MVC your graphics with Eclipse GEF

An introductory tutorial

Bart van Riel

January 2008

www.tutorials.tfo-eservices.eu

Table of Contents

1.Why this tutorial?

This tutorial is about GEF, the *Graphical Editing Framework* for the Eclipse platform. Basically, it's the API which let's you develop anything from a frame with a simple colored square to complex flowchart drawing applications. So it's probably worth a look and I decided some time ago to give it one.
Enter the usual suspect: the Steep Learning Curve! As with most (all?) powerful API's, with great power comes great incomprehensibility. GEF is no exception to the problem that the vast functionality of a powerful API also means you need to spend a considerable amount of time and energy getting to understand all the different class hierarchies, interfaces and properties involved. I had to climb the learning curve.
Now, there's a habit I have developed the past few years (which my loyal readers will recognize): I report my learning curve climbings in tutorials. Basically, if the available on line documentation is lacking or not as usable as I would like it to be, I write a tutorial to fill in those gaps. This tutorial is an example of that modus operandi.

The purpose of this tutorial is not to give an in-depth training on GEF. Actually, it's rather superficial and does not do much more than introduce the basic workings of GEF to use an Eclipse RCP viewer and paint some simple graphics into it. This tutorial aims to provide the entry level introduction I found to be lacking on line. (Other documentation starts by laying out complex application functionality like connecting shapes with lines, drag-and-drop and various editing events. I'm not even going to touch that.)

So here's the quick rundown of this tutorials' contents. First, I start with a very (very, very!) simple example on Draw2d, the basic graphical API in Eclipse. After that, some discussion of the general ideas behind GEF and the responsibilities of the participating classes. After that, naturally, another example but this time using the GEF setup. And of course a Resources Section at the end.

2.Setting up your environment

This tutorial comes with two example projects in a ZIP. They were developed using Eclipse SDK 3.3 and with the “Graphical Editing Framework SDK” added through the update tool. So you'll need those two, and the projects should be OK to import, build and run.

3.First things first: Draw2d

Although GEF simplifies graphical applications by providing a concrete MVC framework for, you still have to draw some graphics on the screen. That's where Draw2d comes in. So before going into GEF, let's have a quick look at Draw2d.
According to the GEF product documentation, “Draw2d provides lightweight rendering and layout capabilities on an SWT Canvas”, which basically means it makes it easy to draw shapes in a window. Draw2d's claim for being 'lightweight' also means it's easy to do simple drawing which the following example will show.
We'll see the example output first and then the code with some explanation. Here's the simple example:

Simple Draw2d GridLayout example

Fancy, 'eh? Now, Draw2d enables all kinds of complex drawings including lines, text labels with markup and so on , but for the sake of the example I'll keep things simple.

Here's the source code for this example:

```
public class SimpleDraw2dGridLayoutTest {

	public static void main(String[] args) {
		// Create window
		Display d = new Display();
		final Shell shell = new Shell(d);
		shell.setSize(300, 300);
		shell.setText("Simple Draw2d GridLayout Test");
		shell.setBackground(new Color(null, 255, 255, 255));

		// Setup contents
		// Start with 'drawing canvas' instantiation
		LightweightSystem lws = new LightweightSystem(shell);
		// Create a base figure to relate other graphical elements to
		Figure container = new Figure();
		// We need a layout, in this case 3 columns wide
		GridLayout gridLayout = new GridLayout();
		gridLayout.numColumns = 3;
		container.setLayoutManager(gridLayout);

		// Create the graphical elements we want to show.
		// In this simple case, all will be rectangles.
		// They will be distributed among 3 columns using
		//  as many rows as needed, so three rows of 3 and
		//   one row of 1
		for(int i = 0; i < 10; i++){
			Shape rect;
			rect = new RectangleFigure();
			//color turns lighter with increment of 'i'
			Color backGroundColor = new Color(null, i*25, i*10, i*20);
			rect.setBackgroundColor(backGroundColor);
			rect.setBounds(new Rectangle(50, 50, 50, 50));
			// Add the new rectangle to the container figure
			container.add(rect);
		}
		//add container to drawing canvas
		lws.setContents(container);

		// show window
		shell.open();
		while (!shell.isDisposed())
			while (!d.readAndDispatch())
				d.sleep();
	}
}
```

Here's a quick rundown on some of the interesting parts in the class.

The line:

```
	LightweightSystem lws = new LightweightSystem(shell);
```

creates the drawing canvas and attaches it to the application window (the `shell`).

The lines

```
Figure container = new Figure();
	GridLayout gridLayout = new GridLayout();
	gridLayout.numColumns = 3;
```

```
container.setLayoutManager(gridLayout);
```

create a base `Figure` instance which serves as a graphical container for all other drawn shapes. We also set the layout to a `GridLayout` of 3 columns wide. The layout will automatically distribute drawn shapes along as many rows as needed.

The lines

```
Shape rect;
rect = new RectangleFigure();
...
container.add(rect);
```

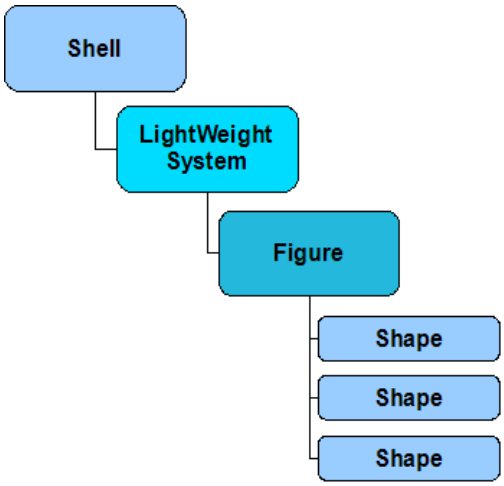
create a new `Rectangle` and add them to the base `Figure`. Note that the GridLayout causes the 10 elements to be distributed over three rows of 3 and one row of 1 element.

Finally, the line

```
lws.setContents(container);
```

Adds the base figure to the canvas.

Note that what is actually created here is a hierarchy of graphical objects of a window which holds a canvas, which in turn holds a Figure which in turn holds Shapes. In graphical form:



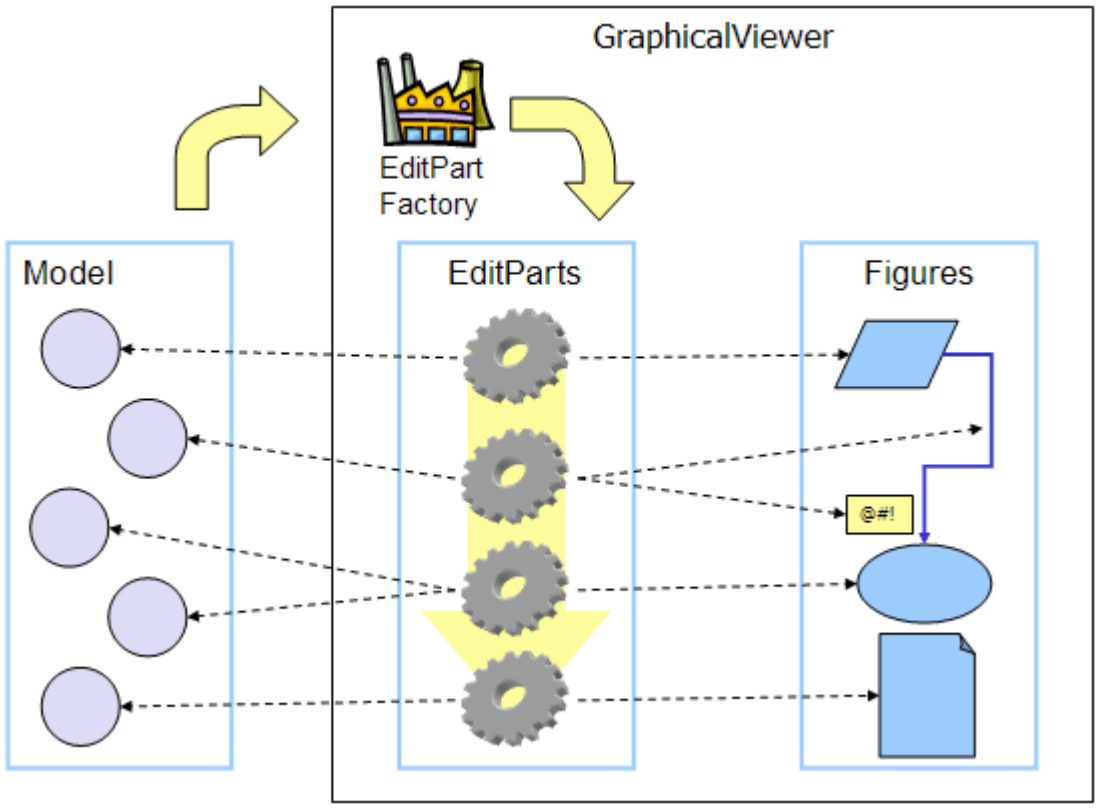
Drawing objects hierarchy

This hierarchical notion is important to keep in mind when examining GEF in the next chapter. The fact that the View part of MVC in GEF has a specific structure associated with it also places some constraints on the Model and Controller parts. We will also see that GEF packs some helper classes which help you with setting up things like the Lightweight System and the Shell.

4.Graphical Editing Framework (GEF)

GEF Basics

As the name suggests, GEF adds editing support to graphical applications. Basically, GEF allows you to detect user events on the graphics and manipulation of those graphics. The possibilities are advanced. But I won't go into them.
What I will get into, is the basic implementation and usage of the MVC framework GEF introduces. Even without direct editing support, GEF offers a very useful way of decoupling data from the graphical display of that data. Modifying the display can simply be done by modifying the data and letting the framework update the graphics. Before we get into this little wizardry, take a look at the schematic below:

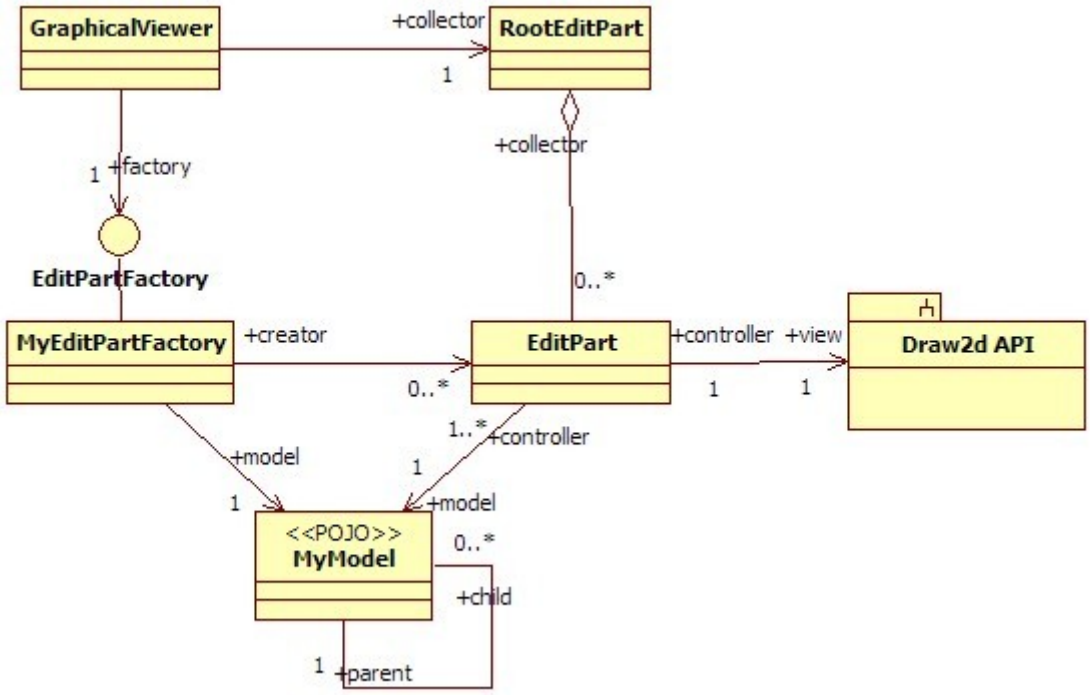


GEF MVC Schematic

Although a picture can say more than 1000 words, let me add these words to the picture:

- You have a *Model* holding your data;
- The model data is displayed in a *View* composed of Figures;
- EditParts fill Figures with Model data (and vice versa through editing support, but we won't cover that in this tutorial), thus acting as *Controllers*;
- EditParts are created and initialized (i.e. injected with Model data etcetera) by an EditPartFactory which does what its name suggests;
- A GraphicalViewer takes care of displaying the figures and calling all the appropriate methods of all the other participants.

The picture above makes for fun viewing, but it does not accurately reflect all participants and their relations. The following diagram does:



Basic GEF participating classes

If nothing else, the above class model demonstrates how thin a veneer over Draw2d GEF acutally is if you leave out all the editing stuff. Even so, the MVC framework it introduces gives you a very useful level of control, as will become clear (I promise) in the upcoming examples.

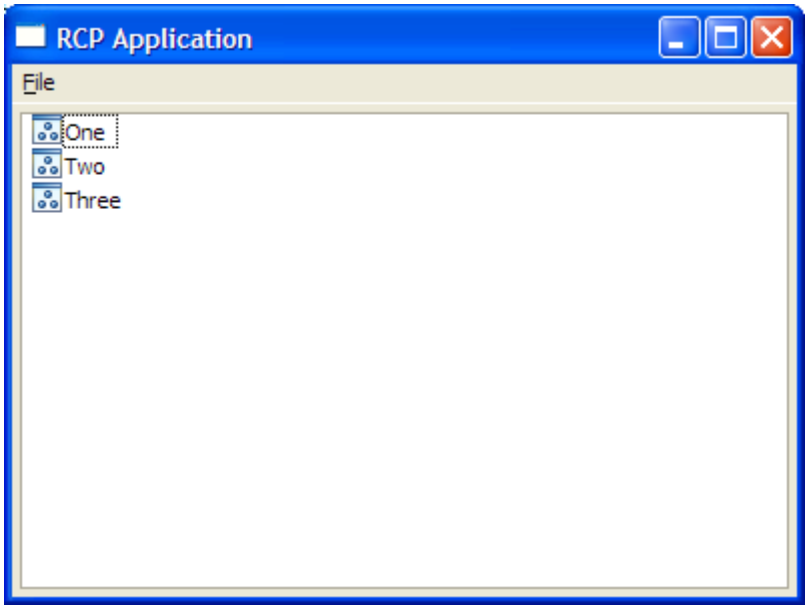
GEF Example

Eclipse plugin project

The example will be the same application as the Draw2d example, but written as an Eclipse plugin and using an Eclipse 'View' by means of a graphics display. I'm not going into details on Eclipse RCP here, except for basic instructions on how to create a starting point plugin (alternatively, get the sourcecode project from the Resources section and import it in Eclipse yourself).
Quick steps:

1. Select File-->New Project-->Plugin Project and click “Next”;
2. Use “GEFTutorial” as the project name and click “Next”;
3. Click “Next” again
4. Select the “RCP Application with a view” template and click “Next”;
5. Click “Finish” (if asked to switch to the “Plugin perspective” choose “Yes”.

The project now opens with the plugin editor open (alternatively, double click the “plugin.xml” in the project folder to open this editor). On the “Overview” tab of the plugin editor in the section called “Testing”, click the link “Launch an Eclipse Application” and you should see this:



Initial RCP app with a view

The MVC classes

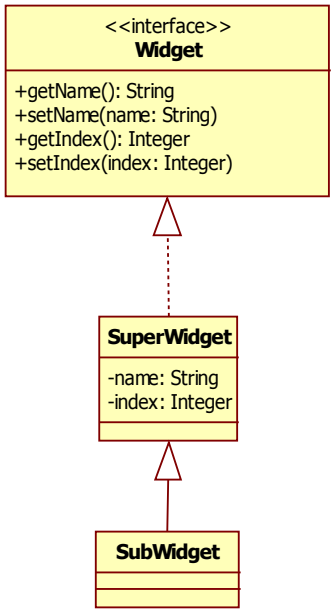
I haven't mentioned this earlier (because I was afraid you might run off) but even with GEF, you actually have to write some code yourself. You have to provide implementations of the following classes:

- A class implementing the `EditPartFactory` interface and the method `EditPartFactory.createEditPart();`
- Specialized `EditPart` classes, which will typically extend `AbstractGraphicalEditPart`, and override the appropriate methods which initialize `Figures` (there they are again) from Model data and/or refresh them as requested (via method calls). Actually you will need at least two types of `EditPart`, one 'base' edit part which holds the root of the model and some other `EditPart` implementation for the various model elements (usually one `EditPart` implementation per model element type);
- Your Model. Obviously.

By the way, the root of your model is NOT associated with the `RootEditPart` class shown in the UML model. The `RootEditPart` holds `EditPart`s, the first `EditPart` holds the model root. Don't worry, it will all get clear when we get to the code.

The Model classes

For the model, let's use one of the most interesting real world entities in existence: the widget. To actually have something to display, we'll give the widget two attributes: a `name` and an `index`. And let's make a distinction between a `SuperWidget` and a `SubWidget` (mostly because a parent-child relation makes sense in any model). Finally, for the sake of polymorphism, we'll throw in the interface `Widget` containing the getters and setters and let it be implemented by the two classes. To cut a long story short:



Widget UML model

And the sourcecode:

Widget.java:

```
public interface Widget {
	public void setName(String name);
	public String getName();
	public void setIndex(int index);
	public int getIndex();
}
```

SuperWidget.java:

```
public class SuperWidget implements Widget {
	private List<Widget> myWidgets = new ArrayList<Widget>();
	private String name;
	private int index;
	private int hashCode =
	  (int)(System.currentTimeMillis() *  (1000*Math.random()));

	public SuperWidget(String name){
		this.name = name;
	}

	public void addWidget(Widget aWidget){
		myWidgets.add(aWidget);
	}

	public List<Widget> getChildren(){
		return myWidgets;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
```

```
            this.name = name;
      }

      @Override
      public boolean equals(Object obj) {
            if(!(obj instanceof Widget) || null == obj){
                  return false;
            } else {
                  Widget widget = (Widget)obj;
                  if(null != widget.getName() && null != name){
                        return name.equals(widget.getName());
                  } else {
                        return false;
                  }
            }
      }

      @Override
      public int hashCode() {
            return hashCode;
      }

      public void createDummyModel(){
            for (int i = 0; i < 10; i++){
                  Widget subWidget = new SubWidget("Widget no. " + i);
                  subWidget.setIndex(i);
                  this.addWidget(subWidget);
            }
      }

      @Override
      public int getIndex() {
            return index;
      }

      @Override
      public void setIndex(int index) {
            this.index = index;
      }
}
```

SubWidget.java:

```
public class SubWidget extends SuperWidget implements Widget {

      public SubWidget(String name){
            super(name);
      }
}
```

The EditPart classes

Usually there will be at least two sorts of EditParts in your application. The first is the 'contents' or 'root' EditPart which holds the root element of your model. The others are true 'controller' EditParts, each responsible for controlling a specific portion of the model (other than the root).
The difference in responsibilities between the EditParts is not only reflected how they handle the model, but also in the graphical behaviour the EditParts control. The 'root' EditPart is responsible for setting up a container Figure with an initial Layout (analogous to the Figure container instance in the Draw2d example, see how useful that simple app is here?). The 'controller' EditParts are responsible for providing the graphics for their associated model elements.

Let's not go into more details here, but take a look a the source code. SimpleGEFContentsEditPart is the 'root' EditPart and WidgetEditPart is the controller for individual Widgets.
SimpleGEFContentsEditPart.java:

```
public class SimpleGEFContentsEditPart extends AbstractGraphicalEditPart {
	@Override
	/**
	 * Create root figure. Use standard FreeformLayer figure here.
	 */
	protected IFigure createFigure() {
		//Create FreeformLayer figure as layout base for other
		// graphical elements
		Figure f = new FreeformLayer();
		f.setBorder(new MarginBorder(1));
		//Create a layout for the graphical screen
		GridLayout gridLayout = new GridLayout();
		gridLayout.numColumns = 7;
		gridLayout.horizontalSpacing = 0;
		gridLayout.verticalSpacing = 0;
		gridLayout.marginHeight = 0;
		gridLayout.marginWidth = 0;
		f.setLayoutManager(gridLayout);
		return f;
	}

	//Return the lower elements of the model
	protected List getModelChildren(){
		return ((SuperWidget)getModel()).getChildren();
	}

	@Override
	protected void createEditPolicies() {
		// Not editing, so keep empty...
	}
}
```

WidgetEditPart.java:

```
public class WidgetEditPart extends AbstractGraphicalEditPart {
	private Font widgetFont;

	@Override
	protected IFigure createFigure() {
		//Create simple rectangle
		IFigure figure = new RectangleFigure();
		//Set default background color to gray
		figure.setBackgroundColor(new Color(null, 200, 200, 200));
		return figure;
	}

	@Override
	public void activate() {
		super.activate();
	}

	@Override
	public void deactivate() {
		//Do cleanup of resources
		if (widgetFont != null) widgetFont.dispose();
		super.deactivate();
	}
```

```
	protected void refreshVisuals() {
		//Get the model element associated with this
		// EditPart instance
		Widget widget = (Widget)getModel();

		//This is where the actual drawing is done,
		// Simply a rectangle with text
		Rectangle bounds = new Rectangle(50, 50, 50, 50);
		getFigure().setBounds(bounds);
		Label label = new Label(widget.getName());
		widgetFont = new Font(null, "Arial", 6, SWT.NORMAL);
		label.setFont(widgetFont);
		label.setTextAlignment(PositionConstants.CENTER);
		label.setBounds(bounds.crop(IFigure.NO_INSETS));
		getFigure().add(label);
		//Make every 'even' SubWidget purple-ish
		if(widget.getIndex()%2 == 0){
		 getFigure().setBackgroundColor(new Color(null, 255, 100, 255));
		}
	}

	@Override
	protected void createEditPolicies() {
		// Not editing, so keep empty...
	}
}
```

The EditPartFactory class

This factory class is probably the most straightforward of them all. The name says it all: `SimpleGEFEditPartFactory`. Basically, it returns an appropriate `EditPart` instance. Note the injection of the model element into the `EditPart.setModel()` method. The `EditPartFactory.createEditPart()` is called with the actual model element being traversed at that time, so it's both safe and necessary to inject the model elements' reference to the controlling (newly created) controlling `EditPart`.

SimpleGEFEditPartFactory.java:

```
public class SimpleGEFEditPartFactory implements EditPartFactory {
	@Override
	public EditPart createEditPart(EditPart context, Object model) {
		EditPart editPart = null;
		if(model instanceof SubWidget){
			//Standard editpart instance for standard model element
			editPart = new WidgetEditPart();
			editPart.setModel(model);
		} else if(model instanceof SuperWidget){
			//Root instance of editpart for root model element
			editPart = new SimpleGEFContentsEditPart();
			editPart.setModel(model);
		}
		return editPart;
	}
}
```

The View class

OK, so we now have the Model and the Controller in place (and the Controller's factory). This leaves the

View. You may not expect this, but the view is actually the simplest part of the application in this case because GEF provides very powerful default implementations of visual components for you. Basically, you implement an RCP View (or Editor) with:

- An instance of `ScrollingGraphicalViewer` to use as a viewport with scrollbars;
- An instance of `ScalableFreeFormRootEditPart` which holds all your `EditPart`s and allows painting of whatever shape you want;
- An instance of your `EditPartFactory`, in this case `SimpleGEFEditPartFactory`;
- A reference to your Model.

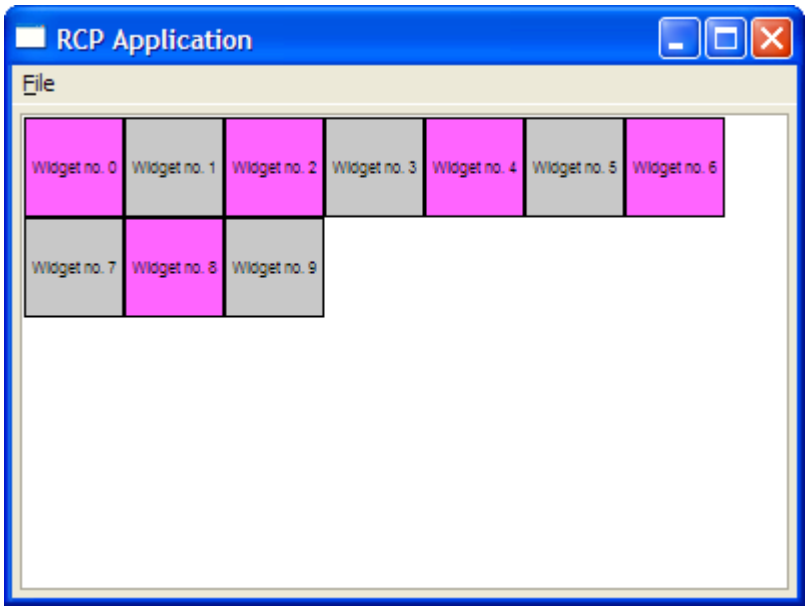
Beyond that, it's a matter of a few lines to setup and link those instances. They configure themselves for the most part. `ScrollingGraphicalViewer` takes care of traversing the Model and calling the appropriate methods it's participants (as we have seen) to create `EditPart` instances and have `EditPart`s create graphics.
Take a look at the source code below, which shows that setting up the viewer involves not much more than creating the appropriate objects and some setup in the `ViewPart.createPartControl()` method.

View.java:

```
public class View extends ViewPart {
	public static final String ID = "GEFTutorial.view";

	//Use a standard Viewer for the Draw2d canvas
	private ScrollingGraphicalViewer viewer = new ScrollingGraphicalViewer();
	//Use standard RootEditPart as holder for all other edit parts
	private RootEditPart rootEditPart = new ScalableFreeformRootEditPart();
	//Custom made EditPartFactory, will automatically be called to create
edit
	// parts for model elements
	private EditPartFactory editPartFactory = new SimpleGEFEditPartFactory();
	//The model
	private SuperWidget model;

	/**
	 * This is a callback that will allow us to create the viewer and
initialize
	 * it.
	 */
	public void createPartControl(Composite parent) {
		//Create a dummy model
		model = new SuperWidget("Model");
		model.createDummyModel();
		//Initialize the viewer, 'parent' is the
		// enclosing RCP windowframe
		viewer.createControl(parent);
		viewer.setRootEditPart(rootEditPart);
		viewer.setEditPartFactory(editPartFactory);
		//Inject the model into the viewer, the viewer will
		// traverse the model automatically
		viewer.setContents(model);
		//Set the view's background to white
		viewer.getControl().setBackground(new Color(null, 255,255,255));
	}

	/**
	 * Passing the focus request to the viewer's control.
	 */
	public void setFocus() {
		viewer.getControl().setFocus();
	}
}
```

Running the example application

When you run the example application, you should see the following:



Example GEF Application Result

Now isn't that impressive?

Congratulations! You have just completed the GEF enabled application. Try fiddling around with the font, colors, layout and so on to familiarize yourself a bit with the possibilities and behaviour. Note that you will probably only need to adjust the `EditPart.createFigure()` and `EditPart.refreshVisuals()` methods because that's where the graphical action takes place. Enjoy!

5.Resources

The Draw2d example was inspired by the Eclipse Corner Article “A Shape Diagram Editor” by Bo Majewski, http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.
The homepage of the Graphical Editor Project at Eclipse.org provides all kinds of useful information, including instructions on downloading and installing the GEF plugins, documentation and libraries: http://www.eclipse.org/gef/.
At IBM DeveloperWorks you'll find this tutorial giving a thorough (although somewhat complex) outline of the use of GEF: “Create an Eclipse-based application using the Graphical Editing Framework” by Chi Aniszcyk et al., http://www.ibm.com/developerworks/opensource/library/os-eclipse-gef11/.
IBM also provides an excellent redbook on GEF (and the Eclipse Modeling Framework by the way): “Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework”, http://www.redbooks.ibm.com/abstracts/SG246302.html?Open.
Finally, for a quick overview you may want to check out the presentation on Draw2d and GEF given at the 2005 EclipseCon. You can download the presentation, “GEF In Depth”, through this page: http://www.eclipsecon.org/2005/tutorials.php.