Component Design

For agentTool III (Static)

Version 1.0

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

Deepti Gupta
CIS 895 – MSE Project
Kansas State University
# TABLE OF CONTENTS

1. Introduction ........................................................................................................................................... 3
2. agentTool III (Static) .......................................................................................................................... 3
3. agentTool III – Package View ........................................................................................................... 3
4. agentTool III – Model Package ......................................................................................................... 3
5. agentTool III – View .......................................................................................................................... 7
6. agentTool III – Controller ................................................................................................................. 8
   6.1 EditPart ........................................................................................................................................ 8
   6.1 Edit Policy ..................................................................................................................................... 8
   6.1 Command ...................................................................................................................................... 9
7. Model – Controller communication ................................................................................................. 16
8. Miscellaneous Classes (Editor Package) .......................................................................................... 18
1. **Introduction**

The purpose of this document is to provide a component design for the agentToolIII (Static). The design will outline the internal design of each component.

2. **agentTool III (Static)**

The agentTool III allows agent system designers to 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 the Multiagent Systems Engineering methodology (MASE). The system design defines the types of agents in the system as well as the possible communications that may take place between agents. The designer can also specify system goals and roles that are responsible for achieving those goals.

3. **agentTool III – Package View**

![Diagram of MVC architecture]

The MVC architecture is inherent to GEF. The model package contains classes that represent the data to be displayed to the user. The view contains classes that serve as the primary representation for the model objects. The controller contains classes that help communication between the model and the view. The editor package is the main package and contains classes that are the main entry point into the application. Each package and the classes included in those packages are detailed below.

4. **agentTool III – Model Package**

The model represents the data to be displayed. It generally knows nothing about its view or the controller. The model may reference them only as listeners to its notification mechanism. GEF provides commands to manipulate the model. The model should be changed only using Commands.
The following class diagram shows the complete object model being used for agentTool III (Static). It includes all the model elements and relationships mentioned in the Vision Document.

**Model Element**

This class represents an element in the agentTool model. It is the super class for every primitive model element; Goal, Role, Organization, Capability, Actor. It holds common information like the element name, its location and bounds.

![Model Element Class Diagram](image)

**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.
• protocolDestRelationships: Is a list of ProtocolRelationship objects where the model element participates as the target entity.
• protocolSrcRelationships: Is a list of ProtocolRelationship objects where the model element participates as the source entity.
• addProtocolDestRelationships: Adds a ProtocolRelationship object to the protocolDestRelationships list
• addProtocolSrcRelationships: Adds a ProtocolRelationship object to the protocolSrcRelationships list
• writeXML: All sub-classes inherit this method to generate their own XML strings.

All model elements like Goals, Roles, Organizations, Capabilities and Actors inherit from the base class Model Element. These classes have a similar structure. The following is a detailed view of the Organization class. All other classes are similar.

**Organization**

This class represents an Organization. It holds information about the relationships between Goals and Organizations and Services and Organizations. All model element classes

```
Organization

- columns: ArrayList
- orgAchievesRelationships: ArrayList
- orgProvidesRelationships: ArrayList
- stereotype: String

Organization():
(name: String, schema: OrgSchema)
addColumn(): Column: void
addColumn(): Column, index: int: void
addOrgAchievesRelationship(): OrgAchievesRelationship: void
addOrgProvidesRelationship(): OrgProvidesRelationship: void
equals(): Object: boolean
columns(): List
getOrgAchievesRelationships(): List
getOrgProvidesRelationships(): List
removeColumn(): Column: void
removeOrgAchievesRelationship(): OrgAchievesRelationship: void
removeOrgProvidesRelationship(): OrgProvidesRelationship: void
switchColumn(): Column, index: int: void
writeXML(): String
```
Attributes and methods:

- columns: A list of column objects needed when the compact view is generated and elements related to the organization entity are embedded within the organization figure.
- orgAchievesRelationships: A list of AchievesRelationship objects where the organization participates as the source element and connects to goals.
- orgProvidesRelationships: A list of ProvidesRelationship objects where the organization participates as the source element and connects to services.
- stereotype: The string containing the word <<organization>> representing the stereotype used for the organization object.
- writeXML: generates the XML string for the organization object.

Relationship

This class represents a relationship in the agentTool model. It is the super class for every type of relationship: Plays, Possesses, Protocol. It holds common information like the relationship name.

```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
</tr>
<tr>
<td>stereotype</td>
<td>String</td>
</tr>
<tr>
<td>visible</td>
<td>boolean</td>
</tr>
</tbody>
</table>
```

Attributes and methods:

- stereotype: stereotype for the relationship. E.g. <<provides>>, <<plays>> etc.
- name: used for the protocol relationship. The user can edit the relationship name.
- visible: an attribute used for implementing the ChangeView functionality. Used for setting the visibility of the relationship. For the compact view, all relationships expect the protocol relationship and set invisible.
All relationships like plays, possesses, provides, achieves, protocol inherit from the base class Relationship. The following is a detailed view of the Plays Relationship class. This relationship joins an Agent with a Role. All other relationship classes have a similar structure.

**PlaysRelationship**

This class represents a relationship between an Agent and a Role. It holds information about the source Agent and the target Role. It inherits the stereotype field from its parent and assigns it to <<plays>>.

<table>
<thead>
<tr>
<th>PlaysRelationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent : Agent</td>
</tr>
<tr>
<td>relationshipType : int</td>
</tr>
<tr>
<td>role : Role</td>
</tr>
</tbody>
</table>

- **getAgent** : Agent
- **getRelationshipType** : int
- **getRole** : Role
- **setAgent** (a: Agent) : void
- **setRelationshipType** (t: int) : void
- **setRole** (r: Role) : void

**Attributes and methods:**

- **agent:** The agent object that represents the source of the relationship
- **role:** The role object that represents the target of the relationship
- **relationshipType:** An integer constant identifying the type of relationship.

5. **agentTool III – View**

The view serves as the primary representation for the model objects. GEF provides visual representations of model objects through draw2d Figures. Each model object has a corresponding view object in the View package. Each view object extends the primary draw2d object – Figure.

All model figures like GoalFigure, RoleFigure, OrganizationFigure, CapabilityFigure and ActorFigure inherit from the base class draw2d Figure supplied by GEF. These classes have a similar structure. The following is a detailed view of the OrganizationFigure class. All other classes are similar.

**OrganizationFigure**
This class is the visual representation of an Organization object. It holds information about the background and foreground colors to be shown to the user for an Organization object. It also consists of a graphical label <<Organization>> as the stereotype for an organization object.

Attributes and methods:
- **nameLabel**: An editable label for the user to be able to change the name of the organization by direct editing.
- **columnsFigure**: An empty rectangular figure used to embed all the relationships of an organization object when the user changes the view to a compact view.
- **stereotypeLabel**: A simple label holding the string <<organization>>
- **painFigure**: Used by the draw2d framework of GEF to render the figure representing an organization entity onto the drawing canvas.

6. **agentTool III – Controller**

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 **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.
An editpart is designed for every corresponding model element. All editparts like GoalPart, RolePart, OrganizationPart, CapabilityPart and ActorPart inherit from the base class GEF GraphicalEditPart supplied by GEF. These classes have a similar structure. The following is a detailed view of the OrganizationPart class. All other classes are similar.

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
- changeLocation: handles the change in location of an object during drag/drop operation performed by the user
- 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.
• getModelChildren: returns the list of children objects associated with the model object linked to this editpart
• getModelSourceConnections: returns a list of Relationship objects where the model associated with the edit part participates in the relation as a source object
• getModelTargetConnections: returns a list of Relationship 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 location of the figure object associated with the edipart when the user drags the object around the canvas
• handleNameChange: handes 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.

6.2 Edit Policy
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. An EditPolicy provides a specific editing role to an EditPart. The following edit policies are used in agentTool
1. XYLayout Policy

This policy handles the layout of all model elements onto the drawing canvas. It is responsible for moving model elements when the user initiates drag/drop requests.

**Methods:**
- `createAddCommand`: returns the Command to perform an Add operation with the specified child and constraint.
- `createChangeConstraintCommand`: Returns the Command to change the constraint of the specified child so as to move the object around the canvas.

2. Node Edit Policy

This policy handles manipulation of relationships between model elements. It receives connection creation requests from the GEF framework and performs validations against those requests. It allows only valid connections to be made depending on the source and target elements.

**Methods:**
- `getConnectionCreateCommand`: Returns the Command that represents the first half of creating a connection. Assigns the source object.
- `getConnectionCompleteCommand`: Returns the Command that will creates the connection. Assigns the target object.

3. Direct Edit Policy
This policy handles the direct edit requests made by the user. When the user wishes to change the name of model elements, the direct edit request is received by the Direct Edit Policy. It is responsible for providing the user with a direct edit box for editing the element 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 `ChangeNameCommand` object will be returned

### 6.3 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.

**GEFCommand**

All commands designed for agentTool inherit from the Command class provided by the GEF framework. Element add commands, move commands, delete commands, relationship create commands and change name commands, all inherit from the base class `Command`.

1. **ModelElementAddCommand**

A `ModelElementAddCommand` class is associated with every model element. E.g `AgentAddCommand`, `CapabilityAddCommand` etc. The AddCommand is responsible for adding the model element to the diagram. Following is detailed view of the `OrganizationAddCommand`. All other classes are similar.

```
OrganizationAddCommand
- index : int
- loc : Point
- org : Organization
- schema : OrgSchema

execute () : void
setIndex (i : int) : void
setLocation (l : Point) : void
setOrganization (o : Organization) : void
setSchema (schema : OrgSchema) : void
undo () : void
```
Attributes and methods:
• loc: represents the x-y location of the object
• org: the organization object to be added onto the canvas
• schema: the parent object representing the canvas onto which all objects are dragged/dropped
• execute(): this function executes the command by adding the org object to the model

2. ModelElementMoveCommand
A ModelElementMoveCommand class is responsible for moving a model element. This is useful when the elements are dragged on the drawing canvas. This class changes the bounds of the model element thus changing its position.

Attributes and methods:
• me: the modelElement object to be moved
• newBounds: the new location/size of the object
• oldBounds: the original location/size of the object

3. ModelElementDeleteCommand
A ModelElementDeleteCommand class is associated with every model element. E.g DeleteAgentCommand, DeleteCapabilityCommand etc. The DeleteCommand is responsible for deleting the model element from the diagram. It is also responsible for deleting all the relationships associated with the model element to be deleted. Following is detailed view of the DeleteOrganizationCommand. All other classes are similar
Attributes and methods:

- **org**: the Organization object to be deleted
- **orgAchievesRelationships**: a list of achieves relationships associated with the Organization object. These also need to be deleted when deleting the organization
- **orgProvidesRelationships**: a list of provides relationships associated with the Organization object. These also need to be deleted when deleting the organization
- **protocolDestRelationships**: a list of protocol relationships associated with the Organization object where the Organization participates as the destination object. These also need to be deleted when deleting the organization
- **protocolSrcRelationships**: a list of protocol relationships associated with the Organization object where the Organization participates as the source object. These also need to be deleted when deleting the organization
- **deleteRelationships()**: deletes all the above mentioned relationships associated with the Organization object.

### 4. RelationshipCreateCommand

A RelationshipCreateCommand class is associated with every relationship. E.g PlaysRelationshipCreateCommand, ProtocolRelationshipCreateCommand etc. The CreateCommand class is responsible for adding new relationships to the diagram. It is also
responsible for validating that the relationship does not already exist. The command executes only when the validations fall through. Following is a detailed view of the ProtocolRelationshipCreateCommand.

<table>
<thead>
<tr>
<th>ProtocolRelationshipCreateCommand</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest : ModelElement</td>
</tr>
<tr>
<td>relationship : ProtocolRelationship</td>
</tr>
<tr>
<td>relationshipType : Int</td>
</tr>
<tr>
<td>src : ModelElement</td>
</tr>
</tbody>
</table>

- canExecute(): determines if the relationship can be created. If the relationship is already present, it returns false. execute() method is called only when canExecute returns true.

### Attributes and methods:
- dest: the ModelElement object that acts as the destination of the protocol relationship
- src: the ModelElement object that acts as the source of the protocol relationship
- relationship: the protocol relationship object that is being created
- canExecute(): determines if the relationship can be created. If the relationship is already present, it returns false. execute() method is called only when canExecute returns true

5. **RelationshipDeleteCommand**

A RelationshipDeleteCommand class is associated with every relationship. E.g DeletePlaysRelationshipCommand, DeleteProtocolRelationshipCommand etc. The DeleteRelationshipCommand class is responsible for deleting relationships from the diagram. The following is a detailed view of the DeleteProtocolRelationshipCommand. All other classes are similar.
Attributes and methods:
- d: the ModelElement object that is the destination of the protocol relationship
- s: the ModelElement object that is the source of the protocol relationship
- relationship: the protocol relationship object that is being deleted

6. ChangeNameCommand
This class is responsible for modifying the name of the model element when the user directly edits the element name in the diagram.

Attributes and methods:
- me: the ModelElement object whose name is to be changed
- name: the new name string
- oldName: the original name string

7. Model – Controller communication
The model in the MVC architecture does not directly talk to the controller. All communication between the model and the controller is via listeners. The model fires event notifications that are received by the controller. When the underlying model changes, event notifications are sent to the controller. The controller then talks to the view in order to update the user interface.
Every model element inherits from an abstract base class called PropertyAwareObject in order to support the event notification mechanism.

The abstract base class contains a few String constants representing the types of model changes it knows about. It uses the `java.beans.PropertyChangeSupport` to provide the "plumbing" for the event handling. It also exposes methods like addPropertyChangeListener and removePropertyChangeListener which observers can use to register and deregister their interest in model changes. Finally, it includes a firePropertyChange() method which subclasses can use to trigger property events.

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.

Similarly every EditPart inherits from an abstract base class called PropertyAwarePart. This class registers and deregisters model listeners through its activate and deactivate methods. It provides base methods for handling model changes which can be overridden by EditPart subclasses to customize reactions to specific changes in the model.
Attributes and methods:

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

8. Miscellaneous Classes (Editor Package)

The following classes are needed for each agentToolIII (Static) diagram to work as an Eclipse plugin. This is the main package of the application.

**DiagramPlugin**

This class is responsible for making the application an Eclipse plugin. It defines the plugin id and with the help of the plugin.xml file, it enables the application to be bundled as a plugin into the eclipse environment.
The following is detailed view of the OrgDiagramPlugin class. Similar classes are present for the other three diagrams: GoalDiagramPlugin, AgentDiagramPlugin, RoleDiagramPlugin.

<table>
<thead>
<tr>
<th>Attr/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUGIN_ID</td>
<td>represents the plugin_id as in the plugin.xml file</td>
</tr>
<tr>
<td>resourceBundle</td>
<td>the resource bundle object needed by Eclipse to run the application as a plugin.</td>
</tr>
<tr>
<td>getWorkspace</td>
<td>returns the workspace instance associated with the plugin</td>
</tr>
</tbody>
</table>

**EditorClass**

This class is the entry point into the application.

The following is detailed view of the OrgDiagramEditor class for the Organization diagram plugin. Similar classes are present for the other three diagrams: GoalDiagramEditor, AgentDiagramEditor, RoleDiagramEditor.
Most of the attributes and methods in this class are GEF specific. They are used by the GEF framework to render the application as a graphical editor.
**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.