Introduction to Graphical Editing Framework

Applies To:

This article applies to all SAP technology groups that are involved in Graphical Editor Development on the eclipse platform. A prior knowledge of the basics of eclipse plugin development is recommended.

Summary

The article showcases the Graphical Editing Framework which can be used to develop Graphical Editors and applications for Application modeling, diagram editing, workflow representation and creation etc. on the eclipse platform.

By: Praveen Sinha

Company: Sap Labs India Pvt. Ltd.

Date: 01 January 2006

Table of Contents

1. What is GEF

Graphical Editor Framework or GEF as it is more commonly referred to as, is a framework to ease the task of building a graphical editor in Eclipse.

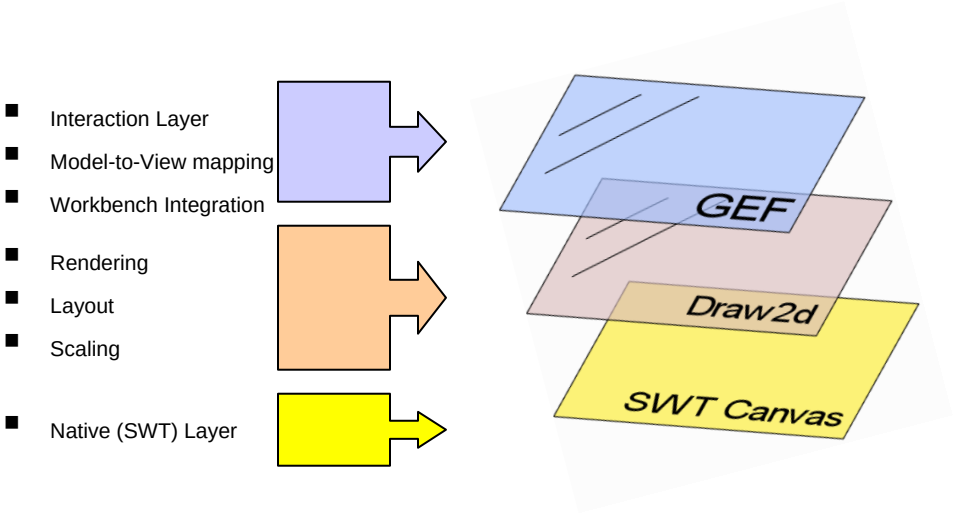


Figure 1. What is GEF

Design wise, GEF follows the MVC architecture, where GEF forms the interaction layer between the model and view, in addition since it is designed to work over eclipse, it provides a seamless eclipse workbench integration. View is contributed by Drw2D (part of GEF feature) figures, which handles all the rendering, layouting and scaling as well. All the drawing and user interaction is done over the SWT canvas.

a. GEF components and Dependencies

Let me try to explain, very briefly, about the composition and usages of the GEF library.

GEF is shipped as a feature of the eclipse workbench, the feature in addition to the GEF plugin also contains the Draw2D libraries. It extends the ui.views and the RCP (Rich Client Platform) features of eclipse and thus has a dependency over them. The draw2D library uses the SWT (Standard Widgeting Toolkit, a widgeting toolkit which uses the native widgets of the platform it runs on.)

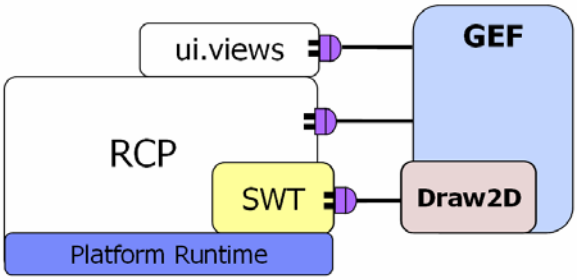


Figure 2. GEF components and dependencies.

The dependencies are realized by just plugging into the extension points that is exposed by these plugins to let others extend their functionality (see Figure 2).

2. Who can use it

The main idea behind GEF is to allow graphically tinkering with any available data model. If somehow you can create an object model of the data you have to tinker with, GEF can be used to give a graphical representation to this model and allows you to not only view or edit it, but it also allows you to create more of the same type.

The model elements listen to the changes being requested from the view as a result of certain actions. The views also listen to the changes in the model through callbacks (or listeners). In the event of any model change a repaint of the view is triggered and the newly painted figure represents the changed model. All of these callbacks, actions, repainting etc. are handled by the Interaction layer, i.e. GEF.

In addition to representing data and their relationships using simple and composite figures and connectors in various layouts, GEF also provides us with some extra features which helps make UIs developed using it really intuitive. Just to list some of the features, GEF supports Moving, Resizing and Creating of figures. It has a customizable palette viewer, which can be used to aggregate all the elements that can be added to the figure, and also the actions related to them. It also supports various editing features like delete, undo, redo and direct-editing of texts etc. For intuitiveness and a better accessibility it has features like figure overview, zooming, keyboard accessibility and animation support. For WYSIWIG editing it supports rulers, guides, grids, scale and snap to grid/geometry. In the next section we will see what these features actually are.

With all these features and a promise for many more to come, I think GEF is one place where developers would surely wish to venture to not least of all because GEF gives them an intuitive editable and configurable graphical representation to their data.

3. How can you use it

Till now I have tried to build an interest around GEF. I have talked about all the goodies you get out of it and all the things it brings with it. Now, let me start off with the part you surely must be itching to get to. How can you get this thing working for you?

I have already mentioned that GEF is based on the MVC architecture. Let's look at it step by step, i.e. for each of the entities, model, view and controller, one by one.

a. Model

Before starting off with the model let me introduce to you two new terms, Business Model and the View Model.

A Business Model is the actual representation of the data you want to work on. Technically we can use any model and make it interact with the GEF. For example if you would like to make a class diagram, the info like class name, methods, attributes and the relationship will be the business model for your editor.

The View model stores data like, where in the root figure (a view compartment where all other figures will be added to) a particular class has to be shown, what will be the height and width of the class figure, what colors, icons etc., would you like it to be decorated with. The figure below with the help of the above mentioned class diagram example, shows these two models with what they can represent.

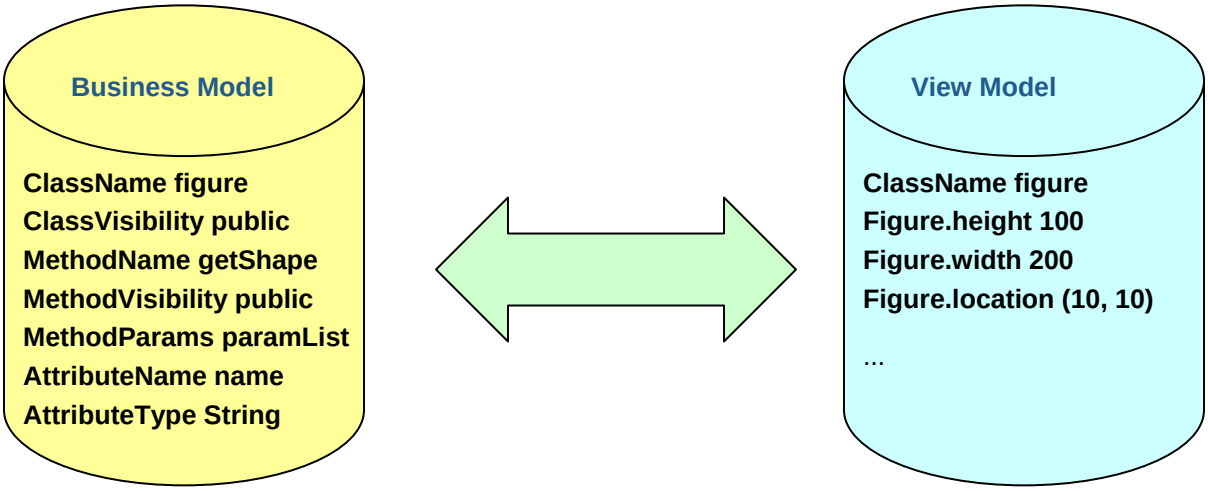


Figure 3. Business Model and View Model

In addition to this the view model can also implement the Interface IPropertySource to display the model element properties in the standard property sheet implementation in eclipse. The model should also implement some notification mechanism so that any change in it can be notified by fire of certain events which can be listened to by its registered listeners. The following code sample gives a sample skeleton for the view Model.

```
//An example of the view model
public class ObjectModel implements IPropertySource
{
   private PropertyChangeSupport listeners;
   private static IPropertyDescriptor [] descriptors;
   private Location location;
   private Dimension size;
   private BusinessModel bmodel;
   public ObjectModel()
   {
         listeners = new PropertyChangeSupport(this);
   }
   public Location getLocation(){
         return location;
   }
   public Dimension getSize(){
         return size;
```

```
}
public void setLocation(Location location){
      firepropertyChange("location", this.location, location);
      this.location = location;
}
public vois setDimesion(Dimension dimension){
      firepropertyChange("location", this.dimension, dimension);
      this.dimension = dimension;
}
public BusinessModel getBModel(){
      return bModel;
}
public void setbModel(BusinessModel bModel){
      this. bModel = bModel;
}
…
//code for rest of the model properties
…
public void addPropertyChangeListener(PropertyChangeListener listener)
{
      listeners.addPropertyChangeListener(listener);
}
public void removePropertyChangeListener(PropertyChangeListener listener)
{
      listeners.removePropertyChangeListener(listener);
}
protected void firePropertyChange(
      String propertyName,
      Object oldValue,
      Object newValue)
{
      listeners.firePropertyChange(propertyName, oldValue, newValue);
}

//for property sheet
public IPropertyDescriptor[] getPropertyDescriptors()
{
      if(descriptors == null)
```

```
                descriptors = new IPropertyDescriptor[0];
        return descriptors;
    }
    public Object getPropertyValue(Object propName)
    {
        return null;
    }

}
```

b. View

A view is contributed by the Draw2D figures. Draw2D is a Lightweight Toolkit built on top of SWT, built with a sole purpose: to support GEF. Concepts here and there are freely borrowed from Swing and other tools.

While creating a figure for the model representation, first a root figure is created, which is painted over the SWT figureCanvas. All the figures get attached to the root figure in a hierarchical manner as shown in the diagram below. The painting of the figure is done in accordance with the following rules,

- When the figures are painted, the order of painting us from top to bottom and from Left to right. Which means the parent figure is painted first followed by its children. Amongst the siblings the leftmost is painted first followed by the ones to its right.
- The parent figure clips its children.
- Last painted is on top
- Hit-Testing is in the reverse order of the painting.

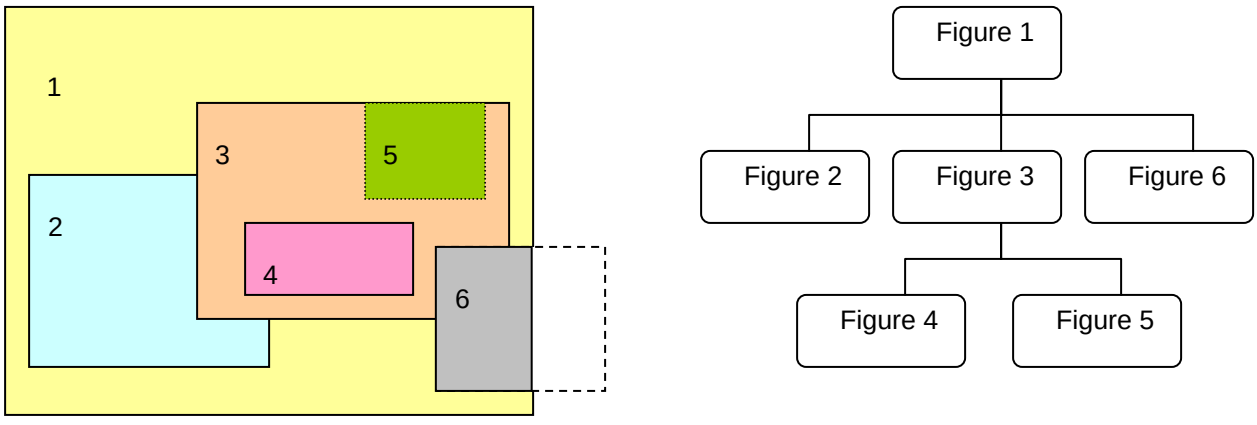


Figure 4. Figure painting, clipping and Hierarchy

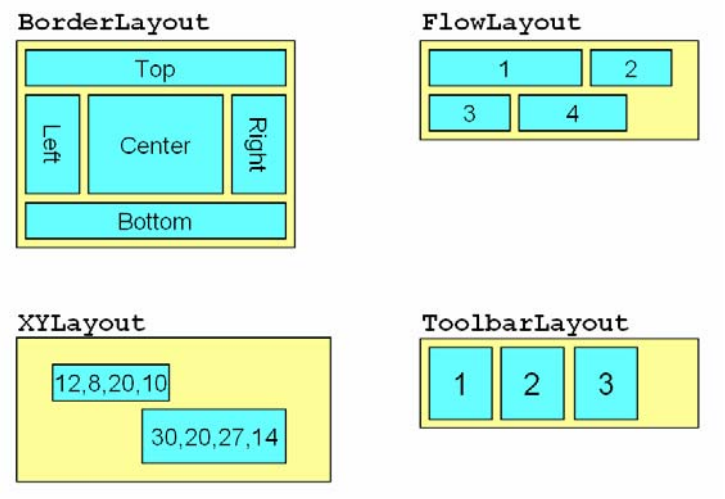
The figure in itself is quite illustrative. It shows the order of painting based on the position of the figure in the figure hierarchy. A lot of problem during painting arise from incomplete understanding of this behaviour of Draw2D.

A typical figure implementation is shown below with the help of a skeleton figure class.

```
public class RootFigure extends Figure
{
  public RootFigure(String text) {
        add(new Label(text));
        setLayoutManager(new ToolBarLayout(ToolBarLayout.VERTICAL));
        setBorder(new MarginBorder(new Insets(2)));
        add(new Label("child1"));
        contentPane = new Figure();
        contentPane.setLayoutManager(new ToolbarLayout());
        contentPane.setBorder(new TitleBorder("Children"));
        add(contentPane)
        …
  }
  public void setText(String text){
  super.setText(text)
  }
}
```

As you can see in the code, every Figure has to extend the class org.eclipse.draw2d.Figure which is the base implementation for graphical figures. The figure has two children, both of them are Label one with the text as param ‘text’ and the other as “child1”. Any type can be added as a child to a figure, if it extends Figure or any of its extensions.

What can also be observed from the code is the call to two setter methods, setLayoutManager and setBorder. Layout manager set for a figure determines how its children will be arranged inside it. Ofcourse you can implement your own layout manager as well. The figure below shows how some of the layoutmanagers shipped with Draw2D library does this.



Figer 5. Various layouts

The setBorder method sets the border for the figure again like the layouts, Draw2D provides many borders and for some specific border you can always go on and implement it.

One more thing worth noticing is the presence of a contentPane. A contentPane serves as the placeholder for any other figure, which is to be *contained* by this figure. This is used by the GEF to put the child elements to the model this figure corresponds to.

c. Controller

Controller is the most important part in any implementation of the MVC; here also it's not much different. Actually the whole gist of GEF is this. It streams the model info from the Model to the View, it takes the user interaction from the views to the model, it forces the view to refresh on any changes in the model. With all this to be done here its implementation is also the most difficult part of the whole story.

I will try to briefly describe the steps needed to come up with a quick implementation. But before this let's have a look at the model and the view as well, so that we can build the whole picture now with all the parts slowly but surely coming into life.

As discussed in the model section we are now in position to define a view model by adding some view details to the business model and also attaching the listeners to it. For each model we must have a corresponding view. Each view can be constructed using multiple figures as shown in the code example above. The contentPane as discussed above serves as a placeholder for the children of the model this figure corresponds to.

Now let us discuss about the part that will bind this two together. First of all the controllers in GEF are called EditParts. **There should be one editpart for each editable model element you want to represent in the views**. To define this part we have to follow these steps. The steps with their description are given below.

i. First we have to subclass AbstractGraphicalEditPart.

```
public class MyEditPart extends AbstractGraphicalEditPart{
}
```

ii. Register to the model change event. Remember when we talked about models, it was said that a model must have some way to notify the views of any changes in it. The editparts registers itself to the model it corresponds to for listening to the model change event fire.

```
public class MyEditPart extends AbstractGraphicalEditPart implements
PropertyChangeListener{
   public void activate() {
         super.activate();
         ((MyModel)getModel()).addPropertyChangeListener(this);
  }
   public void propertyChange(PropertyChangeEvent evt) {
         String prop = evt.getPropertyName();
         if (prop.equals("The property you are listening to")){
         //your reaction to the event resides here
   }
}
```

iii. Override createFigure(). You have already created views for all the model elements you have, now is the time to initialize them. Inside the MyEditPart override the method createFigure of the parent class. Lets look at the this with the help of code sample.

```
protected IFigure createFigure() {
   return new MyFigure();
}
```

iv. Define EditPolicies and create commands. Editparts don't handle editing directly. Instead, they use EditPolicies. Each editpolicy is then able to focus on a single editing task or group of related tasks. This also allows editing behavior to be selectively reused across different editpart implementations. Also, behavior can change dynamically, such as when the layouts or routing methods change.

Defining Edit policies is a complex and a very conceptual task. Normally I would have liked to explain it in a separate section dedicated to it. But just for a start, I don't want to rip deeper into the details and scare you off. First let me point down how this editing is triggered. Any editing is triggered directly or indirectly by some user interaction. The interactions can be of following types,

- Invoking some Action (usually displayed on the toolbar, menubar, or popup)
- Clicking on something
- Clicking and dragging something
- Hovering over something (pausing the mouse for a certain time)
- Dropping something dragged from another source (native Drag-N-Drop)

- Pressing certain keys

Based on the interactions there are certain requests that are created, there are also some EditPolicy Roles that is designated to handle that and inmost cases there are EditPolicy implementation already provided by GEF. The table below lists most of these,

Interaction	Requests	Edit Policies and Roles	Actions
Selection	SelectionRequest	SelectionEditPolicy	SelectAllAction
	DirectEditRequest	DirectEditPolicy	
	REQ_SELECTION_HOVER	SELECTION_FEEDBACK_ROLE	
	REQ_OPEN		
	REQ_DIRECT_EDIT		
Model Delete	REQ_DELETE	COMPONENT_ROLE CONNECTION_ROLE RootComponentEditPolicy	DeleteAction
Model Create	REQ_CREATE Create	CONTAINER_ROLE LAYOUT_ROLE TREE_CONTAINER_ROLE ContainerEditPolicy LayoutEditPolicy	CopyTemplateAction PasteTemplateAction
Moving and Resizing	ChangeBoundsRequest AlignmentRequest REQ_CLONE REQ_ALIGN REQ_RESIZE REQ_MOVE REQ_ADD REQ_ORPHAN	LayoutEditPolicy ResizableEditPolicy ContainerEditPolicy	AlignmentAction MatchSizeAction
Connection Creation	CreateConnectionRequest REQ_CONNECTION_START REQ_CONNECTION_END	GraphicalNodeEditPolicy NODE_ROLE	

Editing Connections	ReconnectRequest REQ_RECONNECT_SOURCE REQ_RECONNECT_TARGET	ConnectionEndpointEditPolicy ENDPOINT_ROLE GraphicalNodeEditPolicy NODE_ROLE	
Bending Connections	BendpointRequest REQ_MOVE_BENDPOINT REQ_CREATE_BENDPOINT	BendpointEditPolicy CONNECTION_BENDPOINTS_ROLE	

Now based on the Interaction and the type of request you are expecting as a result of the interaction you have to define the corresponding EditPolicy. You can either use any concrete EditPolicy Implementation, if available, or you can subclass or implement the one corresponding to your interaction. The process is very well documented in the GEF developer docs, which can be accessed in eclipse workbench, with GEF installed, by going to help-> help contents-> GEF-> editing and edit policies.

You also need to define some commands which will be executed from these edit policies.

v. Override createEditPolocies(). Here you install all the editpolicies defined above. You need to specify the Edit policy Role and the policy implementation for it.

```
protected void createEditPolicies() {

   installEditPolicy(EditPolicy.COMPONENT_ROLE, new
   RootComponentEditPolicy());

   …

   …

}
```

vi. Now after you have defined all your editParts you need to create a factory for them by implementing the EditPartFactory to tell the framework which editParts to load give a model element.

```
public class MyEditPartFactory implements EditPartFactory {
   public EditPart createEditPart(EditPart context, Object model) {
         EditPart part = null;
         if (model instanceof MyModel)
               part = new MyEditPart();
         else
         /*
```

```
                You have enter more conditions one for each of your model
                elements
                */
                return part;
            }
        }
```

With all this you should be able to generate a very basic implementation of GEF. Of course you need to make an editor plugin from where all this can be instantiated. Also there are many things which I have left untouched in this article like PaletteViewer, Connectors, animation, zoom-in, zoom-out, grids, snap to grids etc., but the intention of the article was to help the readers to give a hit at GEF. Maybe in future I may come up with a follow up.

Author Bio

Praveen Sinha is currently working with the Visual Composer Development team in SAP Labs India. Before joining this team he was involved in development of GEF Editors for application modeling and visual tools.

Disclaimer & Liability Notice