MASTER OF SOFTWARE ENGINEERING PORTFOLIO

By

BINTI SEPAHA

A PORTFOLIO

Submitted in Partial Fulfillment of the Requirements for the Degree

MASTER OF SOFTWARE ENGINEERING

DEPARTMENT OF COMPUTING AND INFORMATION SCIENCES

COLLEGE OF ENGINEERING

KANSAS STATE UNIVERSITY
MANHATTAN, KS

2005

APPROVED BY:

DR. SCOTT A. DELOACH
ABSTRACT

The agentTool allows agent system designers to formally specify the required structure and behavior of a multiagent system and semi-automatically synthesize multiagent systems that meet those requirements. The system designer formally defines high-level system behavior graphically using our Multiagent Systems Engineering methodology. The system design defines the types of agents in the system as well as the possible communications that may take place between agents. This system-level specification is then refined for each type of agent in the system. To refine an agent, the designer either selects or creates agent architecture and then provides detailed behavioral specification for each component in the agent architecture. This MSE Project involves the development of agentTool III (Dynamic) software that will help the users to draw dynamic diagrams. These diagrams include the sequence and activity diagrams.
ACKNOWLEDGEMENTS

I would like to acknowledge and thank Dr. Scott A. DeLoach, my major professor, for being inspirational and supportive throughout my Masters Degree. His advises and guidance has been extremely helpful at every step for the completion of this project.

I am grateful to my committee members, Dr. David A. Gustafson and Dr. William Hankley to provide valuable suggestions in throughout the course of the project.

I would like to specially thank the GEF members whose support made it possible to solve intricate problems encountered in this project.

I appreciate my husband, Lalit Parashar, for his support and belief in me which always encouraged me to put in my best.

Lastly, I thank all my friends and family members for exhibiting their faith in me and helping me as and when possible.
TABLE OF CONTENTS

Chapter 1. Vision Document .............................................................................................. 1
Chapter 2. Project Plan ..................................................................................................... 20
Chapter 3. Software Quality Assurance Plan .............................................................. 27
Chapter 4. Architecture Design ...................................................................................... 31
Chapter 5. Component Design ....................................................................................... 54
Chapter 6. Test Plan ....................................................................................................... 81
Chapter 7. Assessment Evaluation ............................................................................... 90
Chapter 8. Project Evaluation ....................................................................................... 94
Chapter 9. Inspection Checklist .................................................................................... 99
Chapter 10. User Manual – Activity Diagram Editor .................................................... 101
Chapter 11. User Manual – Sequence Diagram Editor ............................................... 109
References .................................................................................................................. 117
LIST OF FIGURES

Figure 1. Project Overview ................................................................. 2
Figure 2. Use-Case Model................................................................. 4
Figure 3. Frame and Protocol Name.................................................. 5
Figure 4. Alternative Action............................................................. 6
Figure 5. Reference to Another Protocol .......................................... 7
Figure 6. Synchronous Message ...................................................... 8
Figure 7. Synchronous Message ...................................................... 8
Figure 8. Message that the sender receives ..................................... 8
Figure 9. Message that the sender will not receive ......................... 8
Figure 10. Stop Sign ...................................................................... 9
Figure 11. Initial State .................................................................. 10
Figure 12. Final State ................................................................... 10
Figure 13. Action State .................................................................. 11
Figure 14. Action Flow .................................................................. 11
Figure 15. Outgoing Event ............................................................. 12
Figure 16. Incoming Event ............................................................. 12
Figure 17. Decision Node ............................................................... 12
Figure 18. Flow Final Node ........................................................... 13
Figure 19. State ........................................................................... 13
Figure 20. State with Compartments .............................................. 13
Figure 21. Initial Node .................................................................. 15
Figure 22. Final Node .................................................................. 15
Figure 23. Transition ................................................................... 15
Figure 24. Synchronization Point ............................................... 16
Figure 25. Splitting of Control .................................................... 16
Figure 26. Project Schedule ......................................................... 21
Figure 27. MVC Approach in GEF ............................................... 32
Figure 28. Model and the View .................................................... 32
Figure 29. Package View ............................................................. 33
Figure 30. Model Package ........................................................... 35
Figure 31. Working of Controller in GEF .................................... 39
Figure 32. EditParts Package ........................................................ 40
Figure 33. Commands Package......................................................................................... 43
Figure 34. Figures Package ............................................................................................ 45
Figure 35. Add an Element – Sequence Diagram .......................................................... 47
Figure 36. Move an Element – Sequence Diagram....................................................... 48
Figure 37. Delete an Element – Sequence Diagram....................................................... 49
Figure 38. Package View............................................................................................... 54
Figure 39. ActivityDiagramEditor Class ................................................................. 55
Figure 40. ActivityDiagramPlugin Class ...................................................................... 57
Figure 41. PaletteViewerCreator Class ......................................................................... 57
Figure 42. PaletteFlyoutPreferences Class ................................................................. 58
Figure 43. ActivityDiagramPaletteViewerProvider Class ............................................ 58
Figure 44. GraphicalViewerCreator Class ................................................................... 59
Figure 45. OverviewOutlinePage Class ...................................................................... 59
Figure 46. Model Package............................................................................................ 60
Figure 47. PropertyAwareObject Class........................................................................ 61
Figure 48. Activity Class............................................................................................... 62
Figure 49. ActionState Class........................................................................................ 64
Figure 50. ActionFlow Class........................................................................................ 65
Figure 51. Figures/View Package................................................................................ 66
Figure 52. ActivityDiagramFigure Class .................................................................... 66
Figure 53. ActionStateFigure Class............................................................................ 67
Figure 54. EditParts Package....................................................................................... 68
Figure 55. PropertyAwarePart Class........................................................................... 68
Figure 56. ActivityDiagramPart Class ....................................................................... 69
Figure 57. ActionStatePart Class................................................................................ 70
Figure 58. PropertyAwareConnectionPart Class ....................................................... 72
Figure 59. ActionFlowPart Class................................................................................ 72
Figure 60. ActivityDiagramXYLayoutPolicy Class.................................................... 73
Figure 61. ActionStateNodeEditPolicy Class............................................................ 74
Figure 62. ActionStateDirectEditPolicy Class ......................................................... 74
Figure 63. ActionStateEditPolicy Class...................................................................... 75
Figure 64. Command Package..................................................................................... 75
Figure 65. ActionStateAddCommand Class.............................................................. 76
Figure 66. ActionFlowCreateCommand Class ......................................................... 77
Figure 67. DeleteActionStateCommand Class ................................................................. 78
Figure 68. ChangeElementNameCommand Class .......................................................... 79
Figure 79. ElementMoveResizeCommand Class ............................................................ 79
Figure 70. Phase Breadown ............................................................................................ 96
Figure 71. Phase 1 .......................................................................................................... 96
Figure 72. Phase 2 .......................................................................................................... 97
Figure 73. Phase 3 .......................................................................................................... 97
Chapter 1. Vision Document

1. Introduction

1.1. Motivation

The motivation for this project comes from the need to develop a new version of the agentTool that is able to draw Interaction Diagrams such as Sequence, Activity and State Chart diagrams. The existing tool does not have the feature to draw activity diagrams. Also it couples the controller very tightly with the model and the view. In this version we plan to remove this tight coupling and provide the feature to be able to draw these two mentioned dynamic UML diagrams. agentTool III (Dynamic) will be an independent tool, which will be lately integrated with a similar tool being built for static diagrams.

1.2. agentTool

The agentTool allows agent system designers to formally specify the required structure and behavior of a multiagent system and semi-automatically synthesize multiagent systems that meet those requirements. The system designer formally defines high-level system behavior graphically using our Multiagent Systems Engineering methodology. The system design defines the types of agents in the system as well as the possible communications that may take place between agents. This system-level specification is then refined for each type of agent in the system. To refine an agent, the designer either selects or creates agent architecture and then provides detailed behavioral specification for each component in the agent architecture.
2. **Project Overview**

![Diagram of Project Overview]

**Figure 1. Project Overview**

2.1. **Introduction**

The MSE Project involves the development of agentTool III (Dynamic) software that will help the users to draw dynamic diagrams. These diagrams include the sequence and activity diagrams.

Figure 1 shows at the highest level, how the agentTool will work. It is designed to generate the sequence and activity diagrams using the object model. It is supposed to save the models when the user wants to. This action generated the XML code for the model. Also, when a user imports a particular model, it should display the corresponding diagrams by parsing that XML file and loading its contents in the tool.
2.2. **Goal**
To develop a tool that enables the users to draw the sequence, activity and state chart diagrams. These behavioral diagrams help the user understand the flow of data and actions in an efficient way.

2.3. **Purpose**
To improve on the existing tool by adding more features to it and making the controller loosely coupled with the model and the view.
3. Requirements Specification

3.1. Use Case 1 – Draw Diagrams

3.1.1. Description

This use case describes the capability of drawing the diagrams. It includes three other use cases as follows:
3.1.1.1. **SR1 (Critical Requirement) - Use Case 1.1 - Draw Sequence Diagrams**

This use case describes the capability of drawing sequence diagrams. Sequence diagrams describe interactions among entities in terms of exchange of messages over time. The user of the tool should be able to draw such diagrams to represent the interactions taking place among different entities of an organization. It shows the message sequence that is shared among these entities in the order it is supposed to occur.

The ability to draw all these entities mentioned below used in a sequence diagram should be provided by the tool.

**Frame:** A frame is a rectangular box that delimits the sequence diagram. The purpose of this frame is to encapsulate all the elements used in the interaction protocol as a unit.

- **SR1.1** – Each sequence diagram will automatically have exactly one frame when created. The frame will have a default protocol name.
- **SR1.2** – The user cannot delete this frame. He can only resize the frame.

**Protocol Name:** Each frame has a notation to it that follows the `sd` keyword by a unique name. `sd` stands for sequence diagram. This notation is placed in a “snipped-corner” pentagon in the upper-left corner of the frame. It represents the protocol name. These protocols may have parameters.

- **SR1.3** – Edit the protocol name.
- **SR1.4** – add/edit parameters

![Frame and Protocol Name](http://example.com/frame.png)

*Figure 3. Frame and Protocol Name*
**Alternative Actions:** This frame depicts a set of alternative actions that can be taken by roles. This looks similar to the frame but always comes inside a frame. Dashed horizontal lines will separate the alternate actions.

![Alternative Action Diagram](image)

**Figure 4. Alternative Action**

**SR1.5** – Add a frame for alternative action  
**SR1.6** – Delete the frame  
**SR1.7** – Move it in the main frame  
**SR1.8** – Edit the name

**Action in loop:** This is denoted by a rectangular box similar to an alternative action box. It has a guard in the upper left corner, which determines when the loop exists.

**SR1.9** – Add a frame for loop  
**SR1.10** – Delete the frame  
**SR1.11** – Move it in the main frame  
**SR1.12** – Edit the guard

**Reference to another Protocol:** The call to another protocol is shown by a solid-outlined rectangle with a snipped-corner pentagon with the keyword `ref` (for reference). The content of the rectangle is the name of another interaction protocol.
Figure 5. Reference to Another Protocol

SR1.13 – Add a frame for reference to another protocol
SR1.14 – Delete the frame
SR1.15 – Move it in the main frame
SR1.16 – Edit the name of the reference

Class Roles: Class roles describe the way an object (subsystem/actor, agent and external system roles) will behave in context. These are drawn across the top of the diagram. Notation is a rectangle containing Role Name: Class/Object Name.

SR1.17 – Add class roles to the frame
SR1.18 – Delete/edit Class Roles

Lifelines: Lifelines are vertical lines that indicate the object's presence over time.

SR1.19 – Lifelines will be added with the class roles automatically with the class roles.

Messages: Messages handle the communication between objects. These are denoted by labeled horizontal arrows. It shows the direction of the flow of information. There are two types of messages.
**Synchronous messages:** Synchronous messages are messages with a filled arrowhead.

![Figure 6. Synchronous Message](image)

**Asynchronous messages:** Asynchronous messages are denoted by open arrowheads.

![Figure 7. Synchronous Message](image)

**Messages to Self:** A message can be self looped too. It can start at a lifeline and finish at the same lifeline. In this case the sender may or may not receive the message. These two conditions will be denoted separately as shown below.

![Figure 8. Message that the sender receives](image)

![Figure 9. Message that the sender will not receive](image)

**SR1.20** – Add messages between any two lifelines  
**SR1.21** – Delete messages  
**SR1.22** – Add labels to the messages  
**SR1.23 (Future Requirement)** – Message overlapping will be represented by making a bridge between the messages.
Stop: The stop operator is depicted by a cross in the form of an X at the bottom of a lifeline as shown below. It denotes that the specific role no longer interacts in the diagram.

\[ \times \]

Figure 10. Stop Sign

SR1.24 – Add Stop sign to a lifeline
SR1.25 – Remove stop sign from a lifeline

**Timing Constraints:** Time on sequence diagrams allows designers to represent that some messages have to be received before a certain delay. The message to which the timing constraint is applied has to be received between the lower bound and the upper bound. Lower and upper bounds can be either natural numbers such as \{0..3\} (the message has to be received between now and 3 units of time). Relative time constraints are written as an interval as follows: \{ initial time..final time\}. The interval is rendered as a horizontal bar on the first message in the interval, a horizontal bar on the last message in the interval, a vertical line directed in both ways between the two bars and the timing constraints near the vertical directed line. Absolute time constraints respect this notation except that the interval can be a date or a time or any combination of these two elements.

**SR1.26 (Future Requirement)** – Add timing constraints between two messages
**SR1.27 (Future Requirement)** – Remove timing constraints

3.1.1.2. SR2 (Critical Requirement) - Use Case 1.2 - Draw Activity Diagrams
This use case describes the capability of drawing activity diagrams. An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation. The user should be able to draw these diagrams to model the activity flow. The ability to draw all these entities mentioned below used in a sequence diagram should be provided by the tool.

**Frame:** A rectangular box inside which the activity diagram will be drawn.
**SR2.1** – Each activity diagram will automatically have exactly one frame when created.
Swim Lanes: These are vertical lines in the frame to depict different roles participating in the activity diagram.

SR2.2 – Add swim lanes for different roles in an activity diagram

SR2.3 – A swim lane should determine one role

Initial and Final nodes: The point of starting of an activity is called the initial state and similarly, where it stops is called its final state. The tool should provide the capability to draw these states.

Initial Node: A filled circle followed by an arrow represents the initial action node.

![Figure 11. Initial State](image1)

SR2.4 – Add initial node to the activity diagram.

SR2.5 – There will be no incoming action flow to the initial node.

SR2.6 – There will be only one initial node per diagram.

Final Node: An arrow pointing to a filled circle nested inside another circle represents the final action node. This is where all the processing the activity is stopped.

![Figure 12. Final State](image2)

SR2.7 – Add final node to the activity diagram.

SR2.8 – There will be no outgoing action flow from the final node.

SR2.9 – There will be only one final node per diagram.

Action States: Action states represent the non-interruptible actions of objects. You can draw an action state using a rectangle with rounded corners.
Figure 13. Action State

SR2.10 – Add action states
SR2.11 – Remove action states
SR2.12 – Add names to action states
SR2.13 – Edit names of the action states.

**Action Flow:** Action flow arrows illustrate the relationships among action states. Activities will be connected through activity flow arrows. These arrows may have labels on them.

Figure 14. Action Flow

SR2.14 – Add action flows between 2 activities
SR2.15 – Add action flows from an activity to a synchronization node, outgoing event or to a final state.
SR2.16 – Add action flows from an incoming event to an activity or synchronization node.
SR2.17 – Add action flows from a synchronization node to an activity, final node or outgoing event.

**Synchronization Points:** These are solid bars that are used for synchronization among activities.
SR2.18 – Add synchronization points
SR2.19 – Delete synchronization points

**Outgoing Events:** These are outgoing events from an activity or a synchronization point from an activity diagram. They are denoted by a pentagon as shown below.
Figure 15. Outgoing Event

SR2.20 – Add outgoing events
SR2.21 – Delete outgoing events
SR2.22 – Action flows cannot flow out of the outgoing events

Incoming Events: These are incoming events denoted by the pentagon as shown below.

Figure 16. Incoming Event

SR2.23 – Add incoming events
SR2.24 – Delete incoming events
SR2.25 – Action flows cannot flow in the incoming events

Decision Node: A decision node is a control node that chooses between outgoing action flows. One and only one action flow is chosen. These have guards on them that let only one action flow to be chosen. The following diamond denotes it.

Figure 17. Decision Node

SR2.26 – Add decision nodes
SR2.27 – Remove decision nodes
SR2.28 – Add guards to outgoing action flows
SR2.29 – Edit these guards

Flow Final Node: This denotes that a specific flow has stopped but all the other processing does not stop here. A cross in a circle as shown below represents it.
Figure 18. Flow Final Node

SR2.30 – Add flow final node
SR2.31 – Remove flow final node
SR2.32 – There will be no outgoing action flows from the flow final node.

3.1.1.3. SR3 (Future Requirement) - Use Case 1.3 - Draw State Charts
A statechart diagram shows the behavior of classes in response to external stimuli. This diagram models the dynamic flow of control from state to state within a system. The ability to be able to draw state charts will be a future requirement.

State: States represent situations during the life of an object. States are represented by rectangular boxes.

![State Diagram](image)

Figure 19. State

SR3.1 – Add states to the state chart diagram.
SR3.2 – Remove states
SR3.3 – Add/edit names on the states

State with Compartments: A state may be subdivided into multiple compartments separated from each other by a horizontal line.

<table>
<thead>
<tr>
<th>State Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry / do this</td>
</tr>
<tr>
<td>exit / do this</td>
</tr>
<tr>
<td>activity label / do this</td>
</tr>
</tbody>
</table>

Figure 20. State with Compartments
The compartments of a state are:

**Name Compartment:** This compartment holds the (optional) name of the state, as a string.

**Internal Activities Compartment:** This compartment holds a list of internal activities or state (do) activities that are performed while the element is in the state. The activity label identifies the circumstances under which the activity specified by the activity expression will be invoked. A number of activity labels are reserved for various special purposes and, therefore, cannot be used as event names. The following are the reserved activity labels and their meaning:

- **entry:** This label identifies an activity, specified by the corresponding activity expression, which is performed upon entry to the state (entry activity).
- **exit:** This label identifies an activity, specified by the corresponding activity expression, that is performed upon exit from the state (exit activity).
- **do:** This label identifies an ongoing activity (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the activity expression is completed (the latter may result in a completion event being generated).

**Internal Transition Compartment:** This compartment contains a list of internal transitions, where each item has the form as described for Trigger. Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the activity expression through the current event variable.

**SR3.4** - Add compartments to states

**SR3.5** – Delete compartments from states

**SR3.6** - Add activity labels

**SR3.7** – Edit/delete activity labels
**Initial and Final nodes:** The tool should provide the capability to draw object’s initial and final states.

**Initial Node:** A filled circle followed by an arrow represents the initial action node.

![Figure 21. Initial Node](image)

**SR3.8** – Add initial node to the state chart diagram.

**SR3.9** – There will be no incoming transitions to the initial node.

**SR3.10** – There will be only one initial node per diagram.

**Final Node:** An arrow pointing to a filled circle nested inside another circle represents the final action node.

![Figure 22. Final Node](image)

**SR3.11** – Add final node to the state chart diagram.

**SR3.12** – There will be no outgoing transitions from the final node.

**SR3.13** – There will be only one final node per diagram.

**Transition:** A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it.

![Figure 23. Transition](image)

**SR3.14** – Add transitions from one state to another or to a synchronization point.

**SR3.15** – Remove transitions

**SR3.16** – Add labels to the transition

**SR3.17** – Edit/remove the labels
SR3.18 – Change start and end-points of a transition.

**Synchronization Points:** These are solid bars that are used for synchronization of states.

![Synchronization Point](image)

**Figure 24. Synchronization Point**

SR3.19 – Add synchronization points
SR3.20 – Delete synchronization points

**Splitting of Control:** This represents a splitting of control that creates multiple states.

![Splitting of Control](image)

**Figure 25. Splitting of Control**

SR3.21 – Add splitting of control points
SR3.22 – Delete splitting of control points

3.1.2. **Stimulus/response sequence**

The user selects the option of being able to draw sequence/activity/state diagram. The tool displays the drawing tools necessary for the same. The user can select icons and drag and drop them in the drawing pane as per his requirements and annotate them. A description will be attached with every entity.

3.1.3. **Associated Functional Requirements**

**SR3.23 (Critical Requirement) - System Object Model Generation:** Corresponding to each user action for every diagram, the system object model will be modified by the tool in the background.
3.2.1. **SR4 (Critical Requirement) - Use Case 2 – Import XML Models**

3.2.2. **Description**
This use case describes importing XML representations of the environment model into the tool.

3.2.3. **Stimulus/response sequence**
There must be an existing environment model that the user needs to import. A menu item will be provided to import files. This file will be in XML format describing the organizations sequence and activity diagrams. When the user selects to import a file, he will be provided with a dialog box to help him select the file he needs. When loading, the tool will create the appropriate object model from the XML file format.

3.2.4. **Associated Functional Requirements**

3.2.4.1. **SR4.1 (Critical Requirement) - Loading Diagrams from an XML file**
After importing the file, when the user selects to view the sequence/activity/state chart diagrams, the selected diagram is shown in its respective diagram pane.

3.3. **SR5 (Critical Requirement) - Use Case 3 – Exporting XML Models**

3.3.1. **Description**
This use case describes the feature of exporting the environment models in XML file format.

3.3.2. **Stimulus/response sequence**
A menu item will be provided that enables the user to export/save the models. On saving a model, XML code will be generated corresponding to the object model generated by the tool and saved in a file. The user will be provided with a dialog box that will let the user mention the name of the XML file and the location where he wants to save it.

3.3.3. **Associated Functional Requirements**

**SR5.1 (Critical Requirement) -** The system model will be saved to a file.

**SR5.2 (Critical Requirement) –** The file will be in an XML compatible format.

3.4. **SR6 (Critical Requirement) - Use Case 4 – View System**

3.4.1. **Description**
This use case describes the capability of the user to view a listing of the entire system diagrams and browse through each system diagram.
3.4.2. **Stimulus Response Sequence**
The user loads the system model from a previously saved model file. The tool then displays a textual hierarchical structure of each diagram saved in the system.

3.4.3. **Associated functional requirements**

**SR6.1 (Critical Requirement)** - The user will be able to click on any diagram in the hierarchy to view or edit it.

**SR6.2 (Critical Requirement)** - The user can delete diagrams or create new ones as part of the loaded system.

3.5. **SR7 (Regular Requirement) - Use Case 5 – Printing Diagrams**

3.5.1. **Description**
This use case describes the functionality of being able to print the two diagrams using the printer.

3.5.2. **Stimulus/response sequence**
An icon will be provided to the user to be able to print the diagram it has selected using the appropriate diagram tab. When the user has select sequence diagram, a corresponding print icon will be displayed for it. Similarly, it will also show for the activity diagram. When the user hits the icon, a window will open up displaying the print options such as number of copies, etc. The other options will be determined in the future versions of the requirements specification. On selecting ‘OK’ on this window, printout will appear at the printer terminal.

3.5.3. **Associated Functional Requirements**

3.5.3.1. **SR7.1 (Regular Requirement) - Printing Diagrams on paper**
As mentioned above, a separate print icon will appear with every diagram that appears in its respective pane.

3.5.3.2. **SR7.2 (Regular Requirement) - Select the printer**
The user should be able to select the type of the printer he wants to use for printing. A dialog box will appear for this purpose.

3.5.3.3. **SR7.3 (Future Requirement) - Scale the diagrams to fit the paper**
The diagrams will be made to fit on the paper. The user will have the option to select this in the dialog box mentioned above. By default, the diagram will be printed as it appears on the panel.
4. Assumptions
   • For the tool to work the user should have JVM 1.3.1 or later installed on his system.

5. Constraints
   • Since the language used for coding will be Java, speed will be an issue. But this is required to make the tool platform independent.
   • The tool only deals with dynamic diagrams. The static diagrams are will be developed as another project.
   • The underlying object model for the project will be common to the static tool i.e. agentTool III (Static) and the dynamic tool i.e. agentTool III (Dynamic). The object model will have to be agreed upon by the designers of both tools.

6. Environment
   • The tool will be written in SWT and compiled in the Eclipse environment using the Eclipse GEF framework and Draw2D.
Chapter 2. Project Plan

1. Tasks Breakdown

1.1. Inception Phase

The inception phase focuses on defining the requirements specification. For this purpose, a vision document will be created which gives an overview of the project and elaborates the requirements for the software. The major use cases are defined and elaborated in the requirements analysis.

A project plan will be developed in the inception phase that gives the schedule of all activities required to complete the project. A software quality plan is also laid out in this phase to ensure that the quality of the product is maintained throughout the development life cycle. A prototype is also developed with the motive to show the feasibility and the look and feel of the project.

The inception phase completes when the prototype along with the other documents are approved by the committee.

1.2. Elaboration Phase

In the elaboration phase, the architecture of the software will be finalized. The documents from phase I will be reviewed and completed as desired by the committee. A test plan would be developed which will mention the complete testing process including the reporting of bugs and solving them.

Formal requirements specification will be generated for a part of the project, if not for the entire system. The language used for this would be OCL. Another prototype would be developed to demonstrate more features in the tool. Two technical inspectors will review the design in this phase and report with their findings.

The elaboration phase will be marked as ‘complete’ when the committee reviews the documents and approves of the prototype, with or without changes.

1.3. Production Phase

The production phase deals with the complete implementation and testing of the software. In this phase the user will develop the code and make sure that it is fully
documented. The code will be tested entirely to check if all the requirements are met. The testing results will be documented. A user manual will also be created that aids the user to install and use the tool efficiently. A component design that describes the system at low level will also be developed.

The production phase is marked as ‘complete’ when the committee reviews the documents and approves of the entire source code. The final presentation should be given at the end of the phase.

The Gantt chart below gives a schedule for the completion of the above-defined tasks for each phase. (You can increase the screen view size in percentage to view this clearly)

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement Document</td>
<td>24 days</td>
</tr>
<tr>
<td>Design Plan</td>
<td>4 days</td>
</tr>
<tr>
<td>Development</td>
<td>23 days</td>
</tr>
<tr>
<td>Test Plan</td>
<td>6 days</td>
</tr>
<tr>
<td>Present 1</td>
<td>1 day</td>
</tr>
<tr>
<td>Elaboration Plan</td>
<td>3 days</td>
</tr>
<tr>
<td>Revision Requirements Specification</td>
<td>20 days</td>
</tr>
<tr>
<td>Architecture Design</td>
<td>15 days</td>
</tr>
<tr>
<td>Test Plan</td>
<td>14 days</td>
</tr>
<tr>
<td>Technical Inspection</td>
<td>7 days</td>
</tr>
<tr>
<td>Component Design</td>
<td>20 days</td>
</tr>
<tr>
<td>Present 2</td>
<td>1 day</td>
</tr>
<tr>
<td>Production Plan</td>
<td>67 days</td>
</tr>
<tr>
<td>Component Design</td>
<td>7 days</td>
</tr>
<tr>
<td>Test Plan</td>
<td>37 days</td>
</tr>
<tr>
<td>Test Plan</td>
<td>20 days</td>
</tr>
<tr>
<td>Assessment Evaluation</td>
<td>6 days</td>
</tr>
<tr>
<td>Technical Notes</td>
<td>5 days</td>
</tr>
<tr>
<td>User Manual</td>
<td>10 days</td>
</tr>
<tr>
<td>Project Evaluation</td>
<td>10 days</td>
</tr>
<tr>
<td>Present 3</td>
<td>1 day</td>
</tr>
</tbody>
</table>

Figure 26. Project Schedule

2. Cost Estimate

2.1. COCOMO

This model is based on Barry Boehm's Constructive Cost Model (COCOMO). This is the top-level model, Basic COCOMO, which is applicable to the large majority of software projects. This is a simple on-line cost model for estimating the number of person-months required to develop software. The model also estimates the development schedule in months and produces an effort and schedule distribution by major phases.

Boehm says: "Basic COCOMO is good for rough order of magnitude estimates of software costs, but its accuracy is necessarily limited because of its lack of factors to account for differences in hardware constraints, personnel quality and experience, use of
modern tools and techniques, and other project attributes known to have a significant influence on costs." Due to these facts, the estimate we find out may not be as accurate, but would give us a rough idea about the schedule of the project.

According to Boehm’s classification, this project falls under the ‘organic’ modes of development. The following equations are used to calculate effort ant time:

Effort = 3.2*EAF (Size) ^1.05
Time (in months) = 2.5(Effort) ^0.38

To calculate effort one needs to estimate the Size and EAF values. The Size is measured in KLOC. The EAF value stands for effort adjustment factor and is the product of 15 adjustment factors. Each adjustment factor is classified as very low, low, normal, high, or very high. The value of each adjustment factor lies within a range and the classification will determine where on the range the value will falls. The table below lists all the adjustment factors and their corresponding ranges.

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>EFFORT ADJUSTMENT FACTOR</th>
<th>RANGE</th>
<th>MY CLASSIFICATION AND VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELY</td>
<td>Required reliability</td>
<td>0.75 - 1.40</td>
<td>Normal</td>
</tr>
<tr>
<td>DATA</td>
<td>Database size</td>
<td>0.94 – 1.16</td>
<td>Very low</td>
</tr>
<tr>
<td>CPLX</td>
<td>Product complexity</td>
<td>0.70 – 1.65</td>
<td>High</td>
</tr>
<tr>
<td>TIME</td>
<td>Execution time constraint</td>
<td>1.00 – 1.66</td>
<td>Normal</td>
</tr>
<tr>
<td>STOR</td>
<td>Main storage constraint</td>
<td>1.00 – 1.56</td>
<td>Normal</td>
</tr>
<tr>
<td>VIRT</td>
<td>Virtual machine volatility</td>
<td>0.87 – 1.30</td>
<td>Low</td>
</tr>
<tr>
<td>TURN</td>
<td>Computer turnaround time</td>
<td>0.87 – 1.15</td>
<td>Low</td>
</tr>
<tr>
<td>ACAP</td>
<td>Analyst capability</td>
<td>1.46 – 0.71</td>
<td>High</td>
</tr>
<tr>
<td>AEXP</td>
<td>Applications experience</td>
<td>1.29 – 0.82</td>
<td>Normal</td>
</tr>
<tr>
<td>PCAP</td>
<td>Programmer capability</td>
<td>1.42 – 0.70</td>
<td>High</td>
</tr>
<tr>
<td>VEXP</td>
<td>Virtual machine experience</td>
<td>1.21 – 0.90</td>
<td>Normal</td>
</tr>
<tr>
<td>LEXP</td>
<td>Language experience</td>
<td>1.14 – 0.95</td>
<td>High</td>
</tr>
<tr>
<td>MODP</td>
<td>Use of modern practices</td>
<td>1.24 – 0.82</td>
<td>High</td>
</tr>
<tr>
<td>TOOL</td>
<td>Use of software tools</td>
<td>1.24 – 0.83</td>
<td>High</td>
</tr>
<tr>
<td>SCED</td>
<td>Required development schedule</td>
<td>1.23 – 1.10</td>
<td>High</td>
</tr>
</tbody>
</table>
The EAF value evaluated to 1.36. I estimated the size to be 3500 LOC based on the previous versions of the tool. The effort evaluates to:

\[
\text{Effort} = 3.2 \times 1.36 \times 2.5^{1.05} = 11.4 \text{ staff months}
\]

The time can now be calculated as:

\[
\text{Time} = 2.5 \times 12.9^{0.38} = 6.3 \text{ months}
\]

We will not consider the Time we have calculated here because this calculation takes teamwork into account where there are many developers. This project will only have one developer with a little interaction initially with another developer to come up with the object model. The Effort figure may be large for such a project since the COCOMO model is designed for large projects with the involvement of several large teams in an industrial setting, so it tends to over-estimate this value. I would estimate that 8-9 staff months for Effort would be a more realistic value for this project.

3. Architecture Elaboration Plan

The following documents should be complete before the second presentation is made.

3.1. Revision of Vision Document
After the first presentation, changes as required by the committee should be made in the Vision Document. The major professor should approve these documents.

3.2. Revision of Project Plan
According to the changes suggested by the committee after the first presentation, the project plan should be modified to include those changes. These changes might be regarding the cost estimate or schedule. Also an extra section about the implementation plan should be added before the second presentation.

3.3. Architecture Design
Based on the use case diagram in the vision document and using other UML diagrams, an architecture design should be developed for the second phase. This design should be approved by the committee before proceeding to the actual complete implantation of the tool. It should also undergo formal technical inspection by two other MSE students.

3.4. Development of Prototype
This prototype will include the demonstration of the critical requirements identified in the Vision Document.
3.5. Test Plan
A complete test plan will be developed indicating the testing techniques used and the way bugs will be reported and solved. Unit testing, integration testing and system testing will be performed. This will be done to test whether all the requirements specified in the vision document are met or not.

3.6. Formal Technical Inspection
Deepti Gupta and Dominic Gelinas, two other MSE Students will review the architecture design produced by the developer and submit a report on their findings.

3.7. Formal Requirements Specification
The part of the project describing Sequence Diagrams will be will be formally specified using OCL. The tool used will be USE (UML-based Specification Environment). This will include operations like adding or deleting a class role, messages, alternative decisions etc. Messages would be checked for their source and destinations. Eg: A message to self should have the same source and destination.

4. Cost Estimate (based on current progress)
The project is now at the end of the Elaboration Phase. According to the time log I have spent 200 hours on the project. I have spent 150 hours coding/debugging/testing and 50 hours documenting. The prototype for the project has 5500 SLOC and it implements about 50% of the required features. From this data the following metrics can be calculated.

\[
\frac{5500 \text{ SLOC}}{150 \text{ hours}} = 36.67 \text{ SLOC/hour (productivity)}
\]

\[
\frac{5500 \text{ SLOC}}{.50} = 11000 \text{ SLOC (total SLOC)}
\]

The above calculations show that my productivity was 36.67 SLOC/hour and that there is about 11000 SLOC required for the project. There will be 5500 SLOC left for development. The following calculation estimates how much time will be required for the rest of the code development.

\[
\frac{5500 \text{ SLOC}}{36.67 \text{ SLOC/hour}} = 150 \text{ hours (total remaining development time)}
\]

\[
150 \text{ hours} / 7 \text{ hours/day} = 22 \text{ days}
\]
I would estimate that the remaining documentation will take about 70 hours (10 days). This would make the total time required for the rest of the project about 32 days. At a high level I will break down those days as follows:

Coding/Debugging/Testing + Java Docs – 25 days
Documentation including Component Design, Formal Inspection and References – 10 days

The Implementation Plan described below will provide a detailed WBS.

5. Implementation Plan

5.1 Deliverables

The following are the deliverables for presentation three.

User Manual
Component Design
Source Code
Assessment Evaluation
Project Evaluation
References
Formal Technical Inspection Letters
5.2. Work Breakdown Structure

The follow table breaks down the deliverables into tasks and lists the completion criteria and cost for each task.

<table>
<thead>
<tr>
<th>Deliverables</th>
<th>Tasks</th>
<th>Completion Criteria</th>
<th>Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source Code</strong></td>
<td>Activity Diagram Editor</td>
<td>Executable code</td>
<td>Mar 19</td>
<td>4 days</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagram Editor</td>
<td>Executable code</td>
<td>Apr 8</td>
<td>18 days</td>
</tr>
<tr>
<td></td>
<td><strong>JavaDocs</strong></td>
<td>Approved by Major Professor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JavaDocs</td>
<td>Approved by Major Professor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Assessment Evaluation</td>
<td>All test cases are executed</td>
<td>Apr 11</td>
<td>3 Days</td>
</tr>
<tr>
<td></td>
<td>Document Results</td>
<td>All results are documented</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>User Manual</strong></td>
<td>How to Install the Plugins</td>
<td>Approved by Major Professor</td>
<td>Apr 14</td>
<td>3 Days</td>
</tr>
<tr>
<td></td>
<td>How to use the tool</td>
<td>Approved by Major Professor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Component Design</strong></td>
<td>Design of Activity Diagram</td>
<td>Apr 19</td>
<td>5 Days</td>
</tr>
<tr>
<td></td>
<td>Design of Activity Diagram</td>
<td>All major features have been documented</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>References</strong></td>
<td>All References documented</td>
<td>Apr 21</td>
<td>2 Days</td>
</tr>
<tr>
<td></td>
<td>Received letters from technical inspectors</td>
<td>Approved by Major Professor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3. Software Quality Assurance Plan

1. Purpose
This document describes the steps that will be taken to ensure that agentTool III will achieve a high level of quality. The required documentation is also defined. This document follows the guidelines stated in the IEEE Standard for Software Quality Assurance Plans.

2. Reference Documents
- Vision Document
- Project Plan

3. Management

3.1. Organization
Supervisory Committee
Dr. Scott DeLoach
Dr. David Gustafson
Dr. William Hankley

Major Professor
Dr. Scott DeLoach

Developer
Binti Sepaha

Formal Technical Inspectors
Deepti Gupta
Dominic Gelines

3.2. Responsibilities
Supervisory Committee:
The supervisory committee will be responsible for attending the presentations given by the developer. After each presentation the committee members will provide feedback and suggestions regarding the software.
Major Professor:
The major professor will be responsible for supervisory committee duties and also meeting with the developer on a weekly basis to evaluate progress and provide suggestions.

Developer:
The developer is responsible for all documentation and software development tasks for the software. The project plan describes the tasks to be completed. The developer will also meet with the major professor on a weekly basis to report progress.

Formal Technical Inspectors:
The formal technical inspectors will be responsible for a technical review of the architecture design artifacts and the formal requirements specifications. They are required to submit a report on their findings.

3.3. Tasks
All tasks to be performed have been documented in the Project Plan 1.0. This will be reviewed after the first phase to incorporate any changes. A Gantt chart is included in the project plan to provide a schedule for each task.

4. Documentation
The documentation will consist of a vision document, project plan, software quality assurance plan, formal requirements specification, architecture design, test plan, formal technical inspection, prototype, user manual, component design, source code, assessment evaluation, project evaluation, references, formal technical inspection letters. All documentation will be reviewed by the committee members for final approval. All documentation will be posted on the developer’s web site at http://www.cis.ksu.edu/~binti/MSEProject/agentTool.htm

5. Standards, Practices, Conventions, and Metrics

Documentation Standard
IEEE standards will be used as a guideline to follow.

Coding Standard
The project will use traditional object oriented analysis and design methods. Recommended Java style guidelines will also be followed.

Commentary Standards
- Comments shall be used in the project to give a brief description of the code, focusing on the functionality and purpose of commented areas
• Each block of statements shall be well commented
• Each routine shall have comment consisting of one or two lines at the top of the routine, describing the purpose and limitations of that particular routine
• Each file, module and program shall contain the author’s name, date it was written or last modified, and a description of how this code fits into the end product. This might also include external functions that are referenced within the custom code.

Documentation
JavaDoc will be used for documenting the complete API for the project.

Metrics
Basic COCOMO will be used to estimate the effort and time for the project.

6. Reviews and Audits
The committee members will be conducting reviews of the documentation as well as evaluating the developer’s work at each presentation. They will also comment on the software prototype demonstration to suggest changes and additions to the requirements specifications. Deepti Gupta and Dominic Gelinas will evaluate the architecture design artifacts and report on their findings.

7. Test and Problem Reporting
All tests, along with their results, will be recorded on a time log of the project web page. Unresolved problems will be reported directly to the committee members.

8. Tools, Techniques, and Methodologies
The following tools will be used for coding, testing, and documentation.
• GEF Framework with Draw2d – for coding
• Eclipse with GEF Plug-in as IDE
• Microsoft Word and Visio– for documentation
• Microsoft Software Project 2003 – Project Planning
• USE 2.0.1 – for documentation and testing

9. Media Control
The software will be available on a CD-ROM ready for installation. The executable file will be recorded on it. A user manual soft copy will also be saved in the CD to aid with the installation process and use of the software. Documentation will be available from the developer’s website http://www.cis.ksu.edu/~binti/agentTool.htm
10. Records collection, maintenance and retention
The design documentation will be stored in the University library, the Major Professor and the developer. Entire source code, documentation and web pages for the project website will be submitted to the Major Professor in the form of a CD. This should also be stored with the developer.

11. Deliverables
The following is a set of deliverables in each phase.

Phase I
- Vision Document
- Project Plan
- Software Quality Assurance Plan
- Prototype Demonstration

Phase II
- Vision Document
- Project Plan
- Formal Requirements Specification
- Architecture Design
- Test Plan
- Formal Technical Inspection
- Executable Architecture Prototype

Phase III
- Component Design
- Source Code
- Assessment Evaluation
- User Manual
- Formal Technical Inspection Letters
- Project Evaluation
Chapter 4. Architecture Design

1. Introduction

The purpose of this document is to provide an architectural design for the Agent Tool (Dynamic) III. The design will show class diagrams and sequence diagrams. Each class will have a brief description about its purpose. The last section will provide a formal specification of the Activity Diagram Model.

A detailed definition of all classes can be found at:

http://www.cis.ksu.edu/~binti/MSEProject/Phase2Documents/javadoc/index.html

2. Activity Diagram Editor - Architecture

The agentTool is a graphical tool to create behavioral diagrams such as activity and sequence diagrams.

The tool will have an editor to place the objects. The user will be able to move the objects to the desired location. The following sections will describe the different packages of the agentTool in detail.

The agentTool III will be developed using the GEF (Graphical Editing Framework) technology from Eclipse. The Graphical Editing Framework (GEF) allows developers to create a rich graphical editor from an existing application model. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied to the model from the view.

3. GEF Overview

The Graphical Editing Framework allows users to develop graphical presentations for existing models. It is possible to develop feature rich graphical editors using GEF. All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT from eclipse.org. The editing possibilities of the Graphical Editing Framework allow one to build graphical editors for nearly every model. With these editors, it is possible to perform modifications to the model. e.g. changing model properties. All modifications to the model can be handled in a graphical editor using functions like drag and drop, copy and paste, and actions invoked from menus or toolbars.
Each diagram in agentTool III (Dynamic) will be an Eclipse plug-in in the form of a GEF Eclipse Editor. Eclipse provides an editor class called EditorPart. An editor is built by extending this class. This is the main class of the editor and is responsible for receiving the input, creating and configuring the viewer, handling the input and saving the input. It is the single entry point into the application code.

Each diagram in agentTool III (Dynamic) will have an Editor class: ActivityDiagramEditor and SequenceDiagramEditor.

4. MVC Approach

![MVC Approach in GEF](image)

**Figure 27. MVC Approach in GEF**

The controller is the bridge between the model and the view. The agentTool uses the Graphical editing Framework (GEF) MVC architecture. The controller is referred to as editpart. Every model has a figure and an editpart attached to it.

![Model and the View](image)

**Figure 28. Model and the View**
5. Activity Diagram Plug-in - Package View

![Diagram of Editor, Controller, View, Model and their relationships]

**Figure 29. Package View**

5.1 Editor Package

This package includes the classes required to initialize the editor and its palette.

5.1.1 Class Description

5.1.1.1 ActivityDiagramEditor

This class is the entry point to the application. An editor with a flyout palette is created here.

It initializes the palette, actions and the outline page. The save method is also defined here. It registers with the commandStacklistener as well.

5.1.1.2 ActivityDiagramPlugin

This class is responsible for making the application as a plugin. It defines the plugin id and with the help of plugin.xml file, it enables the application to be bundled as a plugin into the eclipse environment. The plugin.xml file contains the references for the required libraries that this application needs and the extension point of this application in the Eclipse GUI.

5.1.1.3 PaletteViewerCreator

This class is responsible for creating the palette and adding tool entries to it. It even associates a model class with every tool item.

5.1.1.4 PaletteFlyoutPreferences

This class sets the preferences for the palette, for example, when the palette should minimize, its width and height, etc.
5.1.1.5 ActivityDiagramPaletteViewerProvider
This class adds the property of dragging and dropping to the palette.

5.1.1.6 GraphicalViewerCreator
This class is the actual graphical viewer on which the user drops the diagrams.

5.1.1.7 OverviewOutlinePage
This class implements the outline page that appears in the editor. It gives a brief outline of the activity diagram.
5.2 Model Package

The model package is shown in Figure 4. All the model objects are subclasses of the class propertyAwareObject. The associations show that the diagram can have only one initial and final state but as many other objects it needs to have.
5.2.1 Class Description

5.2.1.1 PropertyAwareObject
This class is the super class for all model objects. This class makes the all the objects property aware i.e. they can fire events when their properties like bounds, location, name, etc. are changed.

5.2.1.2 Interface IModel
This interface is implemented by all the model objects and it defines all the methods common to all objects such as getters and setters for name, bounds, the activity diagram it belongs to, etc.

5.2.1.3 Activity
This class represents the Activity Diagram as a whole. All the model elements are added to this after creation. On deletion, these are removed from this class. This class is saved and loaded when the editor is opened.

5.2.1.4 ActionState
This class represents the Action State. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.

5.2.1.5 InitialState
This class represents the Initial State. It holds the necessary information related to the object such as name, location, size, outgoing action flows, etc.

5.2.1.6 FinalState
This class represents the Final State. It holds the necessary information related to the object such as name, location, size, incoming action flows, etc.

5.2.1.7 FlowFinalNode
This class represents the Flow Final Node. It holds the necessary information related to the object such as name, location, size, incoming action flows, etc.

5.2.1.8 OutgoingEvent
This class represents the Outgoing Event. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.
5.2.1.9 IncomingEvent
This class represents the Incoming Event. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.

5.2.1.10 SwimLane
This class represents the Swim Lane. Swim Lane is a vertical line which represents a single role.

5.2.1.11 DecisionNode
This class represents the Decision Node. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.

5.2.1.12 SynchronizationPoint
This class represents the Synchronization Point. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.

5.2.1.13 LabelTag
This class represents the Label Tag. It holds the label name, size and location.

5.2.1.14 ActionFlow
This class represents the Action Flow between objects. It holds the necessary information related to the object such as the source and the target, etc.

5.3 Controller Package
The GEF framework provides EditParts to assist development of the controller. Editpolicies and Commands assist the controller in communicating with the model and the view.

EditPart
An EditPart represents a single conceptual object with which the user can directly or indirectly interact. An EditPart generally directly represents something in the model. The EditPart itself is not visible to the user, but presents itself through the view. An editpart is completely responsible for graphical editing like resizing and moving parts within a layout, creating and editing connections and dropping parts inside other parts.
But the task of graphical editing is not implemented directly by EditParts. Instead, each EditPart installs one or more EditPolicies, each of which focuses on its own editing concern. The EditPart forwards edit requests to every installed EditPolicy.

**Edit Policy**
An EditPolicy provides a specific editing role to an EditPart. A Role might be something like "layout management". That policy's role is loosely defined by the Requests which it understands. An EditPart iterates over all of its EditPolicies to handle Requests. EditPolicies ignore the Requests that don't apply to them.

**Request**
GEF uses requests to communicate with an EditPart. The EditPart delegates all Requests to its installed EditPolicies.

GEF defines a common set of Requests, EditPolicies, and the Roles that those EditPolicies provide. These predefined entities can be used and/or extended for ease in development.

**Commands**
When the user interacts with EditParts, the underlying model is not manipulated directly by the EditParts. Instead, a Command is created that encapsulates the change. Commands can be used to validate the user's interaction, and to provide undo and redo support.

The editparts as designed for agentTool III (Dynamic) are shown in the following class diagram. An editpart is designed for every corresponding model element.

The controller in GEF consists of several packages as shown below.
As mentioned earlier, every model has an associated edit part. This edit part installs edit policies to play certain roles and are responsible for certain actions on the model object like moving or renaming it. These policies then create requests and associates requests.
with commands. When a particular type of request occurs, the control is passed to a command which then performs the desired action. Below are mentioned in detail, the three packages of the controller.

### 5.3.1 EditParts Package

![Diagram of EditParts Package](image)

**Figure 32. EditParts Package**

#### 5.3.1.1 Class Description

##### 5.3.1.1.1 PropertyAwarePart

This class extends the GEF EditPart class and is the super class for all editparts in our application. This class makes all the objects property aware i.e. they can fire events when their properties like bounds, location, name, etc. are changed.

##### 5.3.1.1.2 Interface IPart

This interface is implemented by all the editparts and it defines the methods common to all such as handle name change, bound change, get the associated figure and the model, etc.
5.3.1.1.3 ActivityDiagramPart
This class is the Activity Diagram Controller. All the other editparts are added to this part as its children. There is only a single instance of this part for the diagram and all the children are added to this common instance.

5.3.1.1.4 ActionStatePart
This class represents the Action State EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.5 InitialStatePart
This class represents the Initial State EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.6 FinalStatePart
This class represents the Final State EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.7 FlowFinalNodePart
This class represents the Flow Final Node EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.8 OutgoingEventPart
This class represents the Outgoing Event EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.9 IncomingEventPart
This class represents the Incoming Event EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.10 SwimLanePart
This class represents the Swim Lane EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.11 DecisionNodePart
This class represents the Decision Node EditPart. It is responsible for creating the model and the figure associated with this element.
5.3.1.1.12 SynchronizationPointPart
This class represents the Synchronization Point EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.13 LabelTagPart
This class represents the Label Tag EditPart. It is responsible for creating the model and the figure associated with this element.

5.3.1.1.14 PropertyAwareConnectionPart
This class is the super class for the ActionFlowPart. It makes the action flow property aware and enable to fire events when its target or source is changed, etc.

5.3.1.1.15 ActionFlowPart
This class represents the Action Flow EditPart. It creates the decoration for the end point of the connection.

All these editparts install their own policies to handle requests. These policies are mentioned below.

5.3.2 EditPolicies Package
5.3.2.1 Class Description
5.3.2.1.1 ActivityDiagramXYLayoutPolicy
This policy extends XYLayout Policy which handles the layout and creation of child figures in XYLayout. All the children added to the diagram are placed using this layout.

5.3.2.1.2 GraphicalNodeEditPolicy
This class is extended by all elements that can have action flows coming in and going out of them. It is responsible for creating and reconnecting connections graphically.

5.3.2.1.3 DirectEditPolicy
This policy shows DirectEdit feedback and creates the Command to perform a "direct edit". Direct Edit is when the user is editing a property of an EditPart directly (as opposed to in the Properties View). All elements having name tags on them extend this policy for letting the user directly change its name.
### 5.3.2.1.4 ComponentEditPolicy

A model-based EditPolicy for components within a container. A model-based EditPolicy only knows about the host's model and the basic operations it supports. A component is anything that is inside a container. By default, ComponentEditPolicy understands being DELETEd from its container, and being ORPHANed from its container. All elements that are added to the diagram have this policy installed. In this application, this policy creates a command to delete its associated element from the container i.e. the Activity Diagram.

These policies handle user requests by executing commands.

### 5.3.3 Commands Package

![Commands Package Diagram]

**Figure 33. Commands Package**

#### 5.3.3.1 Class Description

##### 5.3.3.1.1 Command

This is the super class for all the classes in this package. This class is provided by the GEF framework. It has a method canExecute() which checks whether this command can be executed or not. If it can be executed, the control passes to a method called execute().
Undo() and redo() methods are also a part of this class. The user has to provide implementations to all these methods in the subclass.

5.3.3.1.2 AddCommand
This command is executed when an element is dropped from the palette on to the diagram pane. The container policy for the activity diagram creates this command. This command is implemented for all the model objects.

5.3.3.1.3 DeleteCommand
This command is executed when an element is deleted from the diagram pane. The corresponding component edit policy for the model objects creates this command. This command is implemented for all the model objects.

5.3.3.1.4 ChangeElementNameCommand
This command is created by the objects direct edit policy when the user tries to change the name of certain elements.

5.3.3.1.5 ElementMoveResizeCommand
This command is invoked by the ActivityDiagramXYLayout policy when any element is moved or resized on the diagram pane.

5.3.3.1.6 ActionFlowCreateCommand
This command is invoked when an action flow is created between two elements. It is created and invoked by the ActionFlowNodeEditPolicy.

5.3.3.1.7 ReconnectActionFlowStartCommand
This command is invoked when an action flow’s source is changed.

5.3.3.1.8 ReconnectActionFlowEndCommand
This command is invoked when an action flow’s target is changed.
5.4 View Package

![Diagram of Figures Package]

Figure 34. Figures Package

5.4.1.1 ActivityDiagramFigure
This class represents the Activity Diagram Figure. This extends FreeformLayer so that it can extend when the diagram becomes larger. This can not be selected or deleted.

5.4.1.2 EditableLabelFigure
This class represents the editable label tag which is added to classes like ActionStateFigure, IncomingEventFigure, OutgoingEventFigure, etc.

5.4.1.3 ActionStateFigure
This class represents the Action State Figure. It is a rounded rectangular with an editable name label on it.

5.4.1.4 InitialStateFigure
This class represents the Initial State diagrammatically. It is a black circle and only action flows can flow out of it. There can be only one such instance in the diagram.
5.4.1.5 **FinalStateFigure**
This class represents the Final State diagrammatically. There can be only one such instance in the diagram. Action flows cannot flow out of it.

5.4.1.6 **FlowFinalNodeFigure**
This class represents the Flow Final Node diagrammatically. Action flows cannot flow out of it.

5.4.1.7 **OutgoingEventFigure**
This class represents the Outgoing Event diagrammatically.

5.4.1.8 **IncomingEventFigure**
This class represents the Incoming Event diagrammatically.

5.4.1.9 **SwimLaneFigure**
This class represents the Swim Lane diagrammatically. Swim Lane is a vertical line which represents a single role.

5.4.1.10 **DecisionNodeFigure**
This class represents the Decision Node diagrammatically. It is a diamond-shape figure.

5.4.1.11 **SynchronizationPointFigure**
This class represents the Synchronization Point diagrammatically. It is a solid bar which can be expanded.

5.4.1.12 **LabelTagFigure**
This class represents the Label Tag diagrammatically.

6. **Sequence Diagram Editor**
The same above architecture will be followed for the sequence diagram editor. It will be developed as a separate plugin too.
7. Sequence Diagrams
The user scenarios are explained below with the help of a few sequence diagrams.

7.1 Add an element
When a user adds an element from the diagram, the following behavior pattern is followed.

![Add an element – Sequence Diagram](image)

Figure 35. Add an Element – Sequence Diagram
7.2 Move an element

When a user moves an element on the diagram pane, the following behavior pattern is followed.

Figure 36. Move an Element – Sequence Diagram
7.3 Delete an element

When a user deletes an element from the diagram, the following behavior pattern is followed.

![Sequence Diagram](image-url)

**Figure 37. Delete an Element – Sequence Diagram**
8. USE Model

Included below is the formally specified USE Model for the Activity Diagram.

-----------------------------------------------
--------
model agentTool

--
-- CLASSES
--

class ActivityDiagram
attributes
  name : String

operations
  addActionState(a1 : ActionState)
  deleteActionState(a1 : ActionState)
end
class IModel
attributes
  name : String
  x : Integer
  y: Integer
  width: Integer
  height: Integer

operations
  actionFlowClosure(i : Set(IModel)) : Set(IModel) =
    if i->includesAll(i.target->asSet) then i
    else actionFlowClosure(i->union(i.target->asSet))
  endif
  reverseActionFlowClosure(i : Set(IModel)) : Set(IModel) =
    if i->includesAll(i.source->asSet) then i
    else reverseActionFlowClosure(i->union(i.source->asSet))
  endif
end
class ActionState < IModel
end
class InitialState < IModel
end
class FinalState < IModel
end
class FlowFinalNode < IModel
end
class IncomingEvent < IModel
end

class OutgoingEvent < IModel
end

class SynchronizationPoint < IModel
end

class DecisionNode < IModel
end

class SwimLane < IModel
end

--

--ASSOCIATIONS
--

association elements between
ActivityDiagram[*] role belongsTo
IModel[*] role contains
end

association actionFlow between
IModel[*] role source
IModel[*] role target
end

--

--CONSTRAINTS
--

constraints

-- An activity diagram has a unique name
context ad:ActivityDiagram
inv UniqueNames:
ad.contains->forAll(a1, a2 | a1 <> a2 implies a1.name <> a2.name)

-- Each activity should have at least one incoming and one outgoing ActionFlow
context i : IModel
inv atleastOneEntryOneExit:
  (ioclIsKindOf(ActionState) or ioclIsKindOf(SynchronizationPoint) or
   ioclIsKindOf(DecisionNode)) implies (i.target->size >= 1 and i.source->size >= 1)

-- InitialState and IncomingEvent cannot act as target for ActionFlow
context i : IModel
inv notAsTarget:
i.target->select(p | (poclIsKindOf(InitialState) or poclIsKindOf(IncomingEvent)))->
  >size=0
-- FinalState, FlowFinalNode and OutgoingEvent cannot act as source for ActionFlow
context i : IModel
inv notAsSource:
    i.source->select(p | (p.oclIsKindOf(FinalState) or p.oclIsKindOf(OutgoingEvent) or
    p.oclIsKindOf(FlowFinalNode)))->size=0

-- There can be only one InitialState in the system
context IModel
inv onlyOneInitialState:
    IModel.allInstances->select(p | p.oclIsKindOf(InitialState))->size = 1

-- There can be only one FinalState in the system
context IModel
inv onlyOneFinalState:
    IModel.allInstances->select(p | p.oclIsKindOf(FinalState))->size = 1

-- Each IncomingEvent should have at least one outgoing ActionFlow
context i : IncomingEvent
inv atleastOneEntry:
    (i.target->size >= 1)

-- Each OutgoingEvent should have at least one incoming ActionFlow
context i : OutgoingEvent
inv atleastOneExit:
    (i.source->size >= 1)

-- Each FlowFinalNode should have at least one incoming ActionFlow
context i : FlowFinalNode
inv atleastOneExitForFlowFinalNode:
    (i.source->size >= 1)

-- There should be at least one path from the initial state to the final state of the system
context i : InitialState
inv atleastOnePath:
    i.actionFlowClosure(i.target)->select(p | p.oclIsKindOf(FinalState) or
    p.oclIsKindOf(FlowFinalNode))->size >= 1

-- Final state should be reachable from each activity
context i : IModel
inv activityInPathToFinalState:
    (i.oclIsKindOf(ActionState) or i.oclIsKindOf(SynchronizationPoint) or
    i.oclIsKindOf(DecisionNode)) implies i.actionFlowClosure(i.target)
    ->select(p | p.oclIsKindOf(FinalState) or
    p.oclIsKindOf(FlowFinalNode))->size >= 1

-- Each activity should be reachable from the Initial State
context i : IModel
inv activityInPathFromInitialState:
    (i.oclIsKindOf(ActionState) or i.oclIsKindOf(SynchronizationPoint) or
    i.oclIsKindOf(DecisionNode)) implies i.reverseActionFlowClosure(i.source)
    ->select(p | p.oclIsKindOf(InitialState))->size >= 1
context ActivityDiagram::addActionState(a1 : ActionState)
  pre cond1 : self.contains->excludes(a1)
  post cond2 : self.contains = self.contains@pre->union(Set{a1})
  post cond3 : (self.contains - self.contains@pre)->size() = 1

context ActivityDiagram::deleteActionState(a1 : ActionState)
  pre cond1 : self.contains->includes(a1)
  post cond2 : self.contains = self.contains@pre->excluding(a1)
  post cond3 : (self.contains@pre - self.contains)->size() = 1
Chapter 5. Component Design

1. Introduction
This document will provide brief descriptions and class diagrams of the applications and classes for the Activity Diagram Editor. A detail description of the methods and attributes is provided in the Javadoc documentation.

2. Activity Diagram Editor
The activity diagram editor is an Eclipse plug-in that gives the user the capability to draw activity diagrams and save them as XML Models. Following is a description of each class used in this tool.

2.1 Package View

![Diagram]

**Figure 38. Package View**

The controller is the bridge between the model and the view. The agentTool uses the Graphical editing Framework (GEF) MVC architecture. The controller is referred to as editpart. Every model has a figure and an editpart attached to it. The editor package has everything required to instantiate the editor with the palette. It also includes the class that manages the outline view.

3. Editor Package
This package includes the classes required to initialize the editor and its palette.
3.1 ActivityDiagramEditor

This class is the entry point to the application. An editor with a flyout palette is created here. It initializes the palette, actions and the outline page. The save method is also defined here. It registers with the commandStacklistener as well.

```java
class ActivityDiagramEditor {
    public ActivityDiagramEditor() {
        // Initialize components and methods...
    }
}
```

Figure 39. ActivityDiagramEditor Class
Attributes and methods:

- actionRegistry: represents the editor’s action registry. It serves as a container for editor actions.
- editDomain: represents an editDomain associated with the application. An edit domain is an interface that logically bundles together an editor, viewers and tools
- editPartActionIDs: the list of action ids that represent editPart actions
- editActionIDs: the list of action ids that represent editor actions
- graphicalViewer: the graphical viewer associated with the editor.
- isDirty: represents whether the editor contents have been modified. If yes, the save option in the environment gets highlighted.
- overviewOutlinePage: the overview outline page associated with the application
- stackActionIDs: the list of action ids that represent command stack actions
- schema: the parent model object that holds all other objects
- doSave(): saves the model as an object stream. Also saves the model as an XML file
- doSaveAs(): our application does not support this functionality. Thus this method throws an UnsupportedOperationException.
- commandStackChanged(): listens for command stack changes
- setInput(): reads the input file specified by the user into the object model
- createGraphicalViewer(): creates, configures, registers and initializes a new graphical viewer to be associated with the application
- createActions: creates the undo, redo, save, print and delete actions and registers them with the Action Registry
- createPaletteViewer(): instantiates the palette for the editor
- setDirty(): sets isDirty variable when the editor’s state is unsaved
- isSaveAsAllowed(): returns whether saveAs is allowed or not

3.2 ActivityDiagramPlugin

This class is responsible for making the application as an Eclipse plugin. It defines the plugin id and with the help of plugin.xml file, it enables the application to be bundled as a plugin into the eclipse environment. The plugin.xml file contains the references for the required libraries that this application needs and the extension point of this application in the Eclipse GUI.
Attributes and methods:

- PLUGIN_ID: represents the plugin_id as in the plugin.xml file
- resourceBundle: the resource bundle object needed by Eclipse to run the application as a plugin.
- getWorkspace: returns the workspace instance associated with the plugin

3.3 PaletteViewerCreator

This class is responsible for creating the palette and adding tool entires to it. It even associates a model class with every tool item.

Attributes and methods:

- paletteRoot: represents the paletteRoot
- createPaletteRoot(): creates the paletteRoot
- getPaletteRoot(): getter for paletteRoot

3.4 PaletteFlyoutPreferences

This class sets the preferences for the palette, for example, when the palette should minimize, its width and height, etc.
### Figure 42. PaletteFlyoutPreferences Class

Attributes and methods:
- DEFAULT_PALETTE_WIDTH, PALETTE_DOCK_LOCATION, PALETTE_SIZE, PALETTE_STATE: represents the default palette preferences
- The other methods are getters and setters for these variables

### 3.5 ActivityDiagramPaletteViewerProvider

This class adds the property of dragging and dropping to the palette.

### Figure 43. ActivityDiagramPaletteViewerProvider Class

Attributes and methods:
- configurePaletteViewer(): configures the palette for the editor

### 3.6 GraphicalViewerCreator

This class is the actual graphical viewer on which the user drops the diagrams.
Figure 44. GraphicalViewerCreator Class

Attributes and methods:
- actionRegistry: represents the editor’s action registry. It serves as a container for editor actions.
- createGraphicalViewer(): creates the GraphicalViewer
- getEditPartFactory(): returns the instance of the EditPartFactory
- getViewer(): returns this viewer instance

3.7 OverviewOutlinePage

This class implements the outline page that appears in the editor. It gives a brief outline of the activity diagram.

Figure 45. OverviewOutlinePage Class

Attributes and methods:
- overview: represents the canvas on which the overview is drawn
• rootEditPart: instance of the root edit part

4. Model Package

The model package is shown in Figure 9. All the model objects are subclasses of the class PropertyAwareObject. The associations show that the diagram can have only one initial and final state but as many other objects it needs to have.
4.1 PropertyAwareObject

This class is the super class for all model objects. This class makes the all the objects property aware i.e. they can fire events when their properties like bounds, location, name, etc. are changed.

Figure 47. PropertyAwareObject Class

Attributes and methods:

- **BOUNDS, CHILD, INPUT, LAYOUT, LOCATION, NAME, OUTPUT, REORDER, TYPE, VISIBLE**: constant strings identifying the properties that can change.
- **listeners**: the PropertyChange listeners that participate in the event handling framework.
- **firePropertyChange()**: calls the firePropertyChange() method on the listeners.
- **addToXml()**: a common method for all model subtypes that adds their information to the XML file format.

4.2 Activity

This class represents the Activity Diagram as a whole. All the model elements are added to this after creation. On deletion, these are removed from this class. This class is saved and loaded when the editor is opened.
Figure 48. Activity Class
Attributes and methods:

- bounds: Represents an object that holds the width, height, x-co-ordinate and y-co-ordinate of the model element.
- There are several integer variables to keep a count of elements in the diagram.
- These variables have their respective getters and setters
- addButton(): adds an element to the diagram
- removeElement(): removes an element from the diagram
- writeXml(): this method creates the XML file to be written and calls addToXml() method for each of its children

4.3 ActionState

This class represents the Action State. It holds the necessary information related to the object such as name, location, size, outgoing and incoming action flows, etc.
Attributes and methods:

- **bounds**: Represents an object that holds the width, height, x-co-ordinate and y-co-ordinate of the model element.
- **location**: Is a point object that holds the initial location of the model element
- **name**: name associated with this model
- **initiators**: a list of action flows flowing out of this element
- **receivers**: a list of action flows flowing into this model
- **addActionFlowEnd()**: adds the end of an ActionFlow to this model
- **addActionFlowStart()**: adds the starting of an ActionFlow to this model
- **getActionInitiators()**: returns the list of initiators
- `getActionReceivers()`: returns the list of receivers
- `getName()`, `setName()`, `getBounds()`, `setBounds()`, `getSchema()`, `setSchema()`, `setLocation()`: these are the respective getters and setters for the variables
- `modifyBounds()`: handles the change in bounds of this model
- `modifyLocation`: handles the change in location of this model
- `modifyName()`: handles name change for his model

### 4.4 ActionFlow

This class represents the Action Flow between objects. It holds the necessary information related to the object such as the source and the target, etc.

![Figure 50. ActionFlow Class](image)

The other classes in this package follow the same declaration. There is a class for every model element as described in the Architecture Design 1.0 Document.

**Attributes and methods:**
- `name`: name associated with this action flow i.e. the message tag
- `initiator`: the action flow’s source
- `receiver`: the action flow’s target
- `getActionInitiator()`: returns the initiator
- `getActionReceiver()`: returns the receiver
- `setActionInitiator()`: sets the initiator
- `setActionReceiver()`: sets the receiver
- `getName()` and `setName()`: these are the getters and setters for name
- `modifyName()`: handles name change for his model
5. View Package

5.1 ActivityDiagramFigure

This class represents the Activity Diagram Figure. This extends FreeformLayer so that it can extend when the diagram becomes larger. This can not be selected or deleted.

Attributes and methods:

- `paintFigure()`: Used by the draw2d framework of GEF to render the figure representing an organization entity onto the drawing canvas.
- `addSwimLane()`: method to add a swim lane to the diagram
• deleteSwimLane(): method to delete a swim lane from the diagram

5.2 ActionStateFigure
This class represents the Action State Figure. It is a rounded rectangular with an editable name label on it.

![Figure 53. ActionStateFigure Class](image)

**Attributes and methods:**
- nameLabel: An editable label for the user to be able to change the name of the organization by direct editing.
- painFigure(): Used by the draw2d framework of GEF to render the figure representing an organization entity onto the drawing canvas.
- setSelected(): determines how the figure should appear when it is selected

The other classes in this package follow the same declaration. There is a class for every figure as described in the Architecture Design 1.0 Document.

6. Controller Package
The GEF framework provides EditParts to assist development of the controller. Editpolicies and Commands assist the controller in communicating with the model and the view.
6.1 EditParts Package

The diagram shows the class hierarchy of the EditParts Package.

6.1.1 PropertyAwarePart

This class extends the GEF EditPart class and is the super class for all editparts in our application. This class makes the all the objects property aware i.e. they can fire events when their properties like bounds, location, name, etc. are changed.
Attributes and methods:

- **activate**: activates the EditPart by activating all the edit-policies linked with it
- **commitNameChange**: empty function to be overridden by the subclass
- **deactivate**: deactivates the editpart by deactivating all the installed policies
- **handleBoundsChange**: empty function to be overridden by the subclass
- **handleLocationChange**: empty function to be overridden by the subclass
- **handleInputChange**: handles connection changes when a connection is made to an editpart where the editpart is the destination object
- **handleOutputChange**: handles connection changes when a connection is made from an editpart where the editpart is the source object
- **handleChildChange**: handles changes required when a child object is added to a parent object

### 6.1.2 ActivityDiagramPart

This class is the Activity Diagram Controller. All the other editparts are added to this part as its children. There is only a single instance of this part for the diagram and all the children are added to this common instance.

![ActivityDiagramPart Class](image)

**Figure 56. ActivityDiagramPart Class**
Attributes and methods:

- delegatingLayoutManager: manages the layout of object on the diagram
- activate(): activates the EditPart by activating all the edit-policies linked with it
- createEditPolicies(): installs all the edit policies for the editpart
- createFigure(): creates the figure object associated with the editpart
- deactivate(): deactivates the editpart by deactivating all the installed policies
- handleLayoutChange(): handles changes to the location of the children of this edit part
- isSelectable(): determines whether the object is selectable

6.1.3 ActionStatePart

This class represents the Action State EditPart. It is responsible for creating the model and the figure associated with this element.
Attributes and methods:

- **manager**: manages the direct-edit operations to be performed on the object name.
- **activate()**: activates the EditPart by activating all the edit-policies linked with it.
- **commitNameChange()**: handles the change in name when the user commits a direct edit operation.
- **createEditPolicies()**: installs all the edit policies for the editpart.
- **createFigure()**: creates the figure object associated with the editpart.
- **deactivate()**: deactivates the editpart by deactivating all the installed policies.
- **directEdiHitTest()**: checks if the user has clicked on the name of the object.
- **getModelSourceConnections()**: returns a list of Action Flow objects where the model associated with the edit part participates in the relation as a source object.
- **getModelTargetConnections()**: returns a list of Action Flow objects where the model associated with the edit part participates in the relation as a target object.
- **getSourceConnectionAnchor()**: returns a ChopBoxAnchor object used as a connection point when the editpart represents a source object.
- **getTargetConnectionAnchor()**: returns a ChopBoxAnchor object used as a connection point when the editpart represents a target object.
- **handleBoundsChange()**: handles changes to the bounds of the figure object associated with the edipart when the user drags the object around the canvas.
- **handleLocationChange()**: handles changes to the location of the figure object associated with the edipart when the user initially creates the object.
- **handleNameChange()**: handles changes to the object name when the user performs a direct edit operation.
- **performDirectEdit()**: delegates the direct edit request to the DirectEdit manager.
- **performRequest()**: checks the type of request received. If the request is of type direct-edit, calls performDirectEdit.
- **refreshVisuals()**: forces a refresh of this EditPart. All visuals properties are updated, as well as structural features like children.
- **revertNameChange()**: Reverts to the existing name in the model when exiting from a direct edit before a commit.
- **setSelected()**: sets the width of the line when the object is selected/de-selected.
- **getThisModel()**: return the model associated with this edit part.
6.1.4 PropertyAwareConnectionPart
This class is the super class for the ActionFlowPart. It makes the action flow property aware and enable to fire events when its target or source is changed, etc.

```
agenttool.dynamic.activity.part.PropertyAwareConnectionPart

activate () : void
deactivate () : void
getCastedModel () : agenttool.dynamic.activity.model.ActionFlow
propertyChange (evt : java.beans.PropertyChangeEvent) : void
```

Figure 58. PropertyAwareConnectionPart Class

6.1.5 ActionFlowPart
This class represents the Action Flow EditPart. It creates the decoration for the end point of the connection.

```
agenttool.dynamic.activity.part.ActionFlowPart

manager : org.eclipse.gef.tools.DirectEditManager
nameLabel : agenttool.dynamic.activity.figuresEditableLabel
activate () : void
commitNameChange (evt : org.eclipse.jface.util.PropertyChangeEvent) : void
createEditPolicies () : void
createFigure () : org.eclipse.draw2d.IFigure
deactivate () : void
directEditHitTest (requestLoc : org.eclipse.draw2d.geometry.Point) : boolean
getNameLabel () : agenttool.dynamic.activity.figuresEditableLabel
handleNameChange (value : java.lang.String) : void
performDirectEdit () : void
performRequest (request : org.eclipse.gef.Request) : void
refreshVisuals () : void
revertNameChange () : void
setSelected (value : int) : void
```

Figure 59. ActionFlowPart Class

Attributes and methods:
- manager: manages the direct-edit operations to be performed on the object name
- activate(): activates the EditPart by activating all the edit-policies linked with it
- commitNameChange(): handles the change in name when the user commits a direct edit operation.
• createEditPolicies(): installs all the edit policies for the editpart
• createFigure(): creates the figure object associated with the editpart
• deactivate(): deactivates the editpart by deactivating all the installed policies
• directEdiHitTest(): checks if the user has clicked on the name of the object.
• handleNameChange(): handles changes to the object name when the user performs a direct edit operation
• performDirectEdit(): delegates the direct edit request to the DirectEdit manager
• performRequest(): checks the type of request received. If the request is of type direct-edit, calls performDirectEdit
• refreshVisuals(): forces a refresh of this EditPart. All visuals properties are updated, as well as structural features like children.
• revertNameChange(): Reverts to the existing name in the model when exiting from a direct edit before a commit.
• setSelected(): sets the width of the line when the object is selected/de-selected.

The other classes in this package follow the same declaration as the ActionStatePart Class. There is a class for every edit part element as described in the Architecture Design 1.0 Document.

6.2 EditPolicies Package

6.2.1 ActivityDiagramXYLayoutPolicy
This policy extends XYLayout Policy which handles the layout and creation of child figures in XYLayout. All the children added to the diagram are placed using this layout.

![ActivityDiagramXYLayoutPolicy Class](image)

Figure 60. ActivityDiagramXYLayoutPolicy Class

6.2.2 GraphicalNodeEditPolicy
This class is extended by all elements that can have action flows coming in and going out of them. It is responsible for creating and reconnecting connections graphically.
6.2.3 DirectEditPolicy

This policy shows DirectEdit feedback and creates the Command to perform a "direct edit". Direct Edit is when the user is editing a property of an EditPart directly (as opposed to in the Properties View). All elements having name tags on them extend this policy for letting the user directly change its name.

Attributes and methods:
- oldValue: stores the old value of the name label in case the user discards/reverts the change
- getDirectEditCommand: returns the command to perform the direct edit operation. In our case, a ElementChangeNameCommand object will be returned

6.2.4 ComponentEditPolicy

A model-based EditPolicy for components within a container. A model-based EditPolicy only knows about the host's model and the basic operations it supports. A component is anything that is inside a container. By default, ComponentEditPolicy understands being DELETEd from its container, and being ORPHANed from its container. All elements that are added to the diagram have this policy installed. In this application, this policy
creates a command to delete its associated element from the container i.e. the Activity Diagram.

Figure 63. ActionStateEditPolicy Class

Every Edit Part has its own Direct Edit Policy if it has an editable name, Component Edit Policy and Node Edit policy if it acts as a node having the same declaration as the respective policies for ActionState.

6.3 Commands Package

Figure 64. Command Package

6.3.1 AddCommand
This command is executed when an element is dropped from the palette on to the diagram pane. The container policy for the activity diagram creates this command. This command is implemented for all the model objects.
Attributes and methods:
- activityDiagram: the activity diagram that this model element belongs to
- request: the request instance to create the object
- execute(): executes the command
- undo(): the undo version of this command
- setSchema(): sets the parent for this object

### 6.3.2 ActionFlowCreateCommand

This command is invoked when an action flow is created between two elements. It is created and invoked by the ActionFlowNodeEditPolicy.
### ActionFlowCreateCommand Class

<table>
<thead>
<tr>
<th>Attribute/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actionFlow:</td>
<td>The object to be created</td>
</tr>
<tr>
<td>actionFlowSource:</td>
<td>Source of the action flow</td>
</tr>
<tr>
<td>actionFlowTarget:</td>
<td>Target of the action flow</td>
</tr>
<tr>
<td>canExecute():</td>
<td>Returns true if the command can be executed, and false otherwise.</td>
</tr>
<tr>
<td>execute():</td>
<td>Executes the command</td>
</tr>
<tr>
<td>undo():</td>
<td>The undo version of this command</td>
</tr>
<tr>
<td>redo():</td>
<td>Redo version of the command</td>
</tr>
<tr>
<td>getActionResult()</td>
<td>Returns the source</td>
</tr>
<tr>
<td>getActionResult():</td>
<td>Returns the target</td>
</tr>
<tr>
<td>setActionResult():</td>
<td>Sets the source</td>
</tr>
<tr>
<td>setActionResult():</td>
<td>Sets the target</td>
</tr>
</tbody>
</table>

**Attributes and methods:**

- actionFlow: the object to be created
- actionFlowSource: source of the action flow
- actionFlowTarget: target of the action flow
- canExecute(): returns true if the command can be executed, and false otherwise.
- execute(): executes the command
- undo(): the undo version of this command
- redo(): redo version of the command
- getActionResult(): returns the source
- getActionResult(): returns the target
- setActionResult(): sets the source
- setActionResult(): sets the target

### 6.3.3 DeleteCommand

This command is executed when an element is deleted from the diagram pane. The corresponding component edit policy for the model objects creates this command. This command is implemented for all the model objects.
Attributes and methods:
- actionState: the element that has to be deleted
- execute(): executes the command
- undo(): the undo version of this command
- redo(): redo version of the command
- deleteRelationships(): deletes the action flows coming in and out of this element
- restoreRelationships(): restores the action flows coming in and out of this element

6.3.4 ChangeElementNameCommand
This command is created by the objects direct edit policy when the user tries to change the name of certain elements.
Figure 68. ChangeElementNameCommand Class

Attributes and methods:

- **modelObject**: the ModelElement object whose name is to be changed.
- **name**: the new name string.
- **oldName**: the original name string.
- **canExecute()**: returns true if the command can be executed, and false otherwise.
- **execute()**: executes the command.
- **undo()**: the undo version of this command.
- **setName()**: sets the name to be changed.

### 6.3.5 ElementMoveResizeCommand

This command is invoked by the ActivityDiagramXYLayout policy when any element is moved or resized on the diagram pane.

Figure 69. ElementMoveResizeCommand Class

Attributes and methods:

- **modelObject**: the ModelElement object whose name is to be changed.
- **newBounds**: the new bounds.
• oldBounds: the old bounds
• newSize: new size of the model element
• oldSize: old size of the model element
• execute(): executes the command
• undo(): the undo version of this command

Every Edit Part has its own Add and Delete Command having the same declaration as the respective commands for ActionState.

The same design is followed for the Sequence Diagram Editor.
Chapter 6. Test Plan

1. Test Plan Identifier
agentTool-Dynamic-V1.0

2. Introduction
This document describes the methods that will be used to test the entire functionalities of agentToolIII (Dynamic). The tool allows the user to draw behavioral diagrams like Sequence and Activity Diagrams. Each diagram will be treated as a separate module of the system. Each module will be tested with respect to the requirements as described in the vision document.

3. Test Items
The following core system modules will be tested
- Drawing Sequence Diagrams
- Drawing Activity Diagrams
- Importing XML Models
- Exporting XML Models
- View System
- Printing Diagrams

4. Features to be tested
The following list of features will be tested for each diagram in agentTool III (Dynamic). The features reference the Specific Requirements (SR) outlined in the Vision document.

Drawing Sequence Diagrams (SR1)
Features mentioned in Vision Document 1.0 section 3.1.1.1 starting from SR1.1 to SR1.25

Drawing Activity Diagrams (SR2)
Features mentioned in Vision Document 1.0 section 3.1.1.2 starting from SR2.1 to SR2.32
SR3.23 System Object Model Generation - Vision Document 1.0 section 3.1.3
Importing XML Models (SR4)
SR4.1 Loading Diagrams from an XML file - Vision Document 1.0 section 3.2.4.1

Exporting XML Models (SR5)
SR5.1 The system model will be saved to a file - Vision Document 1.0 section 3.3.3
SR5.2 The file will be in an XML compatible format - Vision Document 1.0 section 3.3.3

View System (SR6)
SR6.1 The user will be able to click on any diagram in the hierarchy to view or edit it - Vision Document 1.0 section 3.4.3
SR6.2 The user can delete diagrams or create new ones as part of the loaded system - Vision Document 1.0 section 3.4.3

Printing Diagrams (SR7)
SR7.1 Printing Diagrams on paper – Vision Document 1.0 section 3.5.3.1
SR7.2 Select the printer - Vision Document 1.0 section 3.5.3.2
SR7.3 Scale the diagrams to fit the paper - Vision Document 1.0 section 3.5.3.3

5. Features not to be tested
The following future requirements will not be tested.

Drawing State Chart Diagrams (SR3)
Features mentioned in Vision Document 1.0 section 3.1.1.3 starting from SR3.1 to SR3.22
SR1.23 Message overlapping will be represented by making a bridge between the messages.
SR1.26 Add timing constraints between two messages - Vision Document 1.0 section 3.1.1.1
SR1.27 Remove timing constraints - Vision Document 1.0 section 3.1.1.1

6. Approach
Only functional black box testing will be performed to test the functionality of agentToolIII (Dynamic). The features describe how the user will interact with the system, so the testing will require the tester to interact with the system in the same way a typical
user would. To simulate a typical user’s actions a set of scenarios will be created which describe a set of actions to take in order to achieve a desired result. Each scenario will trace back to a requirement listed in the Vision Document 1.0. However, in order to test that the object model is being correctly maintained, debug statements will be entered into the code and debug logs will be inspected.

7. Item Pass/Fail Criteria
Test cases executed on agentToolIII (Dynamic) will pass if they meet the specific requirements as mentioned in the Vision Document. A test case will fail if any behavioral expectation is not met as described.

8. Suspension Criteria and Resumption Requirements
8.1. Suspension Criteria
If a test case fails, testing will be suspended for all dependent features. The failed test case will be logged into a test log along with a description of the failure.

8.2. Resumption Requirement
Test cases not dependent on the case in which a bug is reported will continue to be executed in parallel to bug fixing. Testing for the failed test case will resume after the bug has been identified and resolved.

9. Test Deliverables
9.1 Test Log
The Test Log will document all test cases and record if the test case passed or failed. A test case that fails will have the reasons for failure and suggested solutions documented.

10. Testing Tasks
10.1 Test Case 1 – SR1.1 - SR1.4 – Sequence Diagrams
Tasks
- Click on the Sequence Diagram option on the tree hierarchy.
- Resize the frame
- Delete the frame
- Edit the protocol name
- Add/Modify parameters to the protocol
Results expected

- A new sequence diagram panel should appear with a default frame and a protocol name.
- The user should be able to resize the frame.
- The frame cannot be deleted.
- The user should be able to modify the protocol name.
- Parameters should be added and modified as required by the user.

10.2. Test Case 2 – SR1.5 - SR1.16

Tasks

- To an existing sequence diagram, add an alternative action frame from the palette.
- Edit its name.
- Move it within the sequence diagram frame.
- Delete this frame.
- Repeat the same process for the ‘action in loop’ and ‘reference to another protocol’.

Results expected

- A new frame for alternative action should be added within the existing sequence diagram frame.
- The user should be able to edit the name.
- The user should be able to move this frame inside the existing sequence diagram.
- The frame should be deleted.
- The same results are expected for ‘action in loop’ and ‘reference to another protocol’.

10.3. Test Case 3 – SR1.17 - SR1.19

Tasks

- Click on the class role icon on the palette and drag it on the sequence diagram panel.
- Delete/Edit these class roles.

Results expected

- A new class role should be added to the sequence diagram frame.
- The user should be able to edit and delete it too.
- The life line should be attached to it when the user drags it.
10.4. Test Case 4 – SR1.20 - SR1.22
Tasks
• Select message icon from the palette and then click on lifelines of two class roles (same or different)
• Add/edit label to the message
• Delete this message
Results expected
• A message should be created from the first class role to the second
• The user should be able to add/edit label to the message
• The message should be deleted

10.5. Test Case 5 – SR1.24 - SR1.25
Tasks
• Select stop icon from the palette and then click on a lifeline of a class role
• Delete this stop sign.
Results expected
• The life line should be marked with a ‘X’ sign which denotes that this class role no longer interacts in the protocol
• The stop sign should be removed

10.6. Test Case 6 – SR2.1 – SR2.3 – Activity Diagrams
Tasks
• Click on the Activity Diagram option on the tree hierarchy.
• Resize the frame
• Add Swim lanes from the palette to this frame
• Associate a role name with the swim lane
• Delete the frame
Results expected
• A new activity diagram panel should appear with a default frame and a protocol name.
• The user should be able to resize the frame
• Swim lanes should be added to the frame
• A role name should be associated with the swim lane
• The frame cannot be deleted

10.7. **Test Case 7 – SR2.4 – SR2.9**

**Tasks**
- To an existing activity diagram, add an initial/final node from the palette
- Add another initial/final node
- Send a message from this node to an activity
- Send a message to this node from an activity
- Delete this node.

**Results expected**
- An initial/final node should be added to the existing activity diagram frame
- The user should not be able to another similar node
- The user should be able to only send a message from an initial node and not from a final node
- The user should not be able to send a message to an initial node but should be able to send one to a final node.
- The added node should be deleted

10.8. **Test Case 8 – SR2.10 – SR2.17**

**Tasks**
- To an existing activity diagram, add an action state from the palette
- Add/Edit its name
- Add an action flow from the palette between two action states
- Delete this action flow
- Delete this action state

**Results expected**
- A new action state should be added to the existing activity diagram frame.
- The user should be able to add/edit the name
- An action flow should be added between two action states
- The action should be deleted
- The action state should be deleted
10.9. Test Case 9 – SR2.18 – SR2.19

Tasks
- To an existing activity diagram, add a synchronization point from the palette
- Add action flows to and from the synchronization point to action states
- Delete it.

Results expected
- A synchronization point should be added to the existing activity diagram frame
- Action flows should be added to and from the synchronization points to action states
- The synchronization point should be deleted

10.10. Test Case 10 – SR2.20 – SR2.25

Tasks
- To an existing activity diagram, add an outgoing/incoming event from the palette
- Add action flows to and from the added event
- Delete the added event.

Results expected
- An outgoing/incoming event should be added to the existing activity diagram frame.
- Action flows can only be sent by incoming events and received by outgoing events and not vice-versa
- The added event should be deleted

10.11. Test Case 11 – SR2.26 – SR2.29

Tasks
- To an existing activity diagram, add a decision node from the palette
- Add/edit guards to the outgoing action flows
- Delete the decision node

Results expected
- A decision node should be added to the existing activity diagram frame
- The user should be able to add/edit the guards on the outgoing action flows
- The decision node should be deleted
10.12. **Test Case 12 – SR2.30 – SR2.32**

**Tasks**
- To an existing activity diagram, add a flow final node from the palette
- Add action flows to and from this node
- Delete the node.

**Results expected**
- A flow final node should be added to the activity diagram.
- Action flows should not be sent from a flow final node but only to it.
- The flow final node should be deleted


**Tasks**
- Bring up each diagram one by one.
- Drag and drop palette components onto the canvas
- Inspect the debug logs being generated when components are dragged and dropped onto the canvas

**Results expected**
- The debug logs should contain print statements detailing the changes to the object model being performed as components are dragged / dropped and changed.


**Tasks**
- Save a diagram by selecting the save option from the menu
- Inspect the generated XML file
- Reload the file into the tool

**Results expected**
- The tool should generate the XML model.
- The diagrams should reappear on loading the file as they were when save was initiated.

10.15. **Test Case 15 – SR6.1 – SR6.2 – View System**

**Tasks**
- Bring up a diagram by clicking on it in the tree hierarchy
- Create a new diagram by right clicking on the tree root
• Delete an existing diagram by right clicking on it

Results expected
• The selected diagram should appear on the canvas
• The tester should be able to create a new diagram. An empty canvas should appear for drawing the diagram
• The deleted diagram should no longer be visible in the hierarchy

10.16. Test Case 16 – SR7.1 – SR7.2 – Printing Diagrams

Tasks
• Bring up a diagram by clicking on the tree hierarchy
• Select the print button from the toolbar
• Select the required printer and then press OK

Results expected
• A printer dialog box should appear
• The printout should come out at the selected printer
Chapter 7. Assessment Evaluation

1. Introduction

This document presents the results of the functional testing. The test cases are in reference to the test cases defined in the Test Plan 1.0 from Phase 2.

Reference:
• Test Plan 1.0 - agentTool-Dynamic-V1.0

2. Test Plan Result Summary

<table>
<thead>
<tr>
<th>Test Case #</th>
<th>Test Unit</th>
<th>SR’s Tested</th>
<th>Results/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequence Diagram</td>
<td>1.1 - 1.4</td>
<td>Passed</td>
</tr>
<tr>
<td>2</td>
<td>“</td>
<td>1.5 - 1.16</td>
<td>Passed</td>
</tr>
<tr>
<td>3</td>
<td>“</td>
<td>1.17 – 1.19</td>
<td>Passed</td>
</tr>
<tr>
<td>4</td>
<td>“</td>
<td>1.20 – 1.22</td>
<td>Passed</td>
</tr>
<tr>
<td>5</td>
<td>“</td>
<td>1.23 – 1.25</td>
<td>Passed</td>
</tr>
<tr>
<td>6</td>
<td>Activity Diagram</td>
<td>2.1 – 2.3</td>
<td>Passed</td>
</tr>
<tr>
<td>7</td>
<td>“</td>
<td>2.4 – 2.9</td>
<td>Passed</td>
</tr>
<tr>
<td>8</td>
<td>“</td>
<td>2.10 – 2.17</td>
<td>Passed</td>
</tr>
<tr>
<td>9</td>
<td>“</td>
<td>2.18 – 2.19</td>
<td>Passed</td>
</tr>
<tr>
<td>10</td>
<td>“</td>
<td>2.20 -2.25</td>
<td>Passed</td>
</tr>
<tr>
<td>11</td>
<td>“</td>
<td>2.26 – 2.29</td>
<td>Passed</td>
</tr>
<tr>
<td>12</td>
<td>“</td>
<td>2.30 – 2.32</td>
<td>Passed</td>
</tr>
<tr>
<td>13</td>
<td>“</td>
<td>3.23</td>
<td>Passed</td>
</tr>
<tr>
<td>14</td>
<td>XML Model</td>
<td>4.1, 5.1 - 5.2</td>
<td>Passed</td>
</tr>
<tr>
<td>15</td>
<td>View System</td>
<td>6.1 – 6.2</td>
<td>Passed</td>
</tr>
<tr>
<td>16</td>
<td>Printing Diagrams</td>
<td>7.1 – 7.2</td>
<td>Passed</td>
</tr>
</tbody>
</table>
3. Test Case Result Details

**Drawing Sequence Diagrams (SR1)**

3.1 Test Case 1 – SR1.1 - SR1.4
This test case successfully passed. A new diagram panel appeared and the frame was resized as the diagram became larger.

3.2 Test Case 2 – SR1.5 - SR1.16
This test case passed. An alternative action frame, reference to another protocol and a loop action could be successfully added and edited.

3.3 Test Case 3 – SR1.17 - SR1.19
This test case successfully passed. A new class role was added and edited as desired. The lifeline was attached to the class role.

3.4 Test Case 4 – SR1.20 - SR1.22
Messages could be successfully added to two different class roles. Labels were added and edited successfully to these messages. Messages of two different types – synchronous and asynchronous were created.

3.5 Test Case 5 – SR1.24 - SR1.25
This test case passed successfully. A new stop sign could be added or removed successfully.

**Drawing Activity Diagrams (SR2)**

3.6 Test Case 6 – SR2.1 – SR2.3
This test case successfully passed. A new diagram panel appeared and the frame was resized as the diagram became larger.

3.7 Test Case 7 – SR2.4 – SR2.9
This test case passed. The nodes were added and the action flows were created as desired.

3.8 Test Case 8 – SR2.10 – SR2.17
This test case passed. An action state was added, deleted and edited.
3.9 Test Case 9 – SR2.18 – SR2.19
This test case passed. A synchronization point was added and deleted successfully.

3.10 Test Case 10 – SR2.20 – SR2.25
This test case passed. Incoming and outgoing events were added, deleted and edited.

3.11 Test Case 11 – SR2.26 – SR2.29
This test case passed. A decision node was added and deleted successfully.

3.12 Test Case 12 – SR2.30 – SR2.32
This test case passed. A flow final node was added and deleted successfully. Action Flows could not flow out of it.

This test case passed. The debug logs behaved as expected.

Importing Models (SR4) and Exporting XML Models (SR5)
3.14 Test Case 14 – SR4.1, SR5.1 and SR5.2 – XML Models
This test case passed. The XML files were generated successfully.

View System (SR6)
3.15 Test Case 15 – SR6.1 – SR6.2 – View System
This test case passed. Diagrams could be loaded in the panel by clicking on the tree hierarchy. A new diagram was created and an existing one was deleted successfully.

Printing Diagrams (SR7)
3.16 Test Case 16 – SR7.1 – SR7.2 – Printing Diagrams
This test case passed. Diagrams were printed on the selected printer successfully.

4 Problems Encountered
Some bugs were encountered while testing the tool. All the bugs encountered were resolved and the tool was re-tested. These bugs passed after the re-testing. The following is a list of bugs encountered and resolved:
4.1 Adding Tags to Messages/Action Flows
The message tags that are added to action flows were allowed to be left blank by the user. But once left blank, the user could not rename them again.

4.2 Resize on Name Change
When the names on elements were changed and made large, the names appeared to be truncated. This was due to fixed size of the object. The size of the object was then resized based on the size of the name.

4.3 Deletion of Swim Lanes
Once added, the swim lanes were not being deleted properly. This bug was fixed.

4.4 Message Position in Sequence Diagram
The messages in sequence diagram were getting re-positioned when the diagram became larger and the user scrolled.

4.5 Positioning of elements after scrolling
When the user scrolled down and created new objects, they were positioned at wrong locations. This bug was fixed by correcting the LayoutPolicy for the containers.

4.6 Resizing of Elements
Some elements in the Activity Diagram were not being resized properly. Even on resizing their size remained the same.

5 Summary
The tool was thoroughly tested based on the test cases outlined in the Test Plan. All test cases passed. Hence the tool is now ready for delivery.
Chapter 8. Project Evaluation

1. Introduction
This document will present a summary of my experiences throughout this MSE Project entitled “agentTool III (Dynamic)”.

2. Problems Encountered

2.1 Comprehending Project Requirements
Initially, understanding the project requirements was tough. Developing the Vision Document took a lot of efforts on my side for understanding the details of the sequence and activity diagrams that are specific to agent-oriented systems. Once the Vision Document was developed, the next problem was visualizing how the tool should be laid out and made very user-friendly. Designing the user interface was one of the most challenging parts of the project. But the toughest to implement was making this tool an eclipse plug-in.

2.2 Learning GEF (Graphical Editing Framework)
In the first phase, I started developing the GUI using Eclipse and Jigloo Plug-in. This was not very difficult but later as the project advanced, I realized that this was not enough for the type of functionality we wanted to provide for these plug-ins. The advanced editing capabilities could only be provided by something like GEF (Graphical Editing Framework). This framework is relatively new and there were not much examples and help online. Understanding the complex GEF had a very steep learning curve. It took almost a month understanding how to build an editor and to combine what parts to get the desired functionality.

2.3 Understanding Plug-in Development
Another tough task was to understand how plug-ins are developed. Eclipse plug-ins need to be built very carefully and require a lot of knowledge on the developer’s part. The learning curve for this was very steep too. But I found helpful manuals and some examples online which made it easier for me to develop the plug-ins.

2.4 Implementation of Some Features
Some features took a lot of time for their implementation. Especially the swim lanes part in the activity diagram editor required lot of efforts. There was a time when I thought that
I would not fulfill this requirement, but I kept trying and finally found a solution to it. Another feature was the frame that appears when a new diagram is created. But with constant efforts, I was able to implement this feature as well. There were other minor things as well that took time but learning all these implementation details was a great experience.

3. **Source Lines of Code**
The first estimate for SLOC was made during Phase 1 and it was estimated to be 3500 SLOC. This estimate was driven from the previous versions of the tool. I believe this estimate was low because I assumed this estimate when I was using Java with Jigloo. But when I moved to using GEF, the source became larger as it required much other fragments to be combined to build an editor. The next estimate was in Phase 2 and it was estimated to be 5500 SLOC per tool. This estimate was driven from the working Activity Diagram Plug-in which comprised 50% of the actual project. It gave me a better idea of how much code it would take to implement the features in Sequence Diagram Editor. These estimates were accurate as the actual SLOC match these estimates.

The following is a break down of the SLOC required for each plug-in:
- Activity Diagram Editor = **5500**
- Sequence Diagram Editor = **5500**
- Total SLOC = **11000**

4. **Project Duration**
The following table show the expected vs. actual completion times for each phase of the project.

<table>
<thead>
<tr>
<th>Phase #</th>
<th>Expected Finish Time</th>
<th>Actual Finish Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>October 20, 2004</td>
<td>November 1, 2004</td>
</tr>
<tr>
<td>2</td>
<td>February 25, 2005</td>
<td>March 15, 2005</td>
</tr>
<tr>
<td>3</td>
<td>April 25, 2005</td>
<td>April 25, 2005</td>
</tr>
</tbody>
</table>
The first 2 phases got delayed because of the steep learning curves involved in plug-in development and understanding GEF. The final phase was completed on time. Time spent in each phase is shown in the following chart:

Figure 70. Phase Breakdown

Following graphs show the breakdown of activities during each phase:

Figure 71. Phase 1
Figure 72. Phase 2

Figure 73. Phase 3
5. Lessons Learnt
This project was a great learning experience for me. It gave me good exposure to what it is like to work on a large project. I learnt a lot of new things, few of them being Eclipse, Plug-in development, GEF, generating Javadocs, etc. The most important thing I learnt is the time management and how one should effectively schedule his/her work to meet the deadlines. Another valuable lesson I will take away from this project is how thinking through the design will make you think more deeply about what you are building. One should not jump to coding without having a strong design in hand.

6. Future Work
A few features in the Vision Document were marked as future requirements such adding timing constraints to sequence diagrams and implementation of state chart diagrams. Other than these, the features that need to be implemented in future are:

- Adding a message to self
- Adding stop signs to loop and alternative frames. These can be done by making the container policies for the parent objects handle the creation of child like loops and alternative frames can contain tags.
Chapter 9. Inspection Checklist

1. Introduction
The purpose of this document is to provide a checklist for the technical inspectors of agentTool III (Dynamic). The checklist will be used to document the items which are to be inspected. The checklist will help the technical inspector to inspect the documents. The goal of the technical inspection is to aid the developer in checking for correctness and consistency with the architectural design and formal specification documents.

2. Items to be inspected
2.1. Class Diagrams
Sequence diagrams
Class descriptions
2.2. Formal Specification
USE model

The inspector will need to cross reference the Vision Document Version 1.0 for completing the inspection. It is available at http://www.cis.ksu.edu/~binti/MSEProject/agentTool.htm

3. Formal Technical Inspectors
Deepti Gupta
Dominic Gelinas
4. **Formal Inspection Checklist**

<table>
<thead>
<tr>
<th>Inspection Item</th>
<th>Pass/Fail/Partial</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The symbols used in the class diagrams conform to UML standards</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>2. The classes in the class diagrams have a corresponding description provided in the Architecture Design document</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>3. The description of the classes in the Architecture Design document are clear and adequate</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>4. The symbols used in the sequence diagrams conform to UML standards</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>5. The sequence diagrams are clear and understandable</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>6. The attributes in the USE model are compatible with the attributes of the corresponding class diagrams</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>7. The multiplicities in the USE model are reflected in the Class diagram</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>8. Classes in the Architecture Design (Activity Diagram) correspond to classes in the USE model</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>9. All model elements and relationships as outlined in the Vision Document (Section 3.1.1.2) are present in the Architecture Design document as classes</td>
<td>Pass</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 10. User Manual – Activity Diagram Editor

1. Introduction

This document explains how to install and use the Activity Diagram Editor Plug-in in Eclipse.

2. Installation

2.1 Required Software

- Java 1.4.2 or later ([http://java.sun.com/j2se/1.4.2/download.html](http://java.sun.com/j2se/1.4.2/download.html))
- Eclipse SDK (3.0 or later) available at [http://eclipse.org/downloads/index.php](http://eclipse.org/downloads/index.php). Unzip the Eclipse-archive into the directory of your choice – refer it to as “eclipsehome”. Run eclipse.exe (Windows) or eclipse.sh (Unix, Linux, etc.).
- GEF-All (3.0.0. or later) available at [http://www.eclipse.org/gef/](http://www.eclipse.org/gef/). This version includes everything required by the plug-in to run.
- Download the version of GEF that matches your installation of Eclipse
- Exit Eclipse and unzip it into the "eclipsehome" directory
- Restart Eclipse and GEF plug-in will appear

2.2 Required Files

- All the code and executables for running the software is included in ActivityDiagramEditor.zip. The zip file is contained in src.zip folder available at [http://www.cis.ksu.edu/~binti/MSEProject/Phase3Documents/src.zip](http://www.cis.ksu.edu/~binti/MSEProject/Phase3Documents/src.zip)

2.3 Installation Steps

- Unzip ActivityDiagramEditor.zip into the plugins folder in your “eclipsehome”
- JRE must be specified on the System Path.
3. Using Activity Diagram Editor
   - Open a New Project (any type) in Eclipse
• Right Click on the project and select
  New->Example…->GEF(Graphical Editing Framework)->ActivityDiagramEditor
The following wizard will appear.

New Example

Select a wizard
Wizard to create an empty or pre-populated Activity Diagram

Wizards:
- Examples
  - Example Creation Wizards
  - Example Visual Classes
  - GEF (Graphical Editing Framework)
    - Activity Diagram Editor
    - Flow Diagram
    - Logic Diagram
    - Schema Diagram Editor
    - Shapes Diagram
  - Java
A new file will open up in Eclipse with an extension `.activity`. You can change its name in the text area named File Name.
The editor will open up a new empty diagram.

Start by adding objects from the palette on the drawing area. You can either drag them or use ‘point and click’.

You can move these objects around.

Use features such as undo, redo, delete and save which appear on right click of mouse.

You can add connections between objects, add swim lanes, edit names of the objects and so many other things.
4. **Few Things to Understand**

- You can open as many `.activity` files as you want in a project
- **Swimlanes work in a different way** – Each time a swimlane is created, it is added at the end. When a swimlane is deleted, automatically the last one is deleted, so that the diagram does not have to be restructured
- The XML models will be created and saved in the project folder itself where the diagram is created.

An example of a sample XML document generated by the editor is shown below. It is a model for a diagram that contains an initial and final state, an action state and an outgoing event. This file includes details of all the objects in the diagram like name (if any) and bounds. It also includes information about action flows coming in out of these objects (if any).

```xml
<?xml version="1.0"?>
<ActivityDiagram>
  <InitialState>
    <bounds>
      <x>98</x>
      <y>97</y>
      <width>25</width>
      <height>25</height>
    </bounds>
  </InitialState>
  <ActionState>
    <name>action state 1</name>
    <bounds>
      <x>296</x>
      <y>222</y>
      <width>105</width>
      <height>55</height>
    </bounds>
  </ActionState>
  <FinalState>
    <bounds>
      <x>579</x>
      <y>506</y>
      <width>25</width>
      <height>25</height>
    </bounds>
  </FinalState>
</ActivityDiagram>
```
<OutgoingEvent>
    <name>outgoing event 1</name>
    <bounds>
        <x>567</x>
        <y>199</y>
        <width>124</width>
        <height>50</height>
    </bounds>
</OutgoingEvent>

<SwimLaneCount>0</SwimLaneCount>
</ActivityDiagram>
Chapter 11. User Manual – Sequence Diagram Editor

1. Introduction
This document explains how to install and use the Sequence Diagram Editor Plug-in in Eclipse.

2. Installation
2.1 Required Software

- Java 1.4.2 or later (http://java.sun.com/j2se/1.4.2/download.html)
- Eclipse SDK (3.0 or later) available at http://eclipse.org/downloads/index.php. Unzip the Eclipse-archive into the directory of your choice – refer it to as “eclipsehome".
  Run eclipse.exe (Windows) or eclipse.sh (Unix, Linux, etc.).
- GEF-All (3.0.0. or later) available at http://www.eclipse.org/gef/. This version includes everything required by the plug-in to run.
- Download the version of GEF that matches your installation of Eclipse
- Exit Eclipse and unzip it into the "eclipsehome" directory
- Restart Eclipse and GEF plug-in will appear

2.2 Required Files

- All the code and executables for running the software is included in SequenceDiagramEditor.zip. The zip file is contained in src.zip folder available at http://www.cis.ksu.edu/~binti/MSEProject/Phase3Documents/src.zip

2.3 Installation Steps

- Unzip SequenceDiagramEditor.zip into the plugins folder in your “eclipsehome”
- JRE must be specified on the System Path.
3 Using Sequence Diagram Editor

- Open a New Project (any type) in Eclipse
Right Click on the project and select
New->Example…->GEF(Graphial Editing Framework)->SequenceDiagramEditor
The following wizard will appear.
• A new file will open up in Eclipse with an extension `.sequence`. You can change its name in the text area named File Name.

• The editor will open up a new empty diagram
• Start by adding objects from the palette on the drawing area. You can either drag them or use ‘point and click’.
• You can move these objects around

• Use features such as undo, redo, delete and save which appear on right click of mouse.
• You can add different types of messages between class roles, edit names of the objects and so many other things.
• Enjoy using the tool.

4 Few Things to Understand
• You can open as many ‘.sequence’ files as you want in a project
• To move a message’s position, you need to select its starting or end point and drag it to wherever the new position is desired to be at.
• To move a frame, just drag it to move to the desired location. The frames can only be placed on the locations where there are no already placed objects.

• To resize a frame, just drag its side to achieve the desired behavior.

• If you need to place a frame inside another, first complete the actions on the frame being placed inside and then add the outer frame. Once the outer frame has been placed over the inner frame, the inner frame cannot be selected unless the outer frame is removed from its top.

• The XML models will be created and saved in the project folder itself where the diagram is created.

An example of a sample XML document generated by the editor is shown below. This file includes details of all the objects in the diagram like name (if any) and bounds. It also includes information about messages sent and received by the class roles (if any) and the message bounds.

```xml
<?xml version="1.0"?>
<SequenceDiagram name='sd: Protocol 1'>
  <ClassRole>
    <name>ABC</name>
    <bounds>
      <x>99</x>
      <y>60</y>
      <width>125</width>
      <height>755</height>
    </bounds>
    <messagesSent>
      <Message type='synchronous' hashCode=5825185>
        <tag>send</tag>
        <bounds>
          <x>177</x>
          <y>230</y>
        </bounds>
      </Message>
    </messagesSent>
    <messagesReceived>
      <Message type='asynchronous' hashCode=19426064>
        <tag>receive</tag>
        <bounds>
          <x>378</x>
          <y>337</y>
        </bounds>
      </Message>
    </messagesReceived>
  </ClassRole>
</SequenceDiagram>
```
<ClassRole>
  <name>XYZ</name>
  <bounds>
    <x>322</x>
    <y>60</y>
    <width>125</width>
    <height>757</height>
  </bounds>
  <messagesSent>
    <Message type='asynchronous' hashCode=19426064>
      <tag>receive</tag>
      <bounds>
        <x>378</x>
        <y>337</y>
      </bounds>
    </Message>
  </messagesSent>
  <messagesReceived>
    <Message type='synchronous' hashCode=5825185>
      <tag>send</tag>
      <bounds>
        <x>177</x>
        <y>230</y>
      </bounds>
    </Message>
  </messagesReceived>
</ClassRole>

<ReferenceFrame>
  <name>Protocol 3</name>
  <bounds>
    <x>44</x>
    <y>428</y>
    <width>445</width>
    <height>200</height>
  </bounds>
</ReferenceFrame>
References


Realsolve Solutions Ltd. 2004.


[5] Bo Majewski. A Shape Diagram Editor

[6] GEF Logic Example – an example plug-in
