Creating a Workflow Editor Eclipse Plug-in Using the Graphical Editing Framework (GEF)

Mei Tang

Carleton University

COMP 4905

Supervisor: Prof. Tony White

August 13, 2004

Abstract

Eclipse platform is a universal tool integration platform. As an important plug-in in Eclipse, Graphical Editor Framework (GEF) provides fundamental operations for developer to map any existing model to a graphical editing environment.

This project makes use of GEF and extends its operations to create a graphical workflow editor Eclipse plug-in, which will have the functionality to drag and drop specific components on the canvas, establish particular associations between them, hook the components with external application model and simulate model behaviors with the workflow diagram. The project implements with Java and can be integrated into Eclipse platform.

This report will give an introduction to the project, briefly overview the Eclipse and GEF, as well outline the project design architecture, implementation and limitation. Moreover it gives a description about how to use this workflow editor.

Acknowledge

I would like to appreciate my supervisor Professor Tony White for his guidance, ideas and advices for this project.

Also I would like to thank the following people for their patient assistant throughout this term:

Zhiyong Liu
Michael Gold
Weimin Mao

Table of Contents

List of Figures

1 Introduction

IBM Eclipse is a microkernel framework, which is designed for building integrated development environment. Software such as modeling tools, programming tools and testing tools are fit into Eclipse as plug-ins to integrate a universal tool platform, which can support the construction of a variety of tools for application development, arbitrary content types (XML, HTML, GIF, etc.) manipulation and graphical user interface development. As an essential plug-in in Eclipse, Graphical Editing Framework (GEF) has provided fundamental functionality for developer to follow and extend so that a graphical editor can be created from an existing application model.

In general the workflow editor is used to draw a workflow diagram for an application model. In this project, the Eclipse plug-in workflow editor will take advantage of the common operations of GEF and extend them such that it has the functionality to draw a workflow diagram and simulate the behaviors for the external application model. The workflow diagram consists of components such as condition, activity and transition. In order to simulate the model behavior, external application model objects also have to be part of the components in the editor. The editor will have a palette, from which those components can be selected and dragged, and a canvas, on which the dragged components will be drawn. Components that represent external application model can be added on the palette via a palette customization user interface. Transitions are established between activities and conditions to indicate the workflow direction, and particular user interfaces are associated with the activity and condition components to define their relationship with the external application model. After the workflow diagram is defined, it can be run to check the correctness of its definition. Furthermore the editor follows an open-save-close life cycle, so the workflow diagram can be open and saved as a file.

Even though the workflow editor plug-in has realized most useful functionality, how to implement a perfect editor is still a complicated issue. Some restrictions are related to this editor, which only can be used to simulate the behavior of the simple application model. Further improvement is necessary.

2 Eclipse and GEF Overview

2.1 Eclipse Platform Working Principal

As we know, the Eclipse Platform can be used to develop various applications. It uses an innovative plug-in architecture allowing near-infinite extensions to the base IDE, the core of which is built in the same way as all the tools that extend it are built. Many useful building blocks and frameworks are provided to facilitate the application tool development.

All functionalities of the Eclipse Platform are achieved with plug-ins, which can do anything from syntax highlighting to interfacing with a source code control system. According to the complexity of its functionality, a tool can be implemented with one or several plug-ins. Generally a plug-in includes a JAR library, a manifest file, and some resource files such as images, web templates, native code libraries, etc. The manifest file is a XML schema that declares the relationship between this plug-in and other plug-ins. For instance, it specifies the extension points, actions contribution and the runtime JAR library, etc. An in-memory plug-in registry will be built to register the available plug-ins when the Eclipse platform is launched, and find the corresponding contributed extensions when a plug-in is activated to run.

The Eclipse Platform UI paradigm consists of views, editors and perspectives. Editors and views are tightly integrated into the workbench window, a GUI that Eclipse Platform interacts with the user. When an editor is activated, actions will be contributed to the workbench menus and toolbars. Views provide information about what user is working with in the workbench. The package explorer, navigator and outline are all views to display different information about the workbench. Modification is tightly consistent between editor and views.

The workbench provides diversity of extension points that allow tools to be integrated into the editors-views-perspectives UI paradigm. The main extension points are editor, view and perspective, which make it easy to add new editors, views and perspectives. Within those parts, new actions can be contributed to the workbench menu and toolbar, related view or editor's pop-up context menu, and view's local menu and toolbar. Moreover, new views, action sets and shortcuts can be added to an existing perspective. Eclipse platform is responsible for the management of the workbench window and platform perspective. It automatically instantiated the required editors and views, disposed of them if no longer needed, and provide workbench services to keep track of the activation and selection.

2.2 GEF Working Principle

Graphical environment is an important functionality in the Eclipse platform. GEF and Draw2d, two major plug-in in the Eclipse Platform, work together and make it possible to create a graphical application from a non-graphical application model. Draw2d is an SWT-based drawing plug-in, which provides a standalone rendering and layout package for an SWT canvas. Common shapes and layouts are defined and can be assembled and extended to render what users want.

The GEF plug-in uses Draw2d for rendering, but employs the MVC (model-view-controller) architecture for its editing framework, which makes sure that most graphical applications look and behave in a similar way. GEF implements a unique MVC architecture, which generally has no link between the view and model. The *model* in the GEF should be independent on any view or controller and can be changed only via GEF Commands. The *view* includes the *visual part,* which serves as the primary representation for the model object(s), and other things that are visible to the users such as feedbacks, tool tips, etc. Draw2d provides the common figures and layouts to compose of the graphical view. The *controller* in the GEF is named Edit Part, which directly represent an object in the model. It plays a vital role between the *view* and the *model.* In one hand, according to its functionality an Edit Part can install one or more Edit Policies that will trigger the execution of specific commands to change the model. On the other hand it works as a modification event handler to enables the visual part consistent with the model. For instance while a deletion operation is made for a visual part, the Edit Policy on the Edit Part will trigger the corresponding delete command to modify the model. Then the model will fire a modification event, and the Edit Part handles it to make corresponding change on the visual part.

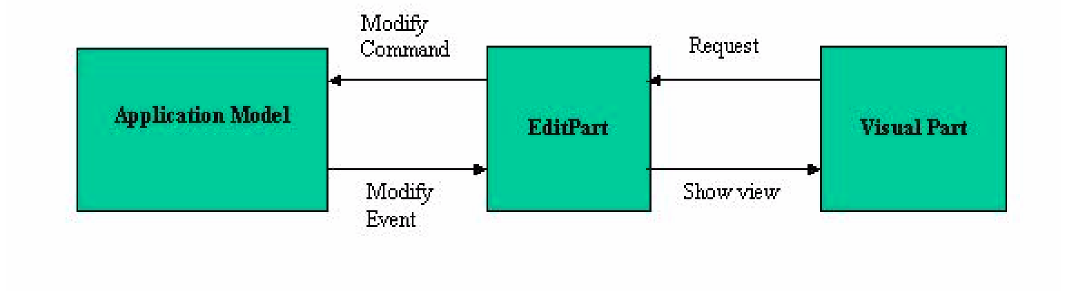

Figure 1: GEF MVC architecture

GEF has a variety of features. It provides a bunch of common tools such as Selection, Creation, Connection and Marquee tools. Users also can define their own components to be used in an editor via a palette. The palette is designed to display all the tools and components that an editor wants to use. Moreover, GEF includes functionality to manipulate the connection, resize objects and display the model in two different ways as Graphical and Tree structure.

The majority of GEF applications will be Eclipse Editors. The workflow editor in this project will be one of the GEF applications. Its implementation will be involved with the Eclipse workbench, GEF, Draw2d and Eclipse PDE (Plug-in Development Environment).

3 Workflow Editor Design Description

3.1 Workflow Editor Design Overview

Workflow Editor is designed to be an Eclipse plug-in. It will be written in Java and the basic unit of the editor is Java class. The extension point is chosen to be the Eclipse Editors.

In order to couple with the GEF plug-in, the design principle will use GEF MVC architecture. The structure of the Workflow Editor will consist of three parts - Application model, Edit Part and Visual part. In order to achieve the interactions between them, it can be divided into more detailed basic components as following:

- Application Model
- Command
- Edit Part
- Edit Policy
- Editor Part

The following diagram has shown interactions between those components.

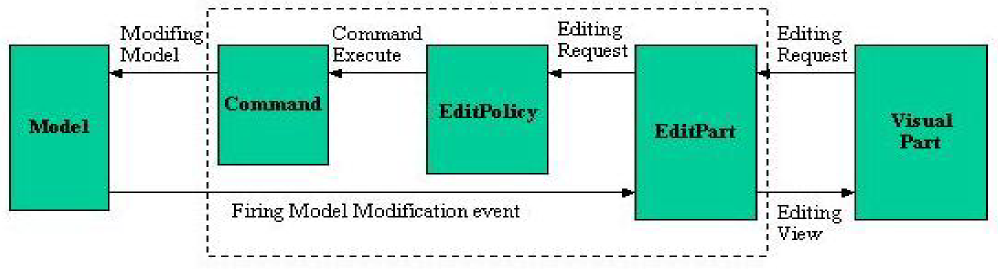

Figure 2: Workflow Editor Structure

3.2 Design Description of Editor Part

As a typical GEF editor, the Workflow Editor will consist of a Palette Viewer, an Editor Part, an Edit Domain and lot of Edit Part Viewers. In this project, the Editor Part is the main frame of the Workflow Editor. When the editor is activated, the Editor Part will configure and initialize an Edit Domain which will create a single editing session for the editor, a Palette Viewer which will display all the editing tools and components, and an Graphical Viewer which is corresponding to a top level Edit Part and works as a blank canvas. Once the canvas is realized, the editor will be ready to populate the viewer with additional Edit Parts.

The Palette Viewer has a palette model that defines the editing tool, such as Selection, Marquee and Connection, and the component groups. A palette factory will define all tool entries and template creation entries for the model. Those template creation entries are the real engine to create component instances on the canvas. For the purpose to add new component on the Palette Viewer, the palette customizer on the context menu will be used to define the component type for the template creation entry.

It is Editor Part's responsibility to registry the editor actions, create the adapters for different views and define how to open and save the workflow diagram as a file.

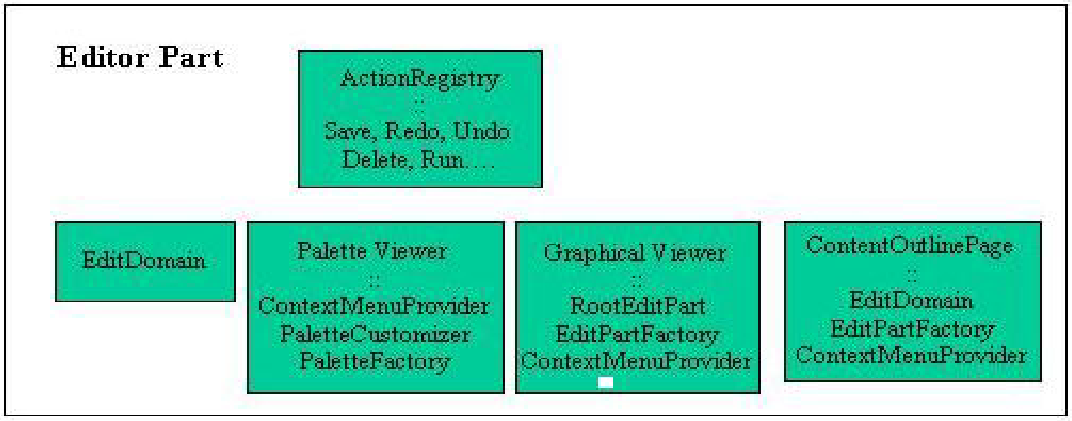

Figure 3: Editor Part Contents

3.3 Design Description of Application Model

The application model is composed of two parts - the components that used to draw the workflow diagram and an interface for the external application model.

A workflow diagram consists of many activities, conditions and transitions. In this project, application model also has those three fundamental components. In case the properties of the components change, the *model* should notify the controller to interact with the visual part. A super class WorkingFlowElement is defined for all model object classes to include the functions such as adding and removing property change listener, firing property change event, setting the property source and so on. Corresponding to the blank canvas initialized by the Editor Part, a class is defined so that the components on the palette can be added and removed. Activity and condition component classes inherit from the canvas class because they also need to add other components inside. For example activity will have a label to describe its action property, and condition can include other condition instances. Conditions can be separated to three classes - simple condition, OR and AND conditions. The simple condition will define the properties that are used to do condition check. The OR and AND conditions are combination of many simple, OR and AND conditions and do corresponding OR or AND condition check. Activity component class needs to define its related object, the action and how to run. Another important class is used to define the transition, which connects the condition and activity and have bendpoints to make it smooth. But each activity only has one outgoing transition, and the condition will have two, one defines as yes and the other defines as no.

An interface is defined for the objects of the external application model. Because user's object will be part of the editor, the interface is an abstract class that inherits from the WorkingFlowElement super class. It specifies the functions that will be used by the activity and condition components, such as user's model objects have to define its properties and actions list, how to run with the given action, which information need to be logged on the visual part and so on.

3.4 Design Description of the Edit Parts

Corresponding to the application model, the Edit Parts also will separate to two groups. The Edit Parts that represent the diagram components will be shown on the canvas, and those that represent the external application model will be shown on the outline view.

Each Edit Part will be a controller to manipulate the application model and the viewer. It will be shown up on the canvas as the content of an Edit Part Viewer to form the workflow diagram, as well handle the modification between the model and viewer. So every Edit Part will implement as a property change listener, create a figure representation and install some edit policies. Edit policy is the actual agent to finish the editing request. For the diagram components, Edit Part classes will be defined for activity, three conditions, transition, start point and the top-level canvas.

In this project, the transition Edit Part class will represent the transition model object and inherit from the GEF AbstractConnectionEditPart. It will have a context menu, from which user can define its status to be yes or no. Via installing specific Edit Policies, it will have the roles to create connection, bendpoints and pop up the context menu. A connection anchor class is defined to specify the connection point on the activity and condition Edit Part for establishing the transition Edit Part.

All other Edit Parts will inherit from the GEF AbstractGraphicalEditPart class. A dialog will be associated with the activity Edit Part to define its related object, the action and the arguments that are used for the action. The activity Edit Part need to install the Edit Policies to work as graphical connection node, and do deletion, creation and move operations. There are three condition Edit Parts to represent the three different condition components. All of them have the editing role as a graphical node for the transition, and can do creation, deletion, move and other general component operation. But they have different visual representation and context menu. The simple condition Edit Part has a dialog that can be opened to define the condition comparison rule. However its editing role will permit only a label child to be added on it. The other two Edit Parts have the capability to add any condition instances inside to form an OR or AND condition, therefore user can assemble the condition that he wants.

A diagram Edit Part will represent the top-level canvas. It installs the editing roles to work as a root component and can add any components inside. Also it is a connection layer with a specific router definition such that the transition Edit Part has a route rule to follow when it creates. A start point Edit Part will be added on the diagram Edit Part when the canvas is initialized and cannot be deleted. It will connect to an activity or condition Edit Part to specify the beginning component for the run command.

Some components on the editor represent external model objects. It is not reasonable to add them as a part of the workflow diagram on the canvas, but they are necessary to achieve the run functionality. So outline viewer is chosen to be a good place to put those instances. From the outline viewer, we can know which external model component is added or deleted, and how their properties change when the diagram is run. An outline page class will be defined to be the adapter for the IContentOutlinePage class. Those Edit Part classes that represent the external model objects inherit from the AbstractTreeEditPart class and have the functionality to be deleted, redo and undo.

In order to realize the editing roles of those Edit Parts, different Edit Policies are defined for them. The Edit Polices are mainly used to finish the graphical node role, component role, layout role, connection role and bendpoint role. Those Edit Policies will trigger the command to execute to do deletion, creation, move, redo, undo and other operations. All those command classes are extended from GEF Command class and define the execution, redo and undo functions to modify the model. Two important commands – run and step command – will define how to run the diagram in different ways. Moreover various action classes is necessary for the operations from the context menu and toolbar, such as actions to run the diagram, open the dialog for the activity and condition Edit Part, set the transition status and so on. Those actions will be registered when the editor part is initialized and can be contributed to the editor toolbar and context menu.

Finally a WorkingFlowPartFactory class is defined to create those Edit Part instances on the canvas when model objects are instantiated. For those on the outline viewer, a FlowTreePartFactory class is clarified to create instances for the outline tree. Then the workflow editor will have all the fundamental parts to finish its functionality.

3.5 Design Description of the manifest file

The manifest XML file is the most essential part of the Eclipse plug-in. In this project, the manifest file defines the extension point to be the Eclipse Editor. The editor file extension, run time JAR library, action bar contribution class, dependency libraries and other information will be specified on this file.

4 Workflow Editor Implementation

As described in last section, the workflow editor will be implemented as an Eclipse plug-in. According to the MVC architecture, its implementation will divided into four primary tasks – application model implementation, edit parts and their edit policies implementation, command and action implementation, editor part and palette implementation. Detailed description of the implementation will be described in this section.

4.1 Application Model Implementation

The application model has a hierarchy structure, which is shown on the following diagram.

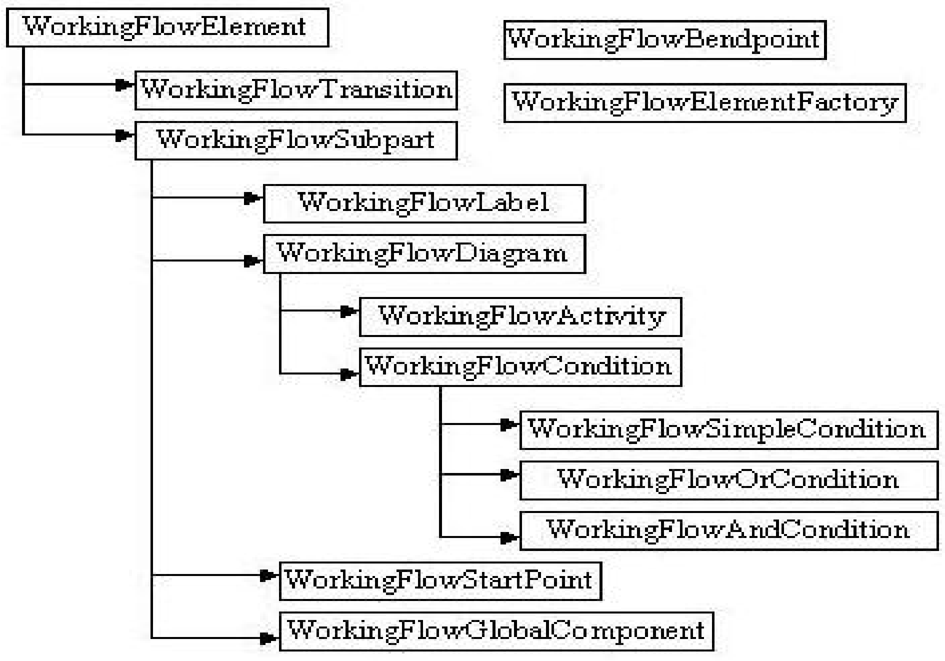

Figure 4: Workflow Editor Application Model Structure

4.1.1 WorkingFlowElement Class

The class is a super abstract class and inherited by all other component classes in this editor. Because the components on the editor has the same attributes such as need to be saved on the file, be responsible to fire model change event and register the edit part as the property change listener, this class will implement IPropertySource, Cloneable and Serializable interfaces.

4.1.2 WorkingFlowSubpart Class

This class has defined the functions for all the components that have the attribute to be a graphical node. It includes method to add and remove the transition, and defines the properties such as size, location and name. Those properties will be used by the Edit Part to create and refresh the visual part.

4.1.3 WorkingFlowDiagram class

This class will define the property of the canvas on the editor. It includes methods to add and remove children and initialize the start point on the canvas. When external model objects are added on the diagram, they will be stored in a hashtable and past to start point, activity and condition object instances such that those instances can define their properties with the information provided by one or some of them. Then the workflow diagram can realize to interact with user's model. For example an activity instance can define that it represent a specific action of one external model object. The WorkingFlowStartPoint class will define a start point for the diagram. It has a function to reset all the property of the user model when the diagram is ready to run.

4.1.4 WorkingFlowActivity Class

This class implements the activity component on the editor. It defines the function to set its related external model object, its action and the action related arguments. A method run() implements what the activity will be run and the next() method is used to interact with the visual part to indicate this activity will be run next step. It also has a function to set the hashtable that holds all external object instances.

4.1.5 Workflow Editor Condition Classes

The condition components in the workflow editor are categorized to three kinds. A super class WorkingFlowCondition defines the function to check whether the condition is success, set the condition comparison rule and register the user model objects. As drawn on the model structure diagram, other three classes will implement their own functions to check the condition in different way. However the WorkingFlowSimpleCondition class is the actual one to interact with external model. It implements all the functions to set the necessary properties for the comparison rule and check whether the comparison is success. The comparison rules of the other two classes are based on the simple condition. They define combination conditions and do OR or AND check for them.

4.1.6 WorkingFlowGlobalComponent Class

This abstract class is the interface that all user model objects need to be extended. Via this class the workflow diagram will have a relationship with user application model. It defines all the functions that other editor components, such as activity and condition, need to use.

- *public abstract void run(String action, Vector arguments);*

  Run the function that matches with the action name and arguments.

- *public abstract List getProperties();*

  Return a list of properties that will be used to define the condition.

- *public abstract List getActions();*

  Return a list of actions that will be used to define the activity.

- *public abstract boolean checkCondition(String prp, String opt, String val);*

  Check condition with the given three arguments.

- *public abstract void initialProperties();*

  Initialize the properties in order to reset the property of the external model object.

- *public abstract String logProperties();*

  Return the string for outline viewer to trace the property changes.

4.1.7 WorkingFlowTransition Class

Transition is an important part on the workflow diagram. It describes which activity will be run next. This class defines the function to add and remove the transition to its source and target node, set the status to be yes/no, and insert bendpoints. The WorkingFlowBendpoint class will set the relative dimensions of a bendpoint, which will be used for the transition Edit Part to draw it on the canvas.

The important classes in the application model have been described above. Other auxiliary classes that used for the editor will not be listed detailed at this time.

4.2 Edit Part Implementation

Edit Part implementation is the most important task in this project. It can be divided into two parts – Edit Part and its Edit Policies. Every Edit Part class will represent an object in the application model. They have the same functionality to be the event handlers for the model modification and create a figure representation on the canvas for the model object. Moreover, the Edit Policy classes will be defined to accomplish corresponding editing roles for the Edit Parts. The commands that are triggered by the Edit Policy will be introduced later. Here the description of each Edit Part class will be focus on its editing role and related Edit Policies. The class hierarchy structure of the Edit Part is described on the following diagram, but Edit Policies classes are not list.

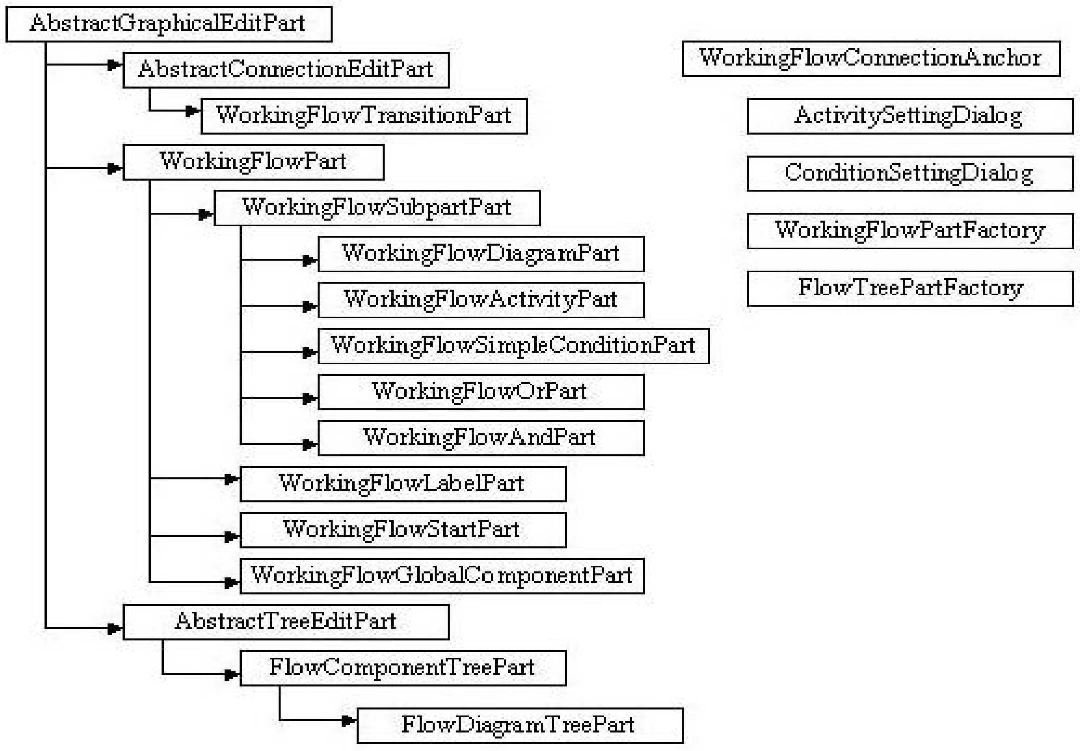

Figure 5: Workflow Editor Edit Part Class Structure

4.2.1 WorkingFlowTransitionPart Class

This class will be the Edit Part of the transition object. It installs the Edit Policies to manipulate the connection. The FlowBendpointPolicy class will be responsible for the `CONNECTION_BENDPOINTS_ROLE` to create command for creating, deleting and moving the bendpoints on the transition figure. The FlowTransitionPolicy class will extend the ConnectionEditPolicy and realize the `CONNECTION_ROLE` to create connection deletion command, and the FlowTranComponentPlicy class will be related to the `COMPONENT_ROLE` to create the command for setting the transition status to be yes or no.

4.2.2 WorkingFlowPart Class

This class is the super class for all the Edit Part classes that have the Edit Policy for the `GRAPHICAL_NODE_ROLE`. The FlowNodeEditPolicy class extends the GraphicalNodeEditPolicy, and creates the commands to create transition and reconnect the transition with its source or target nodes. If the node is an OR or AND condition Edit Part, all the condition Edit Parts in which will not be a valid source or target node. Moreover, this class has defined the connection anchor for the source and target nodes.

4.2.3 WorkingFlowSubpartPart and FlowDiagramTreePart Class

In WorkingFlowSubpartPart class, an important function is defined to extract the user model objects from the children list such that only the components that used to draw the workflow diagram will be shown on the canvas. In the FlowDiagramTreePart class it will have a function to return only the user model objects in the children list.

Therefore user's objects will be displayed on the outline viewer and separate from the workflow diagram.

4.2.4 WorkingFlowDiagramPart Class

This Edit Part class will represent the top-level canvas of the editor. It defines the RootComponentEditPolicy class to handle its COMPONENT_ROLE. Since all other Edit Part instances will be drawn on it, the FlowDiagramLayoutPolicy class is installed for its LAYOUT_ROLE, which will trigger commands to create new components and handle move operation. It defines the canvas to be a FreeformLayer and have a specific router rules for the transition and its bendpoints to follow.

4.2.5 WorkingFlowActivityPart Class

WorkingFlowActivityPart class defines editing role for the activity object in the model. The LAYOUT_ROLE is implemented with FlowActivityLayoutPolicy class, which will permit only one Label object to be added on it to describe the activity definition. The FlowActivityPolicy class is defined for its COMPONENT_ROLE, which will create commands to run or step run the diagram in addition to the deletion command. Also when the diagram runs, this Edit Part will refresh the visual part to indicate the change.

4.2.6 Workflow Editor Condition Edit Part Classes

As drawn on the class hierarchy diagram, the Workflow Editor condition Edit Part will separate to three classes. The WorkingFlowSimpleConditionPart class has all the editing roles as the WorkingFlowActivityPart class, but its COMPONENT_ROLE class only handles deletion command. The WorkingFlowOrPart and

WorkingFlowAndPart classes have the same LAYOUT_ROLE, which will allow the instances of the three kind conditions to be added on them.

4.2.7 WorkingFlowGlobalComponentPart Class

This class will be corresponding to object in user's application model. When the diagram runs, it will keep track of the property change information. But after I decide to display user's model on the outline viewer, this class is deprecated. Now it keeps as a point for future extension.

4.2.8 FlowComponentTreePart Classes

All the objects in user's application model can be mapped to a component on the workflow editor. This class will represent those objects and display them as figures on the outline tree.

4.2.9 ActivitySettingDialog and ConditionSettingDialog Classes

These two classes define the UI to hook the activity and condition components with the real actions and properties in user's application model. The ActivitySettingDialog will provide place for user to choose the activity related object, its action name and action related arguments. The ConditionSettingDialog also allow user to set the condition related object, its property name, compare operator and user defined property value. But now the operator is fixed to be some simple comparison, such as ">", "<" , "==" and so on, so only simple condition can be defined for this workflow editor.

4.3 Command and Action Implementation

Edit Policy will trigger the command to execute, so every command in the editor will do a specific operation to change the model. The CreateCommand and DeleteCommand class will be triggered to response for the LAYOUT_ROLE and COMPONENT_ROLE to create and delete a component in other components. The SetContraintCommand class will set new location and size for the components when they are moved on the canvas. There are six commands to manipulate the transition component. Two commands are used to create and delete the transition, other four commands are used to insert, remove and move the bendpoints in the transition figure. The ReconnectNodeCommand class defines the way to attach the transition with another source or target node.

How to run the diagram is an important issue for this editor, the algorithm is defined on two commands. The FlowRunCommand class defines the principle for running the diagram. From the start point on the canvas, the command will find the first activity component it connects to. Then the first activity begins to run. Via the transition the next component that defines in the diagram will be found, if the component is an activity, keep on running. Otherwise it will be a condition, which then check its condition to decide which transition will be chosen. Because the transition has set its status to be yes/or, the command will follow the matched transition to find the next component. Keep on doing this procedure, until the running process is finished. If the workflow defines an infinite loop, the run command will not stop. The FlowStepCommand class has the same way as the run command to find the next running activity, but it only run one activity, then stop and specify the next activity to be run. This command provides a way for user to keep track of the logic of the workflow diagram and check its correctness.

In the Workflow Editor some operations are done through the context menu or toolbar, for instance setting the activity and condition, running the diagram and resetting the properties of user model. Five actions will be contributed to the toolbar or context menu, which are *FlowActivitySetAction, FlowConditionSetAction, WorkingFlowRunAction, WorkingFlowStepAction and WorkingFlowResetAction.* Except the last action class extends the Action class, the other four all inherit from the SelectionAction. The *FlowActivitySetAction* can be performed by the activity Edit Part and popu up the activity setting dialog; The *FlowConditionSetAction* is performed by the condition Edit Part and pop up the condition setting dialog. The *WorkingFlowRunAction* and *WorkingFlowStepAction* will be performed by the start point on the canvas to begin executing the run or step command. When the step command is executed, the next running activity can perform these two actions to continue the run process.

4.4 Editor Part and Palette Viewer Implementation

Editor Part is the class that will be specified in the manifest file to initialize the workflow editor. The WorkingFlowEditor class is defined to perform this task. It extends the GraphicalEditorWithPalette class so that a Palette Viewer will be created with the canvas. The instance of WorkingFlowDiagram is set to be the content of the

GraphicalViewer, and a WorkingFlowEditorPaletteFactory class will define the content of the Palette Viewer. Components such as selection, marquee, connection, activity, three kind of conditions, label will be created by this factory class and displayed on the palette. A context menu with palette customizer is associated with the palette. In the WorkingFlowEditor class, functions also define to initialize the contents of the GraphicalViewer and PaletteViewer, configure them so that the Edit Part factory, key handler and context menu provider can be used by the editor, get the adapter for the outline viewer, and create actions that will be used to operate the editor. Corresponding functions are defined to save the diagram as object stream.

An OutlinePage class that extends the ContentOutlinePage is defined to be adapter of the outline viewer for the editor. It will synchronize with the selection operation on the editor canvas to create and delete component. The FlowTreePartFactory will be set for it to add user's object in the outline tree.

The difficult implementation for the PaletteViewer is to add new components on the palette. The palette customization dialog has been redefined so that a new EntryPage UI will be shown for user to set the component name, description and its template type. With this object type, the FlowTemplateCreationEntry class that extends ToolEntry can set the template type, and then creates corresponding object instances.

It is a challenge work to implement the workflow editor plug-in, but with the four parts that described above most functionality has been finished. A workflow diagram can be drawn and run on this editor.

5 Workflow Editor Testing

In order to test the functionality of the workflow editor, a clock application model is simulated to apply on the editor.

The clock model consists of three classes – Second, Minute and Hour, which all extends the WorkingFlowGlobalComponent class. Each class defines the content of the six abstract functions, in which it will do necessary task and fire property change event such that editor will handle its change and do corresponding update on the outline viewer.

After the model is finished, the three objects are added on the palette via the customizier on the context menu. Then they can be dragged from the palette and drop on the canvas to create instances. The figure representation will be displayed on the outline viewer. With the other components on the palette, a workflow diagram is drawn for the clock model. After defined the activity and condition components on the diagram with the properties and actions that are provided by the clock model, the diagram can be ready to run.

Different workflow diagrams are assembled for the clock application model on this editor. It proves that the editor has the functionality to draw the workflow diagram, simulate the behavior of a simple model, and check the correctness of the logic of the workflow.

6 Workflow Editor Application Process

The Workflow Editor is implemented with Eclipse 2.1.3 and GEF 2.1.3. It has been tested under Window 2000 and Window XP operating system. Other operating system may not support. Because the Eclipse and GEF is updated all the time, this plug-in may not consist with the new version Eclipse or GEF structure.

This manual has described the process to install and use the Workflow Editor.

- The workflow editor is a generic Eclipse plug-in, so it needs to be copied into the Eclipse plug-in directory.
- Open the Eclipse workbench window and add a new simple project. In the new simple project, create a file with extension ".wfe". A workflow editor with a palette will be opened for this file.
- There is a context menu associated with the palette; user can add new components via the palette customizer. Users' own model objects have to inherit from an abstract class (WorkingFlowGlobalComponent) in order to make it a part of the editor. In the palette customizer, the complete path of the model object has to be specified on the *type* text area such that the object will be created correctly. For instance in the clock application model, the Second class is defined at the package edu.carleton.workingfloweditor.model, so the whole path – edu.Carleton.workingfloweditor.model.Second need to input on the *type* text area.

- The component can be dragged from the palette and dropped on the canvas to create the workflow diagram. The components that represent user's objects will be displayed on the outline viewer.
- Transition can be connected between activities and conditions. From its context menu, user can define its status to be yes or no.
- Select an activity or condition on the canvas. From its context menu, a dialog will be popped up for user to define the action or comparison rule.
- Connect the start point on the canvas with the first activity or condition you want to run.
- Select the start point, then click the run or step on the context menu or the toolbar, the diagram will begin to run. If the run by step is chosen, running process will stop at the next will-running activity, which will display different color. Selects this activity and clicks run or step, the diagram will continue to run.
- There is a reset button on the toolbar. The properties of the components on the outline viewer will be reset after the button is clicked at any time. If the diagram is running by step but not terminated, it will be stopped.
- The diagram file can be saved with the save action on the context menu or the toolbar. It also can be open for editing next time.

7 Workflow Editor Design Limitations

This project has finished many functionalities of the workflow editor, but there are still some improvements for it to be continued, for instance:

- The workflow editor is a single thread program. When the diagram is run to simulate the behavior of the model, each step only one activity can be run or one condition can be checked even thought you can draw the diagram with many parallel activities or conditions.
- The UI of the condition component is still simple such that complicated comparison rule cannot be defined, for instance string and object instance cannot be compared.
- User's model has to extend an abstract class in the workflow editor to define some functions such that activity and condition components can hook with the user model. It's better if the super class is a simple interface that user model can implement but not need to extend.
- Only simple project model can be run on the workflow editor. If the model is involved with complicated object setting or other operations, the workflow editor cannot handle.
- The components that user adds via the context menu cannot be saved for next open.

8 Conclusion

After four months endeavor on this project, a Workflow Editor has been accomplished. Much functionality has been implemented and works robust. Most of all I have experienced a process from a novice that know nothing about the Eclipse to a person that understand the Eclipse Platform, its Plug-in Development Environment and its GEF plug-in, as well as master how to use its Plug-in and how to implement my own plug-in. It is a tough process, but I'm proud that I have made it. Through this process, I also have consolidated my technical skills with Java programming and have a deep understanding about the software design process. I'm so glad that I have experienced a meaningful process.

9 Reference

http://www.eclipse.org

http://www.devx.com/ibm/Article/6884

http://eclipse-wiki.info/GEF

http://eclipse-wiki.info/GefExamples

Eclipse GEF User Forum

Eclipse Logic Example

Eclipse Flow Example

Eclipse Help Manual