

**A Platform Independent Improved GUI for the PeANUt Computer Simulator**

Software Engineering Project(COMP 8790)

The Department of Computer Science

The Australian National University

**AVINASH G PRASAD**

**Supervisor: Dr Peter Strazdins**

## TABLE OF CONTENTS

<b>Abstract.....</b>	<b>4</b>
<b>1 Introduction.....</b>	<b>5</b>
1.1 Project Overview.....	5
1.2 Background.....	5
1.3 Statement of scope.....	8
1.4 Participants.....	9
1.5 Software Overview.....	9
1.6 Background on Technologies incorporated.....	10
1.6.1 Java Swing.....	10
1.6.2 Java Native Interface.....	11
1.7 System and interaction.....	12
<b>2 Plan / Management.....</b>	<b>13</b>
2.1 Scheduled Project Plan.....	13
2.1 Actual Project Plan.....	14
<b>3 Requirements.....</b>	<b>15</b>
3.1 Assumptions and Constraints.....	17
3.2 Risk Analysis.....	17
<b>4 Approach.....</b>	<b>18</b>
4.1 Modelling.....	18
4.2 Implementation.....	24
4.3 Issues.....	25
<b>5 Automated GUI testing.....</b>	<b>26</b>
<b>6 Conclusion.....</b>	<b>28</b>

<b>7 References.....</b>	<b>29</b>
<b>8 Appendix.....</b>	<b>30</b>

## **Abstract**

Nowadays softwares are created with multiple platforms in mind from the beginning. But there are instances when developers overlook this aspect and create a software with a single platform as the basis. This is a drawback for the developer if he wants to make the software multiple platform compatible which lays the foundation for my project, "A Platform Independent Improved GUI for the PeANUt Computer Simulator".

PeANUt is a simple microprocessor software used for the teaching purposes at the Australian National University. Currently it works only with the Linux Operating System. Its functionality was written in C and the scripting was done using Tcl/Tk (Tool Command Language/Toolkit) and the Graphical User Interface in Tix. The use of Tcl/Tk has created maintainability problems along with portability.

This project rewrites the Graphical User Interface of PeANUt in Java making it platform independent so that it works with all the major Operating Systems. The functionality will be imported into Java using the Java Native Interface. The artifact provided will also be more maintainable and any changes can be easily made.

Automated GUI testing was performed with the GUI to increase the robustness of the software. The testing was performed using the Abbot framework and Costello to write and record the scripts.

# **1 Introduction**

## **1.1 Project Overview**

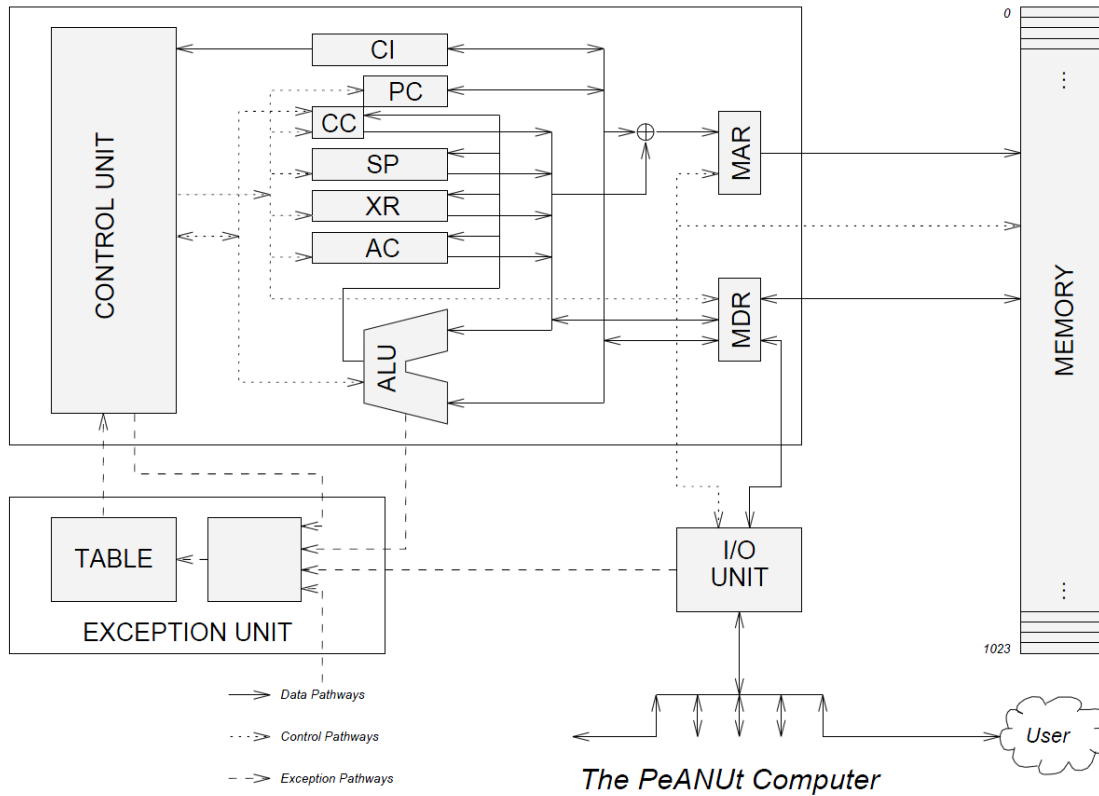
The main purpose of this project was to develop a GUI interface for the PeANUt software by keeping the core functionality intact in C and also make it operating system independent.

## **1.2 Background**

### **Graphical User Interface**

The GUI or the Graphical User Interface as we call it is a program's interface which takes the advantage of the computer's graphical capabilities to make the program easier to use. Initially considered unnecessary Graphic User Interfaces by computer developers, now have become a very important part of the system as it has optimized entry times and reduced mistypes. GUI helps to communicate with computer more effectively by drawing attention to the work in progress. GUI presentation of information has many levels of information which user tends to ignore. A consistent interface is important for quick and easy extraction of information from the screen. As most of users use multiple applications, and upgrades are constantly loaded, the user must make these evaluations daily. GUI Technology over the last couple of decades has seen steady incremental changes built on some core principles. Similar to other technologies, even the idea behind GUI was thought long before technology existed to build such a machine. Over the years GUI Technology has advanced considerably without changing the mouse interface which has remained as the backbone. Much of the core functionality for GUI remains the same but the potential for adding new features remains limitless.

## PeANUt Architecture



**Figure 1. PeANUt Architecture[1]**

### The architecture

The basic structure of the PeANUt microprocessor is given in Figure 1. The processor consists of a memory, a 16 bit arithmetic and logical unit, an addressing unit, an execution unit, a primitive operating system, a set of 16 bit registers and connections to input and output devices. There are 1024 memory cells, with addresses 0 . . . 1023. Cells contain 16 bits, and are called memory[0] . . . memory[1023]. There is a stack, which is based just after the loaded program and data, and grows in the direction of increasing addresses.[1]

## **Registers[1]**

The PeANUt machine has five special registers. Each of these registers is 16 bits in length.

**AC** - The accumulator holds data during execution of the program. It is always one of the sources, as well as the destination, of values computed by arithmetic and logical instructions.

**CI** - The current instruction register holds the instruction currently being executed. It cannot be directly referred to by programs.

**SP** - The stack pointer holds the address of the top of the stack.

**XR** - The index register holds the index for indexed instructions.

**PSW** - The program status word contains 16 bits of status information about the program.

PeANUt was developed in 1992 in the Department of Computer Science. The original design and implementation was by Brendan McKay. The addition of virtual memory features is due to Steve Blackburn. This manual is largely the work of Markus Zellner. Other changes to the design, implementation and documentation have been made over the years by Drew Corrigan, Steve Edwards, Peter Farmer, Malcolm Newey, Robin Stanton, Peter Strazdins, Trevor Vickers, Dave Walsh and Peter Christen. The Version 2 implementation is principally the work of Mark Dixon and Brendan Humphries. The term PeANUt was coined in 1993. After 2002, PeANUt ceased working in the new Operating Systems. In 2005, a study concludes that GUI should be rewritten using Java or Python.[1]

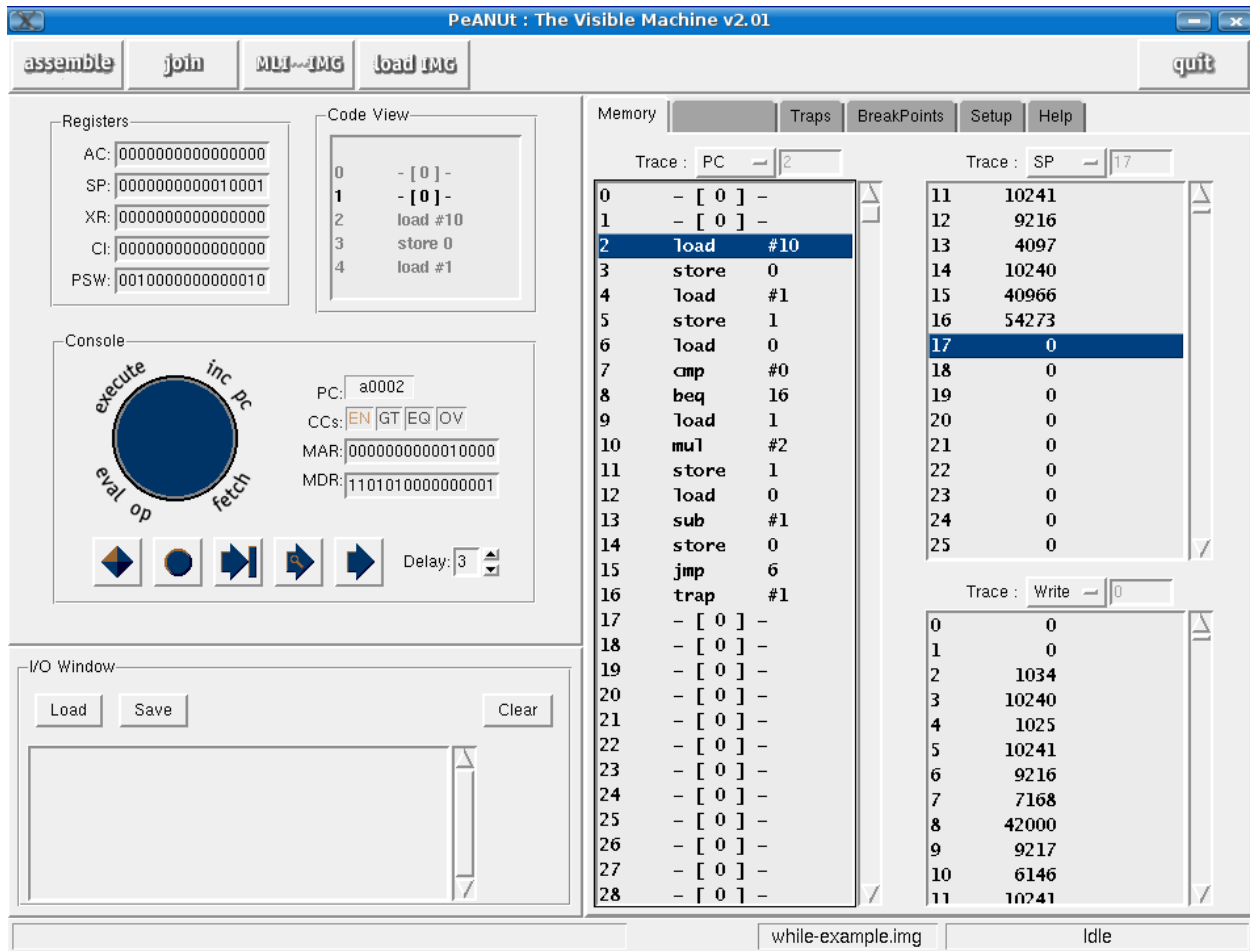


Figure 2. Old PeANUt

### 1.3 Statement of scope

The challenge was to design a GUI and interface it with the existing C code without altering it. Automated testing is supposed to be done only for GUI and it is assumed that there will be no testing required for the functionality of PeANUt. The current PeANUt works perfectly on Linux but doesn't work on any other Operating systems.

## 1.4 Participants

Role	Name	Contact Details
Supervisor	Dr Peter Strazdins	Peter.Strazdins@cs.anu.edu.au
Developer	Avinash G Prasad	U4370159@anu.edu.au

## 1.5 Software Overview

Now we will look at the new PeANUt. The new PeANUt picture is shown below

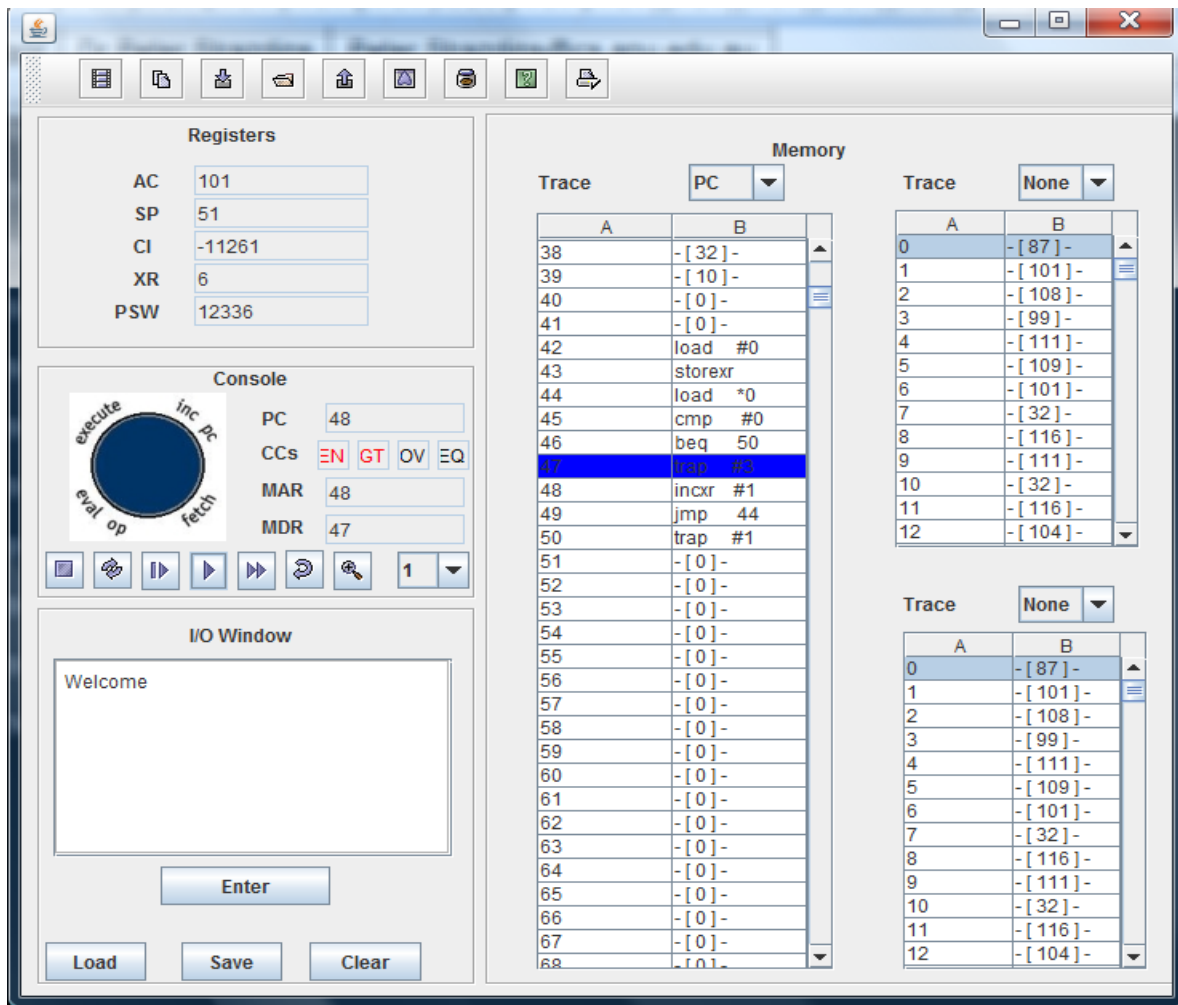


Figure 3. New PeANUt Display

The layout of the new PeANUt is very much similar to the old PeANUt as the original design was deemed to be very user friendly. The new PeANUt displays the memory registers(Accumulator, Current Instruction Register, Stack Pointer and Index Register), On the left hand side is the console which displays the Memory Address Register, Memory Data Register , Condition Codes and the Program Counter. The output area displays the text and also allows the user to input the text when a program requires an input from the user. The memory panel displays the memory traversal when the program is in execution. The memory view allows the Program Counter's movement through the memory, Read and Write access of the memory. It also shows the movement of the Stack Pointer.

## **1.6 Background on Technologies incorporated**

### **1.6.1 Java Swing**

Swing is part of Sun Microsystems Java Foundation Classes (JFC) which is an API for providing a graphical user interface (GUI) for Java programs. Swing provides a sophisticated set of GUI components than the earlier Abstract Window Toolkit. It also provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has got many advantages like platform independence, extensibility. It is also lightweight and configurable.[6]

#### **Platform independence**

Swing is platform independent both in terms of its expression (Java) and its implementation. Its widget system design is not native which makes it platform independent.

## Extensibility

Swing technology is highly extensible, allowing users to plug various custom implementations of specified framework interfaces: Users can provide their own custom implementation(s) of these components to override the default implementations. In general, Swing users can extend the framework by extending existing (framework) classes and/or providing alternative implementations of core components.

### 1.6.2 Java Native Interface

The Java Native Interface (JNI) is a programming framework that allows Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.[2]

JNI allows us to write native methods when an application cannot be written entirely in the Java programming language. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications.

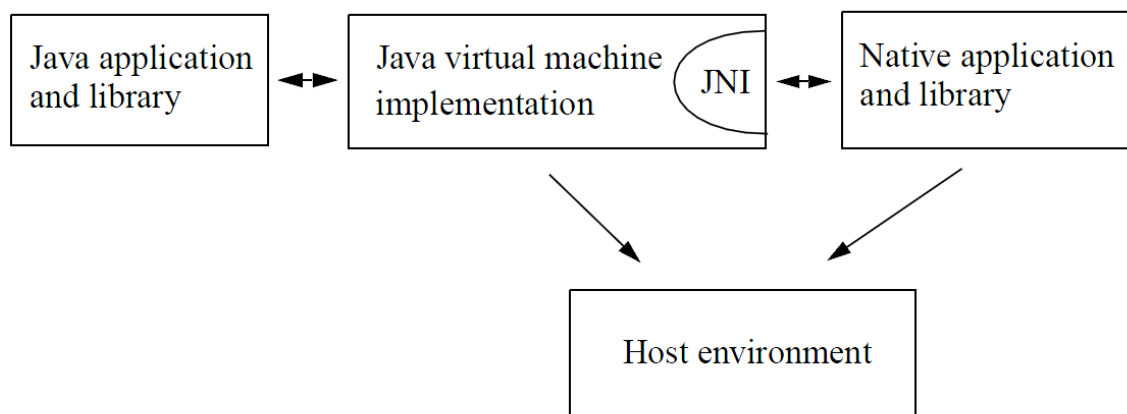


Figure 4 Java Native Interface [4]

The working of JNI is shown in the above figure . The JNI is designed to handle situations where Java applications need to be combined with native code. JNI supports native libraries and native applications.

JNI can be used to write native methods that allow Java applications to call functions implemented in native libraries. Java applications call native methods in the same way that they call methods implemented in the Java programming language.

JNI also supports an invocation interface that allows you to embed a Java virtual machine implementation into native applications. Native applications can link with a native library that implements the Java virtual machine, and then use the invocation interface to execute software components written in the Java programming language.

## 1.6 System and interaction

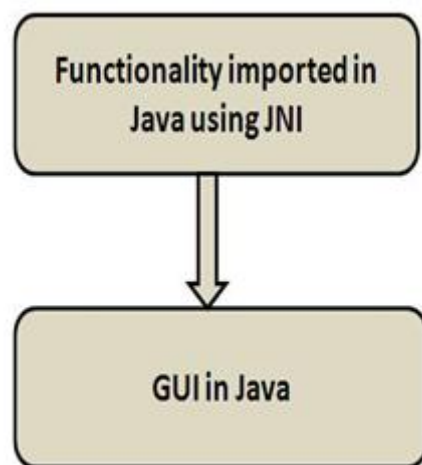


Figure 5. New PeANUt Working

The GUI is created in Java using Swing and Java native Interface acts as the mediator between Java and the C code. A Java virtual machine( Environment) is created so that the native code can run in this environment.

## 2 Plan / Management

This gives out the scheduled plan for the project and also the actual plan for the project

### 2.1 Scheduled Project Plan

#### Project Plan

Week	Tasks	Notes
1-3	Requirements & Background research	<ul style="list-style-type: none"><li>• Understanding requirements</li><li>• Doing Risk Analysis</li><li>• Background research on GUI</li><li>• Decide Tools and software</li><li>• Documentation</li></ul>
4-5	Modeling	<ul style="list-style-type: none"><li>• Analyze the existing artifact</li><li>• Modeling using UML Class diagrams</li></ul>
6-13	Implementation	<ul style="list-style-type: none"><li>• Coding for the PeANUt software which consists of creating a new GUI in Java and porting the functionality using JNI using software design methodologies</li><li>• Testing consists of Automated test cases which automate the tests using scripts</li><li>• Debugging the code and fix the errors</li></ul>

10-14	Report	<ul style="list-style-type: none"> <li>• Completing the final report</li> <li>• Preparing slides for the final presentation</li> </ul>
-------	--------	--

The whole project period was decided to be 14 weeks. Since there was lot of research to be done regarding the technology that is feasible for the project around 3 weeks were given for research. Modelling the class diagrams was given 2 weeks. Implementation which was the biggest part of the project was given 8 weeks. It was decided that report writing should be started earlier, so it was given around 5 weeks time.

## 2.2 Actual Project Plan

### Project Plan

Week	Tasks	Notes
1-4	Requirements & Background research	<ul style="list-style-type: none"> <li>• Understanding requirements</li> <li>• Doing Risk Analysis</li> <li>• Background research on GUI</li> <li>• Decide Tools and software</li> <li>• Documentation</li> </ul>
5-6	Modeling	<ul style="list-style-type: none"> <li>• Analyze the existing artifact</li> <li>• Modeling using UML Class diagrams</li> </ul>

7-14	Implementation	<ul style="list-style-type: none"> <li>• Coding for the PeANUt software which consists of creating a new GUI in Java and porting the functionality using JNI using software design methodologies</li> <li>• Testing consists of Automated test cases which automate the tests using scripts</li> <li>• Debugging the code and fix the errors</li> </ul>
10-15	Report	<ul style="list-style-type: none"> <li>• Completing the final report</li> <li>• Preparing slides for the final presentation</li> </ul>

The background research took a long time for this project as I had to be sure on using Java Native Interface. There was not sufficient coding examples to look at for reference. So modelling time was postponed.

### **3.Requirements**

The requirements of this project are as follows

#### **Make the PeANUt compiler platform independent**

The compiler should run on different Operating systems. The current version of PeANUt runs only on the Linux Operating System. This is due to the fact that it uses Tcl\Tk and Tix libraries which works only on Linux. The main aim of the project is to run PeANUt on most of the other major Operating Systems.

## **Complete port of the GUI Component**

The current coding for PeANUt GUI is very unstable and needs to be repaired quite often which is shown by the fact that it has ceased to work on many occasions. PeANUt stopped working on Linux in 2005 and also on the Solaris operating system in 2007[1]. So a new GUI interface needs to be designed which integrates easily with the existing functionality and is highly maintainable.

## **Functionality should be maintained**

PeANUt was designed in C which is not platform independent. There might be instances where the code will not execute in other operating systems. The second major intention of the project is to maintain the functionality of the PeANUt compiler by not changing the inherent C code.

## **The new PeANUt should be more maintainable**

The current PeANUt has got lots of problems with Tcl/Tk codes which makes it more prone to have incompatibility issues with new version of Linux. This makes it difficult for a developer who has no knowledge of Tcl/Tk to rectify the code. So the new PeANUt should use a more familiar language which should be easy to use and maintain in future.

## **Use Automated GUI Testing**

Automated GUI testing scripts should be used to test the software. This enables the user to run scripts instead of hard coding tests into the software.

### **3.1 Assumptions and Constraints**

1. It is assumed that the existing PeANUt functionality works and there will be no development or testing required confirming it.
2. The main developing language used will be Java and all development will be done using the Eclipse IDE.
3. The C code will be imported to Java and it is out of the scope of the project to do an actual translation of the code into Java.
4. The C compiler used in Windows is Mingw.

### **3.2 Risk Analysis**

1. Background research for GUI Design and Maintainability may require a long time as it is a specialized field in Application Design and Maintainability. To mitigate this risk, we can use the papers and resources available for Application Design and Maintainability (having a heavy emphasis on GUI) and follow a similar method followed in these papers.
2. Full code portability may not be possible i.e. Full C code may not be portable to Java resulting in partial or full implementation of the C code in Java. From the early stages of the project, the parts which cannot be ported needs to be identified and if there is no alternative then we have to do an implementation in Java.
3. All the software aspects may not be tested using automated scripts. Some functionality may have to be manually tested. I have to start writing test cases in parts along with the software implementation so that the parts which can be automated and which cannot be automated are identified.

## **4 Approach**

After analyzing different platforms rigorously and circumstantially, I planned to implement Java as the platform for developing project. The reasons are as follows

Since JNI is a part of the Java platform, programmers can address interoperability issues once, and their solution will work with all implementations of the Java platform. Another advantage is the exhaustive library of GUI components in Java which allows me to develop complex GUI without much trouble. By using Java as the frontend and JNI as the mediator, the execution speed of the software increases as both of them are integrated. Having JNI and Java together makes the code more maintainable. Another very important consideration I had was the developer community for JNI who have extensive knowledge about JNI and are always ready to help JNI beginners.

### **4.1 Modeling**

Modeling is the method of building representations of real world entities and allows ideas to be investigated. It is central to all activities and is the starting point in the process for building or creating an artifact of some form or other.

I decided to model the GUI using use case model[7]. This model allows us to identify the requirements of a particular system and study its high level functionality.

With the help of a use case diagram representing actors and use cases of a system, and activity diagrams representing each use case, a prototype of the user interface having the GUI components was generated. Actors and use cases are the key concepts of the use case models. The users and any other systems that may interact with the system are represented as actors. Use

cases specify the required behaviour of the system. These use cases are defined according to the needs of the actors.

In the use case diagram given in figure 6 and figure 7 the relationships between actors are generalizations, and relationships between uses cases are <<include>> dependences, together with generalizations. The relationships between actors and use cases are generalizations.

Multiplicity, roles, directionality and *extend* dependences will not be considered.

Activity diagrams specify user-system interaction. States represent outputs to the user which are labelled with UML stereotypes representing visual components for data output. Transitions represent user inputs which are labelled with UML stereotypes representing visual components for data input and choices. This finer description allows a mapping with the graphic user interface design. The refinement of uses cases by means of activity diagrams achieves more precise specifications, enabling to detect <<include>> and generalization relationships between use cases [3].

The steps that I followed for creating the diagrams are

1. The first step comprises of generating a high level descriptions of the system using a use case diagram with the actors and the use cases.
2. Based on the use case diagrams the activity diagrams are modelled. For each use case its behaviour is described by means of one or more activity diagrams.
3. Thirdly, the use cases and the stereotyped states and transitions are translated into class diagrams.
4. Finally, the class diagrams obtained in the previous step produce Java implementations which could be considered as GUI component prototypes

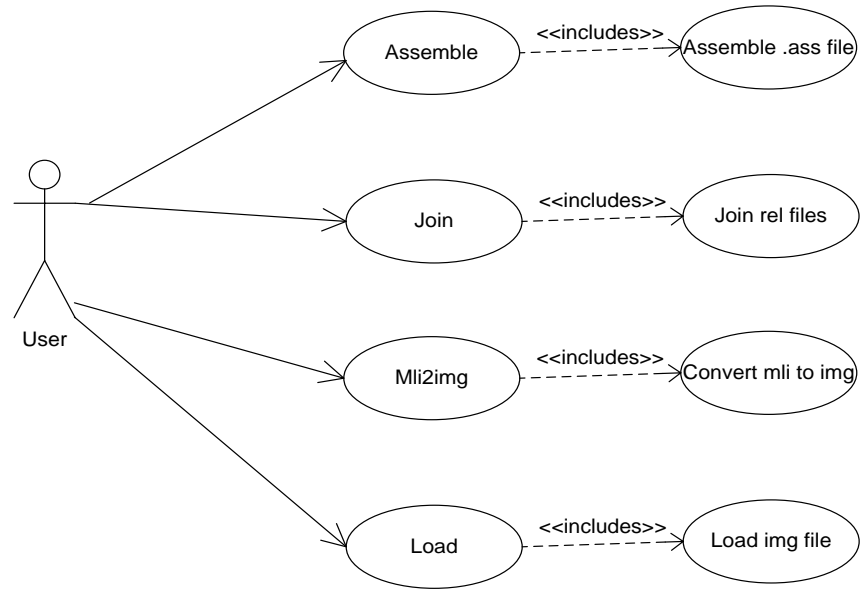


Figure 6- Use case diagram for Assemble, Join, Mli2img and Load

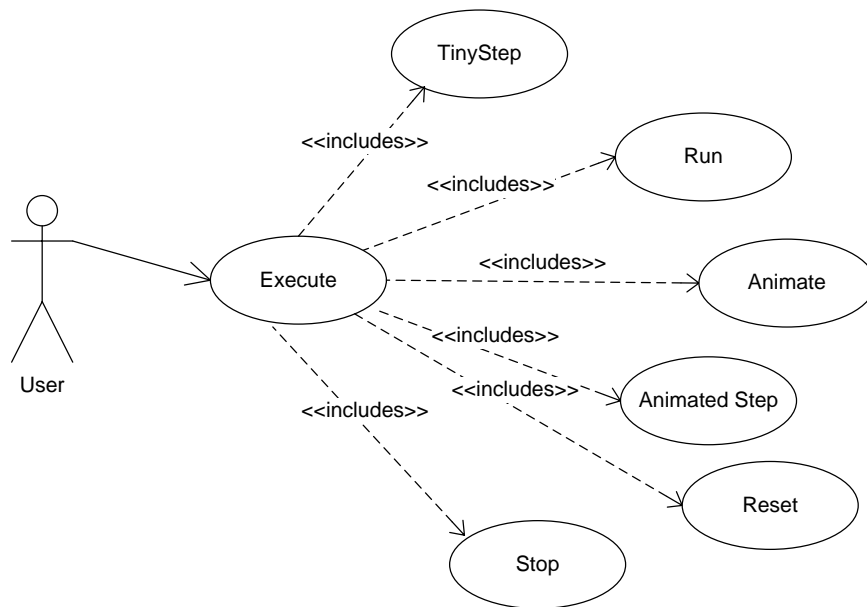


Figure 7-Execute use case diagram

The first task before starting the design was the identification of future windows in the PeANUt system. So I identified the windows which will open once the appropriate event occurs. For

example, when the Assemble button is clicked, the FileChooser window should open. In my design, each use case will represent another window of Frame component doing other tasks. The connection of actor(user) with the use cases will be interpreted as a user generated event in the main window on which the user interacts with the PeANUt system. But the execution use case will not have any window as the execution takes place in the main window. The execution use case depends on the type of execution button the user clicks on. An `<<include>>` relationship between the use cases can mean that a use case could be considered as a composition of two or more other use cases. For instance, the use case Execution is composed of the use cases Run, Reset, Stop, Animate, Tiny Step and Animated Step (i.e., applets or frames). However, an `<<include>>` relationship can also indicate that a use case mandatorily depends on another use case to operate. For example, the "Assemble .ass file " use case depends on the Assemble use case.

### Activity Diagrams

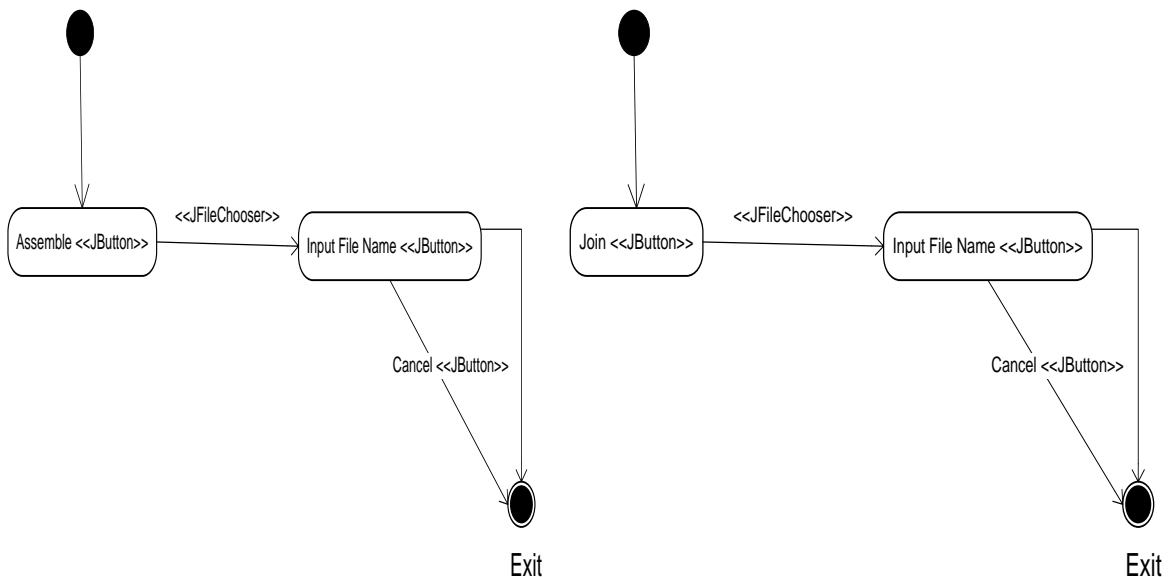


Figure 8- Assemble and Join activity diagrams

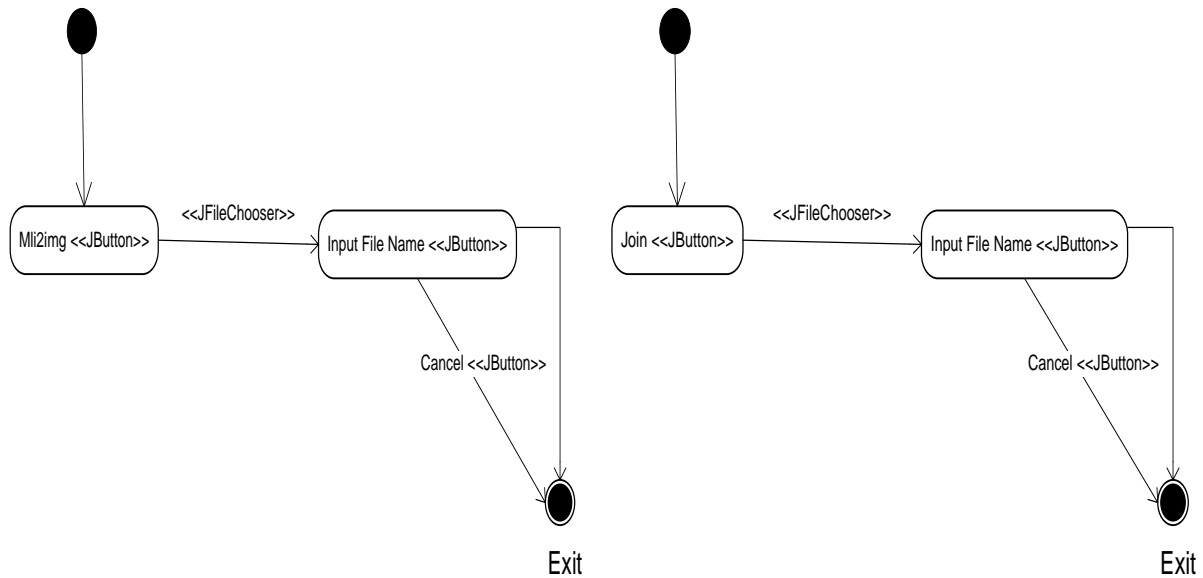


Figure 9-Mli2img and Join activity diagrams

Activity diagrams describe certain graphical and behavioural details about the graphical components of the frame. Graphical components can be classified as input (a text area or a button) and output components (a label or Table). Transitions can be labelled by means of stereotypes, conditions or both together. For instance, a button is connected to a transition by using the <<JButton>> stereotype, and the name of the label is the name of the button. For example, a Assemble transition stereotyped as <<JButton>> will correspond with a button component called “Assemble”. Figure 8 shows the activity diagrams for Assemble, Load, mli2img, Execution and Join use cases. Let us consider the Assemble activity diagram. The behaviour shows how the user begins by assembling the program by clicking the assemble button. A FileChooser window opens up which allows the user to choose the assembly file. He can click on assembling the file or he can cancel and return to the main screen. The mli2img, join and load behaviours are similar to the assemble activity diagrams. Once the activity is completed, a successful operation window comes up. If the operation is unsuccessful then a warning window shows up with a message telling that to the user.

## Class Diagram

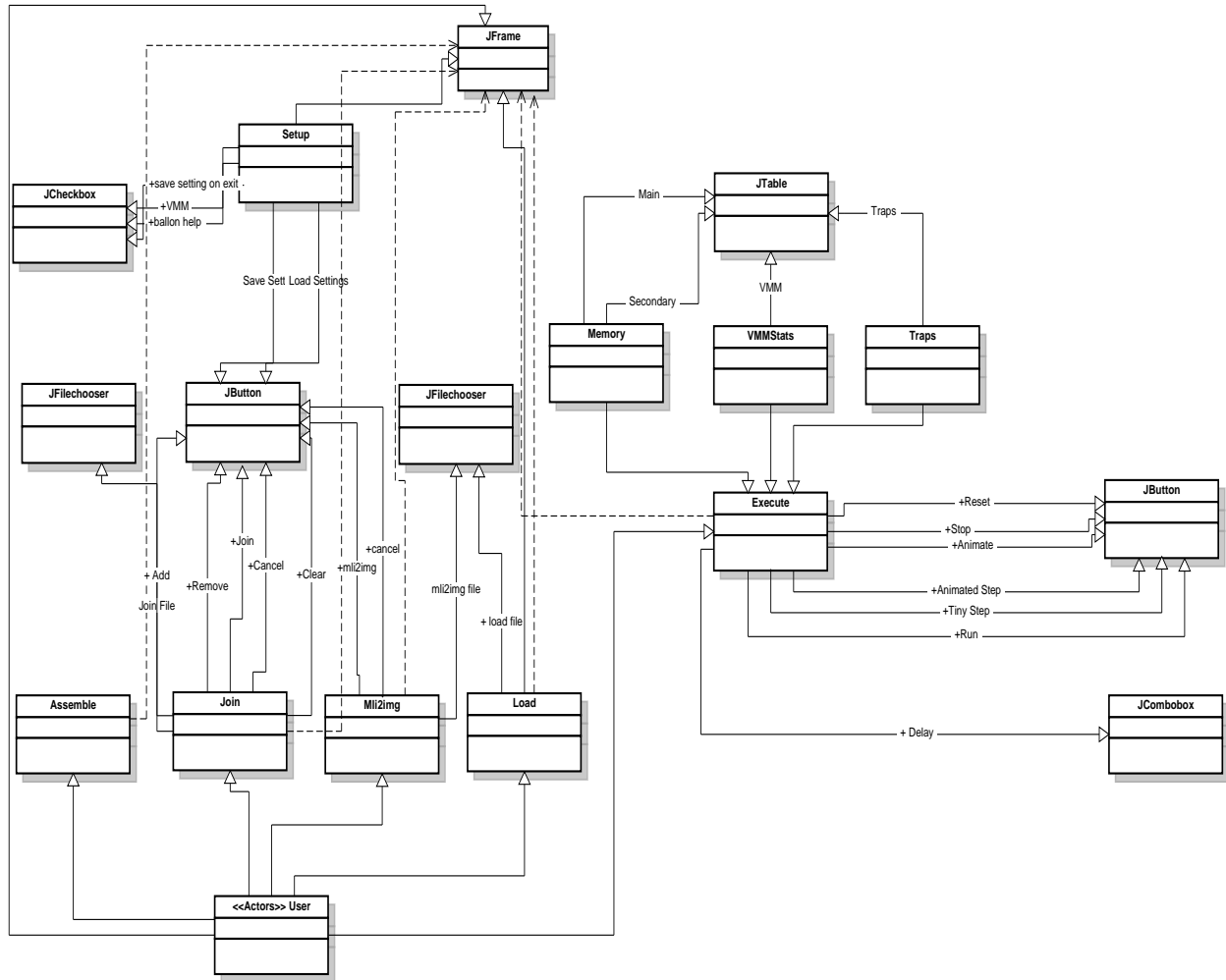


Figure 10-. Class Diagram

Once a formal use case diagram and the respective activity diagrams are obtained we generate a class diagram based on them as shown in figure 10. The class diagrams are built from Java *swing* classes. In the method, each use case corresponds with a frame class. Use cases are translated into classes with the same name as these use cases. The translated classes specialize in a Java *JFrame* class. The components of the frame (use case) are described in activity diagrams. A terminal state is translated into that Java *swing* class represented by the stereotype of the state. The Java *swing* class is connected from the container class (i.e., that class working as an applet window in the use case diagram) and uses an association relationship whose role's name is the

one on the terminal state. For example, those terminal states stereotyped as <<JButton>> are translated into a JButton class in the class diagram. The Java *swing* class is connected from the container class (i.e., that class working as a frame window in the use case diagram) and uses an association relationship whose role's name is the one on the terminal state. For example, those terminal states stereotyped as <<JFileChooser>> are translated into a JFileChooser class in the class diagram. Something similar happens to the rest of stereotyped states and transitions. The non-terminal states of an activity diagram may correspond to some other use cases (frames) or activity sub diagrams. The class diagram contains five classes of the JFrame type, which directly specialize in the JFrame class: the Assemble class, the Join class, the Mli2img class, the Load class and the Execution class. The other classes inherit the JFrame class through their super classes. For example, the VMMstats , Traps and Memory classes inherit the JFrame classes through the super class Execution. These classes which (direct or indirectly) inherit the JFrame class— correspond to use cases in the use case diagram together with the user as the actor.

## **4.2 Implementation**

After finishing the GUI, the next part involved access to the native library routines from the Java program. To accomplish that I use the Java SDK's Java Native Interface (JNI), from the PeANUt program. When Java programs call native methods (C functions), extra arguments are prepended to the argument list of the called C function. These arguments give the C code a handle onto Java methods and properties. However, to call a function that's in object code, an intermediate shared library (Linux) or DLL (Windows) needs to be created. This library is the interface between the Java code and library code.

Java allows us to use Object oriented programming and Structured programming. I was using a mixture of structured programming and Object Oriented programming. The steps that I followed to create JNI methods in C are

1. A native method declaration is written in Java. This declaration includes the keyword *native* to signify to the Java compiler that it is implemented externally.
2. A header file is created for using the native code. This header file contains the declaration of the native method as viewed by the C compiler. It includes the extra arguments required for the C function to access Java methods and properties, and has argument types defined in terms of standard C types.
3. The native method is implemented in C. This function uses the header file in Step 2, makes calls to library functions it needs (possibly back to Java methods), and returns results to Java. This C code is compiled to build a shareable library.

### **4.3 Issues**

There were a few issues which I had to overcome to make PeANUt multi platform compatible. They are as follows

1. I had to create .def files in Windows. These files are called as Definition files which contain all the method names that are created in JNI for a specific Java class. When the shared library is created the Definition file is also passed as an argument to the C compiler while compiling. This helps the compiler to link the method names so that during execution errors are not created.
2. Windows and Linux have different directory notions. Windows uses "\" to separate nested directories and Linux and Mac uses "/" for the same. So I had to modify the code in Java to suit the particular operating systems while dealing with files.

3. The shared libraries which are required for calling the C functions when the program is executing is operating system dependent. Linux and Mac uses "Shared Object" files whereas Windows uses "Dynamic Link-Library" at runtime.

## **5 Automated GUI Testing**

Graphical User Interface testing is a very difficult area because constructing test cases for GUI is more difficult than the application logic. There are many roadblocks to GUI testing like traditional test coverage criteria is not sufficient to trap all the user interaction scenarios. Even end users use a different task interaction model than the one conceived by the development team. Functional GUI testing needs to deal with GUI events as well as the effects of the underlying application logic that results in changes to the data and presentation.[5]

I used Abbot to test the GUI because it is an open source framework which allowed me to quickly come up with an effective and comprehensive test framework. It also allows record and execute scripting.

The script editor Costello will be used to develop a battery of tests. Costello provides the "record and execute" functionality that will allow us to record different user interaction scenarios with the GUI and test those scenarios efficiently. The scripts that record the user interactions are saved as XML.

### **Using Costello**

Costello records and executes scripts which allows us to do tedious testing in a very short time.

In my PeANUt GUI, I clicked on the Assemble button and selected a file. The file assembled

successfully. It showed the message that the file had assembled successfully. I clicked on Join and selected the assembled file and converted into img file. I loaded an image file as well. All the user actions which I was doing were recorded in Costello. Then the script runs when I click on the run button and executes all the actions very quickly thereby testing the GUI. I was also able to save these scripts as XML files and these scripts are dynamically interpreted by Abott which saves us writing test cases. I was even able to add assert statements in Costello which made checking whether the image had loaded correctly or not. Another advantage I had with Costello was that it dynamically stores all the information regarding the GUI component position. With the help of these scripts I was able to rectify GUI errors and make it more robust.

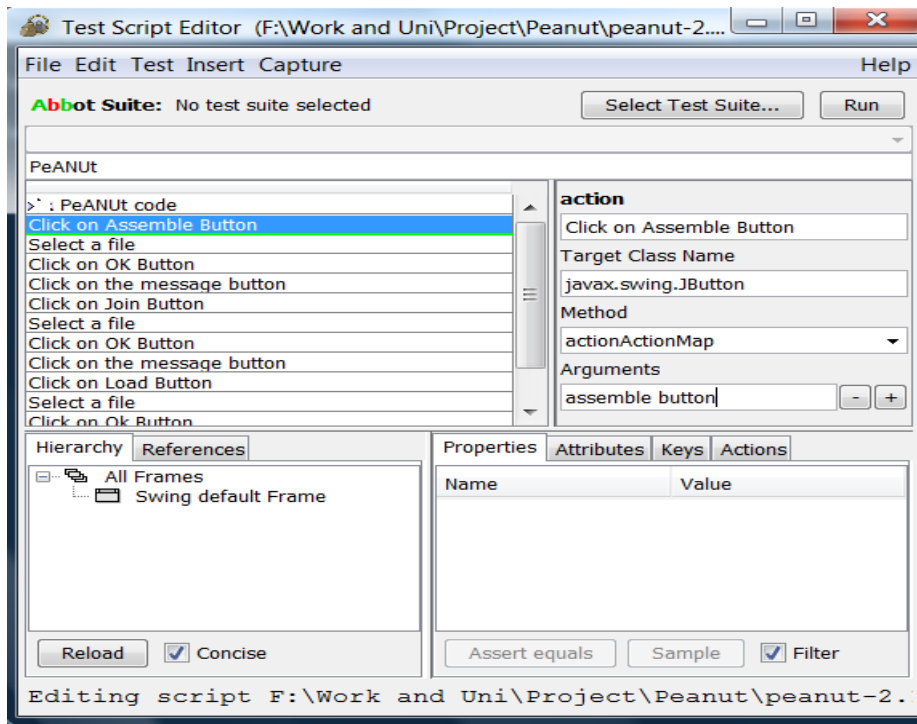


Figure 11- Costello

## **6 Conclusion**

I was able to run PeANUt simulator using Java in all the major operating systems which includes Windows XP, Windows Vista, Fedora 8.0, Ubuntu 8.0 and Mac OS 10.4. The startup time of the simulator was faster compared to the old Tcl/Tk PeANUt. The main aim of the project was a new GUI interface for the PeANUt compiler without destroying the functional integrity and I have accomplished that using Java and Java Native Interface. With the help of Automated GUI testing I was able to identify errors and problems with my GUI and further develop my GUI to make it more user friendly and efficient. The new GUI will enable students and staff to use PeANUt on any system and anywhere for their own use. Since the GUI was developed in Java with object oriented programming, the code is more maintainable.

## 7 References

References:

[1]The PeANUt Computer Specification –Department of Computer Science, The Australian National University.

[2]Dr. Dobb's, "Using the Java Native Interface", July 2003.

[3]Perdita Stevens. On Use Cases and Their Relationships in the Unified Modelling Language. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE'01*, pages 140–155. LNCS 2029, 2001.

[4]Sheng Liang. On Role of the JNI. In "The Java Native Interface Programmer's Guide and Specification". 1999 Sun Microsystems, Inc. ISBN 0-201-32577-2,

[5] Abbot GUI testing framework overview: [http://abbot.sourceforge.net/doc/api/overview-summary.html#overview\\_description](http://abbot.sourceforge.net/doc/api/overview-summary.html#overview_description)

[6]Java: <http://www.java.sun.com>

[7] Jes'us M. Almendros-Jim'enez and Luis Iribarne, " Designing GUI components from UML Use Cases" Dpto. de Lenguajes y Computaci'on. Universidad de Almer'ia, Spain.

## 8 Appendix

### Assemble

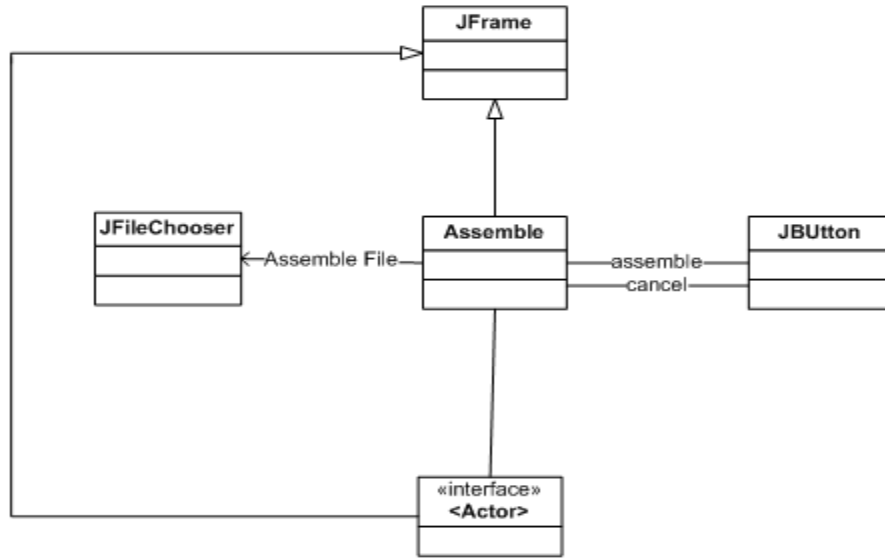


Figure 12- Assemble

### Join

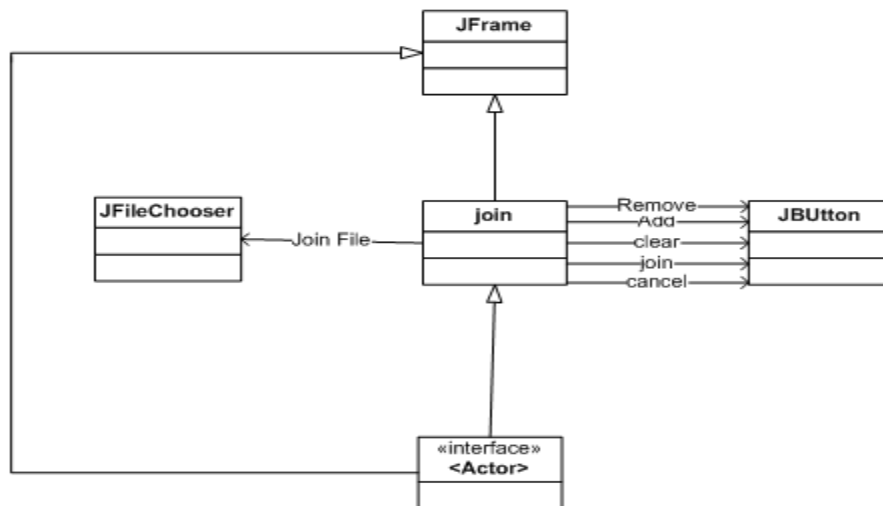


Figure 13- Join

## Mli2img

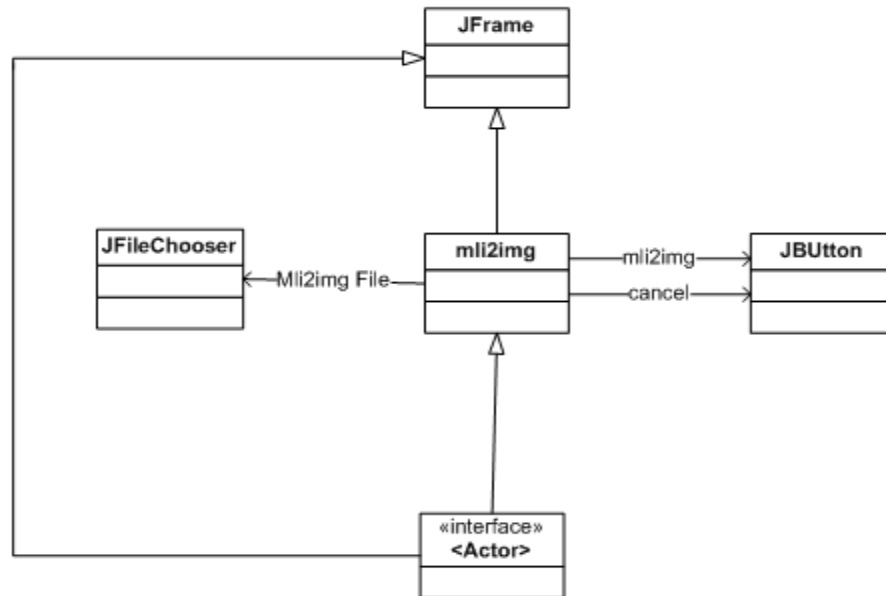


Figure 14- Mli2img

## Execute

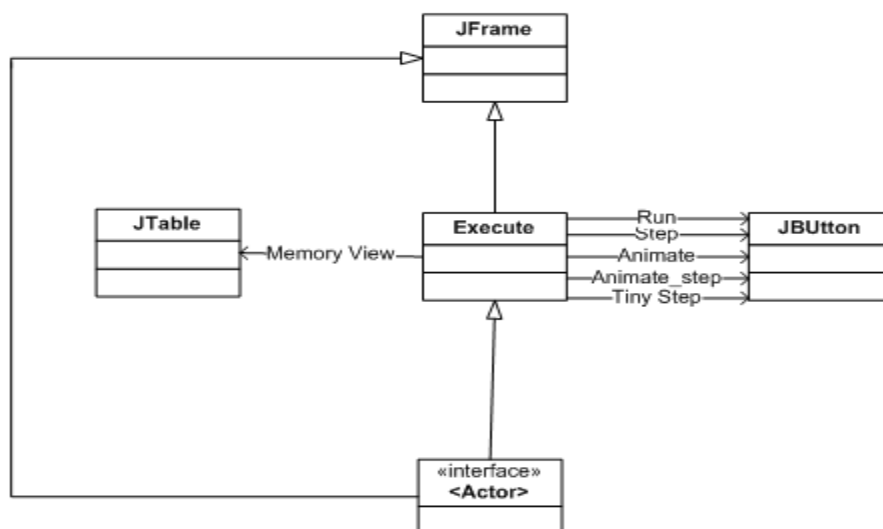


Figure 15- Execute

# Load

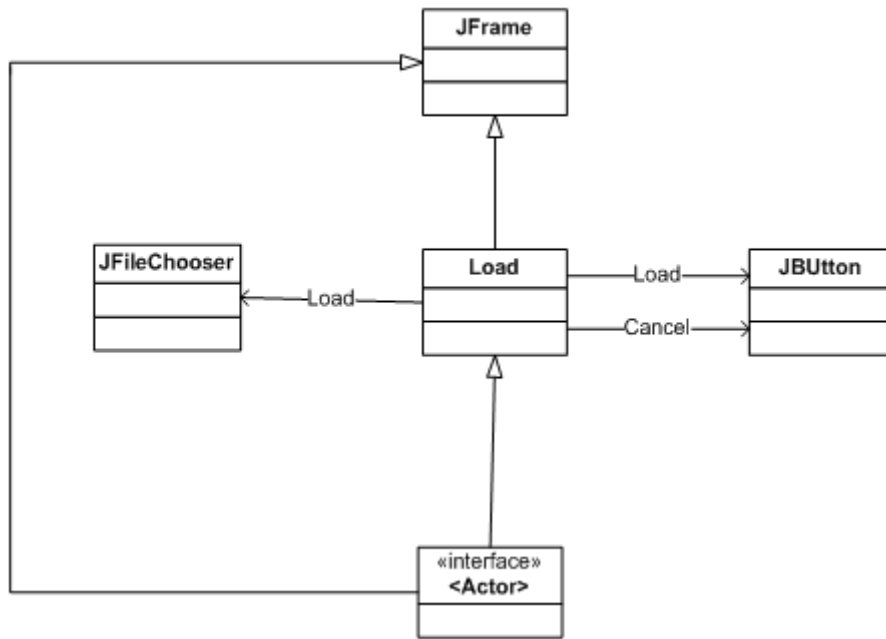


Figure 16- Load

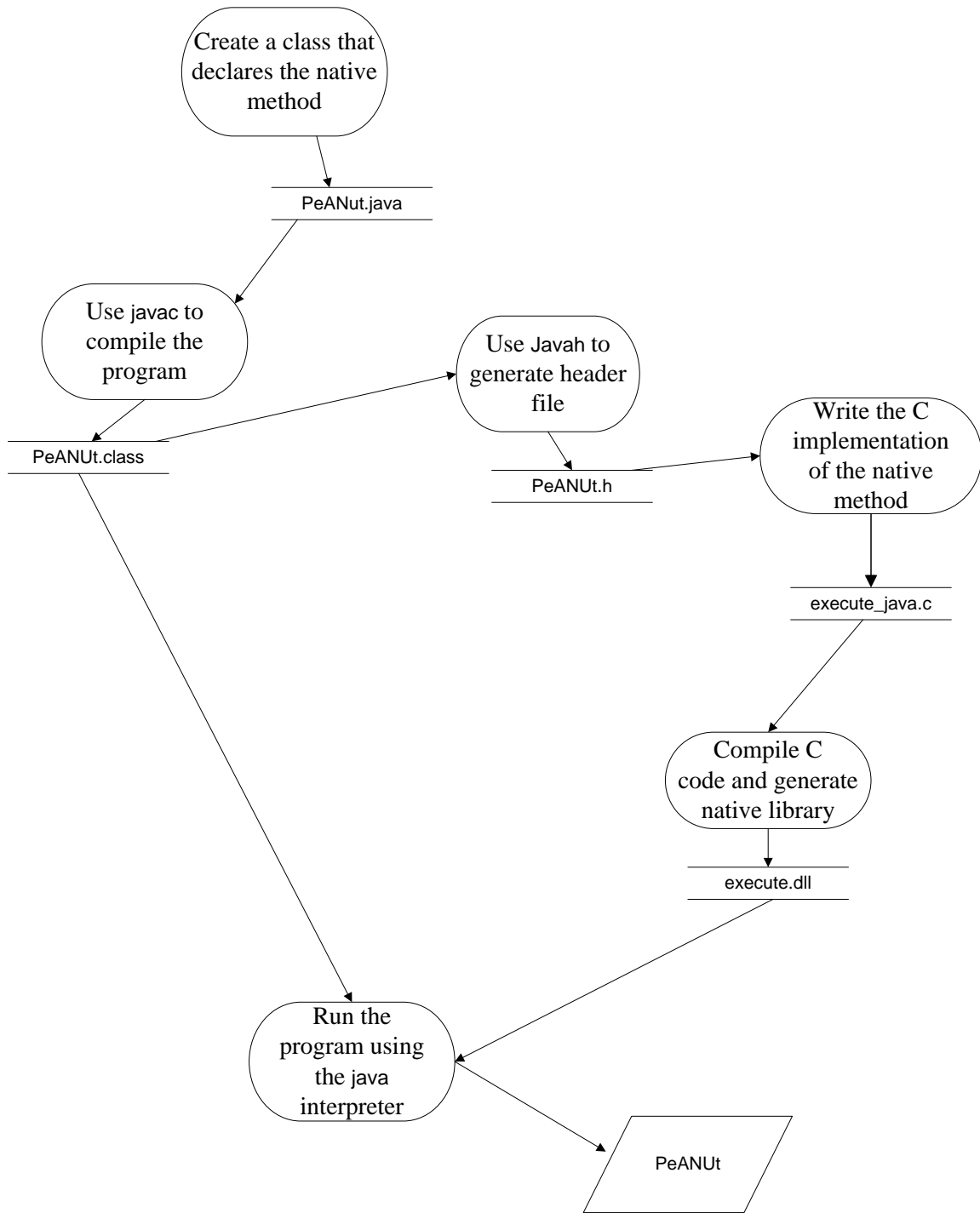


Figure 17-Compilation and Execution Process