Software Design Description

for

AJCON, Applet to JSF Converter

Version 2.0

Prepared by

Anıl Sevim

Berkan KISAOĞLU

Özge TOKGÖZ

09.01.2011

Table of Contents

1. Introduction

This report intends to present complete design and progress of the Applet to Java Server Faces (JSF) Converter (AJCON) project, conducted by Team Teaplet. AJCON is supposed to be a software development tool which helps a software developer to migrate from Applet technology to JSF. This report explains complete descriptions of the proposed software system design. In this design document, general design architecture of the project will be enlightened and current project status will be indicated.

1.1. Problem Definition

Java Applets can provide web applications with interactive features that cannot be provided by HTML. When Java enabled browser is used to view a page that contains an applet, the applet's byte codes are transferred to user's system and executed by browser's Java Virtual Machine (JVM). Nowadays, applet technology has become out of date due to some disadvantages:

- It requires the Java plug-in.
- Some organizations only allow software installed by the administrators. As a result, some users can only view applets that are important enough to justify contacting the administrator to request installation of the Java plug-in.
- As with any client-side scripting, security restrictions may make it difficult or even impossible for an untrusted applet to achieve the desired goals.
- Some applets require a specific JRE. This is discouraged.
- If an applet requires a newer JRE than available on the system, or a specific JRE, the user running it the first time will need to wait for the large JRE download to complete.
- Java automatic installation or update may fail if a proxy server is used to access the web. This makes applets with specific requirements impossible to run unless Java is manually updated. The Java automatic updater that is part

of a Java installation also may be complex to configure if it must work through a proxy.

- Unlike the older applet tag, the object tag needs workarounds to write a cross-browser HTML document.

Meanwhile, with new Java 2 Enterprise Edition (J2EE) technologies, same functional requirements can be met with less dependency. JSF is one of these technologies, but switching from Applet to JSF requires both lots of money and manpower. Also, it is really long-lasting to write a JSF based application which does the same work with Applet from scratch. Even though there are some converters that may help employees at intermediate levels, there is no existing service, which does this conversion.

1.2. Purpose

The purpose of this document is to explain compelete design details of AJCON project. As IEEE standards document indicates, the Design Report show how the proposed software system will be structured in order to satisfy the requirements identified in the Software Requirements Specifications document. In other words, it is aimed to translate software requirements defined in SRS document into a representation of software components, interfaces and data to be used later in implementation phase of the project. However, since every software design is open to changes and modifications, it is highly possible to make changes during implementation and update SRS and SDD documents accordingly.

1.3. Scope

This complete SDD will contain the general definition and features of the project, design constraints, the overall system architecture and data architecture, a brief explanation about our current progress and schedule of the project. With the help of UML diagrams, design of the system and subsystems/modules will be explained visually in order to help the programmer to understand all information stated in this document correctly and easily.

1.4. Overview

This document encompasses a design model with architectural, interface, component level and deployment representations. Design model will be contained in this document, which will be used as a medium for communicating software design information, assessed for quality, improved before code is generated. Many graphical representations and verbal explanations were added to this document to achieve the goal of AJCON.

This document is divided into subsections to make it more understandable. Those are:

Section 2 contains general description about the system components.

Section 3 contains the assumptions made during the design process, dependencies and other constraints.

Section 4 contains general data structures that AJCON used.

Section 5 contains the most important diagrams of the document. Class diagrams, data flow diagrams and sequence diagrams of components are stated in this section. Also a brief explanation about the classes is mentioned.

Section 6 contains the user interface design and some screenshots.

Section 7 contains the detailed design issues and future works.

Section 8 contains the libraries and tools that we will use.

Section 9 contains the change LOG about the SRS.

Section 10 contains the basic timeline of the project.

Section 11 contains the conclusion of the SDD.

Those sections and subsections of them are mentioned in the table of contents more precisely.

1.5. Definitions, Acronyms and Abbreviations

SRS	Software Requirement Specifications
SDD	Software Design Document
AJCON	Applet to JSF Converter
JVM	Java Virtual Machine
JSF	Java Server Faces
J2EE	Java 2 Enterprise Edition
JavaML	Java Markup Language

1.6. References

[1] IEEE Recommended Practice for Software Design Descriptions

[2] AJCON Software Requirements Specifications Document, v1.0

[3] JavaML – A Markup Language for Java Sources, www.cs.washington.edu/research/constraints/web/badros-javaml-www9.ps.gz

[4] Apache Log4j, logging.apache.org/log4j/

[5] Jikes, jikes.sourceforge.net

[6] Apache Tomcat Wikipedia Page, Wikipedia.org/Apache_Tomcat

[7] Richfaces Community, jboss.org/richfaces

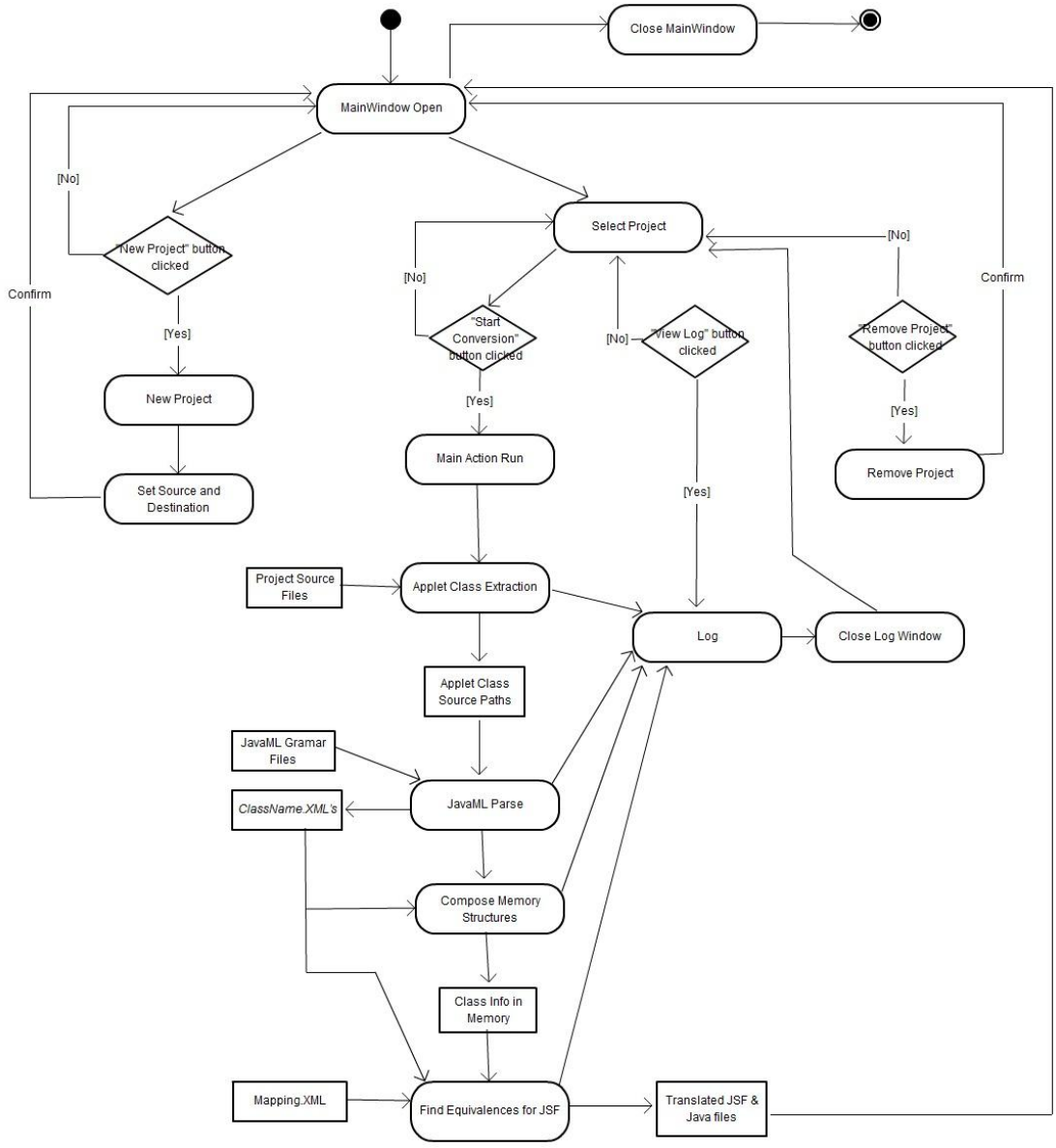
[8] Java Reflection API, http://download.oracle.com/javase/tutorial/reflect/index.html

2. System Overview

Main concern of the AJCON project is to help developers to make their work easy. For an applet project, converting it into a JSF project totally can be costly. With the use of AJCON, cost, man power needs and time needs of converting process can be decreased. It is not possible to convert all the projects with a rate of 100% correctness, but after the convert operation, little changes can raise the output of AJCON up.

In this context, we designed AJCON in a manner stated in section 5 and 7.

General description of the system drawn on the activity diagram stated below. Reactions defined on the user interface depends on the users actions, on the other hand, with the start of the conversion operation it is automated. User decides the operation will be done. Those operations can be adding/removing/selecting/deselecting/converting operations. Once converting operation starts, other related things done by AJCON like finding applets, parsing sources, displaying log information and etc.



3. Design Considerations

3.1. Design Assumptions, Dependencies and Constraints

3.1.1. Design Assumptions

AJCON is a huge project to design and implement. Since we have approximately six months to finish, we are requested by Siemens EC to make some assumptions in order to narrow down project to a certain level.

For complete design, our design assumptions can be stated as:

- This project runs on a Microsoft Windows platform (Vista or later),
- JRE must be installed on running computer,
- Application will be deployed to Apache Tomcat 6.0 or higher server,
- Input Java project should be syntactically correct and runnable.
- Input Java project should include at least one Applet class.
- For final design, inputs will be an Applet embedded html file, but for now, we assume an Applet Desktop Application as input. Later, we will turn to web based ones.
- For start, we will consider converting 8 basic Applet components to JSF. (See Section 4. Data Design)
- We will use JavaML tool [3] for parsing Java source files. Although, this software product is stated to be working for every Java source file, we have to assume that JavaML works properly. It should generate a well-formed and correct XML file, which is a complete self-describing representation of Java source code.

3.1.2. Design Dependencies

For complete design, our design dependencies can be stated as:

- JSF will depend on Java SE 5 (or higher).
- Software should run on a Microsoft Windows platform.

3.1.3. Design Constraints

3.1.3.1. Time

Under the scope of CEng 491-492 courses, we have approximately six months to finish our projects. In order to meet deadlines, we have to obey our schedule strictly. As we mentioned in our SRS document, we will be following agile software development model. Since it is a step-by-step approach, it is a must to update requirements and solutions. According to the feedback we will take, we will improve the general design and process of our project. Thus, we are planning not to fall behind the schedule.

3.1.3.2. Performance

For every software product, performance is an important criteria. Since AJCON project will be run by local clients at Siemens EC and there is no multi-user operation, we expect that conversion from Applet to JSF will end up at most in a few seconds.

3.2. Design Goals and Guidelines

3.2.1. Portability

There will be an installer for AJCON that runs only on Microsoft Windows platform mentioned both in assumptions and dependencies. Although Java ML tool is written in C++, there is only a Microsoft Windows executable publicly available. We are planning to request a Unix platform executable from designer of Java ML tool. If we are able to access that executable, we will make AJCON project portable for every operating system since Java is a machine independent language and works on every platform.

3.2.2. Reliability

Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Responses and the work done by the system should be consistent.

3.2.3. Correctness

AJCON will work correctly if all the requirements and assumptions are met. It will give the same result regardless of time, environment, etc.

4. Data Design

4.1. Data Description

We will keep our data in simple XML files; therefore converting those XML files into data structures in the memory is so simple. Several files are processed during the process of conversion and running of the system. Those are:

4.1.1. Data Objects

4.1.1.1. External Data Objects:

- User defined inputs
- Project input files
- Mapping.xml
- javaml-2.dtd & javaml-2.xsd
- ClassName.xml(Output of the JavaML)
- Output files
- Log files

All the above files except from output files are required to run the system properly. ClassName.xml and output files are constructed during the conversion operation and they are not temporary files. We will keep them to compare the results of the output with the initial sources. Functionalities and structures of those files are described below.

User defined inputs:

User must define source project folder path and destination project folder path via GUI. These data are used to get all project files included in source project folder path and generate output JSF project files in destination project folder path.

Project input files:

Project input files will be specified in run time. With the use of GUI, user specifies the source folder path to get the files. All the files under the path of source folder are the project input files for the system. All the files will be searched and applet classes will be extracted among them.

Mapping.xml

```
<MappingElements>
        <MapElement>
                <Object>
                        <Applet> JButton </Applet>
```

```
                    <JSF> h:Button </JSF>
                </Object>
                <Properties>
                    <Property type= “message”>
                        <Applet> addActionListener </Applet>
                        <JSF> action </JSF>
                    </Property>
                    <Property type= “message”>
                        <Applet> setText </Applet>
                        <JSF>   value </JSF>
                    </Property>
                    <Property type= “message”>
                        <Applet> repaint </Applet>
                        <JSF> rerender </
                    </Property>
                </Properties>
            </MapElement>
            <MapElement>
                <Object>
                    <Applet> JCheckBox </Applet>
                    <JSF> h:selectBooleanCheckbox </JSF>
                </Object>
                <Properties>
                    <Property type=“message”>
                        <Applet> addActionListener </Applet>
                        <JSF> value </JSF>
                    </Property>
                </Properties>
            </MapElement>
        </MappingElements>
```

Mapping.xml file will be used as data storage for the applet components and the equivalences of these components in the JSF. In the run time all the data in this mapping file will be transformed into the memory for further operations.

Above mapping file is an example-mapping file. At initial step, we will not only consider these two components stated in the example mapping file. We will try to convert other components also. All the components that we will try to convert are:

- JButton
- JCheckBox
- JTextField
- JTextArea
- JComboBox
- JLabel
- JRadioButton
- JList

javaml-2.dtd & javaml-2.xsd

javaml-2.dtd and javaml-2.xsd files are reference documents to grammar and lexer rules of JavaML. Both these file constructs the format of the parse and lex operation.

Some parts of javaml-2.dtd and javaml-2.xsd files shown below to make it clear.

Sample javaml-2.xsd:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified">
    <xs:attributeGroup name="location">
    <xs:attribute name="line" type="xs:string"/>
            <xs:attribute name="col" type="xs:string"/>
            <xs:attribute name="end-line" type="xs:string"/>
            <xs:attribute name="end-col" type="xs:string"/>
            <xs:attribute name="commentToken" type="xs:string"/>
            <xs:attribute name="startToken" type="xs:string"/>
```

```
        <xs:attribute name="endToken" type="xs:string"/>
        <xs:attribute name="idkind" type="xs:string"/>
</xs:attributeGroup>
<xs:group name="stmt">
        <xs:choice>
                <xs:element ref="block"/>
                <xs:element ref="local-variable-decl"/>
                <xs:element ref="try"/>
                <xs:element ref="throw"/>
                <xs:element ref="if"/>
                <xs:element ref="switch"/>
                <xs:element ref="loop"/>
                <xs:element ref="do-loop"/>
                <xs:element ref="return"/>
                <xs:element ref="continue"/>
                <xs:element ref="break"/>
                <xs:element ref="synchronized"/>
                <xs:group ref="expr"/>
        </xs:choice>
</xs:group>
<xs:group name="expr">
        <xs:choice>
                <xs:element ref="send"/>
                <xs:element ref="new"/>
                <xs:element ref="new-array"/>
                <xs:element ref="var-ref"/>
                <xs:element ref="formal-ref"/>
                <xs:element ref="field-ref"/>
                <xs:element ref="field-access"/>
                <xs:element ref="array-ref"/>

        …
```

Sample javaml-2.dtd:

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

<!ELEMENT anonymous-class (superclass?, implement*, (constructor | method | field | instance-initializer)*)>

<!ATTLIST anonymous-class
    abstract CDATA #IMPLIED
    final CDATA #IMPLIED
    synchronized CDATA #IMPLIED
    line CDATA #IMPLIED
    col CDATA #IMPLIED
    end-line CDATA #IMPLIED
    end-col CDATA #IMPLIED
    comment CDATA #IMPLIED
>

<!ELEMENT arguments (((send | new | new-array | var-ref | formal-ref | field-ref | field-access | array-ref | paren | assignment-expr | conditional-expr | binary-expr | unary-expr | cast-expr | instanceof-test | literal-number | literal-string | literal-char | literal-boolean | literal-null | this | super)))*>

<!ELEMENT array-initializer (array-initializer | ((send | new | new-array | var-ref | formal-ref | field-ref | field-access | array-ref | paren | assignment-expr | conditional-expr | binary-expr | unary-expr | cast-expr | instanceof-test | literal-number | literal-string | literal-char | literal-boolean | literal-null | this | super)))*>

<!ATTLIST array-initializer
    length CDATA #REQUIRED
>

<!ELEMENT array-ref (base, offset)>

…
```

Both of the javaml-2.xsd and javaml-2.dtd files are too long and complex files. It is not possible to put all the information in them. For further information, you may visit the website stated in the references section.

ClassName.xml

ClassName.xml is the output file of JavaML. ClassName should be the class name that extends the Applet class. This xml file is automatically generated after JavaML's run on Java source code files of the input Applet project. It conforms to javaml-2.dtd and javaml-2.xsd file structures. This file will be parsed with the help of Mapping.xml by Translator component. (Sample ClassName.xml file format is shown in section 8.1)

Output Files

Output files are JSF files that have been converted from input Applet project.

4.1.1.2. Internal Data Objects

Internal Data Objects for each component are shown in diagrams in section 5.

4.1.3. Data Models

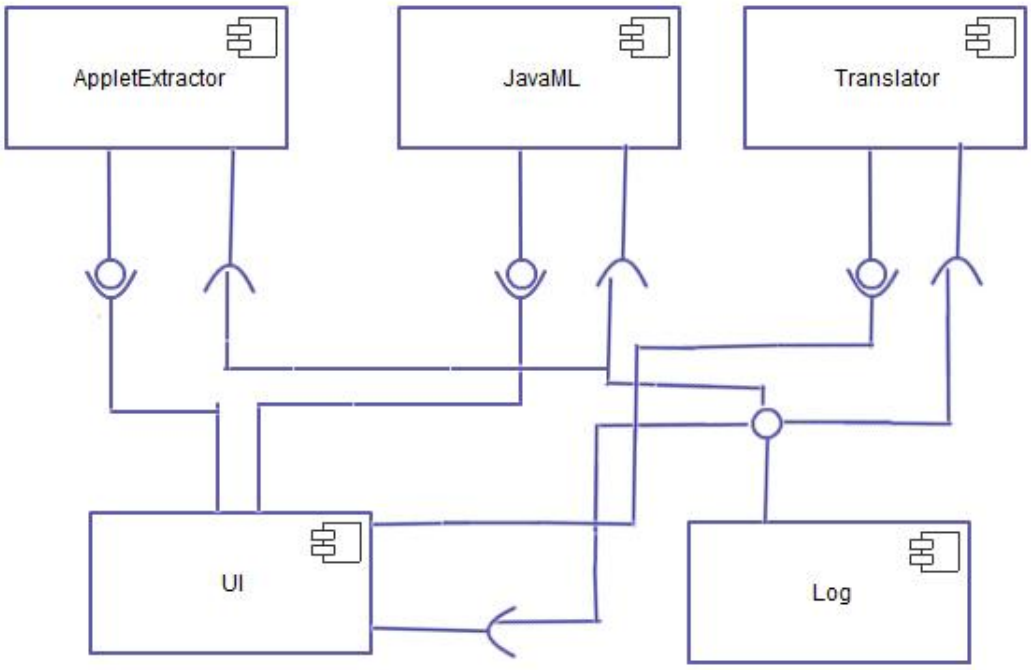
AJCON project does not use database. Therefore, ER Diagram for database modeling is not drawn. For data modeling of the system, data flow diagram is supplied in section 4.1.2.

4.1.4. Data Dictionary

AJCON project does not use database. Therefore, ER Diagram for database modeling is not drawn. For data modeling of the system, data flow diagrams drawn for components are supplied in section 5.2.

5. System Architecture

5.1. Architectural Design



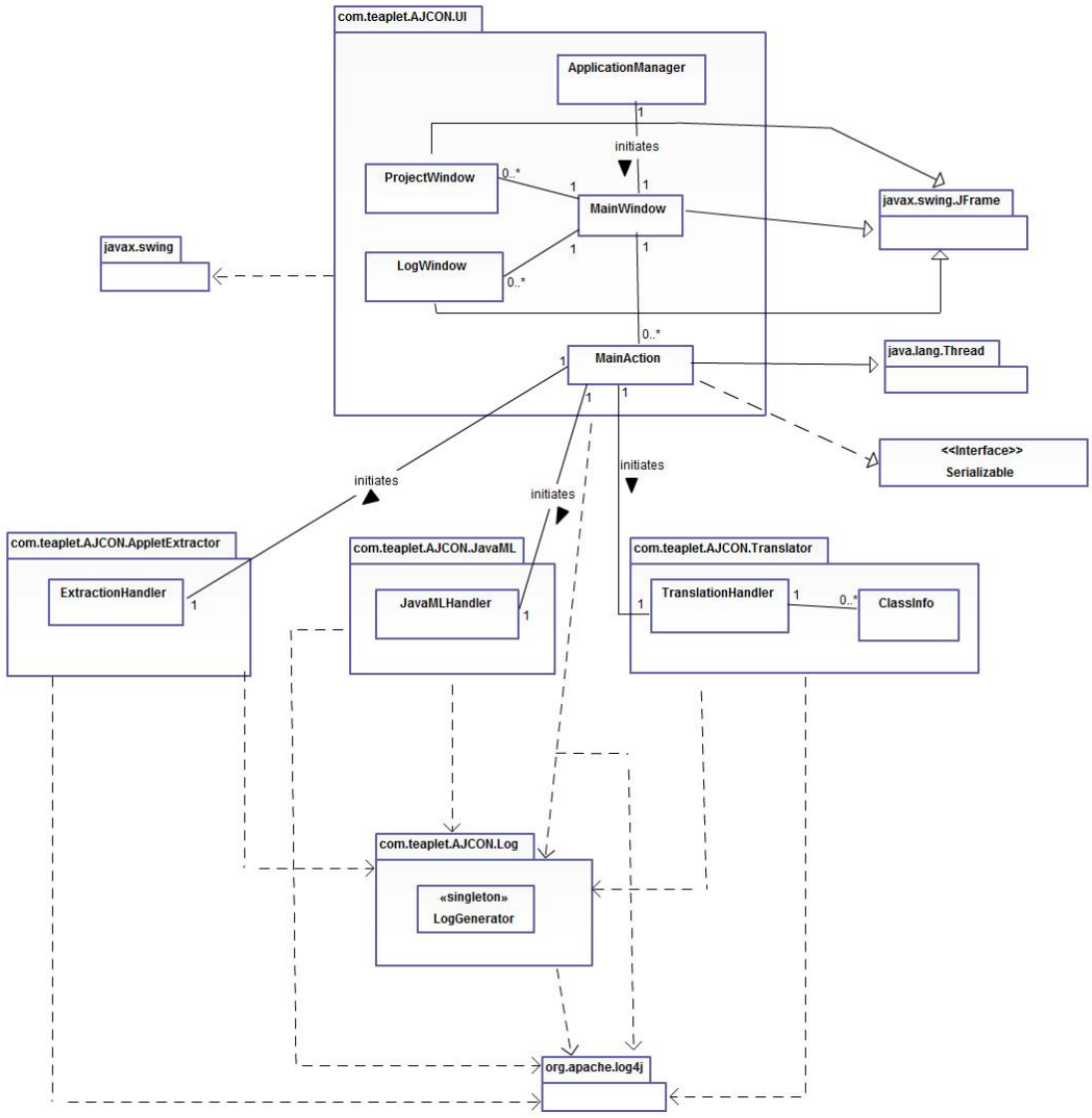
Main concern of AJCON is to convert an Applet project to JSF project. For this purpose AJCON project composed of several components: Log Component, UI Component, Translator Component, JavaML component, Applet Extractor Component.

Those components are interacting with each others. Some of them provide some interfaces to other ones, and some of them use the provided interface. Generally the interfaces provided by the other components are the methods of the classes in it.

Above relations shows that Log component provides an interface to other components and all the other components uses it. By the same way, it is shown that UI Component uses all the interfaces provided in the system. All the existing interfaces and the relations between the components are on the diagram.

5.2. Description of Components

Below there is the package diagram of the overall system. Each package/component will be described in subsections.



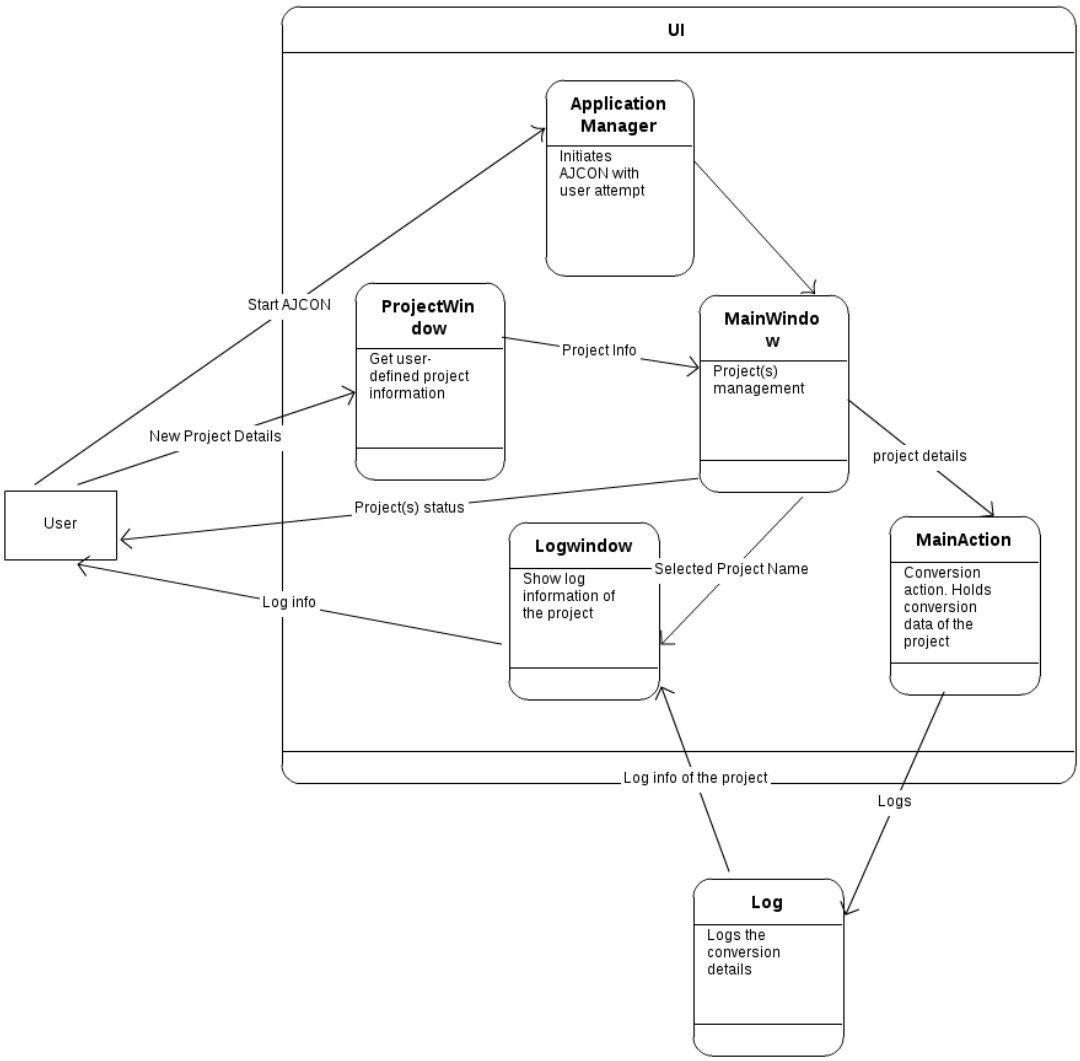
5.2.1. UI Component

5.2.1.1. Processing Narrative for UI Component

This is the component which interacts with user. Since our project does not require lots of user interactions, this component is not complex structured. It has a simple interface and simple purpose. By this component, user can manage projects to be converted with add/remove/select/startConversion options.

At the beginning, “ApplicationManager” class which has the main function initiates the system and shows the user UI main window. Then, when user clicks the “New Project” button, “Project Window” will pop up. By selecting destination and source folder, user adds project to list of project to be converted in main window. User can manage the main window by adding or removing projects with this method and start the conversion of any project that he/she selects. After starting a conversion, user can watch the live continuation of conversion process from main window and see logs.

5.2.1.2. Interface Description of UI Component



5.2.1.3. Processing Detail of UI Component

UI component consists of 5 different classes.

5.2.1.3.1. ApplicationManager Class

This class has the main function of the project. It initiates run of the project and sets MainWindow.

5.2.1.3.1.1. Attributes

- public static MainWindow mainWin: This instance variable is set by Application Manager in main function of the project.

5.2.1.3.1.2. Methods

- public static void main (String[] args): This is the main function of the project. When the project runs, this function is called automatically. In this function, main window will be created and system will be initiated.

5.2.1.3.2. MainWindow Class

This is the window that the user can directly manage all conversion operations. This class extends javax.swing.JFrame class and uses javax.swing components for GUI.

5.2.1.3.2.1. Attributes

- private int[] selectedProjects: This keeps id numbers of the projects that user selected.
- private static ArrayList<MainAction> mainActionList: List of main actions for each project thread.
- private static ArrayList<LogWindow> logWindowList: Keeps list of log windows that user wants to see.

- private javax.swing.JSeperator: Seperator between top-level labels and values.
- private javax.swing.JPanel panel: Contains javax.swing GUI components.
- private const javax.swing.JLabel labelProjectName: Constant header label, set as “Project Name” at first.
- private const javax.swing.JLabel labelSourceFolder: Constant header label, set as “Source Folder” at first.
- private const javax.swing.JLabel labelCreateDate: Constant header label, set as “Create Date” at first.
- private const javax.swing.JLabel labelProgressBar: Constant header label, set as “Progress” at first.
- private const javax.swing.JLabel labelSelected: Constant header label, set as “Selected” at first.
- private javax.swing.JButton buttonNewProject: User can create a new project by pressing this button.
- private javax.swing.JButton buttonRemoveProject: User can remove selected project(s) from the list by pressing this button.
- private javax.swing.JButton buttonStartConversion: User can start conversions of the selected project(s) by pressing this button.
- private javax.swing.JButton buttonViewLog: User can see log(s) of the selected project(s) by pressing this button.
- private static ArrayList<javax.swing.JLabel> listProjectNames: This instance variable keeps names of the projects in the main window.
- private static ArrayList<javax.swing.JLabel> listSourceFolders: This instance variable keeps source folder paths of the projects in the main window.
- private static ArrayList<javax.swing.JLabel> listCreateDates This instance variable keeps creation dates of the projects in the main window.:
- private static ArrayList<javax.swing.JProgressBar> listProgressBars: This instance variable keeps progress bar info of the projects in the main window.
- private static ArrayList<javax.swing.JCheckBox> listCheckBoxes: This instance variable keeps checbox’s status for each project in the main window.

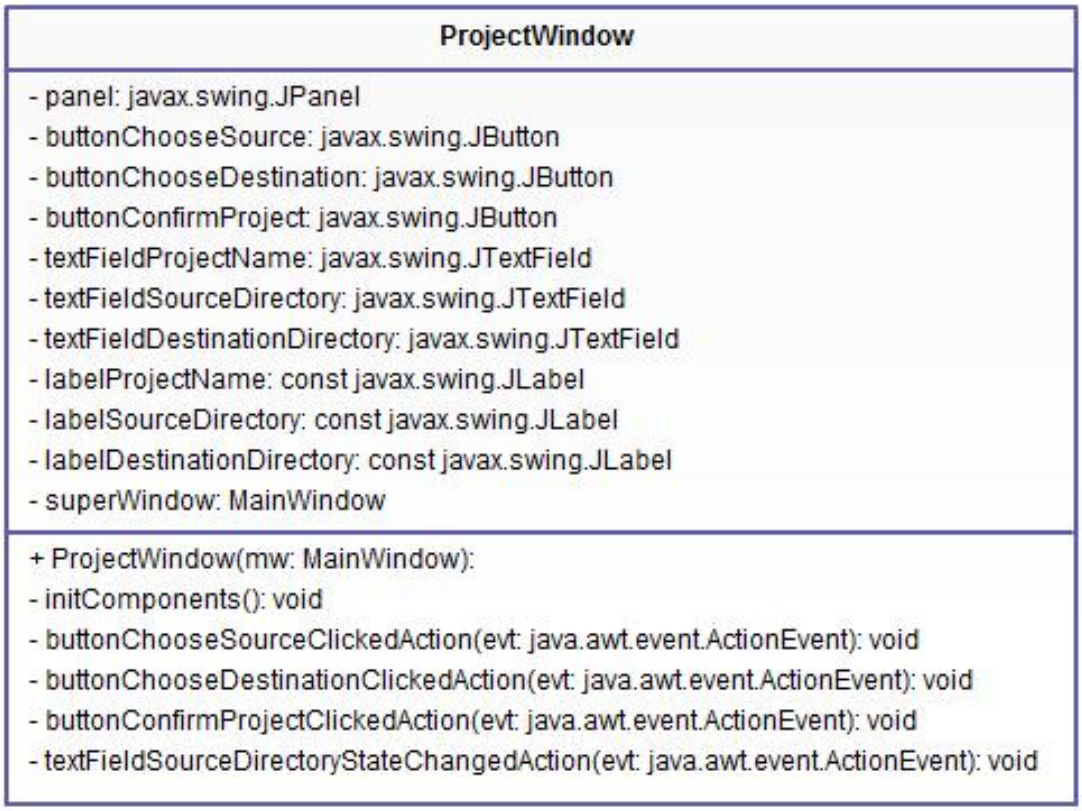
- private org.apache.log4j.Logger logger: This variable is used to log any kind of information inside this class.

5.2.1.3.2.2. Methods

- public MainWindow(): Constructor of the MainWindow class.
- public void initComponents(): Initializes interface components.
- public ArrayList<javax.swing.JLabel> getProjectNames(): Returns list of project names.
- public ArrayList<javax.swing.JLabel> getSourceFolders(): Returns list of source folder paths of the projects.
- public ArrayList<javax.swing.JLabel> getCreateDates(): Returns list of creation dates of the projects.
- public ArrayList<javax.swing.JProgressBar> getProgressBars(): Returns list of progress bar info of the projects.
- public ArrayList<javax.swing.JCheckBox> getCheckBoxes(): Returns list of check box's statuses of the projects.
- public ArrayList<LogWindow> getLogWindows(): Returns list of log windows that user wants to see.
- private void buttonNewProjectClickedAction (java.awt.event.ActionEvent evt, MainWindow mw): When user clicks "New Project", information related to project taken from project window is used as parameter and this function is called.
- private void buttonRemoveProjectClickedAction (java.awt.event.ActionEvent evt): When user clicks "Remove Project", this function is called.
- private void buttonStartConversionClickedAction (java.awt.event.ActionEvent evt): When user clicks "Start Conversion", this function is called.
- private void buttonViewLogClickedAction (java.awt.event.ActionEvent evt): When user clicks, log window(s) open and shows log info to user.

5.2.1.3.3. ProjectWindow Class

This is the class that lets user add a new project with a new window. This class extends javax.swing.JFrame class and uses javax.swing components for GUI.



5.2.1.3.3.1. Attributes

- private javax.swing.JPanel panel: The panel that keeps objects in project window together.
- private javax.swing.JButton buttonChooseSource: Button that is used for choosing source folder.
- private javax.swing.JButton buttonChooseDestination: Button that is used for choosing destination folder.
- private javax.swing.JButton buttonConfirmProject: Button that is used for confirming project conversion.
- private javax.swing.JTextField textFieldProjectName: Text field object that is used for entering project name .

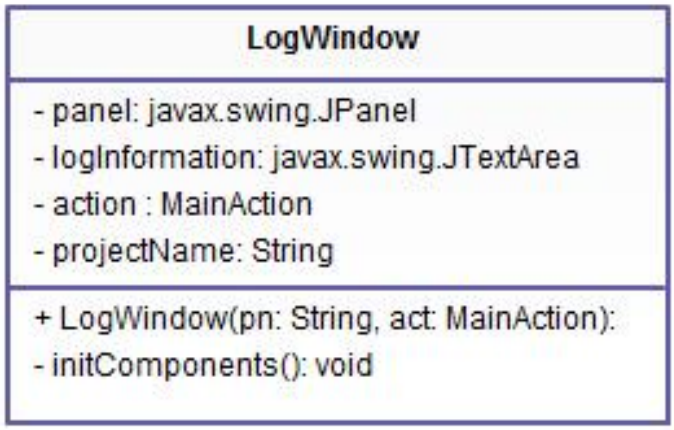
- private javax.swing.JTextField textFieldSourceDirectory: Text field object that is used for entering source directory.
- private javax.swing.JTextFiled textFiledDestinationDirectory: Text field object that is used for entering destination directory.
- private javax.swing.JLabel labelProjectName: Label of project name, that is “Project Name”.
- private javax.swing.JLabel labelSourceDirectory: Label of source directory, that is “Source Directory”.
- private javax.swing.JLabel destinationDirectory: Label of destination directory, “Destination Directory”.
- private MainWindow superWindow: Reference for main window object instance.

5.2.1.3.3.2. Methods

- public ProjectWindow (MainWindow mw): Constructor of ProjectWindow class. Sets mw:MainWindow as its super class object.
- private void initComponents (): Initiates object’s project window components.
- private void buttonChooseSourceClickedAction (java.awt.event.ActionEvent evt): Event handler for clicking “Choose Source” button.
- private void buttonChooseDestinationClickedAction (java.awt.event.ActionEvent evt): Event handler for clicking “Choose Destination” button.
- private void buttonConfirmProjectClickedAction (java.awt.event.ActionEvent evt): Event handler for clicking “Confirm Project” button.
- private void textFieldSourceDirectoryStateChangedAction (java.awt.event.ActionEvent evt): Event handler for text field source directory.

5.2.1.3.4. LogWindow Class

This class shows log information that it takes from Logger object and shows it to user. This class extends javax.swing.JFrame class and uses javax.swing components for GUI.



5.2.1.3.4.1. Attributes

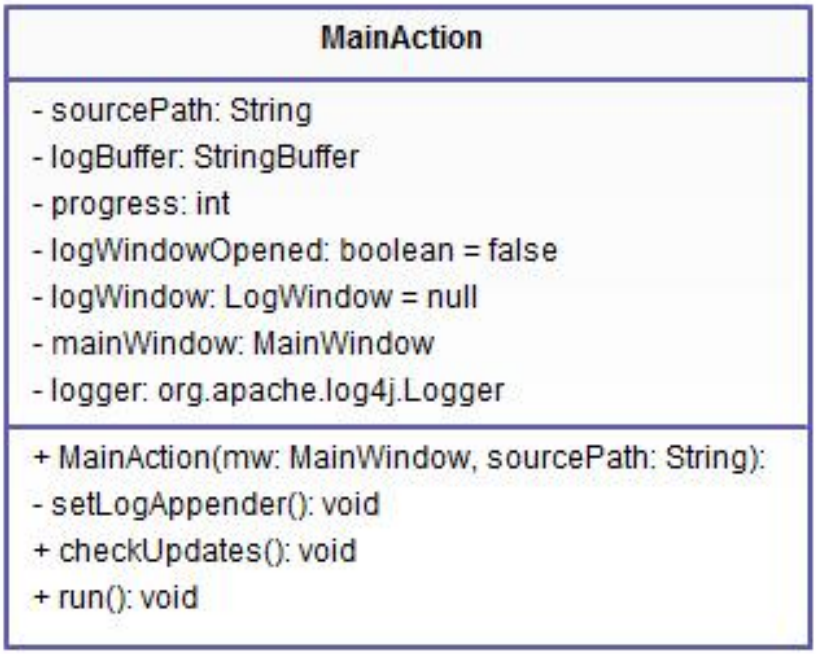
- private javax.swing.JPanel panel: The panel that keeps objects in log window together.
- private javax.swing.JTextArea logInformation: Text area field for log information.
- private MainAction action: Reference for main action object instance.
- private String projectName: Shows name of the project that are being logged.

5.2.1.3.4.2. Methods

- public LogWindow (String pn, MainAction act): Constructor of LogWindow.
- private void initComponents(): Initiates log window components.

5.2.1.3.5. MainAction Class

When user clicks “Start Conversion”, one instance of this class is instantiated for every project and it starts to run. This class also extends Thread and implements Serializable because it uses a multi-threaded approach for every single project run. It lets user to convert several projects at a time.



5.2.1.3.5.1. Attributes

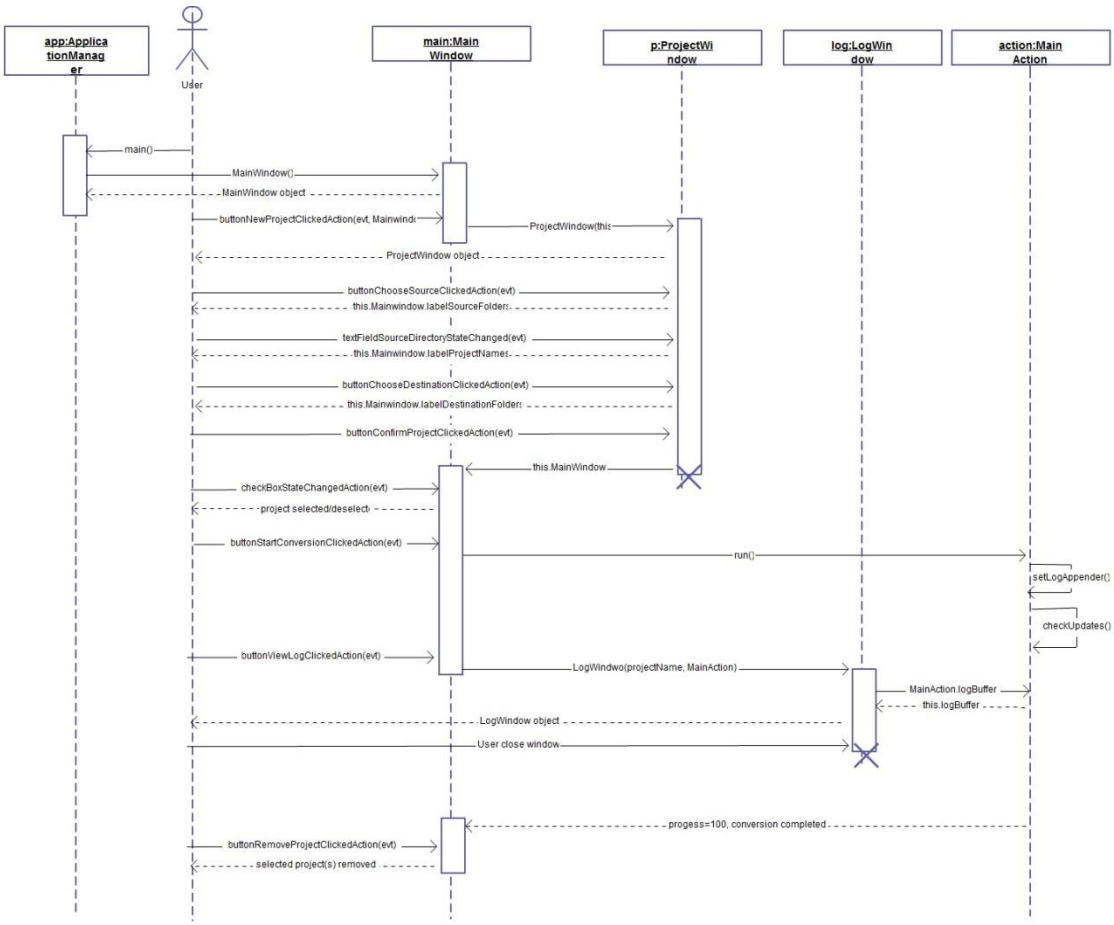
- private String sourcePath: Keeps project source path.
- private StringBuffer logBuffer: The StringBuffer object for logging continuously.
- private int progress: Keeps percentage of the project. Between 0-100.
- private boolean logWindowOpened=false: Boolean value for log window. If open, it is updated in real-time.
- private LogWindow logWindow=null: Reference for LogWindow object instance.
- private MainWindow mainWindow: Reference for MainWindow object instance.
- private org.apache.log4j.Logger logger: Singleton object reference for only one Logger object instance.

5.2.1.3.5.2. Methods

- public MainAction (MainWindow mw, String sourcePath): Constructor of MainAction class.
- private void setLogAppender(): Initiates format of the logger and type of buffer for project.

- public void checkUpdates(): Refreshes the screens.
- public void run(): Function that is needed to be called for thread's start.

5.2.1.4. Dynamic Behavior of UI Component

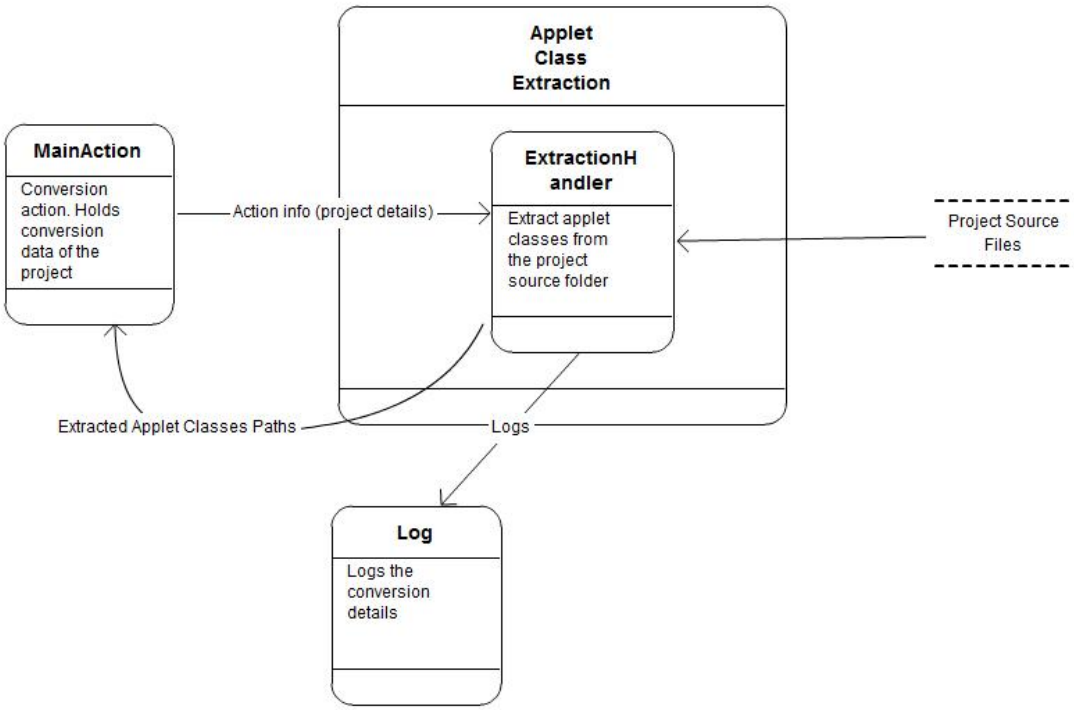


5.2.2. AppletExtractor Component

5.2.2.1. Processing Narrative for AppletExtractor Component

AppletExtractor Component is responsible from finding java sources that extends JApplet class. When the MainAction class is invoked from the user interface, MainAction class constructs an ExtractionHandler in Applet Extractor Component. This component searches the project folder into the deep, and looks all the files in the folders. Component notes down the source files that extend JApplet.

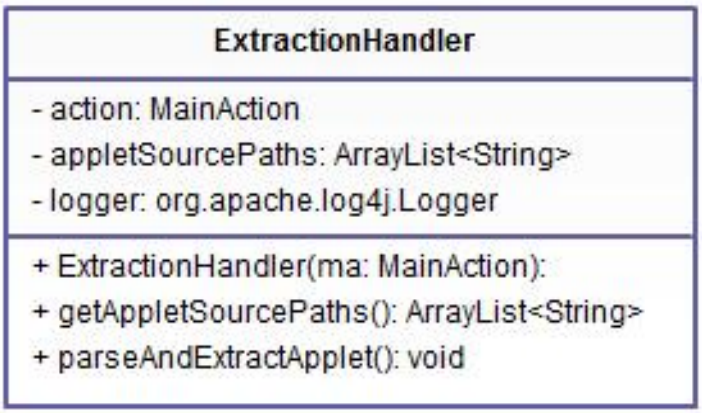
5.2.2.2. Interface Description of AppletExtractor Component



5.2.2.3. Processing Detail of AppletExtractor Component

AppletExtractor component has only one class: ExtractionHandler.

5.2.2.3.1. ExtractionHandler Class



5.2.2.3.1.1. Attributes

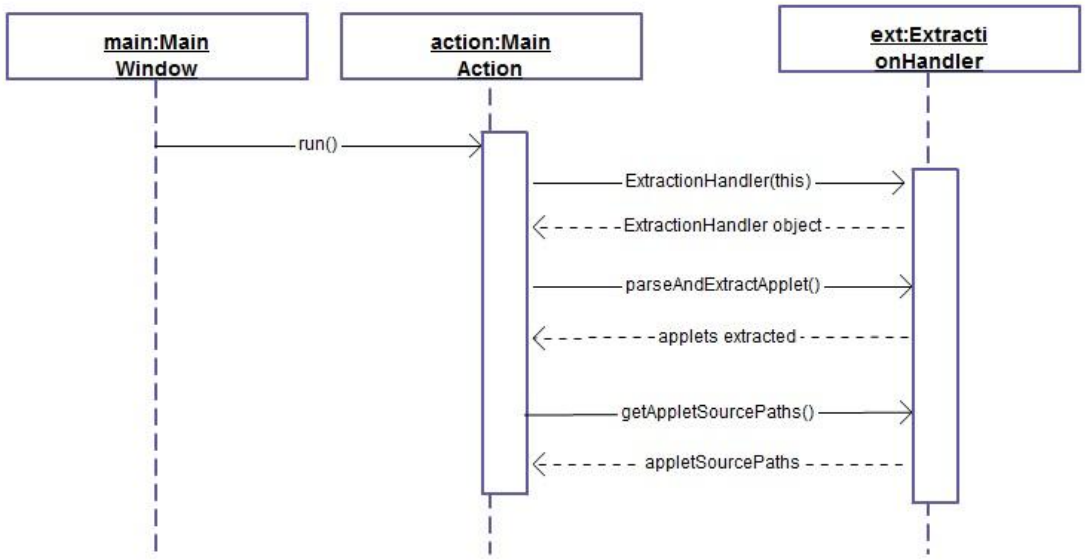
- private MainAction action: Reference to an instance of MainAction class.

- private ArrayList<String> appletSourcePaths: When the class finds a source pushes the file path to list.
- private org.apache.log4j.Logger logger: Singleton object reference for only one Logger object instance.

5.2.2.3.1.2. Methods

- public ExtractionHandler (MainAction ma): Constructor of ExtractionHandler.
- public ArrayList<String> getAppletSourcePaths(): Getter method for field appletSourcePaths.
- public void parseAndExtractApplet():Looks into to deeps of project folder to find source files, which extends JApplet.

5.2.2.4. Dynamic Behavior of AppletExtractor Component



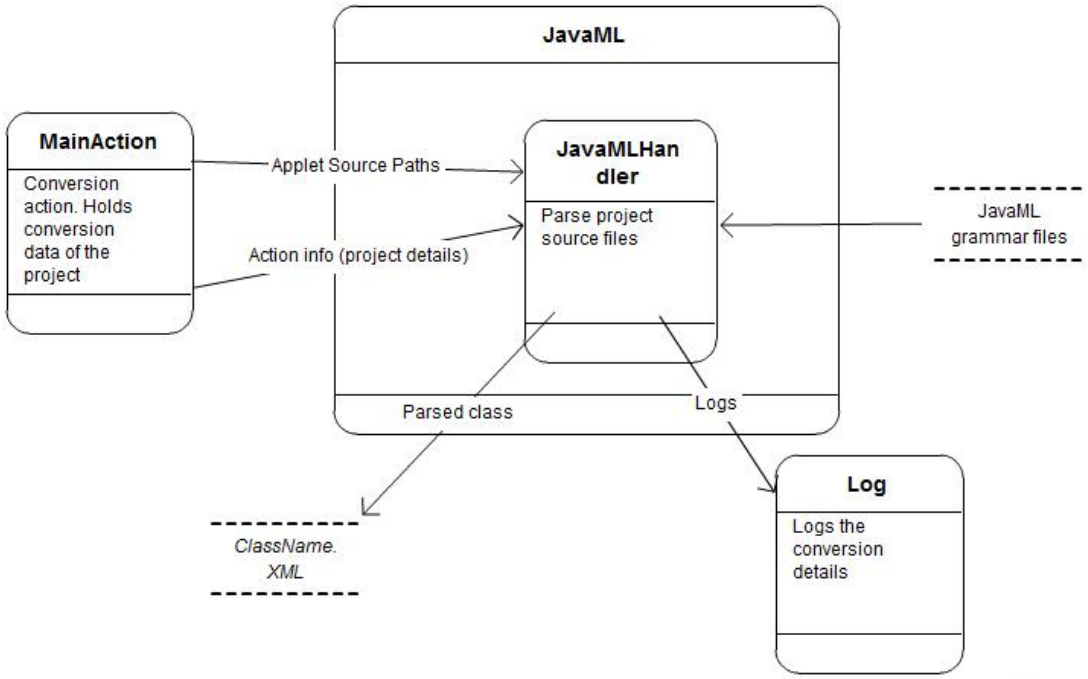
5.2.3. JavaML Component

5.2.3.1. Processing Narrative for JavaML Component

JavaML Component is responsible from lexical analysis and tokenizing the source files. After the process of Applet Extractor Component finishes, MainAction class inititates a JavaMLHandler object. JavaMLHandler object gathers the paths of the source files,

which extends JApplet, from the ExtractionHandler object. After gathering those paths runs Jikes over them.

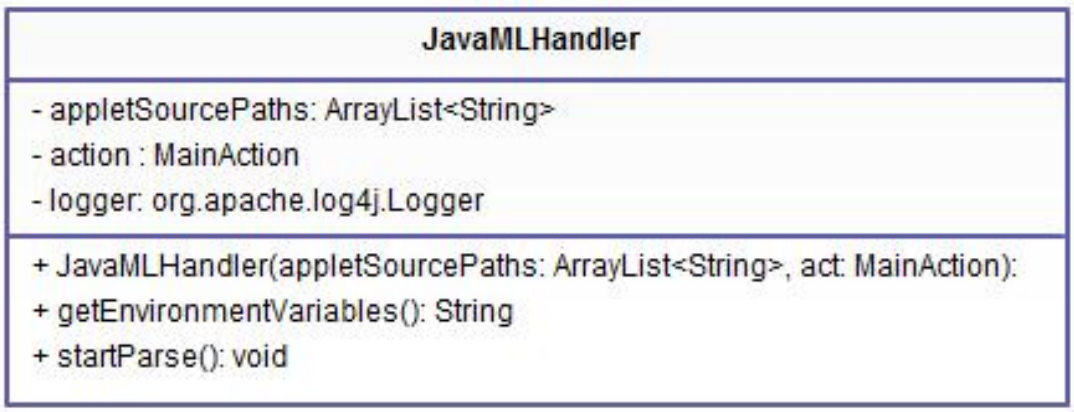
5.2.3.2. Interface Description of JavaML Component



5.2.3.3. Processing Detail of JavaML Component

JavaML component consists of only one class: JavaMLHandler.

5.2.3.3.1. JavaMLHandler Class



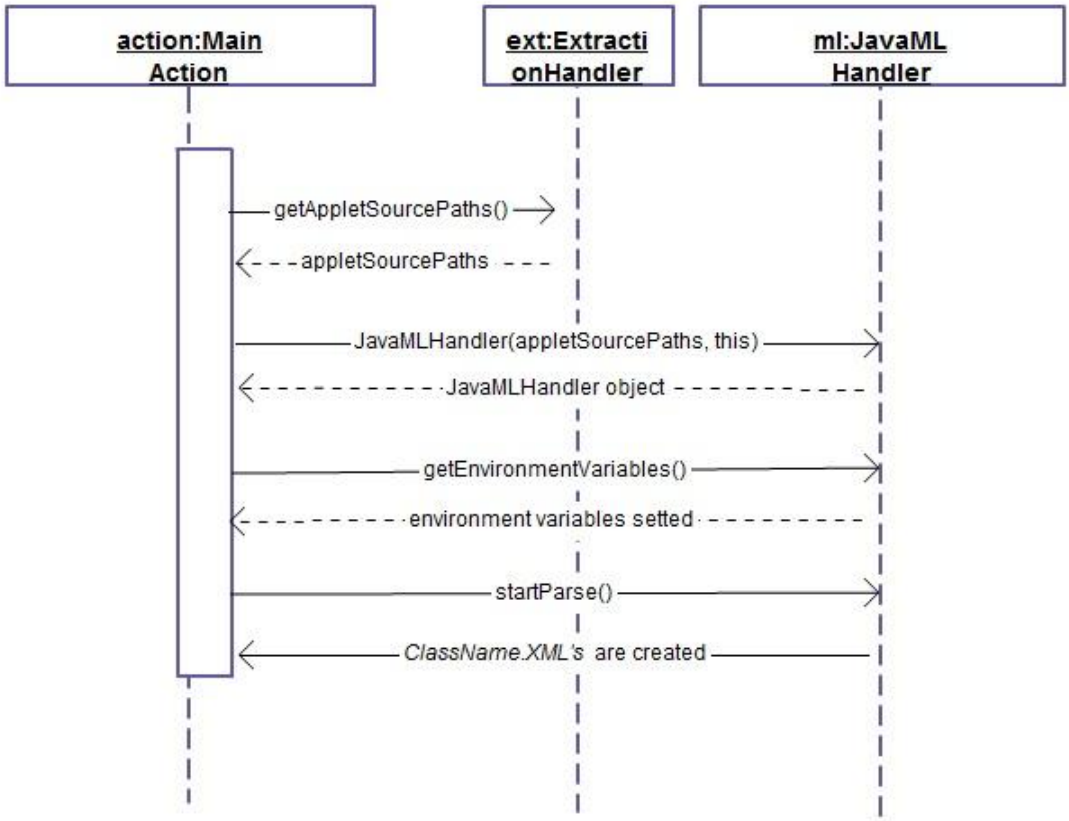
5.2.3.3.1.1. Attributes

- private ArrayList<String> appletSourcePaths: Gathered path information from the ExtractionHandler object.
- private MainAction action: Reference to MainAction instance.
- private org.apache.log4j.Logger logger: Singleton object reference for only one Logger object instance.

5.2.3.3.1.2. Methods

- public JavaMLHandler (ArrayList<String> appletSourcePaths, MainAction act): Constructor for JavaMLHandler class.
- public String getEnvironmentVariables (): Gets the environment variables defined on the system to look for JDK path.
- public void startParse(): Runs JavaML/Jikes over the files.

5.2.3.4. Dynamic Behavior of JavaML Component



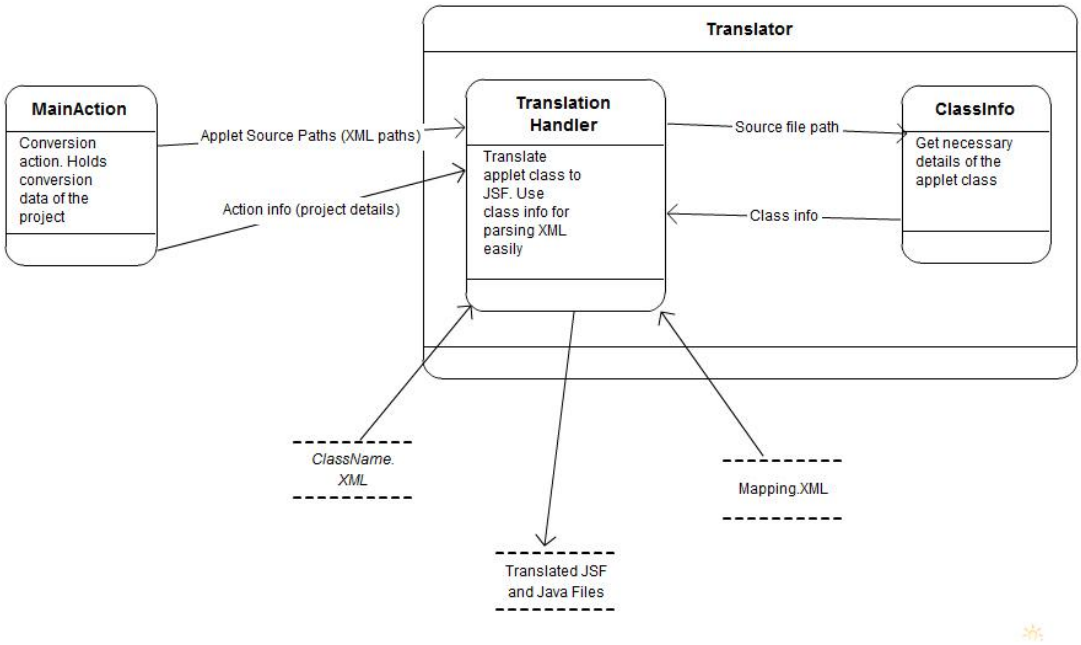
5.2.4. Translator Component

5.2.4.1. Processing Narrative for Translator Component

Translator component uses output of JavaML component – that is ClassName.xml, related ClassInfo object instances and Mapping.xml file in order to generate output files. In this design, we will use Java Reflection API and ClassInfo objects in our design. More information can be found about Java Reflection API at Section 8.6.

This component is going to be instantiated at MainAction class and be triggered from there.

5.2.4.2. Interface Description of Translator Component



5.2.4.3. Processing Detail of Translator Component

Translator component contains two classes: TranslationHandler and ClassInfo.

5.2.4.3.1. TranslationHandler Class

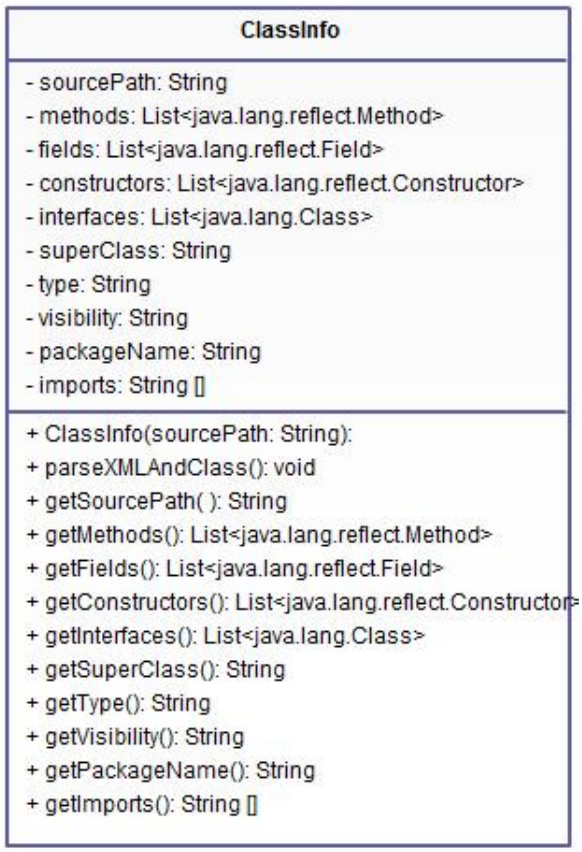
5.2.4.3.1.1. Attributes

- private ArrayList<ClassInfo> listClassInfo: Keeps ClassInfo object instances.
- private MainAction action: Reference for MainAction object instance.
- private ArrayList<String> appletSourcePaths: Keeps paths of java class files which extends JApplet class.
- private org.apache.log4j.Logger logger: Singleton object reference for only one Logger object instance.

5.2.4.3.1.2. Methods

- public TranslationHandler (ArrayList<String> appletSourcePaths, MainAction act): Constructor for TranslationHandler class.
- public void composeMemoryStructure (): Generates ClassInfo objects in memory.
- private void findEquivalentJSF (String filename): Uses Mapping.xml to compare and generate output JSF tags.
- private void write2JSF (String filename): Output stream writer for output JSF files.
- public void findEquivalences(): Interface for MainAction class. Calls findEquivalentJSF and write2JSF.

5.2.4.3.2. ClassInfo Class



5.2.4.3.2.1. Attributes

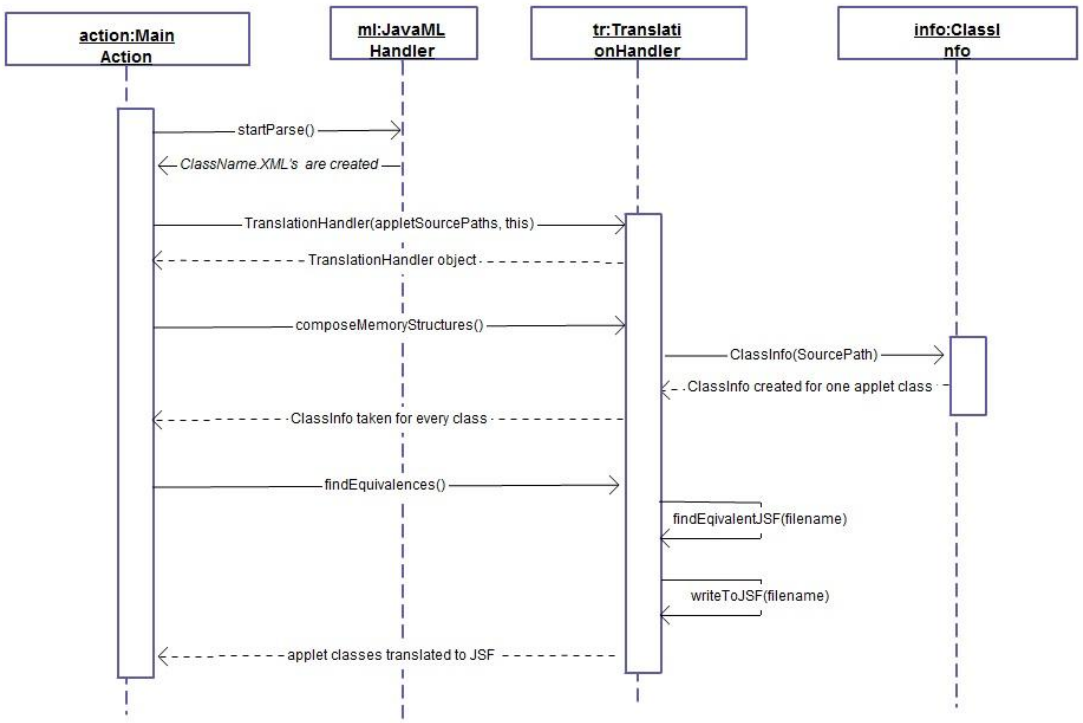
- private String sourcePath: Path of the source file which extends JApplet.
- private List<Java.lang.reflect.Method> methods: Method list of the source file which extends JApplet.
- private List< Java.lang.reflect.Field> fields: Field list of the source file which extends JApplet.
- private List< Java.lang.reflect.Constructor> constructors: Defined constructors on the source file which extends JApplet.
- private List< Java.lang.Class> interfaces: List of the interfaces that class implements.
- private String superClass: Name of the super class.
- private String type: Type of the class:Abstract…
- private String visibility: Accessibility of the class: public, private
- private String packageName: Package of the class.

- private String[] imports: Imported packages of the java source.

5.2.4.3.2.2. Methods

- public ClassInfo (String sourcePath): Constructor for the class ClassInfo.
- public void parseXMLAndClass(): Parses the output of the JavaML and class with Java Reflection API.
- public String getSourcePath(): Getter method for the field “sourcePath”.
- public List<Java.lang.reflect.Method> getMethods(): Getter method for the field “methods”.
- public List<Java.lang.reflect.Field> getFields(): Getter method for the field “fields”.
- public List< Java.lang.reflect.Constructor> getConstructors(): Getter method for the field “constructor”.
- public List< Java.lang.Class> getInterfaces(): Getter method for the field “interfaces”.
- public String getSuperClass(): Getter method for the field “superClass”.
- public String getType(): Getter method for the field “type”.
- public String getVisibility(): Getter method for the field “visibility”.
- public String getPackageName(): Getter method for the field “packageName”.
- public String[] getImports(): Getter method for imports field.

5.2.4.4. Dynamic Behavior of Translator Component



5.2.5. Log Component

5.2.5.1. Processing Narrative for Log Component

Log component is responsible from only logging. There will be only one logger while the system is running. Logger Component will be accessible from all the other components to log appropriate information. Logger will be configured to log different places for each project. It will log into a file named projectName.log and also, it will produce logs on the screen.

Apache log4j library will be used while logging.

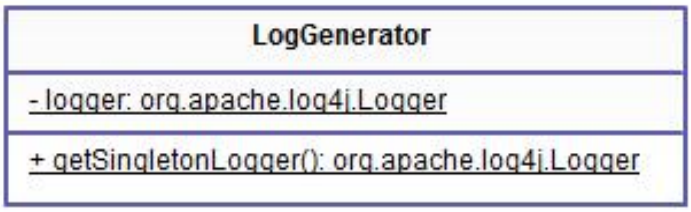
5.2.5.2. Interface Description of Log Component

Log component is not a complex component and there is no complex data flow over the component. Data flow of the Log component described in other components data flow diagrams.

5.2.5.3. Processing Detail of Log Component

Log component consists of only one class: LogGenerator.

5.2.5.3.1. LogGenerator Class



5.2.5.3.1.1. Attributes

- private static org.apache.log4j.Logger logger: Singleton logger object.

5.2.5.3.1.2. Methods

- public org.apache.log4j.Logger getSingletonLogger(): Getter method for the "logger" field.

5.2.5.4. Dynamic Behavior of Log Component

All the other components send log information after all the operations by done the component. So there is no need to show the sequence of the flow in this section. Any component can log any time.

6. User Interface Design

6.1. Overview of User Interface

In this project, there will be no complex user interfaces, because this tool will be a single developer tool. So, we designed our interfaces in that manner.

Our designed user interfaces provide some facilities to users. When the user starts to use the system, main window stated in part 6.2 welcomes the user.

Capabilities of the main window are to:

- Operate over the existing projects

  - Remove an existing project
  - Select an existing project
  - Deselect an existing project
  - Start conversion of selected projects
  - View log information of selected projects
- Add new project

All those operations mentioned above are the directly user related operations. Actions of the user will be converted to system functions related to that action.

This project does not contain a main window only. According to user actions, some other pre-defined user interfaces will appear on the window. When the main window is opened and the user wants to add a new project, another user interface will appear which is stated in 6.2.

Capabilities of the “Project” window are to:

- Select a project folder
- Select a destination folder
- Confirm project details

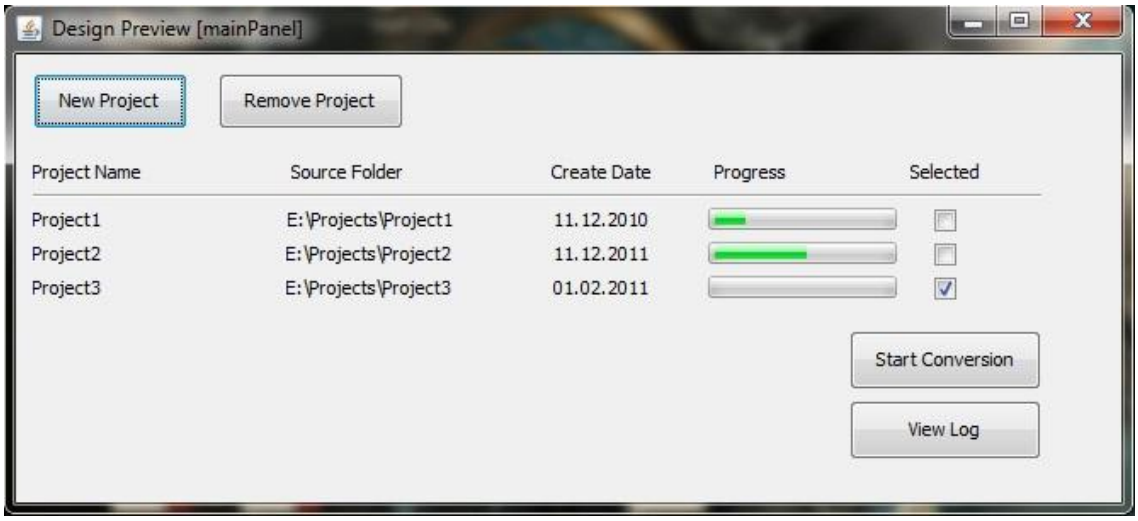
Another window that can be seen via main window is log information window.

Capabilities of the “Log” window are;

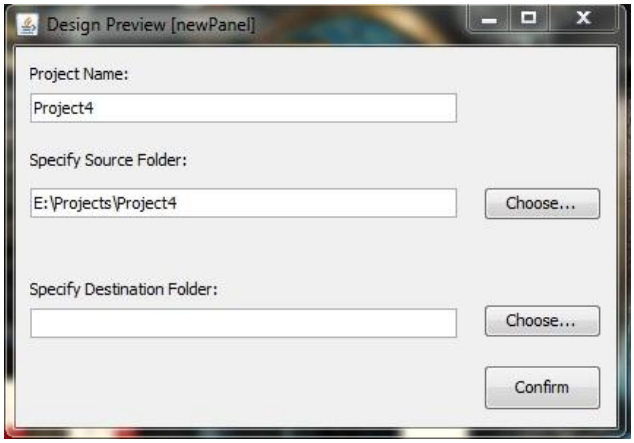
- Display real time information about the project being converted.

All the information stated above is directly from the users perspective. In addition to those, there are some other internal operations that invoke the user interface. According to the conversion process user interface shows the percentage of the conversion.

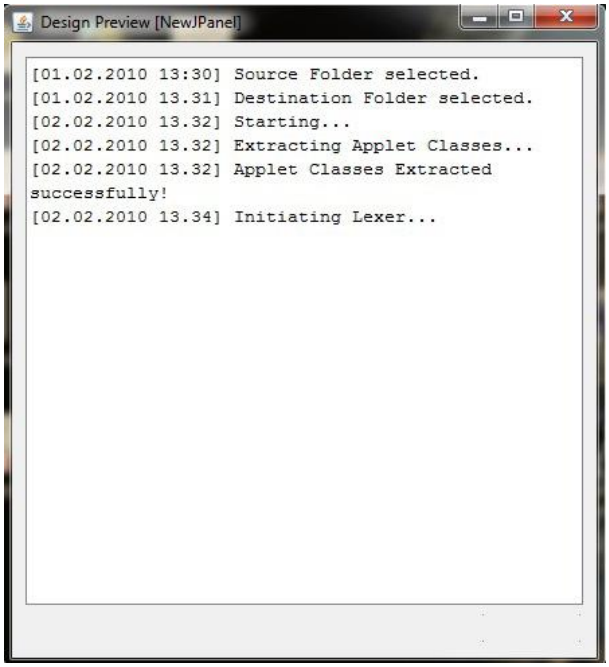
6.2. Interface Screens



Main Window



Project Window



Log Window

6.3. Screen Objects and Actions

This part includes objects on the screen interfaces and the actions linked to that objects.

6.3.1 Screen Objects

For the main window:

- Panel: Panel is to group other objects in the window. There will be only one panel to group objects.
- Buttons
  - buttonNewProject: This button is to add a new project to convert.
  - buttonRemoveProject: This button is to remove an existing project.
  - buttonStartConversion: This button is to start conversion operation of selected projects.
  - buttonViewLog: This button is to view log information.

There are actions linked to those buttons. All the actions are stated below in section 6.3.2.

- Labels
    - labelProjectName: Label for the project name.
    - labelSourceFolder: Label for the source project folder.
    - labelCreateDate: Label for the creation date of the project.
    - labelProgressBar: Header label for the progress bars.
    - labelSelected: Header labels for the checkboxes defined below.

Those labels are the headers. According to the existing projects, there will be some other labels related with each project under above header labels.

- Progress Bars: Progress bars are to show the status of the conversion operation.
- Check Boxes: Checkboxes are to select or deselect a project to operate on it.
- Separator: Separates the headers from the project information.

Progress bars and Check boxes can be more than one according to existing projects. Also there are some actions linked to those checkboxes.

For the project window:

- Panel: Panel is to group other operations on the window.
- Buttons
    - buttonChooseSource: This button is to opens a standard dialog window to select the source folder.
    - buttonChooseDestination: This button is to opens a standard dialog window to select the destination folder.
    - buttonConfirmProject: This button is to confirm project details stated.
- Text Fields
    - textFieldProjectName: This text field is for to specify project name. It is a disabled field and automatically generated with the selected source directory.
    - textFieldSourceDirectory: This text field is to specify source directory. It is an enabled component and also automatically generated with the selection of source directory.

- textFieldDestinationDirectory: This text field is to specify destination directory. It is an enabled component and also automatically generated with the selection of destination directory.

There are actions defined on the objects. Those actions are described in section 6.3.2.

- Labels
  - labelProjectName: Label for the textFieldProjectName .
  - labelSourceDirectory: Label for the textFieldSourceDirectory.
  - labelDestinationDirectory: Label for the textFieldDestinationDirectory.

For the log :

- Panel: Panel is to group another objects together.
- TextArea: TextArea component is to show log information about the process.

6.3.2 Screen Actions and Relations

Defined actions for the interfaces stated below.

For the “Main” window:

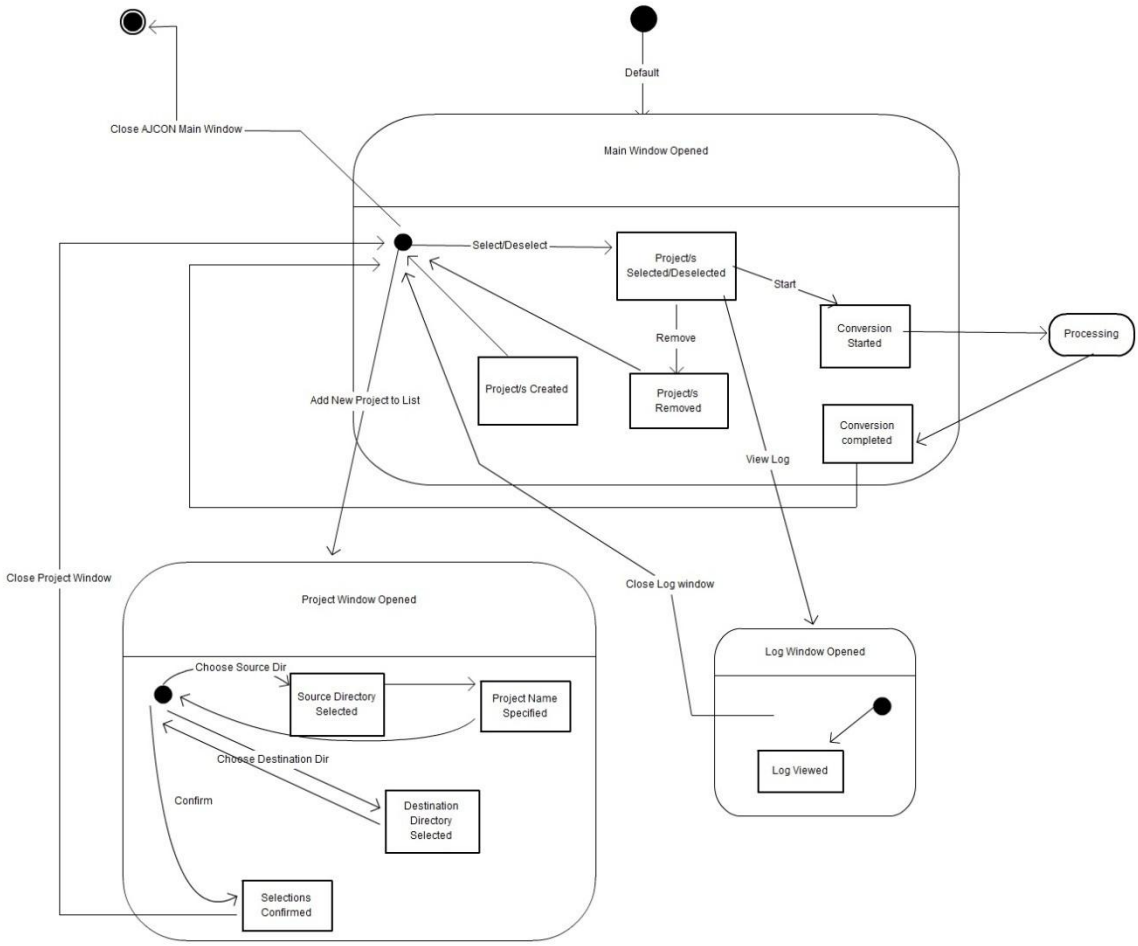
- Actions of Buttons
  - buttonNewProjectClickedAction: Action performed when the buttonNewProject button clicked on the main window. Opens “Project” window stated in section 6.2.
  - buttonRemoveProjectClickedAction: Action performed when the buttonRemoveProject button clicked on the main window. Removes the selected project from the list of existing projects.
  - buttonStartConversionClickedAction: Action performed when the buttonStartConversion button clicked on the main window. Starts the main operation conversion of the selected projects.
  - buttonViewLogClickedAction: Action performed when the buttonViewLog button clicked. Opens “log” window, which is stated in section 6.2.

- checkboxStateChangedAction: Selects or deselects a project from the existing projects.

For the “Project” Window:

- Actions of Buttons
  - buttonChooseSourceClickedAction: Action performed when the buttonChooseSource button clicked in the “Project” window. Action opens a dialog window that contains the system directories to choose source folder.
  - buttonChooseDestinationClickedAction: Action performed when the buttonChooseDestination button clicked in the “Project” widow. Action opens a dialog window which contains the system directories to choose destination folder.
  - buttonConfirmProjectClickedAction: Action performed when the buttonConfirmProject button clicked in the “Project” window. Action closes the current “Project Window” and adds the new project to list of existing projects in the main window.
  - textFieldSourceDirectoryStateChangedAction: Action performed when the state of the textFieldSourceDirectory changed. State of the textFieldSourceDirectory object changes with if any user enters a text. With the change of the state of textFieldSourceDirectory textFieldProjectName field will be automatically generated.

Below, there is a state diagram which summarizes user interface states with respect to user actions. This diagram is similar to diagram drawn in section 2, but here is only related to user and only UI states are shown.



7. Detailed Design

In this section, system architecture of AJCON which is explained in section 5 will be covered lastly and most important points about design will be detailed. While doing this, main components and their classes and most important functions will be handled. (See Section 5.2 Package Diagram)

7.1. UI Component

(See Section 5.2.1 for UI Component related diagrams.)

Classification: Package

Definition: Purpose of this package is to hold classes that is used in interactions with users logically together.

Responsibilities: This class is responsible from all interactions with user. Responsibility of this package will be explained in detail by responsibilities of ApplicationManager, MainWindow, ProjectWindow, LogWindow and MainAction classes.

Uses/Interactions: This component uses interface of the Log component (i,e. subroutine *getSingletonLogger()*) for the purpose of logging. Since Log component import the *org.apache.log4j* package, UI Component also uses it. Component provides an interface to the user.

Processing: Component initiates the system and gets prepared everything for the user to let him/her pick a project and start conversion. Every user-based event processed and according to these events, all actions are handled in order to make system ready for a conversion.

Interface/Exports: The set of services provided by this component is specified by its classes, ApplicationManager, MainWindow, ProjectWindow, LogWindow and MainAction, and their subroutines.

7.1.1. ApplicationManager Class

Classificiation: Class(See class diagram at Section 5.2.1.3.1)

Definition: Purpose of this class is to initiate run of the project.

Responsibilities: This class has the main function of the project. When main function is called by the system, it is responsible from creating main window and initiating the system.

Constraints: There is no time, memory, processor limitiation. To make the class active, user need to run the project.

Uses/Interactions: This class only interacts with MainWindow class. It creates main window of the project and calls its initComponents() function.

Processing: ApplicationManager class simply instantiates a MainWindow object instance and calls its initComponents() function. After calling this method, main thread ends but since there is a MainWindow object instantiation that runs on a JFrame thread, that

windows still stays live for user interactions. Because of this reasong, when main thread dies, system does not halt and keeps working.

Interface/Exports: This class does not provide interface or export anything to any other component.

7.1.1.1 main(args: String[]):void

Classification: Function

Definition: Purpose of this function is to initiate the system. It is called automatically when project is run.

Constraints: There is no time, memory or process limitation.

Uses/Interactions: This function uses MainWindow class interface in order to call its initComponents() method.

Processing: When system is run, main function is called immediately and everything starts from this function. After creating a MainWindow instance, main thread dies but the other thread continues on.

7.1.2. MainWindow Class

Classification: Class(See class diagram at Section 5.2.1.3.2)

Definition: Purpose of this class is to hold everything related to main window and its actions.

Responsibilities: Its behavior in AJCON system is initiated by ApplicationManager class. Class holds all graphical components in main window (labels, buttons, panel, separator, progress bar, check boxes etc). It also handles all actions included in main window, which are new project, remove project, start conversion, view log and check box state change. User can start and do many conversions from MainWindow with the help of multiple threads running on different conversions. In addition to these, this class is responsible from logging everything happened to the user.

Constraints: There is no time, memory, processor limitation. To make the class active, program should be run and main function should start properly.

Uses/Interactions: This class extends javax.swing.JFrame class and uses javax.swing.components for GUI. This class also uses the interface provided by the Log component. Subroutine of Log component *getSingletonLogger()* is used to set class local private field *logger* to static singleton logger object. Its interaction with Log components leads to the creation of a list of log windows and main actions for every single thread that is created during multiple conversions.

Processing: This class simply stores the information every project to be converted. Once a user does an action, its related action handler is triggered. When user clicks New Project button, buttonNewProjectClickedAction is called and a ProjectWindow class object is instatiated. When user clicks RemoveProject button, buttonRemoveProjectClickedAction is called and every check box near to projects is controlled and clicked ones are removed. When user clicks Start Conversion, every check box near to projects is controlled and every clicked project is started to be converted on a different thread. When user clicks View Log, every check box near to projects in controlled and log info of every project is shown to user. All LogWindows objects are also kept as an instance variable, logWindowList:ArrayList<LogWindow>.

Interface/Exports: This provides interface to ProjectWindow class. Every ProjectWindow instance keeps the reference of MainWindow in order to save changes at main window and go back there properly. For each ProjectWindow instance, there will be only one reference to main window in total.

7.1.2.1. initComponents() : void

Classification: Function

Definition: Purpose of this function is to init graphical components and other instance variables in main window. It creates labels, buttons, checkboxes etc. This function also initiates mainActionList, logWindowList and configures logger object.

Constraints: There is no time, memory or process limitation.

Uses/Interactions: This function does not use any interface of any other component.

Processing: When main function of ApplicationManager is executed, an instance of MainWindow is created and initComponents() function is called. After its execution, components are initiated.

7.1.3. ProjectWindow Class

Classification: Class(See class diagram at Section 5.2.1.3.4)

Definition: Purpose of this class is to hold all the information about the window that is created when adding a new project to AJCON system. In other words, it is called “New Project Window”.

Responsibilities: Its behavior in AJCON system is initiated by MainWindow class. Class holds all graphical components (text fields, labels, buttons, panel etc) and a reference to main window. It is basically responsible from adding a new project with all its information to the system properly. After adding a project, it should save changes at main window and return back there properly.

Constraints: There is no time, memory, processor limitation. To make the class active, in main window, user should click New Project button and buttonNewProjectClickedAction function should be triggered.

Uses/Interactions: This class uses the interface provided by MainWindow class. In order to save changes to main window, after creating a new project, main window should be updated.

Processing: This class simply stores the information of a new project to be converted. When user clicks “New Project” button from main window, a ProjectWindow instance is instantiated and shown to the user. User can specify the source and destination paths of projects, enter a new project name and finally click “Confirm Project” button. If user click on the source path text field, buttonChooseSourceClickedAction is triggered and a new browse window is created in order to let user specify the folder. It is same for destination folder selection. After specifying every information, user can click “Confirm Project” button and buttonConfirmProjectClickedAction is triggered.

7.1.3.1. initComponents() : void

Classification: Function

Definition: Purpose of this function is to init graphical components and other instance variables in project window. It creates text fields, labels, buttons etc.

Constraints: There is no time, memory or process limitation.

Uses/Interactions: This function does not use any interface of any other component.

Processing: When user clicks “New Project” button from main window, an instance of ProjectWindow is created and initComponents() function is called in order to initiate all components inside this window.

7.1.4. LogWindow Class

Classification: Class(See class diagram at Section 5.2.1.3.4)

Definition: Purpose of this class is to hold log information that is taken from Logger object and show it to the user.

Responsibilities: Its responsibility includes showing log information of a single project to the user. It also holds a reference to current project’s MainAction object instance.

Constraints: There is no time, memory, processor limitation. To make the class active, user should click “View Log” from main window.

Uses/Interactions: This class uses the interface provided by MainAction class since it should log every action to the user. Other than that, this LogWindow object instance is kept inside an ArrayList of LogWindow in main window, which is logWindowList:ArrayList<LogWindow>.

Processing: When user clicks “Start Conversion” from main window, for every project, there is a different LogWindow instance kept inside logWindowList array list. With the help of MainAction reference inside LogWindow instance, every action is logged to the user to logInformation:javax.swing.JTextArea text field. When user clicks, “View Log” from main window, user is able to see this log window.

Interface/Exports: This class does not provide any interface to any other component.

7.1.4.1. initComponents() : void

Classification: Function

Definition: Purpose of this function is to initialize graphical components and other instance variables in log window. It creates panel and text field, sets project name and reference to MainAction object.

Constraints: There is no time, memory or process limitation.

Uses/Interactions: This function does not use any interface of any other component.

Processing: When user clicks "Start Conversion" button from main window, an instance of LogWindow is created and initComponents() function is called in order to initiate it.

7.1.5. MainAction Class

Classification: Class (See class diagram at Section 5.2.1.3.5)

Definition: Purpose of this class hold all the information about the conversion process of a project.

Responsibilities: Its behaviour in AJCON system is initiated by UI Components' buttonStartConversionClickedAction method. Class holds the necessary information about the conversion process, such as log information, percentage of conversion. Also, this class is responsible from the configuration of logger object.

Constraints: There is no time, memory, processor limitation. To make the class active, *run()* method should be called.

Uses/Interactions: This class uses the interface provided by the java.lang.Thread and interface Serializable, because at a time more than one conversion operation should be handled and all the operations should be thread safe. This class also uses the interface provided by the Log component. Subroutine of Log component *getSingletonLogger()* is used to set class local private field *logger* to static singleton logger object.

Processing: This class simply stores the information comes from the logger object. Once the logger is configured for this MainAction class, it starts to store the log information in the private field *logBuffer* and stores the percentage of the conversion operation in the *progress* field. Also, each *checkUpdates* function call refreshes the UI and provides real time log information when new log information arrives.

Interface/Exports: This class provides the private fields *logBuffer* and *progress* to UI Component to inform the user about the conversion process. For each project existing, there will be a unique MainAction class, and they will carry the specific information about the projects to other components.

7.1.5.1 checkUpdates( ): void

Classification: Function

Definition: Purpose of this function is to refresh user interface when new log information arrives or progress of the conversion changes.

Constraints: There is no time, memory or process limitation.

Uses/Interactions: This function does not use any interface of any other component.

Processing: When any of the components logs information or changes the status of the progress it will immediately call that function via the object. With the execution of the function, opened windows will be refreshed instantly.

7.1.5.2 run( ): void

Classification: Function

Definition: Purpose of this function is to start the execution of a Thread. Since the class MainAction extends java.lang.Thread, execution of the thread will be mention with the *run()* function.

Constraints: There is no time, memory or process limitation. To call the function, there should be at least one project selected and Start Conversion button should be clicked.

Uses/Interactions: This function overrides the function *run()* which is defined in the library *java.lang.Thread.*

Processing: When user selects projects and clicks the Start Conversion button, MainAction objects existing in the system about the selected projects will call the function *run()* to start the process. With the execution of the *run()* function, several threads can be run at the same time.

7.2. Applet Extractor Component

(See Section 5.2.2 for Applet Extractor Component related diagrams.)

Classification: Package

Definition: Purpose of this package is to hold applet extraction related classes logically together.

Responsibilities: Responsibility of the package will be defined by the responsibility of the class: ExtractionHandler.

Uses/Interactions: This component uses interface of the Log component (i,e. subroutine *getSingletonLogger()*) for the purpose of logging. Since Log component import the *org.apache.log4j* package, Applet Extractor Component also uses it. Component also provides an interface to the UI component.

Processing: Component processes the input source path to find classes that extends Applet. Component will search every subfolders and processes .java files under input source folder.

Interface/Exports: The set of services provided by this component is specified by its class *ExtractionHandler* and its subroutines.

7.2.1. ExtractionHandler Class

Classification: Class (See class diagram at Section 5.2.2.3.1)

Definition: Purpose of this class is to find classes which extend Applet under the source folder.

Responsibilities: Its behaviour in AJCON system is initiated by MainAction class's *run()* method. After that, it is responsible for finding applet class source paths.

Constraints: In the input source folder path, there will be at least one .java file which extends java.swing.JApplet class. There is no time, memory, processor limitation.

Uses/Interactions: This class uses the interface provided by the Log component. Subroutine of Log component *getSingletonLogger()* is used to set class local private field *logger* to static singleton logger object. This class also uses static unique MainAction class object for one project and its *checkUpdates()* method. By this way, when logging the process detail, it can append its own log info to the MainActin object's *logBuffer.* Meanwhile, it provides its functions to UI component because in MainAction class, the class object instances are created and class functions are used to extract applet classes.

Processing: Gathers the input source folder path from the UI component. Accesses to the input source folder path and finds .java files. Then; it defines the files that will be sending to the JavaML component. In this process, this component opens all .java files and picks the classes that extend java.swing.JApplet class. By this way, it is aimed to find all classes that include a graphical user interface component. In further stages, only these extracted classes will be processed and converted to JSF components.

Interface/Exports: This class provides extracted classes, which extends java.swing.JApplet, to the JavaML component as data. These extracted class source paths are hold in private field *appletSourcePaths* which is list of strings and provided with *getAppletSourcePaths()* subroutine. Before that, *parseAndExtractApplet()* function is provided to UI component such that it can extract these applet classes.

7.2.1.1 parseAndExtractApplet( ): void

Classification: Function

Definition: Purpose of this function is to find all .java files and classes that extends java.swing.JApplet class in order to find used graphical user interface components in the input Applet project.

Constraints: In the input Applet project, there must be at least one .java file and a class that extends java.swing.JApplet class. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: It gathers the input source path from the UI component. To do this, it uses the MainAction field. It accesses the source folder path and finds .java files. It searches deeps of the folder path and opens all the files. Then, it sets the related field *appletSourcePaths* field according to the results. While doing this, it also uses the *logger* field to log some information to the UI component. If any exception or error occured, it is also logged without violating remaining process of the system.

7.3. JavaML Component

(See Section 5.2.3 for JavaML Component related diagrams.)

Classification: Package

Definition: Purpose of this package is to hold JavaML usage related classes logically together.

Responsibilities: Responsibility of the package will be defined by the responsibility of the class: JavaMLHandler.

Uses/Interactions: This component uses interface of the Log component (i,e. subroutine *getSingletonLogger()*) for the purpose of logging. Since Log component import the *org.apache.log4j* package, JavaML Component also uses it. Component also provides an interface to the UI component.

Processing: Component processes the applet source paths taken from AppletExtractor Component by using JavaML. Detailed process will be explained in JavaMLHandler class processing.

Interface/Exports: The set of services provided by this component is specified by its class *JavaMLHandler* and its subroutines.

7.3.1. JavaMLHandler Class

Classification: Class (See class diagram at Section 5.2.3.3.1)

Definition: Purpose of this class is to execute JavaML command to output JavaML XML.

Responsibilities: Its behaviour in AJCON system is initiated by MainAction class's *run()* method. After Applect Extractor Component's process is finished, it is time for JavaML component. It is responsible for parsing applet source paths and extract detailed parse XML which is shown in data section 4.

Constraints: There will be at least one applet class ,which extends java.swing.Japplet, extracted from Applet Extractor Component. Moreover, there should be JavaML grammar files such as .dtd and .xsd files (explained in section 4). These data files are necessary for JavaML to parse java source according to these rules. Except them, there is no time, memory, processor limitation.

Uses/Interactions: This class uses the interface provided by the Log component. Subroutine of Log component *getSingletonLogger()* is used to set class local private field *logger* to static singleton logger object. This class also uses static unique MainAction class object for one project and its *checkUpdates()* method. By this way, when logging the process detail, it can append its own log info to the MainActin object's *logBuffer.* Meanwhile, it provides its functions to UI component because in MainAction class, the class object instances are created and class functions are used to get JavaML output.

Processing: After applet source paths are extracted, these are given to JavaML class object in its constructor. After that, firstly, in MainAction class, *startParse()* function is called. And in this function, firstly *getEnvironmentVariables()* subroutine is called. Environment variables are taken for JavaML over Jikes compiler execution. Then, JavaML

command is executed and JavaML already parses java source files (applet classes) and give results to its XML files which are on the same directory as applet classes.

Interface/Exports: This class provides JavaML output(s), to Translator component as data. These JavaML output(s) are hold in XML files. Before that, , *startParse()* function is provided to UI component such that it can start JavaML parser execution.

7.3.1.1. startParse(): void

Classification: Function

Definition: Purpose of this function is to start JavaML parse operation with the command

jikes.exe +ux -classpath JDK_PATH\jre\lib\rt.jar ClassName1.java ClassName2.java ...

Constraints: In order to execute JavaML over Jikes compiler, it is necessary to get environment variables. Therefore, firstly they are obtained by *getEnvironmentVariables()* call in this function. Moreover, it is assumed that JavaML over Jikes compiler will work correctly. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: Since this class object is constructed with applet class paths, they are ready for processing. Firstly, with *getEnvironmentVariables()* call, environment variables are taken as a string and it is parsed according to JavaML execution. Then, command will be constructed with applet source paths and executed with the help of *java.lang.Runtime* and *java.lang.Process* objects. In the bakcground, JavaML already parses source files and creates parse XML files. While doing this, it also uses the logger field to log some information to the UI component. If any exception or error occured, it is also logged without violating remaining process of the system.

7.3.1.2. getEnvironmentVariables(): String

Classification: Function

Definition: Purpose of this function is to get environment variables such as JDK path from the user computer.

Constraints: it is assumed that necessary environment variables are set in user's computer environment. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: It will read from system files to get environment variables. After that, it will write their paths to string. While doing this, it also uses the logger field to log some information to the UI component. If any exception or error occured, it is also logged without violating remaining process of the system

7.4. Translator Component

(See Section 5.2.4 for Translator Component related diagrams.)

Classification: Package

Definition: Purpose of this package is to hold translation related classes logically together.

Responsibilities: Responsibility of the package will be defined by the responsibility of the class: TranslationHandler.

Uses/Interactions: This component uses interface of the Log component (i,e. subroutine *getSingletonLogger()*) for the purpose of logging. Since Log component import the *org.apache.log4j* package, Translation Component also uses it. Component also provides an interface to the UI component.

Processing: This component uses JavaML output(s). After process these files, it will give meaning to details and with the help of mapping XML, and it will find eqivalent JSF components/code fragments for applet classes. And then write them to destination paths with keeping other unprocessed source files. Detailed process will be explained in TranslationHandler class processing.

Interface/Exports: The set of services provided by this component is specified by its class *TranslationHandler* and its subroutines.

7.4.1. TranslationHandler Class

Classification: Class (See class diagram at Section 5.2.4.3.1)

Definition: Purpose of this class is to translate applet classes to JSF ones.

Responsibilities: Its behaviour in AJCON system is initiated by MainAction class's *run()* method. After JavaML Component's process is finished, it is time for Translation component. It is responsible for finally translate applet sources to JSF files (.xhtml and .java files) which run in the manner of applet classes. However, since there can be no %100 conversion, some applet source paths will be kept as well while some JSF equivalances are created.

Constraints: There should be JavaML outputs for applet classes. Furthermore, mapping XML should be prepared before according to format given in section 4. Except them, there is no time, memory, processor limitation.

Uses/Interactions: This class uses the interface provided by the Log component. Subroutine of Log component *getSingletonLogger()* is used to set class local private field *logger* to static singleton logger object. This class also uses static unique MainAction class object for one project and its *checkUpdates()* method. By this way, when logging the process detail, it can append its own log info to the MainActin object's *logBuffer.* Meanwhile, it provides its functions to UI component because in MainAction class, the class object instances are created and class functions are used to be able to perform translation.

Processing: After JavaML output(s) are created, in MainAction class, TranslationHandler object is created and applet source paths also given to it. Then, firstly, *composeMemoryStructure()* function is called. Java Reflection API an JavaML output's parse results are merged to compose structures in memory. After that, for each applet file, mapping/translation is done with the help of mapping XML and logic of the mapping algorithms.

Interface/Exports: This class provides final JSF and java files in the destination paths. Before they are created, *composeMemoryStructure() and findEquivalences()* functions are provided to UI component.

7.4.1.1. composeMemoryStructure(): void

Classification: Function

Definition: Purpose of this function is to compose memory structures to make translation easy.

Constraints: JavaML output XML's for each applet class file should be generated before this function call. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: Firstly, it will create ClassInfo instances for each applet source (give *appletSourcePaths* to constructors). Then for these ClassInfo objects, *parseXMLAndClass()* calls are done separately. By this function, ClassInfo object's are fulfilled with memory structures for each applet class. With the help of getter functions of ClassInfo, these structures can be accessed. While doing this, it also uses the logger field to log some information to the UI component. If any exception or error occured, it is also logged without violating remaining process of the system.

7.4.1.2. findEquivalences(): void

Classification: Function

Definition: Purpose of this function is to find equivalence JSF's for applet classes.

Constraints: JavaML output XML's for each applet class file should be generated before this function call. Furthermore, mapping XML should be prepared before according to format given in section 4. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: After memory structures are created (ClassInfo object list) with *composeMemoryStructure()* function, this function is called in the MainAction class.

Within that function, for each applet class, *findEquivalentJSF(String fileName)* and *write2JSF(String fileName)* functions are called by giving applet source paths indicating which source is mapped. *findEquivalentJSF* function uses ClassInfo object for this applet class. Using Mapping.xml and mapping algorithms, it will generate JSF tags and necessary java code fragments. After that stage, *write2JSF* function write these partially equivalent JSF related codes to files. Destination paths are taken from its own MainAction class. While doing this, it also uses the logger field to log some information to the UI component. If any exception or error occured, it is also logged without violating remaining process of the system.

7.4.2. ClassInfo Class

Classification: Class (See class diagram at Section 5.2.4.3.2)

Definition: Purpose of this class is to hold java class details in memory as structures.

Responsibilities: Its behaviour in AJCON system is initiated by TranslationHandler class's *composeMemoryStructure()* method. It is responsible for getting details of applet source files from JavaML outputs and Java Reflection API, and putting them in memory in more organized manner for easy access.

Constraints: There should be JavaML outputs for applet classes. There is no time, memory, processor limitation.

Uses/Interactions: This class uses Java Reflection API and JavaML outputs. Therefore, only interaction can be proposed for JavaML component, but after its process.

Processing: With the call of *composeMemoryStructure()* of TranslationHandler class in MainAction class, ClassInfo objects are created for each applet source. Then, *parseXMLAndClass()* function is called within *composeMemoryStructure()*. With this function, all necessary details of one java source file are obtained and set to local fields *methods, fields, constructors, interfaces, superClass, type, visibility, packageName, imports.*

Interface/Exports: This class provides *methods, fields, constructors, interfaces, superClass, type, visibility, packageName, imports* to TranslationHandler class. TranslationHandler class can acces them through getter's of them provided by this class.

7.4.2.1. parseXMLAndClass(): void

Classification: Function

Definition: Purpose of this function is to parse JavaML output *ClassName.XML* and parse source file with Java Reflection API to get main fiels like method names, interface name etc. Then, merge these information to create java class structures.

Constraints: JavaML output XML's for specified applet class file should be generated before this function call. There is no time, memory or process limitation.

Uses/Interactions: This function uses the interface provided by the log component.

Processing: Firstly, Java Reflection API is used. It will give the method names, fields declared in source class, implemented interfaces etc. main details. Then these names are used to parse JavaML output XML parsing and get more detailed information about these names. Details are set to local fields *methods, fields, constructors, interfaces, superClass, type, visibility, packageName, imports* whenever detail is obtained.

7.5. Log Component

(See Section 5.2.5 for Log Component related diagrams.)

Classification: Package

Definition: Purpose of this package is to hold Logging related classes logically together.

Responsibilities: Responsibility of the package will be defined by the responsibility of the class: *LogGenerator*.

Uses/Interactions: Log component imports the *org.apache.log4j* package for logging utiliy.

Processing: Component appends the log information sent by the other components.

Interface/Exports: The set of services provided by this component is specified by its class *LogGenerator* and its subroutines.

7.5.1. LogGenerator Class

Classification: Class (See class diagram at Section 5.2.5.3.1)

Definition: Purpose of this class is to log information sent by the other components to a related field.

Responsibilities: Its behaviour in AJCON system is initiated by MainAction class's *run()* method. After calling MainAction class's *run()* method, components can send log information to logger object at any time. Acquired log information will be appended to a field.

Constraints: There is no time, memory, processor limitation for LogGenerator class. Only constraint is that there will be only one LogGenerator object in the whole system (singleton object).

Uses/Interactions: This class does not use any interface provided by other components. Meanwhile, it provides its functions to other components because in MainAction class, there exists a *logBuffer* to log and all the other components will send information to log.

Processing: With the start of the system, singleton logger object will be created and this class will only provide the created singleton object. Other logging functionalities will be made via this logger object. Logger object will be an instance of *org.apache.log4.Loggerj* class. All the functions provided by the *org.apache.log4j.Logger* class.

Interface/Exports: This class provides an instance of *org.apache.log4j.Logger* to other components. Other components access to this component with the service *getSingletonLogger()* provided by Log Component.

7.5.1.1. getSingletonLogger(): org.apache.log4j.Logger

Classification: Function

Definition: Purpose of this function is to return the reference of the logger object of the system. If there exists no logger object in the system, it will create one, and then it will return the created logger object references.

Constraints: There exists no constraint.

Uses/Interactions: This function uses the interface provided by *org.apahce.log4j.Logger*.

Processing: It will look for the static field logger in the class. If there exists a logger object, it will return with the reference of the object. If not, it will create an instance of the *org.apache.log4j.Logger* and then returns with the reference of the newly created object. If any exception or error occurred, it is also logged without violating remaining process of the system.

8. Libraries and Tools

8.1. JavaML [3]

The Java Markup Language (JavaML) [4] builds a bridge between Java and XML. It generates a self-describing representation of Java source code. Its nested representation in XML-based syntax directly reflects the structure of software artifact. It has many advantages because since XML is a text-based representation, it still keeps the classical source representation. XML files are also very easy to parse with external Java parsers (Apache Xerxes DOM, SAX etc.)

JavaML is defined by document type definition (DTD) in [4]. In JavaML, concepts such as methods, superclasses, message sends and literal numbers are all directly represented in the elements and attributes of the document contents. The representation reflects the structure of the programming language in the nesting of the elements.

In our project, we will use JavaML in order to parse Java source code and generate corresponding XML file. It will enable us to see hierarchical structure of Java classes and create mapping file.

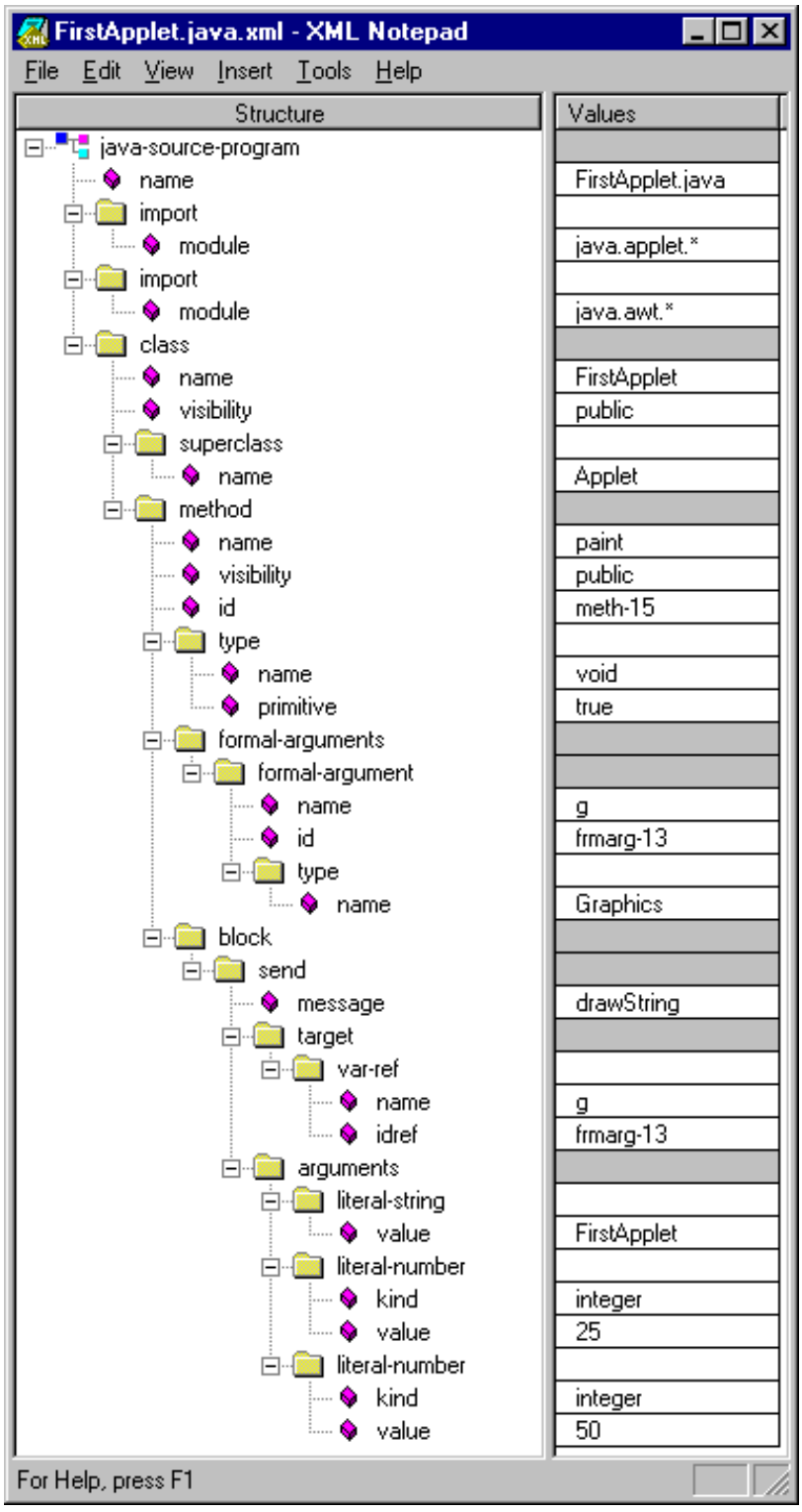
In order to understand the concept, lets look at the sample Java code.

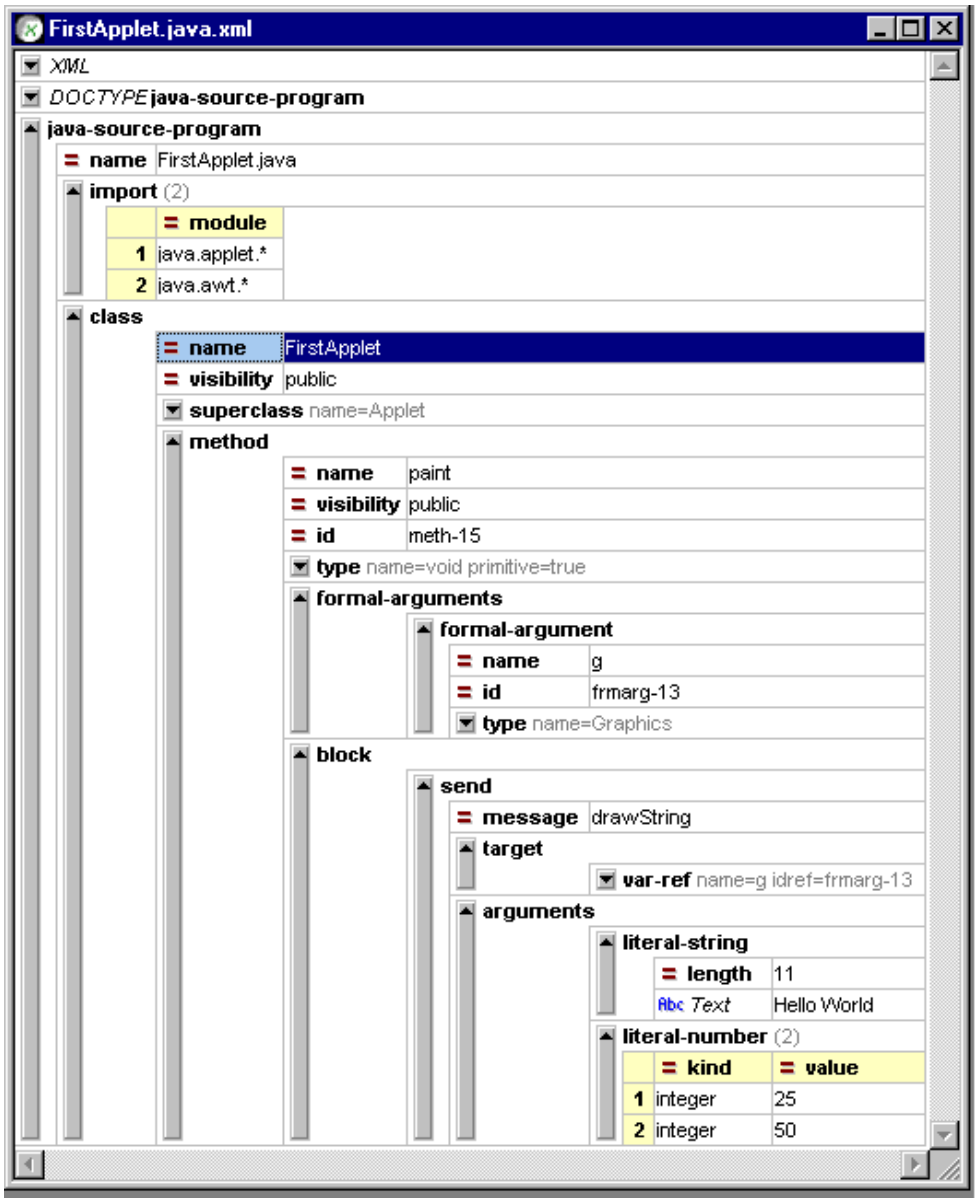
```
import java.applet.*; import java.awt.*;
public class FirstApplet extends Applet
{
     public void paint(Graphics g)
     {
          g.drawString("HelloWorld!",25,50);
     }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">

<java-source-program name="FirstApplet.java">
  <import module="java.applet.*"/>
  <import module="java.awt.*"/>
  <class name="FirstApplet" visibility="public">
    <superclass class="Applet"/>
    <method name="paint" visibility="public" id="meth-15">
      <type name="void" primitive="true"/>
      <formal-arguments>
        <formal-argument name="g" id="frmarg-13">
          <type name="Graphics"/>
        </formal-argument>
      </formal-arguments>
      <block>
        <send message="drawString">
          <target>
            <var-ref name="g" idref="frmarg-13"/>
          </target>
          <arguments>
            <literal-string value="HelloWorld!"/>
            <literal-number kind="integer" value="25"/>
            <literal-number kind="integer" value="50"/>
```

```
          </arguments>
        </send>
      </block>
    </method>
  </class>
</java-source-program>
```

Hierarchical structure of the corresponding XML file can be seen in the figures below:


FirstApplet.java.xml - XML Notepad
File Edit View Insert Tools Help
Structure
Values
java-source-program
name
FirstApplet.java
import
module
java.applet.*
import
module
java.awt.*
class
name
FirstApplet
visibility
public
superclass
name
Applet
method
name
paint
visibility
public
id
meth-15
type
name
void
primitive
true
formal-arguments
formal-argument
name
g
id
frmarg-13
type
name
Graphics
block
send
message
drawString
target
var-ref
name
g
idref
frmarg-13
arguments
literal-string
value
FirstApplet
literal-number
kind
integer
value
25
literal-number
kind
integer
value
50
For Help, press F1



8.2. Log4J[4]

In order to decrease the size of the code in the project, we have decided to use Apache Log4J[4] for Logger component. With log4j it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary.

Logging equips the developer with detailed context for application failures. One of the distinctive features of log4j is the notion of inheritance in loggers. Using a logger

hierarchy it is possible to control which log statements are output at arbitrarily fine granularity but also great ease. This helps to reduce the volume of logged output and the cost of logging.

The target of the log output can be a file, an OutputStream, a java.io.Writer, a remote log4j server, a remote Unix Syslog daemon, or many other output targets.

8.3. Jikes[5]

Jikes is a compiler that translates Java source files into the byte coded instruction set and binary format. We know that java is also a Java compiler that Sun provides free with its SDK. However, Jikes has some advantages that make it a valuable contribution to the Java community. It is open source and strictly Java compatible. Its performance is high and also its dependency analysis concept provides two very useful features: incremental builds and makefile generation. In order to use JavaML, it is a must to use Jikes compiler because JavaML library is integrated to Jikes compiler and comes with it.

8.4. Apache Tomcat[6]

Apache Tomcat is an open source servlet container developed by the Apache Software Foundation. Tomcat implements the Java Servlet and the JavaServer Pages (JSP) specifications from Sun Microsystems, and provides a pure java HTTP web server environment for Java code to run. We will use Apache Tomcat in order to test the output of our conversion operation. It is needed for testing JSF outputs to ensure their correctness.

8.5. Richfaces[7]

RichFaces is an open source Ajax enabled component library for JavaServer Faces (JSF), hosted by JBoss.org. It allows easy integration of Ajax capabilities into enterprise application development. We will use Richfaces components for mapping Applet components to JSF ones.

8. 6 Java Reflection API[8]

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Class Browsers and Visual Development Environments:** A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- **Debuggers and Test Tools:** Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Drawbacks of Reflection

- Performance Overhead:
- Security Restrictions:
- Exposure of Internals

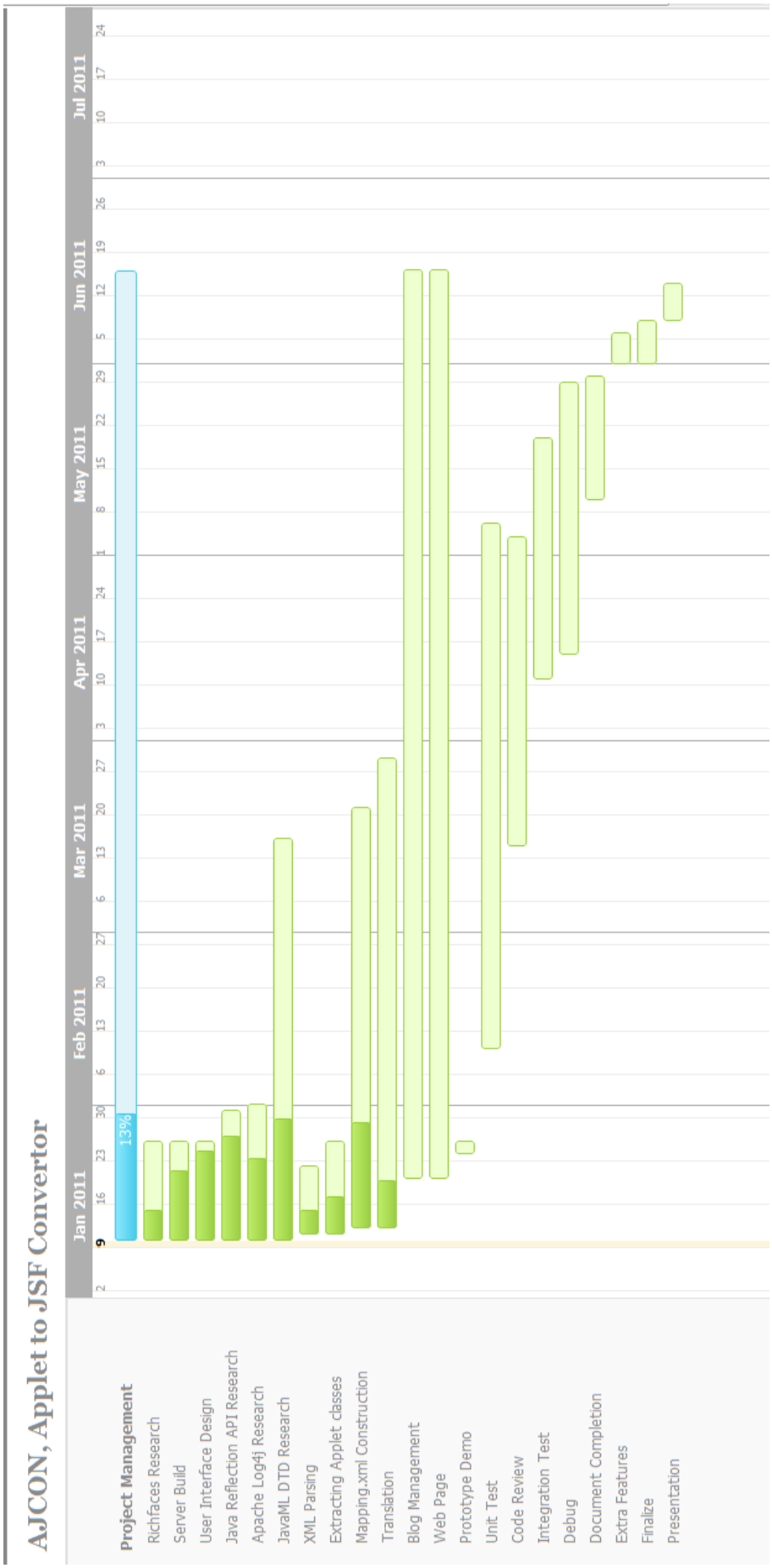
9. Change LOG

There are some changes so far with respect to Software Requirement Specifications.

SDD version 1.0 (this document)	SRS version 1.0
User can understand that conversion is completed by looking at the progress bar's %100 value	Reference sections are 2.2 "Product Functions" and 3.2.1.6 for conversionCompleted() product function
Our system will run only on Microsoft Windows platform (Vista or later)	Reference sections are 2.3 "Constraints, Assumptions and Dependencies" and 3.3.4.2.1 "Adaptability" for working platforms (OS)
Parser and Lexer component are combined into JavaML component	Reference sections are 2.1 "Product Perspective", 2.2. "Product Functions", 3.2.3. "Lexer Component Functions", 3.2.4. "Parser Component Functions" related to Lexer and Parser component

10. Time Planning (Gantt Chart)

Gantt chart about time planning and project management is stated in next page.


AJCON, Applet to JSF Convertor
Jan 2011
Feb 2011
Mar 2011
Apr 2011
May 2011
Jun 2011
Jul 2011
Project Management
13%
Richfaces Research
Server Build
User Interface Design
Java Reflection API Research
Apache Log4j Research
JavaML DTD Research
XML Parsing
Extracting Applet classes
Mapping.xml Construction
Translation
Blog Management
Web Page
Prototype Demo
Unit Test
Code Review
Integration Test
Debug
Document Completion
Extra Features
Finalize
Presentation

11. Conclusion

In this document, design considerations for project AJCON were dealt with. How our system work, how our system was decomposed, how these components work, their design architecture and connections, data design and flows were stated both by UML diagrams and by explanations. Moreover, user interactions were determined through user interfaces design. Libraries and tools which will be used during system development and operation were presented.