Use-Cases
This Section describes Use-Cases of the Software Architecture Checker. For better understanding of Use-Cases, we will clarify the Use-Case model first and later on we will describe each Use-Case in detail.

3.3.1 Use-Case Model
Use-Case model for the actor “End-User” of the Software Architecture Checker can be seen in Figure 3.1

Use-Case model for the actor “Developer” exactly matches with the Use-Case model of VizzAnalyzer and already addressed by (4) and (2).

```
Figure 3.1 Use-Case Model for Software Architecture Checker
```

3.3.2 Use-Case Descriptions
In this section Use-Cases are described in detail. To achieve the goals and realize the features of the system we define each Use-Case with following specialties: Goal, precondition, post-condition, trigger event that releases the Use-Case, extensions and alternatives.
**Use-Case UC1**  
**Define Component**  
*Feature: 1*

<table>
<thead>
<tr>
<th>Goal</th>
<th>Defining the software components that are going to be analyzed by the help of regular expressions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Condition</td>
<td>Graph file that contains the information of implemented system is already selected.</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>New component defined. Add component or Add Sub-component buttons are activated.</td>
</tr>
<tr>
<td>Actors</td>
<td>End-User</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>User clicks Next button just after selection of Graph file and output files.</td>
</tr>
<tr>
<td>Description</td>
<td>User enters a descriptive name for component. User enters the regular expression to define the matching packages and/or classes for the component.</td>
</tr>
<tr>
<td>Extensions</td>
<td>-</td>
</tr>
<tr>
<td>Alternatives</td>
<td>-</td>
</tr>
</tbody>
</table>

**Use-Case UC2**  
**Add Component**  
*Feature: 1*

<table>
<thead>
<tr>
<th>Goal</th>
<th>Add defined component to the Component Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Condition</td>
<td>Component is already defined</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>Component is added to Component Tree</td>
</tr>
<tr>
<td>Actors</td>
<td>End-User</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>User enters some information to define the component with given fields and Add Component button is activated.</td>
</tr>
<tr>
<td>Description</td>
<td>User clicks Add Component Button</td>
</tr>
<tr>
<td>Extensions</td>
<td>Add Sub-Component</td>
</tr>
<tr>
<td>Alternatives</td>
<td>-</td>
</tr>
</tbody>
</table>

**Use-Case UC3**  
**Add Sub-Component**  
*Feature: 1*

<table>
<thead>
<tr>
<th>Goal</th>
<th>Add defined component to the Component Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Condition</td>
<td>Component is already defined</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>Component is added to Component Tree; as a sub-component for selected component</td>
</tr>
<tr>
<td>Actors</td>
<td>End-User</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>User enters some information to define the component with given fields and selects the parent component. Add Sub-Component button is activated.</td>
</tr>
<tr>
<td>Description</td>
<td>User Clicks Add Sub-Component</td>
</tr>
<tr>
<td>Extensions</td>
<td>-</td>
</tr>
<tr>
<td>Alternatives</td>
<td>User double clicks on Component tree to give up adding sub-component.</td>
</tr>
</tbody>
</table>

**Use-Case UC4**  
**Remove Component**  
*Feature: 1*

<table>
<thead>
<tr>
<th>Goal</th>
<th>Remove Already defined component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Condition</td>
<td>Component is already defined and selected</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>Selected component is removed</td>
</tr>
<tr>
<td>Actors</td>
<td>End-User</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>User selects a component to remove</td>
</tr>
<tr>
<td>Description</td>
<td>User clicks Remove Component Button</td>
</tr>
<tr>
<td>Extensions</td>
<td>-</td>
</tr>
<tr>
<td>Alternatives</td>
<td>-</td>
</tr>
<tr>
<td>Use-Case UC5: Define Rule</td>
<td>Feature: 2</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>Define a rule between two components of implemented Software System.</td>
</tr>
<tr>
<td><strong>Pre-Condition</strong></td>
<td>Components are already defined.</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>Rule is defined and Allow/Disallow buttons are activated.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>End-User</td>
</tr>
<tr>
<td><strong>Triggering Event</strong></td>
<td>User ends up with defining components and clicks Next button</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>User selects the source and target components. User selects the rule that supplied by the VizzAnalyzer framework.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use-Case UC6: Add Rule</th>
<th>Feature: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Add Defined rule to the Rules Table</td>
</tr>
<tr>
<td><strong>Pre-Condition</strong></td>
<td>Rule is already defined</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>Rule is added to the Rules Table</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>End-User</td>
</tr>
<tr>
<td><strong>Triggering Event</strong></td>
<td>User selects the source/target components and the rule to activate Allow/Disallow Buttons</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>User clicks Allow or Disallow Button to add the defined rule to the Rules Table</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use-Case UC7: Remove Rule</th>
<th>Feature: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Remove defined rule from Rules Table</td>
</tr>
<tr>
<td><strong>Pre-Condition</strong></td>
<td>Rule is already defined and selected</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>Rule is removed from Rules Table</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>End-User</td>
</tr>
<tr>
<td><strong>Triggering Event</strong></td>
<td>User selects the rule is going to be removed to activate Remove Rule Button</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>User clicks Remove Rule Button to remove the selected rule from table.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use-Case UC8: Select Graph File</th>
<th>Feature: 3,5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>User selects a graph file that produced by VizzAnalyzer and contains the information about implemented software system. This graph is basically a data model for implementation of the software system that is going to be checked by the Software Architecture Checker.</td>
</tr>
<tr>
<td><strong>Pre-Condition</strong></td>
<td>Graph file is already exits.</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>Next button is activated to be able to define components.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>End-User</td>
</tr>
<tr>
<td><strong>Triggering Event</strong></td>
<td>User selects Software Architecture Checker from Eclipse Menu.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>User clicks Browse button to select the graph file in the file system. User Selects the graph file.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td>-</td>
</tr>
</tbody>
</table>
### Use-Case UC9

**Select Output Files**

**Feature:** 4

<table>
<thead>
<tr>
<th><strong>Goal</strong></th>
<th>Select Output file formats and location to store files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pre-Condition</strong></td>
<td>User selects Software Architecture Checker from Eclipse Menu.</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>End-User</td>
</tr>
<tr>
<td><strong>Triggering Event</strong></td>
<td>User selects Software Architecture Checker from Eclipse Menu.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>User selects xml and/or text format for reporting. User selects the location for files to be stored.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

### 3.4 Requirements

This section provides information about functional requirements of the Software Architecture Checker. There are three types of requirements level: Essential, Desirable and Optional. All these requirements are derived from Use-Case descriptions.

<table>
<thead>
<tr>
<th><strong>Requirement</strong></th>
<th><strong>Type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Software Architecture Checker should be implemented as a plug-in for existing VizzAnalyzer system. It must be implemented as an Eclipse plug-in for easy integration with existing VizzAnalyzer implementation.</td>
<td>Essential</td>
</tr>
<tr>
<td>2. Software Architecture Checker should extend the Eclipse Architecture by the help of wizards to get the project information that is going to be analyzed. The user should define components and rules for the system in this wizard.</td>
<td>Essential</td>
</tr>
<tr>
<td>3. Software Architecture Checker needs to be started from the Eclipse menu. VizzAnalyzer extends Eclipse by the help of menu extension point and supplies an entry point for user interface.</td>
<td>Essential</td>
</tr>
<tr>
<td>4. A wizard page should be implemented for selecting input/output files.</td>
<td>Essential</td>
</tr>
<tr>
<td>4.1. The graph file that contains information about the project, that is going to be analyzed by Software Architecture Checker, should be selectable by user</td>
<td>Essential</td>
</tr>
<tr>
<td>4.2. The file formats for the report like xml or raw text and location of files must be selectable by user</td>
<td>Desirable</td>
</tr>
<tr>
<td>5. A wizard page should be implemented for defining components and sub components</td>
<td>Essential</td>
</tr>
<tr>
<td>5.1. Components must be defined by regular expressions and must have a short description.</td>
<td>Essential</td>
</tr>
<tr>
<td>5.2. Components must be defined in a hierarchical way.</td>
<td>Essential</td>
</tr>
<tr>
<td>5.3. Defined component must be stored in components tree.</td>
<td>Essential</td>
</tr>
<tr>
<td>5.4. Define component should be able to be removed</td>
<td>Desirable</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5.5.</td>
<td>Import component definitions from external source.</td>
</tr>
<tr>
<td>5.6.</td>
<td>Export component definitions to an external file.</td>
</tr>
<tr>
<td><strong>6.</strong></td>
<td>A Wizard page should be implemented for defining allowed or disallowed rules between components</td>
</tr>
<tr>
<td>6.1.</td>
<td>Source and destination components must be selected from the list of defined components.</td>
</tr>
<tr>
<td>6.2.</td>
<td>VizzAnalyzer supplied call mechanisms must be used to define the rules between components.</td>
</tr>
<tr>
<td>6.3.</td>
<td>Defined rules must be stored in rules table.</td>
</tr>
<tr>
<td>6.4.</td>
<td>Defined rules should be able to be removed</td>
</tr>
<tr>
<td>6.5.</td>
<td>Import rules table from external source.</td>
</tr>
<tr>
<td>6.6.</td>
<td>Export rules table to an external file.</td>
</tr>
<tr>
<td><strong>7.</strong></td>
<td>Original graph file should be edited through test results to visualize components and violations</td>
</tr>
</tbody>
</table>

**Table 3-1 Requirements Analysis for Software Architecture Checker**
4 Architecture
Software Architecture Checker is going to be a sub module for VizzAnalyzer. Because of this requirement it is very important to design Software Architecture Checker as an Eclipse plug-in which can extend the specification of existing VizzAnalyzer implementation.

On the other hand Software Architecture Checker should be usable as a framework or component even though there is no Eclipse installation. This means getting apart the architecture checker from the plug-in user interface is quite important. The approach behind this goal is designing the Software Architecture Checker in different components like architecture checker core, configuration and plug-in user interface. Through this approach it is possible to use the architecture checker in many different ways.

For better understanding of the approach explained above, we are going to briefly discuss Eclipse plug-in architecture and VizzAnalyzer Architecture.

4.1 Architecture of Eclipse
Eclipse Rich Client Platform stands on plug-in based extensible architecture. Since Eclipse Integrated Development Environment is also a specified set of specialized plug-ins over Eclipse RCP. Eclipse supplies several extension points to implement such extensible architecture. New tools or plug-ins like JDT or PDE extends platform over these extension points. Figure 4.1 shows the Architecture of Eclipse.

![Figure 4.1 The Eclipse Architecture (7)](image)

The workbench is the Graphical User Interface of the Eclipse Platform and uses SWT and JFaces technologies to provide a GUI.

4.2 Plug-in Model of Eclipse
Eclipse architecture supports plug-ins that can be easily integrated. The “plugins” folder inside the Eclipse installation directory contains the needed information to integrate plug-ins. Plug-ins that work on Eclipse platform may also interact with each other.

Each plug-in is defined by a manifest file that is called “plugin.xml”. The Eclipse platform activates and integrates all plug-ins in “plugins” directory with the help of this manifest file. Figure 4.2 shows an example of this manifest file.
The “plugin.xml” file can extend Eclipse over different extension points. In the example above, `org.Eclipse.core.runtime.applications` extension point used to extend Eclipse platform. And `exampleplugin.Application` class is going to be run when plug-in is activated.

4.3 Architecture of Software Architecture Checker

The Software Architecture Checker consists of different components. In that way it is possible to use Software Architecture Checker either as an API or application. Architecture can be seen in Figure 4.3.

When we think about an architectural view of the whole software system, Software Architecture Checker and Eclipse are both components on their own. Software Architecture Checker consists of three different components: Eclipse plug-in UI, Configuration, Analyzer core.
4.3.1 Eclipse Plug-In UI
Eclipse plug-in user interface is the component that is designed for interacting between user and Analyzer core component. The goal of this component is getting Software Architecture Checker related information from user and giving out the reports.

4.3.2 Configuration
Configuration component of Software Architecture Checker holds the information for defined Components and Rules. It is initialized by User interface and used by the analyzer core.

4.3.3 Analyzer Core
Analyzer components is responsible for analyzing the architectural view of given software system through the configuration. After checking the architecture of software system it produces a report.
5 Design and Implementation

In the previous section we described the architecture of Software Architecture Checker. This section covers the design and implementation of Software Architecture Checker. First of all, we will look at the class diagram of the Software Architecture Checker. After describing the class diagram and design aspects we will continue with describing implementation of this System.

5.1 Class Diagram

Figure 5.1 depicts the class design of the Software Architecture Checker.

The Configuration component consists of:
- Configuration
- ComponentsModel
- Component
- RulesModel
- Rule

Classes;

The Analyzer core component is AnalyzerCore class;

The Eclipse plug-in UI component consists of:
- ArchitectureCheckerHandler
- ArchitectureCheckerWizard
- ComponentDefinitionWizardPage
- RuleDefinitionWizardPage

Classes;
Figure 5.1 Class Diagram of Software Architecture Checker
5.1.1 Configuration Class

*Configuration* class is designed for collecting and holding the information related with designed components. This configuration accesses and modifies *RulesModel* and *ComponentsModel* to store information.

*AnalyzerCore* class has an association with this class because Architecture Checker needs rules to check the architecture of given software system.

```
Configuration
  - components : ComponentsModel
  - rules : RulesModel
```

Figure 5.2 Configuration Class

5.1.2 ComponentsModel Class

*ComponentsModel* stores the root component, which is a static instance of *Component* class. This root component is a abstract component without children and parent. Logically, each given software system that is going to be analyzed is also a component on its own. This root component is an entry point to the definition of components and sub-components in a hierarchical way.

```
ComponentsModel
  - root : Component
```

Figure 5.3 ComponentsModel Class

Another idea behind the *ComponentsModel* class is to be able to get apart the Software Architecture Checker from the Eclipse plug-in UI. This gives opportunity to define data model and the viewer separately. Otherwise we could implement the data in Eclipse plug-in related classes and this operation would result in nested/integrated components to each other.

In this way Software Architecture Checker can store the component definitions in *ComponentsModel* class and Eclipse plug-in UI can also access the component definition data model to display by the help of *ComponentsLabelProvider*, *ComponentsContentProvider* classes.

5.1.3 RulesModel Class

*RulesModel* stores the list of rule definitions that created by the user. “rules” attribute of this class is an empty list with static modifier and used for storing rule definitions.

```
RulesModel
  - rules : List<Rule>
```

Figure 5.4 RulesModel Class

Just like *ComponentsModel*, *RulesModel* is also designed through “getting apart the analyzer and UI” aspect. Since *RulesModel* can store needed information in its own; *RulesContentProvider* and *RulesLabelProvider* classes can be used by Eclipse plug-in UI to display the data.
5.1.4 Component Class

*Component* class is designed to define software components of given software system. The abstraction of a software component, from the view point of Software Architecture Checker, needs a name and regular expression that defines the included classes for the component. An instance of this class may also contain a list of class names that is mapped by regular expression.

![Figure 5.5 Component Class](image)

Because of requirement 5.2, defining components in a hierarchical way, there is a parent child association between component objects. “parent” attribute supplies navigability through the parent component and “children” attribute is a list of sub-components.

5.1.5 Rules Class

Rule class is designed to define rules for Software Architecture Checker. An instance of Rule class has two components those are going to be checked against the rule. “fromComponent” and “toComponent” attributes defined because of the rule is designed as Directed. This means ‘X’ rule from component ‘A’ to component ‘B’ is different from the ‘X’ rule that is from component ‘B’ to component ‘A’.

![Figure 5.6 Rules Class](image)

“rule” attribute has a type of TypeValues class which is coming from existing VizzAnalyzer implementation. The graph file that is supplied by VizzAnalyzer framework contains call operations those are defined by TypeValues class.

“allowed” attribute is a Boolean value that implies whether rule between components is allowed or not.

“regex” attribute is designed to store regular expression version of the defined rule to check against implemented architecture.

5.1.6 TypeValues Class

*TypeValues* class is part of VizzAnalyzer framework and it is used to define call operations between different classes. These call operation values stored on the call graph that is created by VizzAnalyzer framework after analyzing operation of the software system.

From the perspective of the Software Architecture Checker we just used some part of these call operations those could be listed as:

- InvokesSuper
• Invokes
• CreatesClassInstance
• Extends
• InheritsFrom
• InvokesConstructor
• IsOfType
• Accessess
• InvokesSuperConstructor
• IsActualParameterOf
• Implements

5.1.7 AnalyzerCore Class

AnalyzerCore class is the most important component of the Software Architecture Checker. All checking operations are done and this class produces reports. Analyzer class has two attributes: “gml” and “configuration”.

```
AnalyzerCore
  gml : File
  configuration : Configuration

Figure 5.7 AnalyzerCore Class
```

“gml” attribute is the input graph file that is supplied by VizzAnalyzer framework. “configuration” attribute stands for configuring analyzer and contains component and rule definitions.

This design of Software Architecture Checker increases reusability of the code. Architecture Checker is designed as three components and it is able to work with given configuration for analyzer core component.

These classes and components that described above are sufficient to work with Software Architecture Checker. Now we will look into the classes those are needed by Eclipse plug-in UI.

5.1.8 ArchitectureCheckerHandler Class

ArchitectureCheckerHandler class is the entry point for plug-in user interface. It manages user interface related operations and main controller for plug-in component. This class extends Eclipse platform with a class named ArchitectureCheckerWizard, which is a subclass of Wizard class in Eclipse Platform.

ArchitectureCheckerHandler uses the AnalyzerCore component to check the given software system and gives opportunity to user for configuring Software Architecture Checker.

5.1.9 ArchitectureCheckerWizard class

This class is a subclass of Wizard class in Eclipse Platform. User interface of Software Architecture Checker consist of a wizard to define components and rules. When user executed plug-in ArchitectureCheckerHandler creates an ArchitectureCheckerWizard and displays it. ArchitectureCheckerWizard have three pages: One for selection of graph file and report files. The second one is to define the components and the last one is to define rules between components.

```
ArchitectureCheckerWizard
  -eg : SelectGraphFileWizardPage
  -cd : ComponentDefinitionWizardPage
  -rd : RuleDefinitionWizardPage

Figure 5.8 ArchitectureCheckerWizard Class
```
5.1.10 SelectGraphFileWizardPage Class
This class is the first page of ArchitectureCheckerWizard and responsible for selecting graph and report files.

![SelectGraphFileWizardPage](image1)

**Figure 5.9 SelectGraphFileWizardPage Class**

5.1.11 ComponentDefinitionWizardPage Class
This class is responsible for defining the components for given software system. There is a two text fields to enter description and regular expression information of the software component.

![ComponentDefinitionWizardPage](image2)

**Figure 5.10 ComponentDefinitionWizardPage Class**

Add button is responsible for adding the defined components to the component Tree if there is some components in component tree and some of them are selected the add button turns from “Add Component” to “Add Sub-Component”

It is possible to add a component as sub-component for several different components.

Remove button is responsible for removing selected components from component tree.

Tree is responsible for displaying defined components.

5.1.12 RuleDefinitionWizardPage Class
This class is responsible for definition of rules those are going to be used for checking the architecture of given software system. Definition of a rule for Software Architecture Checker needs a source and destination component. Because of this need there are two trees named as “clone1” and “clone2”. They display already defined components. And also user can select one of VizzAnalyzer supported call operations. “callOperation” attribute stands for displaying those operations in a list.

![RuleDefinitionWizardPage](image3)

**Figure 5.11 RuleDefinitionWizardPage Class**

Rules are, just like components, stored in their own model and a table viewer displays them.

“allow” and “disallow” buttons are responsible for adding defined rules to the “rules” table;
5.1.13 ComponentsContentProvider Class
This class is used by Eclipse plug-in UI to display ComponentsModel in a tree.

5.1.14 ComponentsLabelProvider Class
This class is used by Eclipse plug-in UI to display ComponentsModel in a tree.

5.1.15 RulesContentProvider Class
This class is used by Eclipse plug-in UI to display RulesModel in a tree.

5.1.16 RulesLabelProvider Class
This class is used by Eclipse plug-in UI to display RulesModel in a tree.
5.2 Implementation of Software Architecture Checker

Previously we have seen the design model of Software Architecture Checker. This section provides information about implementing the Software Architecture Checker.

We will give information about how did we use extension points of Eclipse and how did we configured plug-in entry points.

5.2.1 Plugin.xml

As we already described in chapter four, Eclipse plug-ins stored in “plugins” directory of Eclipse installation path and configured through plugin.xml file. The configuration of Software Architecture Checker could be seen in Figure 5.12.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <extension point="org.Eclipse.ui.commands">
    <category name="Sample Category">
      id="se.vxu.msi.vizzanalyzer.architecturechecker.commands.category">
    </category>
    <command name="Architecture Checker" categoryId="se.vxu.msi.vizzanalyzer.architecturechecker.commands.category">
    </command>
  </extension>
  <extension point="org.Eclipse.ui.handlers">
    <handler commandId="se.vxu.msi.vizzanalyzer.architecturechecker.commands.sampleCommand" class="se.vxu.msi.vizzanalyzer.architecturechecker.handlers.ArchitectureChecker">
    </handler>
  </extension>
  <extension point="org.Eclipse.ui.bindings">
    <key commandId="se.vxu.msi.vizzanalyzer.architecturechecker.commands.sampleCommand" contextId="org.Eclipse.ui.contexts.window" sequence="M1+6" schemeId="org.Eclipse.ui.defaultAcceleratorConfiguration">
    </key>
  </extension>
  <extension point="org.Eclipse.ui.menus">
    <menuContribution locationURI="menu:org.Eclipse.ui.main.menu?after=additions">
      <menu label="Architecture Checker" mnemonic="C">
        <command commandId="se.vxu.msi.vizzanalyzer.architecturechecker.commands.sampleCommand" mnemonic="S">
          <command>
        </menu>
      </menu>
    </menuContribution>
  </extension>
</plugin>
```
org.Eclipse.ui.commands extension point is used to define a category named “Sample Category” and a command named “Architecture Checker” is created in that category.

This command is used to define the entry point of Eclipse plug-in UI. org.Eclipse.ui.handlers mappes this command to the ArchitectureCheckerHandler class which is the entry point for user interface.

org.Eclipse.ui.menus extension point is used to extend Eclipse with Architecture checker menu and a toolbar. Menu item and Toolbar item use “Architecture Checker” command to execute the ArchitectureCheckerHandler class over handler. These extensions could be seen in

![Figure 5.13 Eclipse Extensions of Wostware Architecture Checker](image)
5.2.2 Package Structure of Software Architecture Checker

Implementation of Software Architecture Checker consists of several different packages. These packages created through functional decomposition of Software Architecture Checker. In other words each component of designed system relies on its own package.

The package structure for Software Architecture Checker can be seen in Figure 5.14.

More technical information about the packages and implementation of system can be reached on Appendix A.
6 Conclusion and Future Work
This chapter gives information about the results of project and future work.

6.1 Conclusion
In this thesis, we described a methodology to define software components and their intercommunication. This methodology is used to develop an approach in solving the problem.

As stated before in the first chapter, the problem was design and implementation of an Eclipse plug-in to define and check the architecture of a software system. Since definition support for software components and interfaces is not supplied by any programming languages, including Java, it is a non-trivial job.

Solving this problem was quite important because architecture of a software system is not easily controllable in manual ways. Design documents like class diagrams, sequence diagrams and implementation code of program should be checked against each other. Manual checking operation is not possible especially for big projects.

VizzAnalyzer has developed an approach for call-operations of the software system. Our component and rule definition approach is used together with VizzAnalyzer’s call-operation approach to check the implemented software architecture is as designed or not.

To solve the problem we discussed goals and criteria before.

The first goal was defining the components of software system. This goal has been met by abstraction of a software component to the Component class and second wizard page of user interface. ComponentDefinitionWizardPage class provides a user interface to define a software component.

Second goal was defining sub-components for components. This goal has been met by parent-child association between instances of Component class and component definition user interface.

The third goal was mapping defined components to the implementation of the software system. A software component is expected to consist of several classes. This goal is met by regular expression field of a component. Those classes can be defined by regular expressions.

The fourth goal is defining rules between already defined components. This goal is met by RuleDefinitionWizardPage class which provides user interface to define rules between already defined components. And rules are derived from call-operations which are supplied by VizzAnalyzer framework.

The fifth goal was reporting violations of the given software system through given component and rule definition. This goal has been met by reporting abilities of Software Architecture Checker. Violations can be reported both in XML and raw text format. Figure 6.1 shows an example report in XML format and Figure 6.2 shows an example report in raw text.
Figure 6.1 XML Report of Software Architecture Checker
Figure 6.2 Text File Report of Software Architecture Checker

The sixth goal was grouping the defined components in the graph file supported by VizzAnalyzer. Also a modified call-graph, which contains parameters of components and violations, is generated and met this goal.

Since all goals are met clearly we can result that the problem is successfully solved. Personally it was a great experience for me to solve this problem. I learned a lot of things about Eclipse and plug-in architecture. I studied various approaches for software architecture and importance of software architecture. I am sure that I will use this experience in my future studies or works. On the other hand, since Software Architecture Checker will be a part of VizzAnalyzer, it is very motivating to be able to participate in development of VizzAnalyzer.

6.2 Future Work

The implementation of Software Architecture Checker is successfully met all goals and criteria but some more optional requirements appeared during the development phase.

It would be better if Software Architecture Checker could present an import/export mechanism for defined component and also for defined rules. Since it can remember the configuration between different sessions but if Eclipse exits these configurations also get lost.

On the other hand documentation of the implemented System should be done. Both end user and developer need some documentation for using the Software Architecture Checker or adding more features.
7 References


8 Appendices

Appendix A - Package Structure of Software Architecture Checker

A.1 Package se.vxu.msi.vizzanalyzer.architecturechecker

This package is the main container for Software Architecture Checker. Different components like Configuration or Analyzer Core are located in different sub-packages.

A.2 Package se.vxu.msi.vizzanalyzer.architecturechecker.configuration

This package contains the classes related with the Configuration component. Configuration class of this package is the core of Configuration component. Figure 8.1 shows the list of classes in this package.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.configuration.Component</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.configuration.ComponentsModel</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.configuration.Configuration</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.configuration.Rule</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.configuration.RulesModel</td>
</tr>
</tbody>
</table>

Figure 8.1 List of classes for configuration

A.3 Package se.vxu.msi.vizzanalyzer.architecturechecker.core

This package consists of only AnalyzerCore class which performs the check operation and generates the reports.

AnalyzerCore class is used by ArchitectureCheckerWizard to perform the tests and generate report files. Main goal of this class is checking software architecture of given software system.

“checkArchitecture()” method of AnalyzerCore performs the check operation over given graph and configuration. The perform operation starts with iterating over the edges of graph and checking whether they are matching defined rules or not.

“writeGmlReport(File)” method is responsible for generating new call-graph which contains information about violations and grouped components.

“writeXmlReport(File)” method is responsible for generating report file in XML format.

“writeTxtReport(File)” method is responsible for generating report file in Text format.

A.4 Package se.vxu.msi.vizzanalyzer.architecturechecker.handlers

This package consists of only ArchitectureCheckerHandler class. It is the part of Eclipse plug-in UI component of Software Architecture Checker.

ArchitectureCheckerHandler class is the entry point for plug-in. The main goal of this handler class is initializing the ComponentsModel class and creating a wizard to display user interface.

A.5 Package se.vxu.msi.vizzanalyzer.architecturechecker.wizards

This package presents the main classes for Eclipse plug-in UI component. Figure 8.2 contains a list of classes inside this package.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.ArchitectureCheckerWizard</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.ComponentDefinitionWizardPage</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.ComponentsContentProvider</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.ComponentsLabelProvider</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.DataChangedEvent</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.DataChangedEventListener</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.RuleDefinitionWizardPage</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.RulesContentProvider</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.RulesLabelProvider</td>
</tr>
<tr>
<td>se.vxu.msi.vizzanalyzer.architecturechecker.wizards.SelectGraphFileWizardPage</td>
</tr>
</tbody>
</table>

Figure 8.2 List of Classes for wizard package
A.5.1 ArchitectureCheckerWizard Class
The instance of this class is created by ArchitectureCheckerHandler class. User select files and folders, defines components and rules in the wizard pages those created by this class.

The “performFinish()” method is executed when user clicks finish button in wizard. This method uses AnalyzerCore to check architecture and generate reports. Figure 8.3 shows the method and usage of AnalyzerCore component in this method. “sg” attribute below is the instance of SelectGraphFileWizardPage class.

```java
@override
public boolean performFinish() {
    Configuration c = new Configuration();
    c.setGmlReport(sg.reportGml());
    c.setXmlReport(sg.reportXml());
    c.setTxtReport(sg.reportTxt());

    ac = new AnalyzerCore(sg.getSourceLocation().toString(), c);
    ac.checkArchitecture();

    final String outputPrefix = sg.getDestinationLocation().toOSString() + sg.getSourceLocation().lastSegment();
    if (c.isXmlReport()) {
        File f = new File(outputPrefix + "xml");
        try {
            ac.writeXmlReport(f);
        } catch (TransformerConfigurationException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        }
    }
    if (c.isTxtReport()) {
        File f = new File(outputPrefix + "txt");
        try {
            ac.writeTxtReport(f);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    if (c.isGmlReport()) {
        File f = new File(outputPrefix + "gml");
        ac.writeGmlReport(f);
    }
    return false;
}
```

The “performCancel()” method is implemented to execute cleaning code if user clicks “Cancel” button in wizard. Basically removes listeners those are used to update component trees in UI.
A.5.1 ComponentDefinitionWizardPage Class
This class is implemented to define components and already described in previous section. Figure 8.4 depicts execution result of this class.

![Figure 8.4 Component Definition Wizard Page](image)

A.5.2 ComponentsContentProvider Class
This class is implemented to provide component data to the component trees. The role of this class is very important to separate the AnalyzerCore from Eclipse plug-in UI. TreeViewer classes use this class to get the data model for defined components.

To update different viewers those rely on same ComponentsModel this class implements event-listener methodology. When data changed in ComponentsModel “ComponentsContentProvider.inputChanged(Viewer, Object, Object)” method is called and this method sends DataChangedEvent to its listeners.

The “addDataChangedListener(DataChangedEventListener)” method is implemented to add listeners those that to be informed when data is changed.

A.5.3 ComponentsLabelProvider Class
This class implemented to provide labels for TreeViewer objects. It helps to display the name and regex of defined component.

A.5.4 DataChangedEvent Class
This class extends the java.util.EventObject and implemented to invoke DataChangedEventListener instances.
A.5.5 DataChangedEventListener Class
This is an interface that extends java.util.EventListener class to listen DataChangedEvent instances those sent by ComponentsContentProvider. An example usage of this interface can be seen in Figure 8.5. This example is taken from RuleDefinitionWizardPage. The “tv” and “tv1” attributes point instances of TreeViewer class.

```java
ComponentsContentProvider.addDataChangedListener(new DataChangedEventListener(){
    @Override
    public void dataChanged(DataChangedEvent e) {
        tv.refresh();
        tv1.refresh();
    }
});
```

Figure 8.5 Example Usage of DatachangeEventListener

A.5.6 RuleDefinitionWizardPage Class
This class is implemented to define rules and already described in previous section. Figure 8.6 depicts execution result of this class.

Figure 8.6 Rule Definition Wizard Page
A.5.1 RulesContentProvider Class
This class is implemented to provide rule data to the rules table. The role of this class is very important to separate the AnalyzerCore from Eclipse plug-in UI. Instances of TableViewer class use this class to get the data model for defined rules.

A.5.2 RulesLabelProvider Class
This class implemented to provide labels for TableViewer objects. It helps to display the information about defined rule.

A.5.3 SelectGraphFileWizardPage Class
This class is implemented to select the graph file and report types. User can specify the types and target location of report files. Normally reports are generated in the same directory with graph file. Figure 8.7 shows execution result of this class.

![Select Files Wizard Page](image_url)

Figure 8.7 Select Files Wizard Page