TITLE

INSIGHT: An Eclipse Plug-in Example
By Charlie Thornton

CONCEPTS

Eclipse Plug-in (View) Development
Working with the Graphical Editor Framework (GEF)
Working with the Eclipse JDT Debug Interface

SUMMARY

This document and supporting code is an introduction to GEF plug-in development. We construct a view plug-in for the eclipse IDE that displays information from the built-in java debugging environment. After completing this tutorial, developers should be able to easily expand on their GEF understanding to create more complicated views and editors. Also, the debugging interface in this example targets java, but it can be extended to support other languages in a straightforward manner.

INTRODUCTION

This is a guide though the included source code. It does not contain descriptions of any of the relevent frameworks or much motivating commentary. Suffice it to say, the debug framework has a steep learning curve and GEF adds considerably to the grade. The following sections will provide a description of the plugin, and guides to each of the packages within the source of the plug-in. The package structure (parts, figures, model) was designed to match other examples provided with GEF.

Before continuing, make sure you have created a project for the plug-in source code and that it will compile. If you have not installed the version of GEF suitable for your version of eclipse do so now. Plug-in projects have a huge number of dependencies (on other eclipse plug-ins). In eclipse 3.0 they can resolved by editing the package properties and adding the library "Required Plug-ins" to the java build path, but you may need to resolve the dependencies differently based on your version of eclipse.

DESCRIPTION

We want a graphical representation of the java debugging environment (org.eclipse.jdt.debug) included with eclipse. Java is simply one of the many implementations of the debugging interface (org.eclipse.debug) in the IDE. We are specifying that our visualization will be java-specific because we want to get at some extra java-specific constructs that aren't supported by the debugging interface. Details about dependencies are given in the section of this document discussion the model package (cis690.insight.debug).

The root level elements in our visualization will be threads, and we should be able to expand a thread's stack frame to see what objects it links to. We will also use an icon to differentiate between non-thread objects and threads. If we attempt to expand an object already referenced in the view (i.e. by another thread), then the view will be smart enough simply to draw an additional link. Fundamental data types and wrapper classes should be displayed in-line. We will also include the ability to "shade" and "unshade" our objects to help reduce visual clutter. Refer to the accompanying screen shot of this plug-in in action for an alternate description.

TOP LEVEL
[cis690.insight]

The plug-in entry point is contained in the View class located in this package. View extends GEF's ViewPart and is the component that is plugged in to the eclipse workbench. The eciting code lies in the "createPartControl" method. The reason it is exciting is because ordering matters and you have to do some things that you might try to skip. First and foremost, none of the fancy GEF automatic behavior will come to life unless the edit domain has been set. Passing in a DefaultEditDomain as shown is good enough, but it must be done. Otherwise the edit domain is null and none of your policies will be activated.

The remaining code in createPartControl primarily establishes how the view will be populated. The factory and contents are the keys. After setting the contents, the edit part will create more "child" edit parts based on what children the contents (or model) reports. This process is fully automated and will work properly if your model properly reports children and your factory can build edit parts for those model objects.

The other two classes at this level are convenience classes to store static instantiated color and font objects (imagine java's Color.blue).

DEBUG
[cis690.insight.debug)

This package contains the data model for our view. Conveniently, eclipse manages most of the debugging data, so this package's job is simply to provide access to the information inside eclipse's debugging engine. Because of the GEFs edit part structure, each edit part must contain some model element. This model is what is used to construct the part. The two primary model classes used in this plug-in are DebugModel and DebugObject.

DebugModel is the top-level model element. It's corresponding model element (cis690.insight.parts.DebugModelEditPart) occupies the entire visible area of our view. The model itself will not actually be a rendered element, but its children (the DebugObjects) will be. We now find ourselves at the most confusing descision in developing this plugin.

The eclipse debug model has a natural tree-like interface. Each stack frame refers to objects which refer to other objects and so on. If you are familiar with the edit part api, you know that edit parts are created in a tree-like fashion as well (i.e. AbstractGraphicalEditPart's "getModelChildren" method). The first pass would then be to have each debug object parent its child debug objects as needed. This would be an elegant solution. Unfortunately, during the development of this plug-in I was never able to make this approach work. Any child part was created within (and limited to) the bounding box of its parent's edit part. Because of this "box crowding" I switched to the current model structure in which the DebugModel simply manages all of the DebugObjects as its children. Thus they are all contained within the bounding box of the DebugModelEditParts FreeformLayer.

This package also includes interfaces for the two model objects (IDebugModel and IDebugObject). We refer to the objects by interface as often as possible in other packages to simplify refactoring. There is a class called DOVector which ensures type safety and allows .contains method calls on this type of vector to return true if two objects refer to the same underlying debug object (rather than reference equality). The interface to the java debugging layer also lives here -- DebugUtil. The original motivation for the debug utility was to transform all of the exception throwing debugger interfacing to safe methods. Eventually it became the single window into the java-specific side of the debugging framework.

Almost all deviation from the standard debugging interface takes place in the model class "cis690.insight.debug.DebugUtil". The exception is a use of the class org.eclipse.jdt.debug.core.IJavaThread in the DebugObject and DebugModel to contain data about a thread's name and variables (which are fetched via the DebugUtil). Aside from this, efforts to convert the debugging interface could be localized to the DebugUtil class.

VIEW PARTS
[cis690.insight.parts]

The "parts" package houses the GEF "ViewPart" related elements. It contains a class called InsightPartFactory which is used to manufacture edit parts based on model objects (used in the top-level section). Also, the EnclosingLayoutPolicy can be found here. This policy allows proper dragging (and disallows resizing) of the edit parts displayed in the view. Though it appears to support "undo", I was never able to get it working -- perhaps this would be a nice excercise.

The most mysterious class in the parts package is "AsyncRefresh". It turns out, if you try to call an edit part's "refresh" method (say to update some text) from any thread other than the UI thread (equivalent to the AWT thread in java GUI development) eclipse will throw an exception and your plug-in will stop working. This is inconvenient if you would like to update following a notification from the debugger thread. AsynchRefresh's "refresh" method allows you to refresh the UI be remembering the UI thread and forcing it to refresh. It is equivalent to the repaint posting behavior found in java applications.

The DebugModelEditPart class found here wraps the DebugModel object (its model), and provides GEF with suitable hooks to manipulate its connections. The DebugObjectEditPart class behaves similarly. There is a considerable amount of code here, but the only traps lie in implementing by accident or failing to implement methods that the EditParts use to turn on all of their extra functionality. Study the API carefully and think twice before naming a method in these classes. GraphicalEditParts have many methods and they all turn some bell or whistle on or off -- be wary. Also note that the edit parts make use of the AsyncRefresh class, this allows them to localize the strange refresh handling by intentionally reimplementing the refresh class to reroute calls to the UI thread.

FIGURES
[cis690.insight.figures]

The figures package creates the visual representations of the edit parts. Where the EditParts were the controllers, the figures provide the view in the MVC system. The five figures contained in this implementation are all the pieces of a single big object figure or "ObjFig". In addition to managing the rendering and updating of the model data, the ObjFig class also listens for the shade/unshade action. The state variable monitoring this behavior is actually in the DebugObject class so there are a number of methods in place to facilitate this exchange.

The next level down in the hierarchy is the VarFig class. These figures are the horizontal bars designed to display information about particular variables visible within an object. Variable expansion (the method by which our figures multiply) is handled by this figure in conjunction with the appropriate model objects. Special rendering of descriptive strings is delegated to the DebugUtil object mentioned in the model section above.

The idea that any figure can be properly laid out with a combination of the border and toolbar layouts seemed to guide the GEF at the time this plug-in was made. This restriction created some awkwardness in the component layouts, but luckily these figures are simple enough to comply. Fortunately, when working with figures, the full power of the SWT layout system is available and they have a number of more convenient models to choose from.

FINALY

With the package descriptions in hand it should be possible to browse and alter the original source code without much mysticism. In particular the DebugUtil object could be swapped out or recoded to work with another language supported by the eclipse debugging framework. This document cannot stand in isolation. You must read the introductory articles on the eclipse plug-in framework, SWT, and GEF in order for the source listings in this plug-in to make any sense. Eclipse plug-in development has a very steep learning curve (and it doesn't level off very fast, either!), but with a little patience it will are start to make sense.