Architecture Design

For agentTool III (Dynamic)

Version 1.0

Submitted in partial fulfillment of the requirements of the degree of MSE

Binti Sepaha
CIS 895 – MSE Project
Kansas State University
# Table of Contents

1. Introduction..........................................................................................................................................................3
2. Activity Diagram Editor - Architecture ...............................................................................................................3
3. GEF Overview .....................................................................................................................................................3
4. MVC Approach....................................................................................................................................................4
5. Activity Diagram Plugin - Package View ............................................................................................................4
   5.1 Editor Package.................................................................................................................................................5
   5.1.1 Class Description........................................................................................................................................5
6. Model Package....................................................................................................................................................6
   5.2.1 Class Description.......................................................................................................................................7
7. Controller Package............................................................................................................................................8
   5.3.1 EditParts Package.......................................................................................................................................11
   5.3.1.1 Class Description.................................................................................................................................11
   5.3.2 EditPolicies Package ................................................................. ............................................................................13
   5.3.2.1 Class Description................................................................................................................................13
   5.3.3 Commands Package ................................................................................................................................14
   5.3.3.1 Class Description.................................................................................................................................14
   5.4 View Package..............................................................................................................................................15
8. Sequence Diagram Editor ..................................................................................................................................17
9. Sequence Diagrams.............................................................................................................................................18
   7.1 Add an element..............................................................................................................................................18
   7.2 Move an element...........................................................................................................................................19
   7.3 Delete an element .......................................................................................................................................20
10. USE Model....................................................................................................................................................20
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 agentToolIII (Dynamic) will be an Eclipse plugin 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 agentToolIII (Dynamic) will have an Editor class: ActivityDiagramEditor and SequenceDiagramEditor.
4. MVC Approach

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.

5. Activity Diagram Plugin - 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 entires 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 Fig 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.

Fig 4. Model Package
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 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]

**Fig 6. 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 the 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

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
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.
7.2 Move an element

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

[Diagram showing the process]
7.3 Delete an element
When a user deletes an element from the diagram, the following behavior pattern is followed.

```
User      DeleteAction      EditPart    ComponentEditPolicy    DeleteCommand    Diagram
DeleteElement
DeleteRequest  Get Delete Command
DeleteCommand   DeleteCommand
DeleteCommand  Execute Command
Remove child
```

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 | (poclIsKindOf(FinalState) or poclIsKindOf(OutgoingEvent) or
     poclIsKindOf(FlowFinalNode)))->size=0

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

-- There can be only one FinalState in the system
context IModel
  inv onlyOneFinalState:
    IModel.allInstances->select(p | poclIsKindOf(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