Component Design

For agentTool III (Dynamic)

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 ............................................................................................... 4
   2.1 Package View ........................................................................................................ 4

3. Editor Package ................................................................................................................. 4
   3.1 ActivityDiagramEditor ...................................................................................... 4
   3.2 ActivityDiagramPlugin ...................................................................................... 6
   3.3 PaletteViewerCreator ......................................................................................... 7
   3.4 PaletteFlyoutPreferences ................................................................................... 7
   3.5 ActivityDiagramPaletteViewerProvider ......................................................... 8
   3.6 GraphicalViewerCreator ................................................................................... 8
   3.7 OverviewOutlinePage ......................................................................................... 9

4. Model Package ............................................................................................................... 10
   4.1 PropertyAwareObject ...................................................................................... 10
   4.2 Activity .............................................................................................................. 11
   4.3 ActionState ....................................................................................................... 13
   4.4 ActionFlow ....................................................................................................... 15

5. View Package ............................................................................................................... 15
   5.1 ActivityDiagramFigure .................................................................................... 16
   5.2 ActionStateFigure ............................................................................................ 17

6. Controller Package ...................................................................................................... 17
   6.1 EditParts Package ............................................................................................ 18
      6.1.1 PropertyAwarePart ................................................................................... 18
      6.1.2 ActivityDiagramPart ................................................................................ 19
      6.1.3 ActionStatePart ....................................................................................... 20
      6.1.4 PropertyAwareConnectionPart ............................................................. 21
      6.1.5 ActionFlowPart ....................................................................................... 22
   6.2 EditPolicies Package .......................................................................................... 23
      6.2.2 ActivityDiagramXYLayoutPolicy .......................................................... 23
      6.2.3 GraphicalNodeEditPolicy ...................................................................... 23
      6.2.4 DirectEditPolicy ..................................................................................... 24
      6.2.5 ComponentEditPolicy ............................................................................. 24
   6.3 Commands Package ............................................................................................. 25
      6.3.1 AddCommand .......................................................................................... 25
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

![Fig. 1 Package View](image)

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.

Fig 2. 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 entries 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.
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.

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.
Fig 7. 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.

Fig 8. 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 Fig 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.
Fig 10. 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.
Fig 11. 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
- `addElement()` 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.
### Fig 12. ActionState Class

**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
- **receiver**: 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.

![Fig 13. 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

![Diagram of Draw2D Figure]

### 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.

```java
agenttool.dynamic.activity.figures.ActivityDiagramFigure
gen
```

#### 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.

![ActionStateFigure Class Diagram]

Attributes and methods:

• nameLabel: An editable label for the user to be able to change the name of the organization by direct editing.

• paintFigure(): 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

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.

Fig 18. PropertyAwarePart Class
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 Diagram](image)

**Fig 19. 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.

![ActionStatePart Class](agenttool.dynamic.activity.part.ActionStatePart)

**Fig 20. ActionStatePart 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.
- `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 is 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.
6.1.5 ActionFlowPart

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

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.

Fig 23. 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.
Fig 26. 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

![Diagram showing the structure of the commands 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.
Fig 28. ActionStateAddCommand Class

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

#### 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
- getActionSource(): returns the source
- getActionTarget(): returns the target
- setActionSource(): sets the source
- setActionTarget(): 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.
Fig 30. DeleteActionStateCommand Class

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.
Fig 31. 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.

Fig 30. 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.