Create an Eclipse-based application using the Graphical Editing Framework

How to get started with the GEF and other options for the graphically inclined Eclipse developer

Chris Aniszczyk, Software Engineer, IBM
Randy Hudson, Software developer, IBM

Summary: Learn the initial steps involved in creating an Eclipse-based application using the Graphical Editing Framework (GEF). Also, discover the options you have these days to bootstrap the process of creating graphical editors in Eclipse.

Date: 27 Mar 2007
Level: Intermediate
Activity: 2646 views
Comments: 0 (Add comments)

★★★★☆ Average rating

***Editor's note**: The following article, originally written by Randy Hudson in July 2003, was updated in March 2007 by Chris Aniszczyk.*

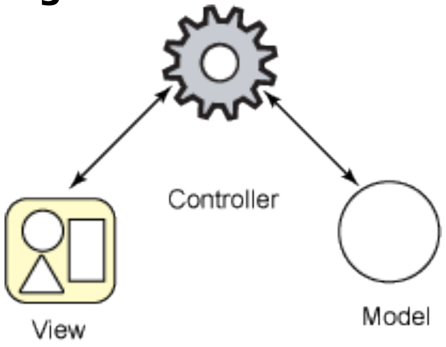
This article walks through the steps for using the Graphical Editing Framework (GEF). Rather than finishing each step in its entirety, we'll use a subset of your application's model and get that working first. For example, we might initially ignore connections or focus on just a subset of the types of graphical elements in your application. Next, learn what other technologies are available to add graphical editing to your applications. In the past, stand-alone GEF used to be the only option for graphical editing in Eclipse, but this has changed as Eclipse has evolved.

Overview of GEF

GEF assumes you have a model you would like to display and edit graphically. To do this, GEF provides viewers (of the type `EditPartViewer`) that can be used anywhere in the Eclipse workbench. Like JFace viewers, GEF viewers are adapters on an SWT Control. But the similarity stops there. GEF viewers are based on a Model-View-Controller (MVC) architecture.

The controllers bridge the view and model (see Figure 1). Each controller, or `EditPart` as they are called here, is responsible for mapping the model to its view and for making changes to the model. The `EditPart` also observes the model and updates the view to reflect changes in the model's state. EditParts are the objects with which the user interacts. `EditPart`s are covered in detail later.

Figure 1. Model-View-Controller



GEF provides two viewer types: graphical and tree-based. Each hosts a different type of view. The graphical viewer uses figures that paint on an SWT `Canvas`. Figures are defined in the Draw2D plug-in, which is included as part of GEF. The `TreeViewer` uses an SWT `Tree` and `TreeItem`s for its view.

Step 1: Bring your own model

GEF knows nothing about a model. Any model type should work, as long as it meets the properties described below.

What's in a model?

Everything is in the model. The model is the only thing persisted and restored. Your application should store all important data in the model. During the course of editing, undo, and redo, the model is the only thing that endures. Figures and EditParts will be garbage collected and recreated over time.

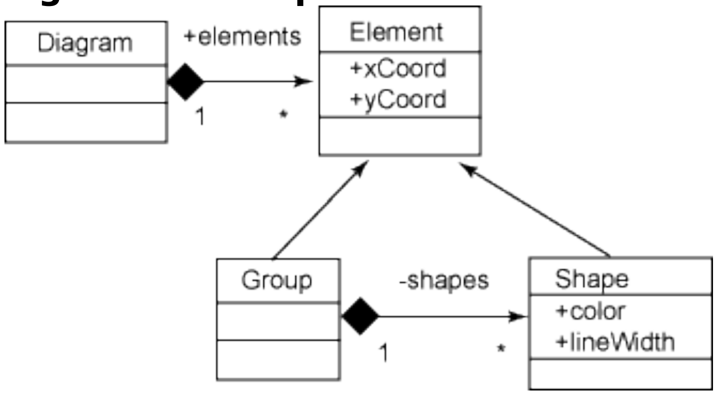
When the user interacts with `EditPart`s, the model is not manipulated directly by the `EditPart`s. Instead, a `Command` is created that encapsulates the change. `Command`s can be used to validate the user's interaction, and to provide undo and redo support.

Strictly speaking, `Command`s are also conceptually part of the model. They are not the model per se, but the means by which the model is edited. Commands are used to perform all of the user's undoable changes. Ideally, commands should only know about the model. They should avoid referencing an `EditPart` or figure. Similarly, a command should avoid invoking the user interface (such as a pop-up dialog) whenever possible.

A tale of two models

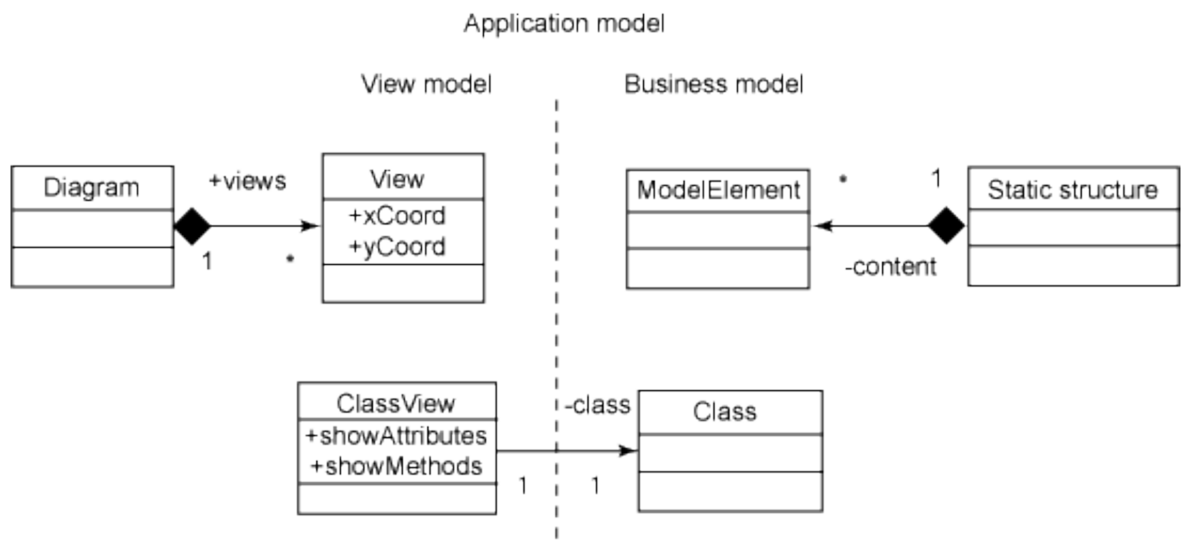
A straightforward GEF application is an editor for drawing diagrams. (Here, *diagram* means just a picture, not class diagram.) A diagram can be modeled as some shapes. A shape might have properties for location, color, etc., and may be a group structure of multiple shapes. There are no surprises here, and the previous requirement is easily maintained.

Figure 2. A simple model



Another common GEF application is a UML editor, such as a class diagram editor. One important piece of information in the diagram is the (x, y) location where a class appears. You might assume that the model must describe a class as having an x and y property. Most developers want to avoid polluting their models with attributes that don't make sense. In such applications, the term *business model* can be used to refer to the base model in which the important *semantic details* are stored. While diagram-specific information is stored in the *view model* (which means a view of something in the business model; an object may be viewed multiple times in one diagram). Sometimes the split is even reflected in the workspace, where different resources might be used to persist the diagram and business model separately. There may even be several diagrams for the same business model (see Figure 3). A common term for view models is *notational models*.

Figure 3. A model split into business and view models



Whether your model is split into two parts, or even multiple resources, it does not matter to GEF. The term *model* is used to refer to the whole application model. An object on screen may correspond to multiple objects in the model. GEF is designed to allow you to handle such mappings easily.

Notification strategies

Updates to the view should almost always be the result of notification from the model. Your model must provide some notification mechanism, and this mechanism must be mapped to the appropriate updates in your application. Exceptions might be read-only models or models that cannot notify, such as a file system or a remote connection.

Notification strategies are usually distributed (per object) or centralized (per domain). A domain notifier knows about every change to any object in the model and broadcasts these changes to domain listeners. If your application employs this notification model, you will probably add a domain listener per viewer. When that listener receives a change, it will look up the affected `EditPart`s, then redispatch the change appropriately. If your application uses distributed notification, each `EditPart` will typically add its own listeners to whichever model objects affect it.

Step 2: Define the view

The next step is decide how you will display your model using figures from the Draw2D plug-in. Some figures can be used directly to display one of your model's objects. For example, the Label figure can be used to display an Image and String. Sometimes the desired results can be achieved by composing multiple figures, layout managers, or borders. Finally, you may end up writing your own figure implementations that paint in a way specific to your application. More information about composing or implementing figures and layouts can be found in the Draw2D developers guide included with the GEF SDK.

When using Draw2D with GEF, you can make your project easier to manage and more flexible to changing requirements by following these guidelines:

Don't reinvent the wheel
: You can combine the provided layout managers to render most things. Consider composing multiple figures using combinations of the toolbar layout (in vertical or horizontal orientation) and the border layout. Only as a last resort should you write your own layout manager. As a reference point, look at the palette provided in GEF. The palette is rendered using many of the standard figures and layouts from Draw2D.

Keep a clean separation between EditPart and figure
: If your EditPart uses a composite structure of several figures, layouts, and borders, keep as much of that detail hidden from the EditPart. It is possible -- although not a good idea -- to have the EditPart build everything itself. But doing this does not lead to a clean separation between controller and view. The EditPart has intimate knowledge of the figure structure, so reusing that structure with a similar EditPart is not possible. Also, changing the appearance or structure may lead to unexpected bugs. Instead, you should write your own subclass of Figure that hides the details of its structure. Then define the minimal API on that subclass that the EditPart (the controller) uses to update the view. This practice, referred to as the *separation of concerns*, results in greater reuse and fewer bugs.

Don't reference the model or EditPart from the figure
: The figure should not have access to the EditPart or model. In some situations, the EditPart may add itself as a listener to the figure, but it will only be known about as a listener, not as the EditPart. This practice of de-coupling also yields greater reuse.

Use a contents pane
: Sometimes you have a container that will contain other graphical elements, but you need decorations around the outside the container. For example, a UML class is typically shown as a box, where the top portion is labeled with the class name and perhaps some stereotypes, and the bottom portion is reserved for attributes and methods. This can be done by composing multiple figures. The first figure is the title box for the class, while another figure is designated as the contents pane. This figure will eventually contain the figures for attributes and methods. When writing the EditPart implementation later, it is trivial to indicate that the contents pane should be used as the parent for all children elements.

Step 3: Write your EditParts

Next, we'll bridge the model and view with the controller, or EditPart. This is the step that puts the "framework" in GEF. The provided classes are abstract, so clients must actually write code. It turns out that subclassing is not only familiar but is probably the most flexible and straightforward way to map from model to view.

There are three base implementations provided for subclassing. Use `AbstractTreeEditPart` for EditParts that appear in the tree viewer. Extend `AbstractGraphicalEditPart` and `AbstractConnectionEditPart` in graphical viewers. We will focus on graphical EditParts here. The same principles apply to use in the tree viewer.

EditPart life cycle

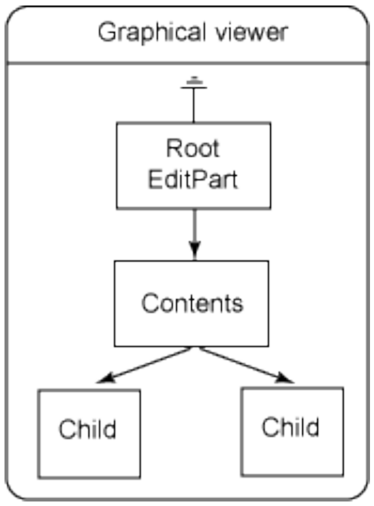
Before you write your EditParts, it helps to know where they come from and where they go when they are no longer needed. Each viewer is configured with a factory for creating EditParts. When you set the viewer's contents, you do so by providing the model object that represents the input for that viewer. The input is typically the top-most model object, from which all other objects can be traversed. The viewer then uses its factory to construct the contents EditPart for that input object. From then on, each EditPart in the viewer will populate and manage its own children -- and connection -- EditParts, delegating to the EditPart factory when new EditParts are needed, until the viewer is populated. As new model objects are added by the user, the EditParts in which those objects appear will respond by constructing the corresponding EditParts. Note that the view construction parallels the EditParts construction. So, after each EditPart is constructed and added to its parent EditPart, the same happens with the view, whether figures or tree items.

EditParts are thrown out as soon as the user removes the corresponding model object. If the user undoes a delete, it is a different EditPart that is recreated to represent the restored object. This is why EditParts cannot contain long-term information and should not be referenced by commands.

Your first EditPart: Contents

The first EditPart you write is the EditPart that corresponds to the diagram itself. This EditPart is referred to as the contents of the viewer. It corresponds to the top-most element in the model and is parented by the viewer's root EditPart (see Figure 4). The root lays the foundation for the contents by providing various graphical layers, such as connection layers, handle layers, etc., and possibly zoom or other functionality at the viewer level. Note that the root's functions are not dependent on any model object and that GEF provides several ready-to-use implementations for roots.

Figure 4. EditParts in a viewer



The content's figure is not too interesting and is often just an empty panel that will contain the diagram's children. Its figure should be opaque and should be initialized with the layout manager that will layout the diagrams' children. It will, however, have structure. The diagram's immediate children are determined by the list of child model objects returned. Listing 1 shows a sample contents EditPart that creates an opaque figure that will position its children using the `XYLayout`.

Listing 1. Initial implementation for the contents EditPart

```
public class DiagramContentsEditPart extends AbstractGraphicalEditPart {
    protected IFigure createFigure() {
        Figure f = new Figure();
        f.setOpaque(true);
        f.setLayoutManager(new XYLayout());
        return f;
    }

    protected void createEditPolicies() {
        ...
    }

    protected List getModelChildren() {
        return ((MyModelType)getModel()).getDiagramChildren();
    }
}
```

To determine the items on the diagram, `getModelChildren()` is implemented. This method returns the list of child model objects, such as the nodes in the diagram. The superclass will use this list of model objects to create the corresponding EditParts. The newly created EditParts are added to the part's list of children EditParts. This, in turn, adds each child's figure to the diagram's figure. By default, an empty list would have been returned, which indicates no children.

More graphical EditParts

Your remaining EditParts (representing the items in the diagram) will probably have data to be displayed graphically. They may also have their own structure, such as connections or their own children. Many GEF applications depict labeled icons with connections between them. Let's assume your EditPart is going to use a Label as its figure and that the model provides a name, icon, and connections going to and from the label. Listing 2 shows a first pass at implementing this type of EditPart.

Listing 2. Initial implementation for a node EditPart

```
public class MyNodeEditPart extends AbstractGraphicalEditPart {
    protected IFigure createFigure() {
        return new Label();
    }
    protected void createEditPolicies() {
        ...
    }
    protected List getModelSourceConnections() {
        MyModel node = (MyModel)getModel();
        return node.getOutgoingConnections();
    }
    protected List getModelTargetConnections() {
        MyModel node = (MyModel)getModel();
        return node.getIncomingConnections();
    }
    protected void refreshVisuals() {
        MyModel node = (MyModel)getModel();
        Label label = (Label)getFigure();
        label.setText(node.getName());
        label.setIcon(node.getIcon());

        Rectangle r = new Rectangle(node.x, node.y, -1, -1);
        ((GraphicalEditPart) getParent()).setLayoutConstraint(this, label, r);
    }
}
```

A new method, `refreshVisuals ()`, is overridden. This method is called when it is time to update the figure using data from the model. In this case, the model's name and icon are reflected in the label. But more importantly, the label is positioned by passing its layout constraint to the parent. In the contents EditPart, we used an XY layout manager. This layout uses a Rectangle constraint to determine where to place the children figures. A width and height of "-1" indicate that the figure should be given its preferred size.

Tip No. 1

Figures should never placed using the `setBounds(...)` method. The use of layout managers like `XYLayout` ensures that scrollbars will be updated correctly. Also, `XYLayout` handles converting relative constraints to absolute locations, and it can be used to size a figure to its preferred size automatically (in other words, when the constraint's width and height are -1).

The method `refreshVisuals()` is called only once during the initialization of the EditPart and is never called again. When responding to model notification, it is the responsibility of the application to call `refreshVisuals()` again as needed to update the figure. To improve performance, you may wish to factor out the code for each model attribute into its own method (or a single method with a "switch"). That way, when the model notifies, you run the least amount of code to refresh only what has changed.

The other interesting difference is the code for connection support. Similar to `getModelChildren()`, `getModelSourceConnections()`, and `getModelTargetConnections()` should return the model objects representing the links between nodes. The superclass creates the corresponding EditParts when necessary and adds them to the list of source and target connection EditParts. Note that a connection is referred to by the nodes at each end, but that its EditPart need only be created once. GEF ensures that a connection is only created once by first checking to see if it exists already in the viewer.

Making connections

Writing a connection `EditPart` implementation is not much different. Start by subclassing `AbstractConnectionEditPart`. As before, `refreshVisuals()` may be implemented to map attributes from the model to the figure. A connection may also have a constraint, although constraints are slightly different than before. Here, constraints are used by connection routers to bend the connection. Furthermore, a connection EditPart's figure must be a Draw2D Connection, which introduces one more requirement: connection anchors.

A connection must be anchored at both ends by a `ConnectionAnchor`. Therefore, you must indicate, either in the connection EditPart or in the node implementations, which anchors to use. By default, GEF assumes that the node EditParts will provide the anchors by implementing the `NodeEditPart` interface. One reason for this is that the choice of anchor is dependent on the figure being used by the nodes at each end. The connection EditPart shouldn't know anything about the figures being used by the nodes. Another reason is that when the user is creating a connection, the connection EditPart doesn't exist, so the node must be able to show feedback on its own. Continuing Listing 2, we add the necessary anchor support in Listing 3.

Listing 3. Adding anchor support to the node EditPart

```
public class MyNodeEditPart
    extends AbstractGraphicalEditPart
    implements NodeEditPart
{
    ...
    public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }
    public ConnectionAnchor getSourceConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }
    public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }
    public ConnectionAnchor getTargetConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }

    ...
}
```

The methods that take a connection are the ones used when setting the anchors on an existing connection EditPart. The other two methods take a Request. These methods are used during editing when the user is creating a new connection. For this example, a chopbox anchor is returned in all cases. A chopbox anchor just finds the point at which a line intersects the bounding box of the node's figure. Implementing the connection EditPart is relatively straightforward. Note that it isn't even necessary to create the figure, since the default creation of `PolylineConnection` is suitable for most uses (see Listing 4).

Listing 4. Initial connection EditPart implementation

```
public class MyConnectionEditPart extends AbstractConnectionEditPart {
    protected void createEditPolicies() {
        ...
    }

    protected void refreshVisuals() {
        PolylineConnection figure = (PolylineConnection)getFigure();
        MyConnection connx = (MyConnection)getModel();
        figure.setForegroundColor(MagicHelper.getConnectionColor(connx));
        figure.setRoutingConstraint(MagicHelper.getConnectionBendpoints(connx));
    }
}
```

Listening to the model

After creation, an EditPart should start listening for change notifications from the model. Since GEF is model neutral, all applications must add their own listeners and handle the resulting notifications. When notification is received, the handler may call one of the provided methods to force a refresh. For example, if a child has been deleted, calling `refreshChildren()` will cause the corresponding EditPart and its figure to be removed. For simple attribute changes, `refreshVisuals()` can be used. As was mentioned previously, this method may be factored into multiple parts to avoid needlessly updating every displayed attribute.

Adding listeners and forgetting to remove them is a **frequent cause for memory leaks**. For this reason, the point where you add and remove listeners is clearly spelled out in the API. Your EditParts must extend `activate()` to add any listeners that must later be removed. Remove those same listeners by extending `deactivate()`. Listing 5 shows the additions to the node EditPart implementation for model notification.

Listing 5. Listening for model changes in a node EditPart

```
public class MyNodeEditPart
extends AbstractGraphicalEditPart
    implements NodeEditPart, ModelListener
{
    ...

    public void activate() {
        super.activate();
        ((MyModel)getModel()).addModelListener(this);
    }

    public void deactivate() {
        ((MyModel)getModel()).removeModelListener(this);
        super.deactivate();
    }

    public void modelChanged(ModelEvent event) {
        if (event.getChange().equals("outgoingConnections"))
            refreshSourceConnections();
        else if (event.getChange().equals("incomingConnections"))
            refreshTargetConnections();
        else if (event.getChange().equals("icon")
          || event.getChange().equals("name"))
            refreshVisuals();
    }

    ...
}
```

Editing the model

Tip No. 2

Don't forget to actually implement the `NodeEditPart` interface. Otherwise, your methods will never get called.

Most importantly, know when to use ConnectionEditParts and when not to use them. A connection EditPart is used when there is something that the user can select and interact with. It probably has a direct correlation to an object in the model and can generally be deleted by itself.

If you just have a node or container that needs to draw a line, just draw the line in the figure's paint method or compose a figure that contains a Polyline figure.

A connection must have a source and a target at all times. If you need a connection that can exist without source or target, you are better off extending just `AbstractGraphicalEditPart` and using a connection figure.

So far, we have covered how EditParts are created, how they create their visuals, and how they update themselves when the model changes. In addition to this, EditParts are also the primary players in making changes to the model. This happens when a request for a command is sent to the EditPart. Requests are also used to ask EditParts to show feedback such as during a mouse drag. The EditPart either supports, prevents, or ignores a given request. The types of requests that are supported or prevented determines the EditPart's behavior.

The focus so far has been on mapping the model's structure and properties into the view. It turns out, this is basically all you do in the EditPart class itself. Its behavior is determined by a set of pluggable helpers called EditPolicies. In the provided examples we have ignored the method `createEditPolicies()`. Once you implement this method, you are pretty much done with your EditPart. Of course, you'll still need to provide edit policies that know how to modify your application's model. Since editing behavior is pluggable, when developing your various EditPart implementations, you can create a class hierarchy based around the task of mapping the model to the view and handling model updates.

Step 4: Bringing it all together

At this point, you have all the pieces needed to graphically display your model. For final assembly, we will be using an `IEditorPart`. However, GEF's viewers can also be used in views, dialogs, or just about anywhere you can place a control. For this step, you must have your UI plug-in, which will define the editor and file extension for the resources being opened. Your model may be defined in the same or in a separate plug-in. You will also need a pre-populated model, since there is no editing functionality yet.

There are several ways to provide sample model data. For the purposes of this example, we will create the model in code when the editor is opened, ignoring the actual contents of the file. To do this, we'll assume the existence of a test factory. Alternatively, you can create an example wizard that pre-populates the resource with data (normal wizards would just create an empty diagram). Finally, you may be able to write the document's contents by hand with a text editor.

Now that you have a sample model, let's create the editor part that will display the model. A quick way to get started is to subclass or copy GEF's `GraphicalEditor`. This class creates an instance of `ScrollingGraphicalViewer`, and constructs a canvas to serve as the editor's control. It is a convenience class provided to help you get started with GEF; a properly behaved Eclipse editor has many other things to consider, such as pessimistic team environments, the resource being deleted or moved, etc.

Listing 6 shows a sample editor implementation. There are several abstract methods that must be implemented. For the scope of this article, we will be ignoring model persistence and markers. You must do two things to get your diagram to appear in the graphical viewer: configure the viewer with your own EditPart factory to construct the EditParts from Step 3, then pass in the diagram model object to the viewer.

Listing 6. Implementing your editor part

```
public class MyEditor extends GraphicalEditor {
    public MyEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }

    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer(); //Sets the \
        viewer's background to System "white"
        getGraphicalViewer().setEditPartFactory(new MyGraphicalEditpartFactory());
    }

    protected void initializeGraphicalViewer() {
        getGraphicalViewer().setContents(MagicHelper.constructSampleDiagram());
    }
    public void doSave(IProgressMonitor monitor) {
        ...
    }
    public void doSaveAs() {
        ...
    }
    public void gotoMarker(IMarker marker) {
        ...
    }
    public boolean isDirty() {
        ...
    }
    public boolean isSaveAsAllowed() {
        ...
    }
}
```

Next steps

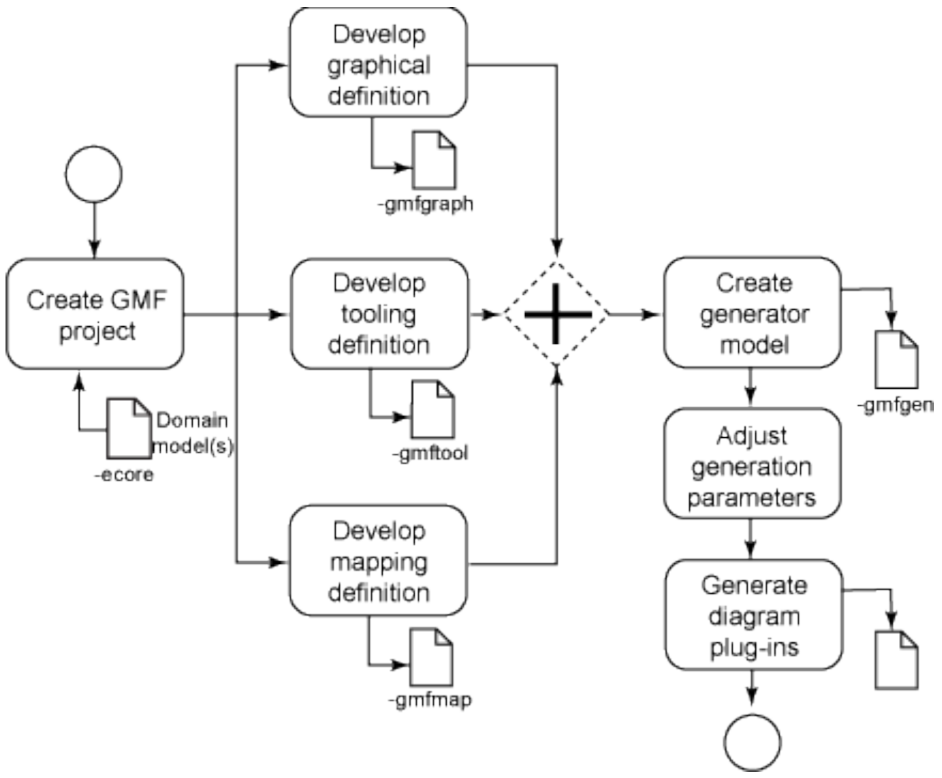
We've gone from having just a model to displaying that model in a graphical editor. But we have only laid the foundation. You can get more information about GEF by reading the GEF Programmer's Guide, provided with the GEF SDK. Since no sample directly accompanies this article, I recommend that GEF beginners look at the shapes example, along with the excellent article "A Shape Diagram Editor," by Bo Majewski. (See Resources.)

As GEF continues to mature, there have been a few robust frameworks that have been built on top of GEF to ease the development of graphical editors in Eclipse. These frameworks need to be mentioned because in many cases, developers are using these frameworks instead of directly building on top of GEF itself.

Graphical Modeling Framework (GMF)

The GMF project was born out of the frustration in creating graphical editors manually (especially in the context of using the Eclipse Modeling Framework). GMF allows you to effectively map (see Figure 5) your semantic (business logic) model to a notional (graphical) model. Once this mapping is complete using a few configuration files, GMF will generate a fully functional graphical editor for you.

Figure 5. GMF development flow

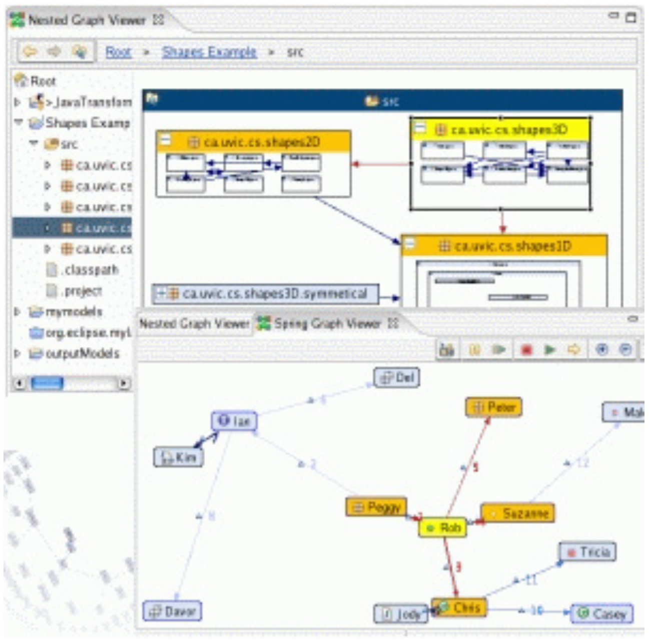

Develop graphical definition
-gmfgraph
Create GMF project
Domain model(s)
-ecore
Develop tooling definition
-gmftool
Develop mapping definition
-gmfmap
Create generator model
-gmfgen
Adjust generation parameters
Generate diagram plug-ins

If your use cases have you using the Eclipse Modeling Framework for your model, it is highly recommend that you use GMF instead of GEF directly. Read Chris Aniszczyk's "Learn Eclipse GMF in 15 minutes" for more information (see Resources).

Zest

Zest is a lightweight visualization toolkit (see Figure 6) that provides a JFace-like wrapping on top of traditional GEF editors. Zest has been modeled after JFace, and all the Zest views conform to the same standards and conventions as existing Eclipse views (think label and content providers). This means that the providers, actions, and listeners used within existing applications can be leveraged within Zest. Also, Zest has reusable layouts that can be applied to your visualizations with ease.

Figure 6. Sample Zest Visualizations



The Zest API is straightforward and to create a graph, you simply create a new Graph object and add the respective nodes and edges.

Listing 7. Sample Zest code

```
public static void main(String[] args) {
                                 // Create the shell
                                 Display d = new Display();
                                 Shell shell = new Shell(d);
                                 shell.setText("GraphSnippet1");
                                 shell.setLayout(new FillLayout());
                                 shell.setSize(400, 400);

                                 Graph g = new Graph(shell, SWT.NONE);

                                 GraphNode n = new GraphNode(g, SWT.NONE, "Paper");
                                 GraphNode n2 = new GraphNode(g, SWT.NONE, "Rock");
                                 GraphNode n3 = new GraphNode(g, SWT.NONE, "Scissors");
                                 new GraphConnection(g, SWT.NONE, n, n2);
                                 new GraphConnection(g, SWT.NONE, n2, n3);
                                 new GraphConnection(g, SWT.NONE, n3, n);
                 g.setLayoutAlgorithm(
                     new SpringLayoutAlgorithm(
                         LayoutStyles.NO_LAYOUT_NODE_RESIZING),true);

                                 shell.open();
                                 while (!shell.isDisposed()) {
                                                 while (!d.readAndDispatch()) {
                                                                 d.sleep();
                                                 }
                                 }
```

```
                    }
```

If your use cases only worry about visualization and not editing, it's recommended that you use the facilities that Zest provides. If you happen to be working with another visualization toolkit, Zest's layouts were developed in such a way that they can be reused by other toolkits.

Conclusion

The goal of this article was to provide a solid introduction to GEF, which was discussed in the first five sections of this article. After that, we discussed several options available for the graphically inclined Eclipse developer outside the GEF realm. In the end, it's important for you to evaluate your use cases and see what graphical framework matches up best to your needs.

Resources

Learn

- Visit Eclipse Graphical Editing Framework (GEF) to learn more about the Graphical Editing Framework.
- New to GEF and want a simple yet concrete example? Check out the shapes example article by Bo Majewski: "A Shape Diagram Editor" at Eclipse.org.
- Check out the GEF schema editor article "Building a Database Schema Diagram Editor with GEF" at Eclipse.org.
- Read the GEF Programmer's Guide for (almost-)definitive documentation on GEF.
- Check out the Eclipse Modeling Framework (EMF) for tutorials and articles that can be useful for the EMF novice.
- The EMF team has a good introductory tutorial titled "Generating an EMF Model."
- If you need the definitive source on EMF, the book *Eclipse Modeling Framework: A Developer's Guide* is a great reference.
- For a quick overview of GMF, read "Learn Eclipse GMF in 15 minutes."

- Frederic Plante wrote an article titled "Introducing the GMF Runtime," detailing the advanced features of the GMF runtime.
- The Graphical Modeling Framework wiki offers tutorials and plenty of information about advanced GMF features.
- Learn about using the various Eclipse modeling frameworks via this webinar with Richard Gronback and Ed Merks.
- Learn about Zest: The Eclipse Visualization Toolkit at Eclipse.org.
- For an introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."
- Check out the "Recommended Eclipse reading list."
- Browse all the Eclipse content on developerWorks.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Stay current with developerWorks' Technical events and webcasts.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download Eclipse and get started with Eclipse now.
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- Want to contribute to GEF? Check out the GEF development mailing list.
- Check out the GMF newsgroups to learn about the intricacies of GMF. (Selecting this will launch your default Usenet news reader application and open eclipse.modeling.gmf.)
- The Eclipse Platform newsgroups should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- Check out the GEF newsgroups for development help. (Selecting this will launch your default Usenet news reader application and open eclipse.tools.gef.)
- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the authors

Chris Aniszczyk is a software engineer at IBM Lotus focusing on OSGi related development. He is an open source enthusiast at heart, and he works on the Gentoo Linux distribution and is a committer on a few Eclipse projects (PDE, ECF, EMFT). He's always available to discuss open source and Eclipse over a frosty beverage.

Randy Hudson is a software engineer for IBM at Research Triangle Park, North Carolina. As the technical lead for the Graphical Editing Framework (GEF), he has helped transition the once internal project to an open source technology. His current work focuses on usability, graphical editing, graph layout, and edge routing. You can contact Randy at buchu at nc.rr.com.