Architecture 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 – Architecture .................................................................................................................. 3
4. GEF Overview ........................................................................................................................................ 3
5. agentTool III – Package View .................................................................................................................. 4
6. agentTool III – Model ............................................................................................................................... 4
   6.1 Class Structure ................................................................................................................................... 5
7. agentTool III – View ................................................................................................................................. 7
   7.1 Class Structure ................................................................................................................................... 8
8. agentTool III – Controller ......................................................................................................................... 9
   8.1 Class Structure ................................................................................................................................... 10
   8.2 Commands ...................................................................................................................................... 11
9. Sequence Diagrams ................................................................................................................................. 13
   9.1 Deleting a model element .................................................................................................................. 13
   9.2 Moving a model element ................................................................................................................. 13
10. Miscellaneous Classes ........................................................................................................................... 14
11. USE Model ............................................................................................................................................ 16
1. **Introduction**

The purpose of this document is to provide an architectural design for the agentToolIII (Static). 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 agentToolIII (Static) application.

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 – Architecture**

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.

Each diagram in agentTool III (Static) will be developed as an Eclipse plugin. The same architecture as outlined below will be followed for each diagram.

4. **GEF Overview**

The Graphical Editing Framework allows to easily 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 like changing element 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 (Static) 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 (Static) will have an Editor class: GoalDiagramEditor, AgentDiagramEditor, OrganizationDiagramEditor, RoleDiagramEditor. These classes along with other GEF specific classes required for each diagram to run as an Eclipse plugin are further discussed in Section 10. These classes are contained in a package called Editor.

The following links provide a comprehensive overview of GEF and building UI applications using GEF:

5. agentTool III – Package View

![Controller Diagram]

The MVC architecture is inherent to GEF. Each package and the classes included in those packages are detailed below.

6. agentTool III – Model

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. The diagram
shows the hierarchical class structure of the model elements and relationships between different model elements.

6.1 Class Structure

Diagram
This class stores all the model elements and relationships that the user draws. It is the highest level class in the model and contains all model objects.

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.
Goal
This class represents a Goal. It holds the information about its connections/relationships to other goals as either sub goals or parent goals.

Role
This class represents a Role. It holds information about the relationships between Roles and Agents and Roles and Organizations.

Organization
This class represents an Organization. It holds information about the relationships between Roles and Organizations and Capabilities and Organizations.

Capability
This class represents a Capability. It holds information about the relationships between Capabilities and Agents and Capabilities and Organizations.

Agent
This class represents an Agent. It holds information about the relationships between Roles and Agents and Capabilities and Agents.

Actor
This class represents a UML Actor. It holds information about the protocol relationships existing between the Actor and the system

Service
This class represents a Service. It holds information about the relationships between Roles and Services, Organizations and Services, Agents and Services.

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.

PlaysRelationship
This class represents a relationship between an Organization/Agent and a Role. It holds information about the source Organization/Agent and the target Role. It also holds the stereotype <<plays>>.

PossessRelationship
This class represents a relationship between an Organization/Agent and a Capability. It holds information about the source Organization/Agent and the target Capability. It also holds the stereotype <<possesses>>.

AND/OR Relationship
This class represents a relationship between a parent Goal and a sub Goal. It holds information about the source parent Goal and the target sub Goal. It holds information about the relationship type: AND or OR.
RequiresRelationship
This class represents a relationship between a Role and a Capability. It holds information about the source Role and the target Capability. It also holds the stereotype <<requires>>.

AchievesRelationship
This class represents a relationship between an Organization/Role/Age/nt and a Goal. It holds information about the source Organization/Role/Agent and the target Goal. It also holds the stereotype <<achieves>>.

ProvidesRelationship
This class represents a relationship between an Organization/Role/Agent and a Service. It holds information about the source Organization/Role/Agent and the target Service. It also holds the stereotype <<provides>>.

Protocol Relationship
This class represents a protocol relationship. It holds information about the model elements it connects. Protocols can connect Actors, Organizations, Roles and Agents.

7. 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. The following class diagram shows the classes used in the View package along with the relationships between them.
7.1 Class Structure

**Draw2d Figure**
This class is the base implementation provided by GEF for graphical figures. It holds information used to display the visual part of the model.

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

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

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

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

**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.

**ActorFigure**
This class is the visual representation of an Actor. It displays the UML actor notation. It holds information about the background and foreground colors to be shown to the user for an Actor object.

**ServiceFigure**
This class is the visual representation of a Service. It holds information about the background and foreground colors to be shown to the user for a Service object. It also consists of a graphical label "<<Service>>" as the stereotype for an agent object.
8. 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.

**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 (Static) are shown in the following class diagram. An editpart is designed for every corresponding model element.
8.1 Class Structure

GEF GraphicalEditPart
This class is the base implementation provided by GEF for graphical editing. It holds information used to edit the visual part of the model.

GoalPart
This class installs the edit policies required for a Goal object. It creates the GoalFigure object associated with the model goal object.

RolePart
This class installs the edit policies required for a Role object. It creates the RoleFigure object associated with the model role object.

AgentPart
This class installs the edit policies required for an Agent object. It creates the AgentFigure object associated with the model Agent object.

CapabilityPart
This class installs the edit policies required for a Capability object. It creates the CapabilityFigure object associated with the model goal object.

OrganizationPart
This class installs the edit policies required for an Organization object. It creates the OrganizationFigure object associated with the model Organization object.
**ServicePart**
This class installs the edit policies required for a Service object. It creates the ServiceFigure object associated with the model Service object.

**ActorPart**
This class installs the edit policies required for an Actor object. It creates the ActorFigure object associated with the model Actor object.

**RelationshipPart**
This class installs the edit policies required for a Relationship object. It creates the relationship connectors associated with the corresponding model object. It also associates the connectors with label stereotypes depending on the relationship type.

**8.2 Commands**
The following generic command classes are designed to be used for achieving the editing capability required for agentTool III.

![Command Class Diagram](image)

**GEFCommand**
This is a class provided by the GEF framework. It is an abstract Command class implementation. This class can be extended to create new application specific commands. Hence all application specific command classes described below are subclasses of this class.

**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.
**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.

**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.

**RelationshipCreateCommand**
A RelationshipCreateCommand class is associated with every relationship. E.g PlaysRelationshipCreateCommand, PossessRelationshipCreateCommand 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.

**RelationshipDeleteCommand**
A RelationshipDeleteCommand class is associated with every relationship. E.g DeletePlaysRelationshipCommand, DeletePossessRelationshipCommand etc. The DeleteRelationshipCommand class is responsible for deleting relationships from the diagram.

**ChangeNameCommand**
This class is responsible for modifying the name of the model element when the user directly edits the element name in the diagram.
9. Sequence Diagrams
The following sequence diagrams show some of the main user scenarios for agentToolIII (Static)

9.1 Deleting a model element

The steps as shown in the following diagram are executed while deleting a model element. When the user tries to delete a model element, the GEF framework instantiates a DeleteAction. The DeleteAction then creates a DeleteRequest object and passes the Request onto the EditPart. The EditPart further delegates the Request to the ComponentEditPolicy. The EditPolicy then creates a DeleteCommandObject and returns it so that it reaches the DeleteAction. The DeleteAction then invokes the execute method on the Command object and the necessary delete functionality is completed by removing a child (i.e. the model element being deleted) from the Diagram.

9.2 Moving a model element
The steps as shown in the following diagram are executed when a user tries to move a model element by using drag and drop.
10. Miscellaneous Classes (Editor Package)

The following classes are needed for each agentTool III (Static) diagram to work as an Eclipse plug-in. This is the main package of the application.

EditorClass
This class is the entry point into the application. An Eclipse UI editor with a flyout palette is created here. It initializes the palette, actions and the outline page.

DiagramPlugin
This class is responsible for making the application an Eclipse plug-in. 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 plugin.xml file is an Eclipse specific file and contains the references for the required libraries that this application needs. It also contains other information like the extension point of the agentTool application in the Eclipse GUI.
PaletteViewerCreator
This class is responsible for creating the palette and adding tool entries to it. It even associates a model element with every tool item. Each model element appears as a new drawing component in the palette. Each relationship appears as a connecting tool.

PaletteViewerProvider
This class adds the functionality of dragging and dropping to the agentToolIII (Static) application

GraphicalViewerCreator
This class is the actual graphical viewer on which the user drags/drops the model elements and connects them with relationship connectors.

OverviewOutlinePage
This class implements the outline page that appears in the editor. It shows the overview of the editor canvas in the bottom left corner of the editor. The portion of the editor canvas that is in focus is visible in the overview page.
11. USE Model

-- This is the USE model for only part of agentTool. Specifically for Goal Diagram
-- Hence the model assumes that the diagram contains only Goals
-- Hence only the association between Goal and Diagram is included here and only the AND/OR
-- relationship is included.

model agentTool

--
-- CLASSES
--

class Diagram
end

class ModelElement
attributes
    name : String
    stereotype : String
    bounds : String
end

class Organization < ModelElement
end

class Goal < ModelElement
attributes
    currentChildNumber : String

operations
    addGoal(o : Diagram) : Set(Goal) = o.hasGoals->union(Set{self})
deleteGoal(o : Diagram) : Set(Goal) = o.hasGoals - (Set{self})
addSubGoal(o : Diagram, g: Goal, c: Relationship)
end

class Role < ModelElement
end

class Capability < ModelElement
end

class Agent < ModelElement
end

class Service < ModelElement
end

class Relationship
attributes
    name : String
    stereotype : String
    srcElement : ModelElement
    targetElement : ModelElement
end

class AndORRelationship < Relationship
attributes
    type : String
parentGoal : Goal
subGoal : Goal
end

--
-- ASSOCIATIONS
--

association GoalBelongsTo between
  Goal[*] role hasGoals
  Diagram[1] role belongsTo
end

association RelBelongsTo between
  Relationship[*] role hasRelationships
  Diagram[1] role belongsTo
end

association ANDORRel between
  Goal[1] role connects
  AndORRelationship[*] role isConnectedTo
end

--
-- CONSTRAINTS
--

constraints

--
-- Unique names. Goal Names should be unique
--
context o:Diagram
inv UniqueGoals:
o.hasGoals->forAll(g1, g2 | g1 <> g2 implies g1.name <> g2.name)

-- OPERATIONS

-- Adding a goal
-- Added object must be new
-- Object should be added to the list of goals in the diagram
--
context Goal::addGoal(o : Diagram): Set(Goal)
pre cond1 : o.hasGoals->excludes(self)
post cond2 : o.hasGoals = o.hasGoals@pre->union(Set{self})
post cond3 : result = o.hasGoals
post cond4 : (o.hasGoals - o.hasGoals@pre)->size() = 1

-- Deleting a goal
-- Object to be deleted must exist in the organization
-- Object should be deleted from the list of goals in the diagram
--
context Goal::deleteGoal(o : Diagram): Set(Goal)
pre cond1 : o.hasGoals->includes(self)
post cond2 : o.hasGoals = o.hasGoals@pre->excluding(self)
post cond3 : result = o.hasGoals
post cond4 : (o.hasGoals@pre - o.hasGoals)->size() = 1

-- Adding a sub goal
-- The goals to be connected via the parent-child relationship should both exist in the organization
-- The name of the sub-goal should become one more than the child number of the parent
-- The relationship type of all sub goals of the parent goal should be same.
-- AND / OR connections not allowed together.

context Goal::addSubGoal(o : Diagram, g: Goal, c: AndORRelationship)
    pre condpre1 : o.hasGoals->includes(self)
    pre condpre2 : o.hasGoals->includes(g)
    pre condpre3 : c.parentGoal = self

    post condpost1 : g.name = self.name->concat('.')->concat(self.currentChildNumber)
    post condpost2 : c.subGoal = g
    post condpost3 : AndORRelationship.allInstances->select(c1 | c1.parentGoal = self)->forAll(c2 | c2.type = c.type)