

NOTE:

moreinfo - needs more information

TODO

- Eclipse version

- Introduction

Conventions

- classes will be camel cased

- methods will be postfix with '()', ie. two brackets without their parameters. If necessary the parameters will be included and described.

## 1 Chapter 1

Starting with a clean slate we will create the plugin project and rip out and modify the default editor that is created by the wizard.

1. *Creating and setting up the base of the plugin*, create a new plugin of type Editor
2. modify the file extension in the wizard or plugin.xml, this project will use the extension 'precall'
3. add the org.eclipse.gef library in the dependencies tab of plugin.xml (what about draw2d?)
4. *We are now ready to start coding the initial classes before we start integrating the model*, remove org.eclipse.ui.workbench.texteditor (why?)
5. create a class extending GraphicalEditor called MainViewer, two methods will be implemented by default
  - (a) create a constructor for MainViewer, setting the edit domain to DefaultEditDomain
  - (b) override the method configureGraphicalViewer
    - i. ensure that super.configureGraphicalViewer() is called first
    - ii. get the Graphical Viewer in order to configure it
    - iii. set the root edit part which will be the base layer for all edit parts, ScalableRootEditPart is easiest
    - iv. set the class BaseFactory (currently nonexistent) as the viewer's factory. It will create the required EditPart based on its type. This class will have to be updated as you model is extended into its visualization

To create the factory BaseFactory, create a new package in which to place the factory. In this instance it was created in the subpackage 'factory'. The factory implements EditPartFactory and will be used by GEF to produce the required edit parts as the different model elements are encountered. No hierarchical

information is maintained in the factory despite any possible occurrences or limitations existent within the model.

Now we will remove the existing editor and place our class in its place so that it is instantiated when the plugin loads. Remember eclipse's infrastructure is such that classes of installed plugins are not instantiated until the plugin is actually required.

- in plugin.xml, replace the 'class' field with MainViewer
- clear the 'contributorClass' field
- update the id of the plugin
- set the name of the plugin if desired
- delete the package baseplugin.editors and all classes within

Now we can test whether we have a working plugin by right-clicking on the project and selecting 'Run As' and then selecting 'Eclipse Application'.

Another Eclipse instance will be run and once loaded, we will want to instantiate the editor we just created with a file. In order to do so, make sure the 'Package Explorer' view is visible and create a new project if necessary, then create a new file with the 'precall' extension and open it. A white window will appear which performs no actions, indicating that our plugin is working.

Before integrating an external package into our plugin, we shall display a simple box using some basic figures to gain some experience with draw2D. The figure will be displayed in the white area that appeared as our editor.

There are several concepts relating to figures:

- drawing basic shapes, this is very easy
- layouts, similar to swing, no extensive coverage here
- placement, very important and can be tricky

Let's draw a simple box which will represent an empty model object for now, and then we will extend this into a sequence of boxes that will represent some associated sequence from a model.

First, we need to create a model object. This is easy, it will not have any variables or methods for now. The class for this is `edu.toronto.model.example.simple.SimpleA`

Second we will create an `EditPart` that will interact with the model object and the `Figure`.

The new class is created as `baseplugin.editpart.SimpleAEditPart` which extends `AbstractGraphicalEditPart`.

Third, in `createFigure()` of `SimpleAEditPart`, we create a rectangle to display on the page.

Fourth, we tie the model class with the edit part that will represent this model object. This is done in `BaseFactory`.

Fifth, we set the model of the viewer to an object of `SimpleA` which will then be displayed. This is done in `MainViewer.initializeGraphicalViewer()`.

If you run the plugin application and open the file with the 'precall' extension you will notice that the whole screen is painted blue. This is because the model object is a single element and it is presumed to be the main diagram (moreinfo). In order to have the box appear as we would like to have the main diagram with all the layout settings made so that multiple component or the diagram's actual components can be displayed and maintained separately.

As such, a new model class `edu.toronto.model.example.simple.SimpleAHolder` is created and an associated edit part `baseplugin.editpart.SimpleAHolderEditPart`. In that class you will see that no actual figures are really created, only the layout and content pane `RectangleFigure` is setup. (more info on toolbar layout and content pane). The model object and associated edit part are also placed in the factory. The last thing to do with this model modification is modifying the actual model passed into the viewer which is done in `MainViewer`.

\*\*\*

Now we can begin working on building/integrating our model.

In this tutorial, I will demonstrate how to interface existing code which provides a result after running an algorithm to display the results using a basic graph.

- import/copy the package into the project