Developer's Manual

For agentTool 3

Kyle Hill
November 25, 2007
Kansas State University

Table of Contents

1. Introduction

The agentTool website describes agentTool as:

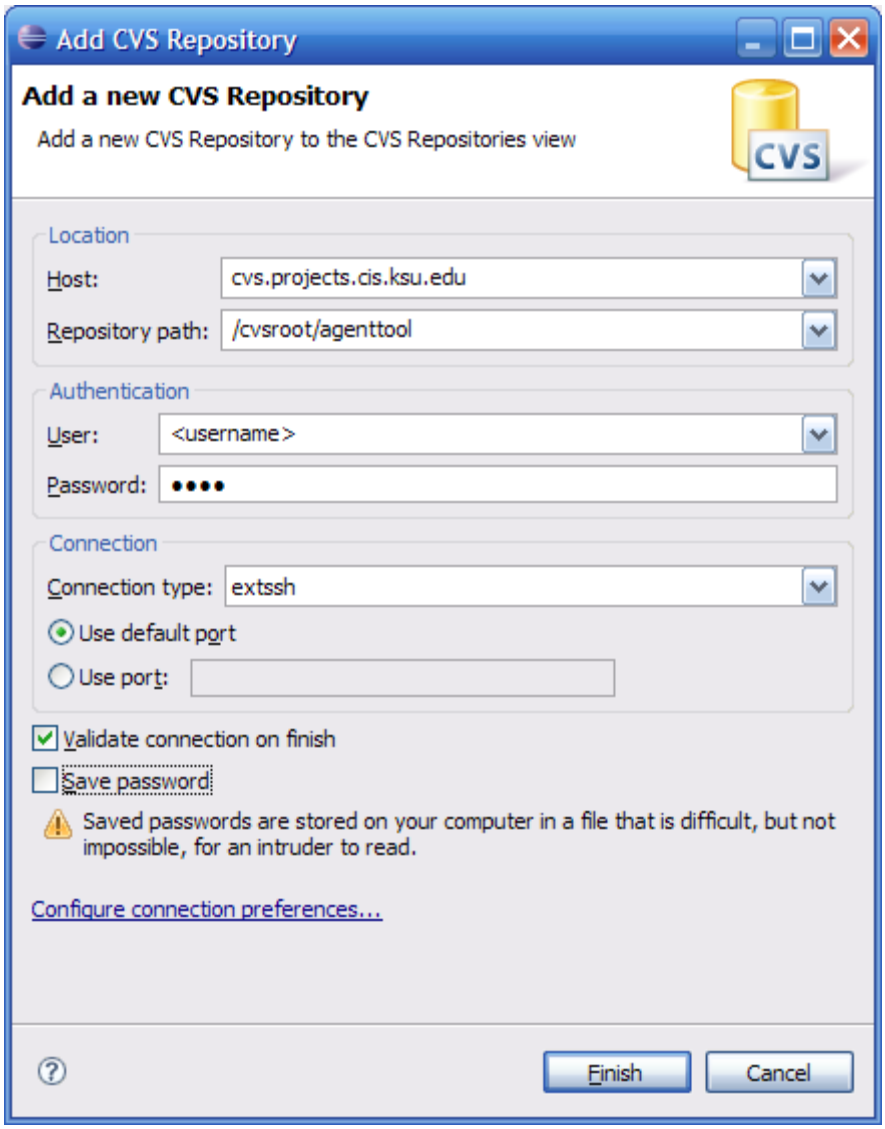
> "… a Java-based graphical development environment to help users analyze, design, and implement multiagent systems. It is designed to support the new Organization-based Multiagent Systems Engineering (O-MaSE) methodology. The system designer defines high-level system behavior graphically using the flexible O-MaSE methodology."

agentTool is a suite of Eclipse plug-ins that facilitate the design and analysis of multiagent systems. Currently, agentTool provides eight diagramming plug-ins. The following plug-ins are included in agentTool as of the 1.0.5 release:

- Agent Diagram
- Capability Diagram
- Domain Diagram
- Goal Diagram
- Organization Diagram
- Plan Diagram
- Protocol Diagram
- Role Diagram

Additionally, agentTool provides validation, code generation, and policy creation plug-ins. The code generation and policy creation plug-ins are currently under development are expected to be included in the next agentTool release.

To get started developing agentTool, you must first check out the latest development code from the K-State CIS projects server. You must first setup a CVS account by requesting one from the projects server administrator. Once you have your account, open the "CVS Repository Browsing" perspective in Eclipse. Next, right-click in the "CVS Repositories" view and select "New Repository Location" to start the wizard. Then fill out the text boxes as follows:

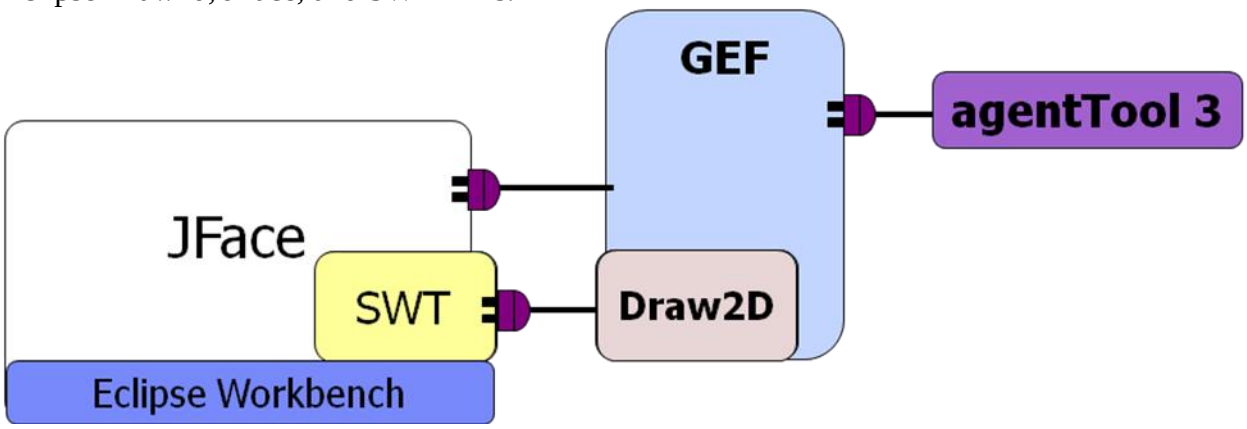


The connection should be successful and a new CVS location will appear in the "CVS Repositories" view. Expand the HEAD node in the tree and select the desired project to check out. Note that you must also check out the Core project as all other agentTool projects depend upon Core.

You should now be ready to start development in agentTool. Always keep in mind that code in the CVS repository represents the official state of the project, you must always be sure to synchronize your local copy with the repository when you make changes.

2. Relationship to GEF

agentTool is built as an Eclipse plug-in and requires JDK 1.5 or greater to compile. agentTool makes use of the Eclipse Graphical Editing Framework (GEF), which in turn relies on the Eclipse Draw2d, JFace, and SWT APIs.



Adapted from Eclipsecon 2005 GEF Tutorial

To compile and run agentTool, the GEF feature must be installed on your local copy of Eclipse. Specifically, the following packages must be available at runtime:

- org.eclipse.gef
- org.eclipse.ui.views
- org.eclipse.core.runtime
- org.eclipse.ui.forms
- org.eclipse.ui
- org.eclipse.ui.ide
- org.eclipse.core.resources

Additionally, the agentTool Core Plug-in must also be available at runtime:

- edu.ksu.cis.agenttool.core

Since agentTool inherits the vast majority of its functionality from GEF always attempt to extend existing GEF classes when implementing new features. This helps keep us up to date with the latest GEF API and allows us to inherit a large number of features "for free". Also, keep in mind that GEF provides a wide array of functionality that we do not currently take advantage of in agentTool. When implementing a new feature, avoid re-inventing the wheel and check to see if GEF has already done the hard work for you.

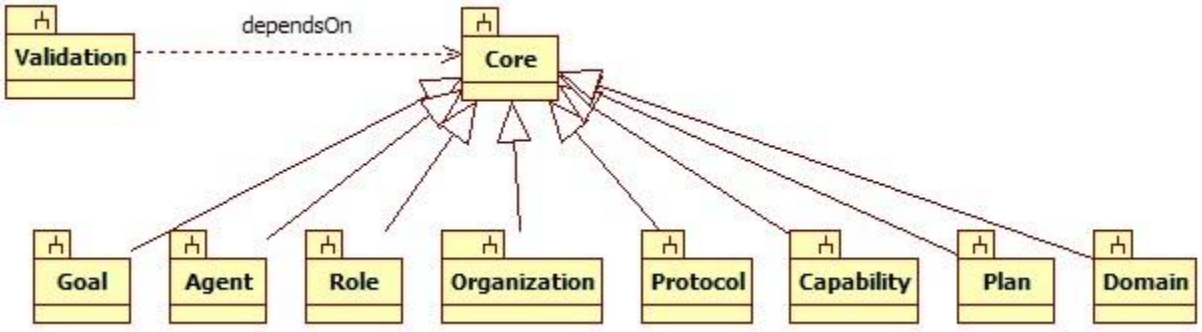
3. The Core Plug-in

agentTool uses a strong Object-Oriented and Design Pattern-based architecture. This Object-Oriented architecture extends to the relationships between plug-ins. The bulk of agentTool code is located in the Core plug-in. This plug-in contains code common across all other agentTool plug-ins. Other plug-ins should extend the base classes that are in core to provide their own diagram-specific functionality.

All classes that define the model should be located in the Core plug-in so that any classes can reference the complete agentTool model without needing to know about other diagrams. For example, the Validation plug-in only has to rely on the Core plug-in to have access to the entire agentTool model. This decouples design plug-ins that manipulate the model from analysis plug-ins that simply need to reference the model.

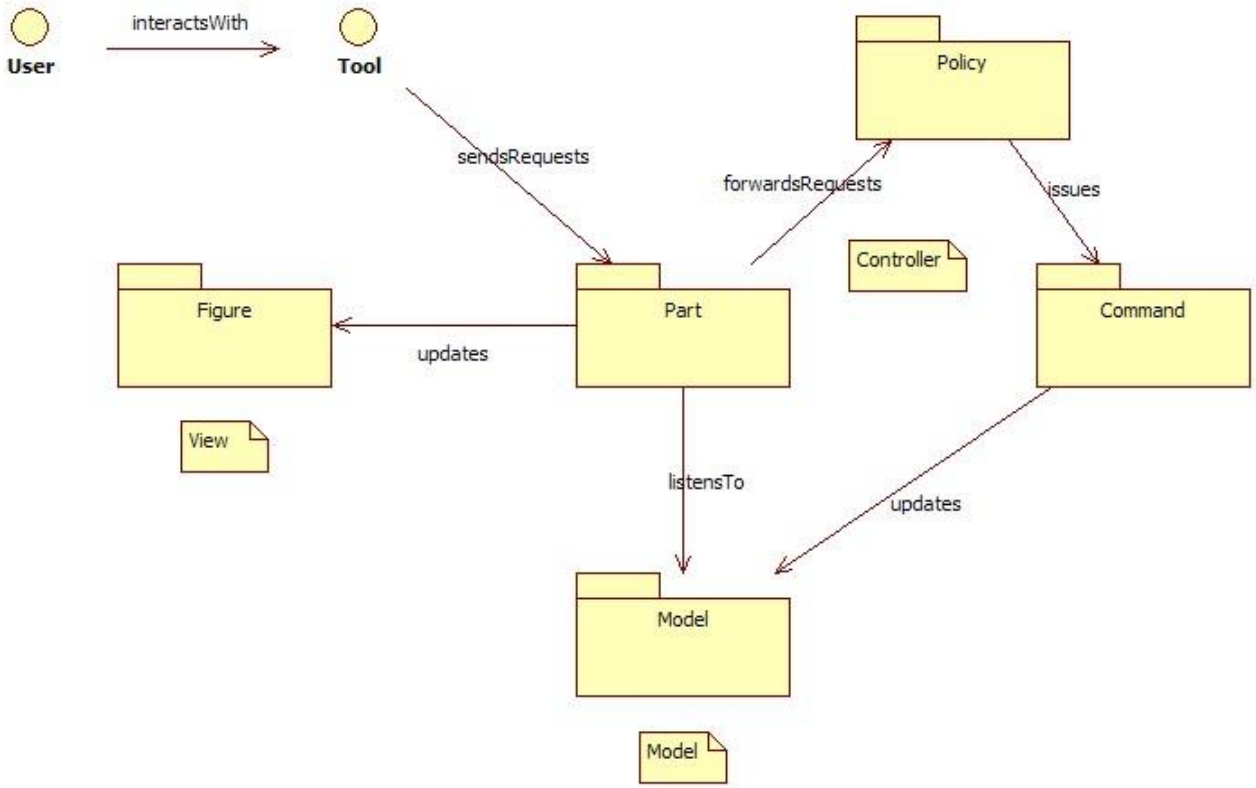
Classes in core that are intended to be extended for diagram-specific functionality contain the Abstract prefix. Classes that can reasonably be reused as-is with no modification contain the Core prefix. However, diagram-specific behavior occasionally requires these classes to be extended.

The following figure shows the current agentTool plug-in hierarchy:



4. agentTool Package Overview

agentTool is built upon GEF, which makes use of the MVC architecture and a variety of Object-Oriented design patterns. agentTool inherits GEF's MVC architecture, and the package structure of agentTool reflects this. agentTool can be broken down into a small set of packages that each perform a well-defined role.



- Action – The Action package handles integration with the Eclipse workbench. Its job is to define Toolbars, Menus, Shortcut-Keys, and their associated actions.

- Command – The Command package abstracts manipulation of the diagram model by encapsulating changes within well-defined commands. This decoupling allows for undo/redo support and increased code reuse.

- DirectEdit – The DirectEdit package provides utility methods for interacting with the canvas through direct-edit requests. Direct editing consists of double-clicking on a figure in the canvas and modifying its contents without launching a new view or dialog.

- Editor – The Editor package contains all editors, views, and wizards. This package is mainly concerned with interfacing agentTool with the Eclipse workbench.

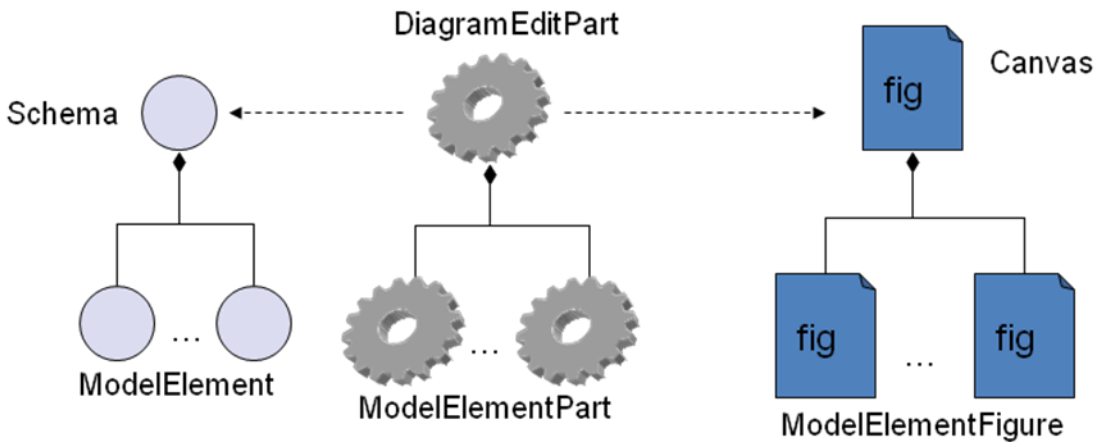
- Figure – The Figure package defines the view for the plug-in. All visible items on the canvas are defined in this package.

- Model – The Model package defines the data model for the plug-in. It contains the data and methods that define a given diagram. When developing a new diagram, one should always define classes in the Model package first.

- Part – The Part package contains controllers that are each associated with an object from the Model package in a 1-to-1 fashion. All property changes in the model trigger handlers defined in the part package. These handlers ensure that the view of the model always stays up to date.

- Policy – The Policy package defines the rules for user interaction with the model. It determines the set of legal actions and then dispatches an appropriate command to update the model.

- XML – The XML package handles model serialization. The XMLWriter class contains methods to help write valid and well-formed XML. The XMLParser class reads the XML file from the disk and returns a new diagram schema.



Adapted from Eclipsecon 2005 GEF Tutorial

In MVC terms, the Figure package makes up the view, the Model package is the model, and the Part, Policy, and Command packages together make up the controller. Every object in the model has an EditPart associated with it. ModelElements are associated with ModelElementParts, Schemas with DiagramEditParts, and Relationships with RelationshipEditParts. The part acts as the controller for its model object. Every EditPart is also associated with a figure that represents the state of its model element on the canvas.

The following is a simple example of how all these pieces fits together. In this example, we will assume that the user selects the new Goal tool from the palette.

1. When the user clicks on the diagram with the tool, the DiagramEditPart detects the click and forwards the request to its associated CreateEditPolicy.
2. First, the CreateEditPolicy determines if a new Goal can be constructed based on the request. If it can, it then creates a new CreateGoalCommand and executes it.
3. The CreateGoalCommand constructs a new Goal ModelElement and adds it to the Schema's list of children.
4. This addition causes the Schema to fire a new PropertyChangeEvent that is detected by the DiagramEditPart.
5. The DiagramEditPart calls its refreshChildren() method which causes all child EditParts to refresh their Figure to reflect this change.
6. When the Goal was created, a new ModelElementPart was also created using the EditPartFactory. The EditPartFactory defines the association between model objects and their EditParts.
7. Finally, ModelElementPart determines what Figure should be associated with Goals and paints the correct Figure in response to the refreshChildren() call from DiagramEditPart.

4.1 The Command Package

The Command package abstracts manipulation of the diagram model by encapsulating changes within well-defined commands. This decoupling allows for undo/redo support and increased code reuse.

There are five general classes of commands. Each class reflects the various ways the model can be modified:

- Create – Creates a new PropertyAwareObject. These commands setup a new PropertyAwareObject's fields and then add it their parent's list of children. For example, the CreateActorCommand assigns a new Actor a unique name, bounds, and a parent. It then adds the new Actor to its parent's list of children. Each of these changes modifies some part of model, which, in turn, fires a PropertyChangeEvent that updates the view to reflect these changes.

- Delete – Removes an existing PropertyAwareObject from its parent. This class of commands, as well as all others, fires PropertyChangeEvents similarly to the Create class. Executing a delete command will remove a PropertyAwareObject from the model.

- Move – Modifies the view of a model object in some way. This class of commands simply changes the bounds of a PropertyAwareObject in the model.

- Change – Modifies any other property of a model object. This class of commands modifies some property of a PropertyAwareObject.

- Set – Changes a group of properties in a model object at the same time. This class of commands is generally used to assign a list of children to a PropertyAwareObject. For example, a Goal object can be assigned a list of Parameters using the SetParametersCommand.

Whenever you wish to update the model in response to a user action, you should encapsulate that change inside of a command. Commands are executed through the Workbench's CommandStack and therefore inherit support for undo and redo operations. Additionally, by forcing model updates to be performed through a small set of well-defined commands, the model can be effectively decoupled from the controller.

4.2 The Editor Package

The Editor package contains all editors, views, and wizards. This package is mainly concerned with interfacing agentTool with the Eclipse workbench.

- DiagramEditors – DiagramEditors define the canvas and palette on which a plug-in operates. The AbstractDiagramEditor class is one of the most important classes in AgentTool. This class handles most of the interfacing between AgentTool and the rest of Eclipse. To modify a property that affects all diagrams, edit the AbstractDiagramEditor class.

- Palettes – Palettes define the set of tools that a user can use to modify a diagram. Typically, the set of tools consists of a selection tool and various component and relationship creation tools.

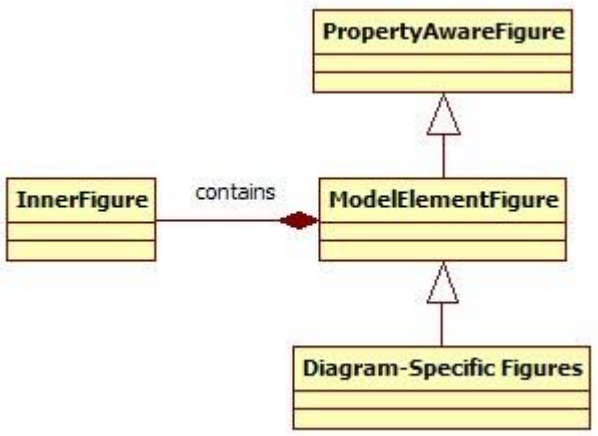
- Views – AgentTool makes use of a variety of PropertiesViews to provide a user interface for editing various object model properties. For example, the GoalPropertiesView provides an interface to modify a Goal's name, number, description, definition, preference, and set of associated parameters. Views allow complex user interactions to be encapsulated within a compact interface.

- Wizards – AgentTool uses Wizards to generate an initial diagram or rule. They typically prompt the user for required information and construct a skeleton diagram before handing over control to the appropriate editor.

The Editor package is tightly integrated with the Eclipse workbench. Often, when new Eclipse versions are released, classes in the Editor package need to be updated to provide access to new features, or accommodate API changes.

4.3 The Figure Package

The Figure package defines the view for the plug-in. All visible items on the canvas are defined in this package.



- PropertyAwareFigure – PropertyAwareFigure is the base class for all other figures. This class adds support for name labels.

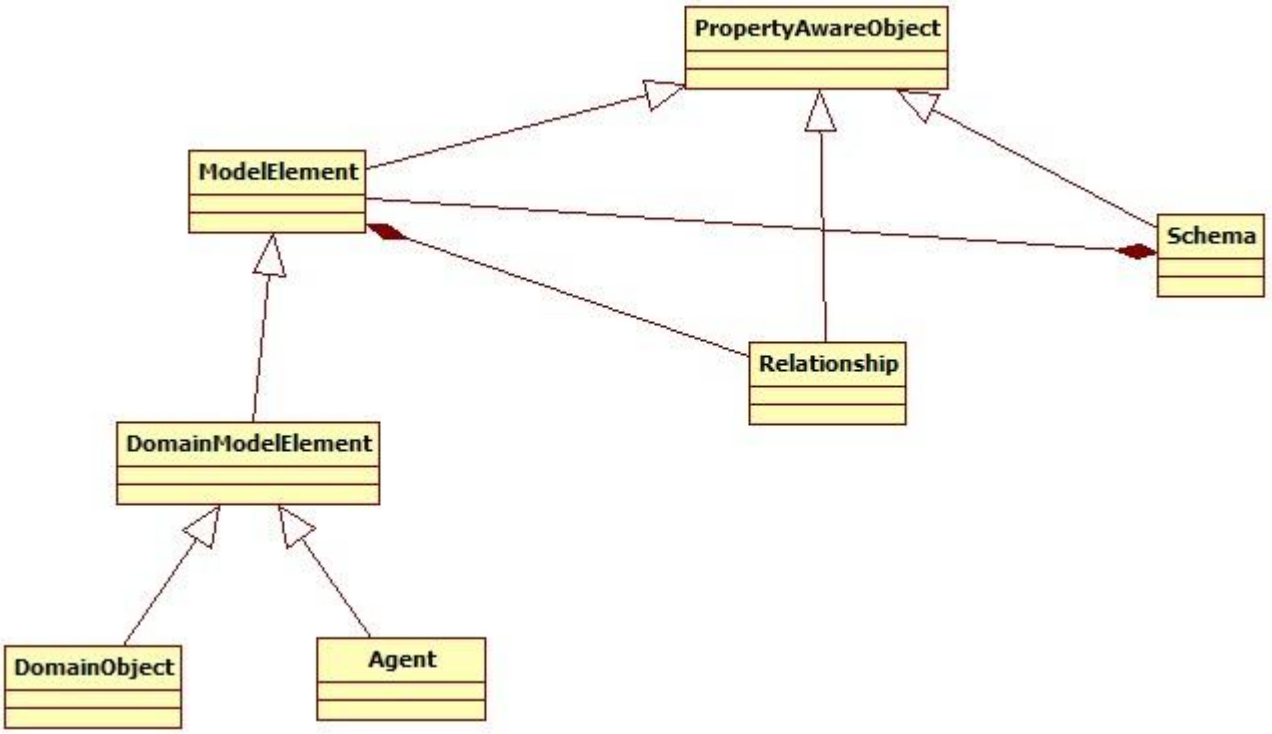
- ModelElementFigure – ModelElementFigure provides an area, InnerFigure, that allows child figures to be associated with a figure. Nearly every ModelElementPart uses ModelElementFigure, or one of its subclasses, to represent its model. Additionally, this class adds support for stereotype labels.

- InnerFigure – InnerFigure provides an area for child figures, usually Columns or Labels, to be added to a parent.

Most diagrams contain their own diagram-specific figures that add support for user interface elements that are not needed by other diagrams. For example, the Goal diagram creates its own GoalFigure that adds support for a numberLabel, and a list of child parameters. Nevertheless, diagram-specific figures still must extend a base class in the Core plug-in to avoid breaking compatibility.

4.4 The Model Package

The agentTool object model closely resembles the model defined by the GEF. The data model is defined by objects in the edu.ksu.cis.agenttool.core.model packages. Each diagram defines its own object model. However, all data models are similar, for example, the DomainDiagram's model looks like this:



- PropertyAwareObject – PropertyAwareObject is the base model class in agentTool. All other model objects should extend this class. PropertyAwareObject adds support for EditPart change listeners, as well as object name and visibility attributes.

- Schema – Schema is the primary model object in the diagram. One could think of the Schema as the object that represents the canvas; it contains all ModelElements and their associated Relationships. In addition, this class contains a number of helper methods to make model manipulation easier. Whenever a new diagram type is created, the Schema's DiagramType enumeration must be updated to add support for the new diagram.

- ModelElement – ModelElement is the base class for all diagram-specific model objects. This class adds support for relationships and figure bounds attributes. Additionally, this class contains a number of helper methods for manipulating ModelElements. Whenever a new ModelElement type is created, the ModelElementType enumeration must be updated to add support for the new ModelElement.

- Relationship – Relationship represents all connections between ModelElements. This class is only subclassed to add support for additional diagram-specific attributes. In general, however, this class does not need to be extended by diagrams wishing to use it. Diagrams must simply add a new relationship type to the RelationshipType enum to add support for new relationships.

Additionally, each diagram will also stipulate its own diagram-specific model. However, unlike other packages, the entire model must be located inside the Core plug-in. When diagram-specific classes are created, they should be placed in the diagram's sub package within the edu.ksu.cis.agenttool.model package.

4.5 The Part Package

The Part package contains controllers that are each associated with an object from the Model package in a 1-to-1 fashion. All property changes in the model trigger handlers defined in the part package. These handlers ensure that the view of the model always stays up to date.

- AbstractModelElementPart – AbstractModelElementPart acts as the controller for all ModelElements. This part does not install any policies, but instead relies on specific diagrams to install their own policy support. For example, the ProtocolDiagram allows some of its ModelElements to contain other ModelElements, however, not all diagrams do. Therefore, the ProtocolDiagram contains a ModelElementPart that extends AbstractModelElementPart that installs the correct edit policies. In addition to policies, this part delegates figure creation to its subclasses. This allows all ModelElements to have common event handling code, but separates presentation and diagram-specific logic.

- AbstractRelationshipPart – The AbstractRelationshipPart works similarly to the AbstractModelElementPart, except it acts as the controller for Relationships. One important difference, however, is that this part installs a set of policies that are common among all Relationships.

- CoreDiagramPart – CoreDiagramPart acts as the controller for the canvas. Primarily, this class is concerned with ModelElement and Relationship layout. Unlike other Core part classes, this class usually does not need to be extended. Occasionally, however, a diagram will need to override the default functionality provided by this class. For example, the ProtocolDiagram extends CoreDiagramPart to remove support for automatic relationship layout.

Every diagram will include its own diagram-specific parts. Typically, the part and policy packages contain very diagram-specific code. Classes in the Core plug-in provide a solid framework to build upon, but it is up to the individual plug-in to complete the implementation.

4.6 The Policy Package

The Policy package defines the rules for user interaction with the model. It determines the set of legal actions and then dispatches an appropriate command to update the model. All policies are installed by and owned by specific EditParts.

- AbstractRelationshipCreateEditPolicy – AbstractRelationshipCreateEditPolicy specifies which Relationships can connect to which ModelElements. This class is very diagram-specific and contains the bulk of the logic that differentiates one diagram from another.

- CoreElementDeleteEditPolicy – CoreElementDeleteEditPolicy simply bridges delete requests from all ModelElementParts with their appropriate DeleteElementCommand. This class rarely needs to be extended.

- CoreLayoutEditPolicy – CoreLayoutEditPolicy defines how ModelElements can be laid out on the canvas. Typically, this class restricts certain ModelElements from being resized. This class also handles the translation between absolute and relative coordinates.

- CoreRelationshipDeleteEditPolicy – CoreRelationshipDeleteEditPolicy simple bridges delete requests from all RelationshipParts with their appropriate DeleteRelationshipCommand. This class has yet to be extended.

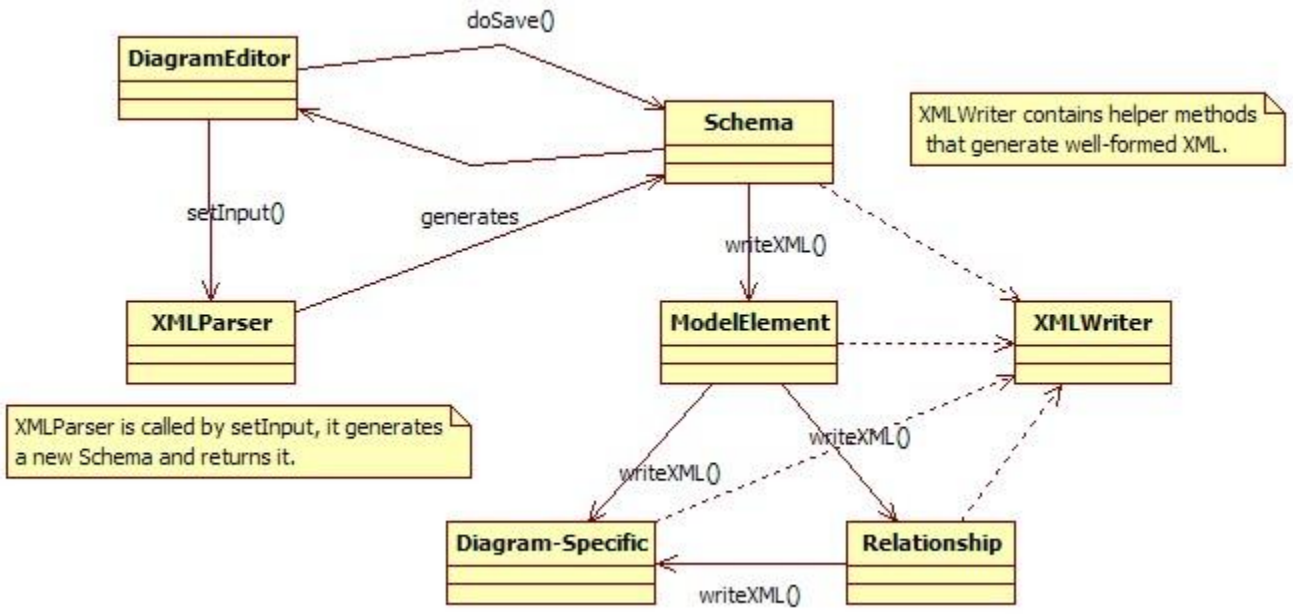
In addition to the above classes, each diagram can provide its own diagram-specific policies.

4.7 The XML Package

The XML package handles model serialization. The XMLWriter class contains methods to help write valid and well-formed XML. The XMLParser class reads the XML file from the disk and returns a new diagram schema.

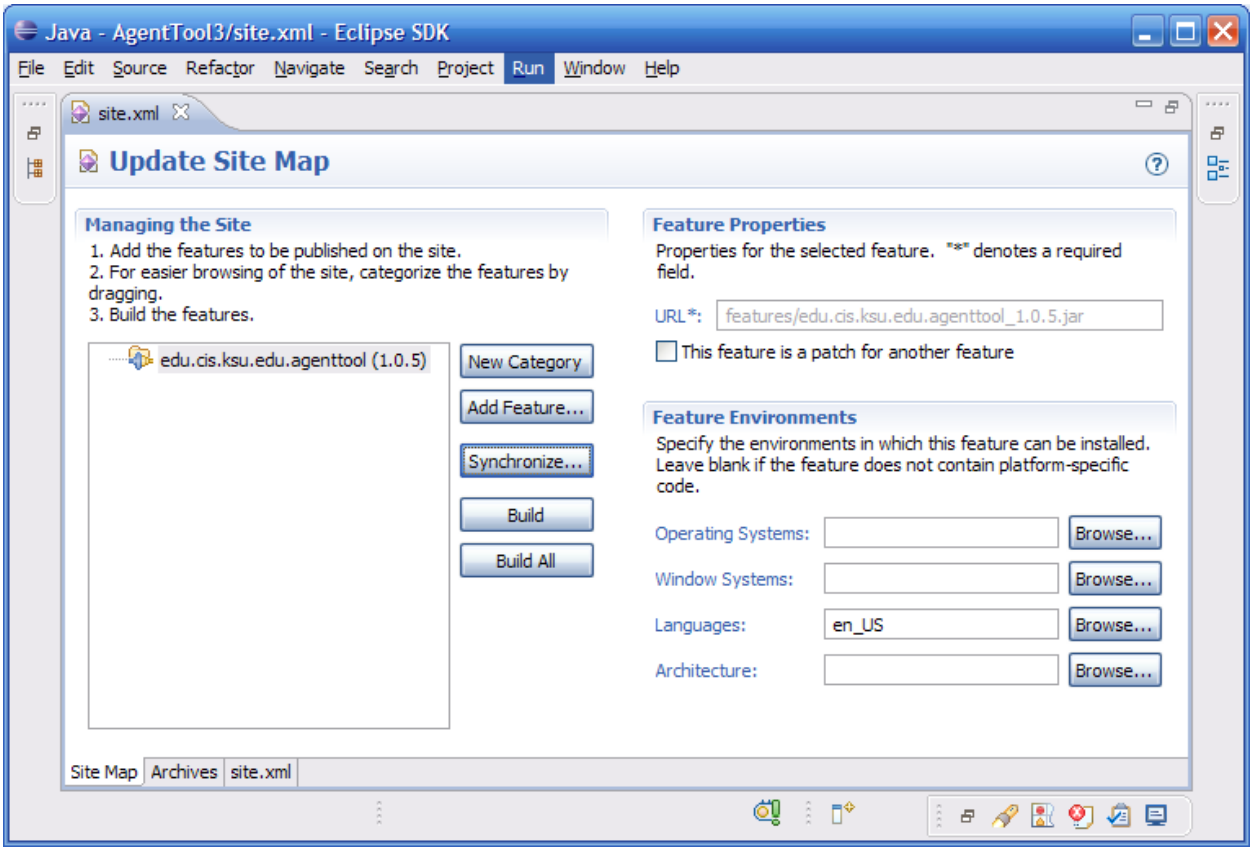
agentTool uses XML to serialize its object model. XML allows a wide variety of other tools to interface with agentTool without having to have knowledge of agentTool's inner workings. The Validation plug-in takes advantage of this decoupling and has very little in common with agentTool besides the shared data model.

When a diagram is saved the DiagramEditor creates a stub XML file and then calls the writeXML() method in the Schema class, which in turn calls the writeXML() method in all of its children objects. To read in the XML document, the DiagramEditor creates a new XMLParser and passes the path of the XML file to read. The XMLParser then uses the built-in Java DOM parser to construct an in-memory object model that is represented by the Schema object.

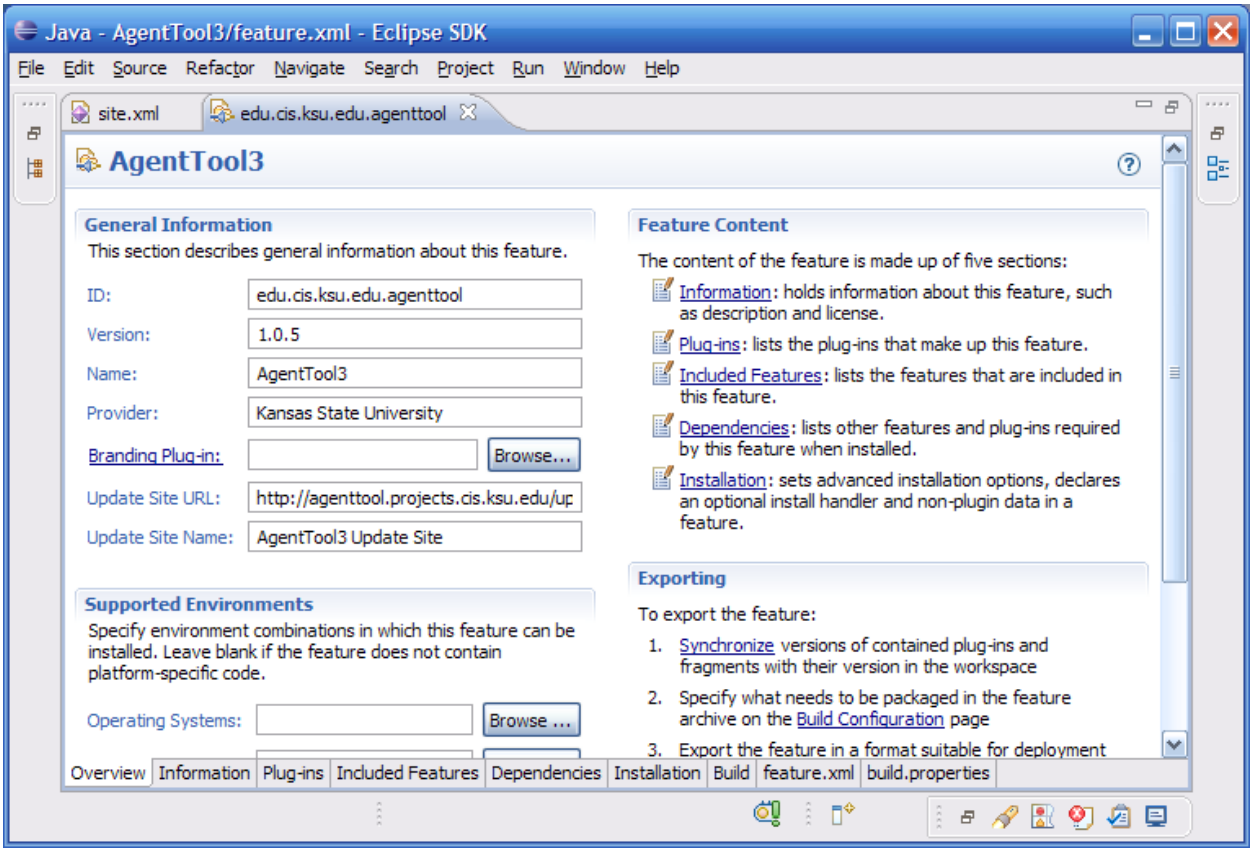


5. Releasing an Update

agentTool releases are very simple to build and maintain. The meta-project AgentTool3 contains all the necessary files to create a new official release and update site. When you wish to release a new version of agentTool, simply check out the AgentTool3 project from CVS and open the site.xml file in Eclipse. This will open the "Update Site Editor":



Double-click on the edu.ksu.cis.agenttool feature from the site dialog; this will open the "Feature Editor":



Increment the version number by one to signify a new release. Note that Eclipse version numbers have special significance for backwards compatibility. Do not bump the first or second number unless you intend to break backwards compatibility with this release.

Next, close the "Feature Editor" and return to the "Update Site Editor". To build a new release, click the "Build All" button. The Eclipse builder will run and generate a new set of .jar files for the new release.

From here, you can upload the contents of the AgentTool3 project to: http://agenttool.projects.cis.ksu.edu/update/ to allow users to automatically update their pre-installed older version. Alternatively, you can package the contents of the AgentTool3 project in a .zip file to allow web distribution.

See the article: Keeping Up To Date: http://www.eclipse.org/articles/Article-Update/keeping-up-to-date.html for additional information about the automatic updates process.

6. Extending agentTool: An Example

agentTool is designed to be easily extended to add new features and support new modeling capabilities. In this example, we will walk through the addition of simple ModelElement called Plan to the Capability-Action Diagram. For simplicity in our example, the Plan ModelElement will not contain any children, such as parameters, attributes, or constraints. It will also not be the source of any Relationships to other ModelElements. The Plan ModelElement will only allow the "performs" relationship to connect to it.

The first thing that one must consider when adding new modeling capabilities is how the data will be structured and stored between editing sessions. These considerations are the responsibility of the Model package. The Model package is contained entirely within the Core plug-in.

First, since we are creating a new ModelElement type, we must create a new entry in the ModelElementType enumeration in the ModelElement class, we will call our new entry ModelElementType.PLAN. Additionally, we need to add a new String entry to the XMLConstants class:

```
public static final String PLAN = "Plan";
```

Next, we must define the Plan Model class. This class will extend ModelElement and implement its abstract methods, namely writeXML() and getType():

```
public class Plan extends ModelElement {
    public Plan() {
        super();
    }

    public Plan(String n, Rectangle bounds, Schema s) {
        super(n, bounds, s);
    }

    @Override
    public ModelElementType getType() {
        return ModelElementType.PLAN;
    }

    @Override
    public String writeXML() {
        // open element
        StringBuilder xml = new StringBuilder();
        xml.append(XMLWriter.openElement(XMLConstants.PLAN, this));

        // this element has no children

        // close element
        xml.append(XMLWriter.closeElement(XMLConstants.PLAN));
        return xml.toString();
    }
}
```

The getType() method is used to identify this ModelElement by assigning it a unique identifier. The writeXML() method is called by the Schema object to generate the correct XML for a Plan object.

Next, we need to define the controller for our new Plan ModelElement. In the Part package of the CapabilityDiagram plug-in, we need to create a new class called PlanPart. This class will extend AbstractModelElementPart and implement its abstract createEditPolicies() and createFigure() methods:

```
public class PlanPart extends AbstractModelElementPart {
    @Override
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
            new RelationshipCreateEditPolicy());
        installEditPolicy(EditPolicy.COMPONENT_ROLE,
            new CoreElementDeleteEditPolicy());
        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE,
            new CoreDirectEditPolicy());
    }

    @Override
    protected IFigure createFigure() {
        return new ModelElementFigure(getModel().getName(), "«Plan»");
    }
}
```

The createEditPolicies() method essentially assigns roles that this EditPart plays by associating it with various Policies. In our example, PlanPart is assigned roles that allow it to be deleted, direct-edited, and allow relationships to be connected to it.

Now that we have created our Model and our controller, we need to associate them with each other. To make this association, we need to add a new case to CapabilityDiagram's DiagramEditPartFactory class. Simply add the following lines to the createEditPart() method right before the last statement:

```
else if (obj instanceof Plan) {
    editPart = new PlanPart();
}
```

Our next task is to make sure that the policies that we installed in PlanPart can accommodate the addition of a new Part. Neither the CoreElementDeleteEditPolicy or the CoreDirectEditPolicy need any modifications. However, we will have to update the RelationshipCreateEditPolicy in the CapabilityDiagram Plug-in to add support for Plan objects. In this case, we simply need to change one line in the getConnectionCompleteCommand() method to allow "performs" relationships to connect to PlanParts:

```
case PERFORMS:
    if ((child instanceof ActionPart) || (child instanceof PlanPart)) {
        cmd.setChild(element);
        return cmd;
```

```
    }
    break;
```

In addition to the previously mentioned policies, we must always update our diagram's LayoutEditPolicy to add support for our new Plan object. We must add a new case to the getCreateCommand() method:

```
case PLAN:
  return new CreatePlanCommand((Plan) modelElement
      (Rectangle) getConstraintFor(request), (Schema) getHost().getModel());
```

This policy handles requests that are generated by tools from the palette. We will update the Palette shortly to add support for our new Plan object. However, first, we must define the CreatePlanCommand class that we just referenced in LayoutEditPolicy:

```
public class CreatePlanCommand extends Command {
    private final Rectangle bounds;
    private final Plan plan;
    private final Schema schema;

    public CreatePlanCommand(Plan p, Rectangle b, Schema s) {
        plan = p;
        bounds = b;
        schema = s;
    }

    @Override
    public void execute() {
        if (plan.getName().equals("")) {
            plan.setName(schema.generateUniqueName(XMLConstants.PLAN));
        }
        plan.setParent(schema);
        plan.setBounds(bounds);

        schema.addChild(plan);
    }

    @Override
    public void undo() {
        schema.removeChild(plan);
    }
}
```

This class initializes a new Plan object by assigning it an initial name and bounds, and then associating it with its parent, the Schema.

Now we need to update this diagram's palette so that we can create new Plan objects. Inside the PaletteViewerCreator class, we need to define a pair of String constants (we will assume that the icon file already exists):

```
private static final String PLAN = "Plan";
private static final String PLAN_ICON = "icons/c_plan.png";
```

And add the following line in the CreatePaletteRoot() method:

```
compDrawer.add(new CombinedTemplateCreationEntry(PLAN, DESCRIPTION + PLAN + "
", Plan.class, new SimpleFactory(Plan.class),  getIcon(PLAN_ICON),
getIcon(PLAN_ICON)));
```

The Plan object can now be added to Capability diagrams. The only other thing that we must do is update the XMLParser class to add support for reading. In the XMLParser class we have to define a new method, addPlans():

```
private void addPlans(Schema schema) {
    // add plans to schema
    NodeList list = document.getElementsByTagName(XMLConstants.PLAN);
    for (int i = 0; i < list.getLength(); i++) {
        Node node = list.item(i);
        try {
            Plan plan = new Plan(getRequiredString(node, XMLConstants.NAME),
              getBounds(node), schema);
            schema.addChild(plan, false);

            // plans don't currently have any children

         } catch (XMLParseException e) {
            IStatus status = new Status(IStatus.WARNING,
              CorePlugin.PLUGIN_ID, 0, e.getMessage(), e);
            CorePlugin.getDefault().getLog().log(status);
            continue;
        }
    }
}
```

And then call it from the Capability diagram case in the getSchema() method:

```
case CAPABILITY:
    // add capability model elements to schema
    addCapabilityActions(schema);
    addCapabilities(schema);
    addPlans(schema);
    break;
```

We have now finished adding our new ModelElement to the Capability diagram! Often, new features will require changes other than ones that have been outlined here; however, this example provides an excellent starting point for adding new features.

7. Additional Reading

agentTool is distributed with a full set of Javadoc documentation. This documentation should be the first place to look for answers to agentTool development questions. The most current version of Javadoc documentation can be checked out from CVS. Each plug-in contains its own /doc/ directory that holds the relevant documentation.

Since agentTool closely mimics the structure of the GEF framework, a variety of relevant GEF documentation exists:

- Building a Database Schema Diagram Editor with GEF
  http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html

- Create an Eclipse-based application using the Graphical Editing Framework
  http://www-128.ibm.com/developerworks/opensource/library/os-gef/

- Displaying a UML Diagram using Draw2D
  http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

- EclipseWiki GEF
  http://eclipsewiki.editme.com/GEF

- Other GEF Documentation
  http://www.eclipse.org/gef/reference/articles.html

- GEF Redbook
  http://www.redbooks.ibm.com/redbooks/SG246302/

- GEF Tutorial
  http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/47_Hudson.pdf

Also, the GEF examples plug-in is also very instructive. The following diagrams and their source code are included in the org.eclipse.gef.examples package:

- LogicDiagram
- ShapesDiagram
- FlowDiagram

Finally, since agentTool was originally built through a series of MSE projects, the following links may also be of use:

- MSE Portfolio – Deepti Gupta (Agent, Goal, Role, Organization Plug-ins)
  http://mse.cis.ksu.edu/deepti/

- MSE Portfolio – Binti Sepaha (Plan and Protocol Plug-ins)
  http://mse.cis.ksu.edu/binti/

- MSE Portfolio – Patrick Gallagher (Validation Plug-in)
  http://people.cis.ksu.edu/~psg9999/classes/MSE/index.html

Note that agentTool has undergone very significant structural changes since these MSE projects have been completed. Not all of the information on these pages is accurate.