Best practices for developing Eclipse plugins
Using markers, annotations, and decorators

Andy Flatt  
Mickael Maison  
August 16, 2011

This tutorial highlights best practices when marking information to resources using markers, and then introduces annotations and decorators that you use to highlight markers within the workbench. By extending extension points, you can reuse and adapt the built-in functions in Eclipse and perform advanced resource marking, such as moving a text marker when editing text. We discuss methods that take advantage of the plugin model, which allows for an efficient, high performance, and integrated look and feel plugin.

Introduction
When developing a plugin for an IDE Eclipse environment, you have several design considerations. These considerations ensure that you:

• Do not lock up the user interface thread.
• Decorate the UI without impacting performance.
• Process data in the background.

This tutorial discusses how to process and display data associated with resources stored in the workspace with these design considerations. We will examine how Eclipse provides a marker interface to store and handle information about a resource.

We provide best practices for handling data markers against a resource. First, we show how to mark data, then build knowledge to represent the marker on the user interface, and then update the marker as the resource changes. In this context, resources are Eclipse objects that implement the IResource interface, such as projects, files, folders, and Java™ objects (including packages, classes, and source).

This tutorial is intended for developers who can write a basic plugin but want to learn best practices when dealing with Eclipse resources.

Eclipse.org documents individual extension points and interfaces. This article provides assistance in choosing the best practice of using them in combination. Read more to learn how to take advantage of existing Eclipse functions to provide new features.
Part 1: Create your own marker

What are markers?

Markers are used to link information to a resource without having to change the resource. Common examples of marked information are breakpoints, bookmarks, and compilation errors. Taking the example of compilation errors, every time a compile job is executed, the job will look through the source and highlight errors by creating a new marker.

Extend markers

In the first part of this tutorial, we will create our own type of marker. This is done by use of the marker extension point and writing just one line of code.

You can create your own markers by extending the IMarker extension point org.eclipse.core.resources.markers. Begin by adding the code in Listing 1. to your plugin.xml file. This says you are going to create a marker called My Marker with the id com.ibm.mymarkers.mymarker. Its supertype is org.eclipse.core.resources.marker. This is the most basic marker type. You can extend other supertypes that offer functions that are slightly more specific. The other supertypes available are:

- org.eclipse.core.resources.problemmarker
- org.eclipse.core.resources.textmarker

Note: You can use as many of these supertypes as suitable.

Listing 1. Marker extension definition from the plugin.xml

```xml
<extension point="org.eclipse.core.resources.markers" id="com.ibm.mymarkers.mymarker"
  name="My Marker">
  <super type="org.eclipse.core.resources.marker"/>
  <persistent value="false"/>
  <attribute name="description"/>
</extension>
```

Notice in Listing 1. the two elements persistent and attribute. These are properties given to the mymarker marker. Persistent states if the markers should be stored within the workspace. There is also a new attribute description given to mymarker.

Use the new marker

You can create a marker by calling the createMarker method on the desired resource. You may also wish to set its attributes.

The method shown in Listing 2 can also be used to create a marker. Its only required input is the IResource that links to the marker. Once created, you can set its attributes.
Listing 2. Java code creating the new marker type

```java
public static IMarker createMarker(IResource res) throws CoreException {
    IMarker marker = null;
    //note: you use the id that is defined in your plugin.xml
    marker = res.createMarker("com.ibm.mymarkers.mymarker");
    marker.setAttribute("description," "this is one of my markers");
    //note: you can also use attributes from your supertype
    marker.setAttribute(IMarker.MESSAGE, "My Marker");
    return marker;
}
```

Find your marker

To find the markers associated with an IResource you can query for a resource to get all the markers of a specific id and specify whether to search related resources. To search for markers, call the findMarkers method on the resource. Specify the arguments such as the marker type and the depth of search.

The code in Listing 3 finds the markers directly linked with that resource by using the IResource.DEPTH_ZERO argument.

Listing 3. Java code finding the new marker type

```java
public static final String MARKER = "com.ibm.mymarkers.mymarker";

public static List<IMarker> findMarkers(IResource resource) {
    try {
        return Arrays.asList(resource.findMarkers(MARKER, true, IResource.DEPTH_ZERO));
    } catch (CoreException e) {
        return new ArrayList<IMarker>();
    }
}
```

Changing the resource depth to IResource.DEPTH_INFINITE will return all markers related to this or any sub-resources. For example, you could pass in a package and get all the markers linked with resources from that package.

Part 2: Display and update markers using annotations

What is an annotation?

Annotations are used to mark an area of text within an editor. They are used widely in Eclipse for displaying information such as errors, warnings, build issues, tasks, or breakpoints. Annotations are visible on the editor's rulers and on the text. Figure 1 shows an annotation displaying a syntax error.

Figure 1. Example of an annotation
Common implementations of annotations generate marks while parsing the file and picking out things such as errors and "todo" tags. This is normally completed at build time. Others are linked to persistent markers such as break points. This example uses a persistent marker.

The way the user views annotations can be customized in the preferences panel under **General > Editors > Text Editors > Annotations**. This means no additional work is required to allow for user customization of annotations.

**Figure 2. A screenshot of the annotation preferences panel**

View a larger version of Figure 2.

**The marker extension point properties**

In Part 1 of this tutorial, the marker only extended the default marker type. In Part 2, we will extend the text marker type so that we can mark a text location. This type defines two key attributes: `charStart` and `charEnd`. We also want the marker to persist between sessions so we change the persistent value to `true`. To do this, we need to update the marker as defined in **Listing 4**.

**Listing 4. Marker extension definition from the plugin.xml**

```
<extension point="org.eclipse.core.resources.markers" id="com.ibm.mymarkers.mymarker" 
    name="My Marker">
    <super type="org.eclipse.core.resources.textmarker"/>
    <super type="org.eclipse.core.resources.marker"/>
    <persistent value="true"/>
</extension>
```

**Define your annotation specification extension**

To create your own annotations, use the extension point `org.eclipse.ui.editors.markerAnnotationSpecification`. (See **Listing 5**.) This defines the annotation's properties and its default display options.
Listing 5. Annotation extension definition from the plugin.xml

```xml
<extension point="org.eclipse.ui.editors.markerAnnotationSpecification"
  id="myannotationspecification" name="MyAnnotation">
  <specification annotationType="com.ibm.example.myannotation"
    label="MyAnnotation"
    icon="icons/sample.gif"
    overviewRulerPreferenceKey="clruler"
    overviewRulerPreferenceValue="true"
    colorPreferenceKey="clcolor"
    colorPreferenceValue="255,255,0"
    textPreferenceKey="cltext"
    textPreferenceValue="true"
    verticalRulerPreferenceKey="clvertical"
    verticalRulerPreferenceValue="true"
    textStylePreferenceKey="clstyle"
    textStylePreferenceValue="BOX">
  </specification>
</extension>
```

The parts of the XML that we will use to link to the specification are the `id` and `annotationType` attributes. Eclipse.org documents the other attributes that are used for customization (see Related topics).

The next step is to link the existing marker to the new annotation specification by using the extension point `org.eclipse.ui.editors.annotationTypes`. We link the specification using the annotation type from the specification and the `id` from the marker definition.

Listing 6. Annotation type definition from the plugin.xml

```xml
<extension point="org.eclipse.ui.editors.annotationTypes">
  <type markerSeverity="0"
    super="org.eclipse.ui.workbench.texteditor.info"
    name="com.ibm.example.myannotation"markerType="com.ibm.mymarkers.mymarker"/>
</extension>
```

See Related topics to find more information on this extension point from Eclipse.org.

Create and add an annotation to the editor

The code in Listing 7 is used to create and add the annotation to an editor.

Listing 7. Add a new annotation to the editor

```java
public static void addAnnotation(IMarker marker, ITextSelection selection,
    ITextEditor editor) {
    //The DocumentProvider enables to get the document currently loaded in the editor
    IDocumentProvider idp = editor.getDocumentProvider();

    //This is the document we want to connect to. This is taken from
    //the current editor input.
    IDocument document = idp.getDocument(editor.getEditorInput());

    //The IannotationModel enables to add/remove/change annotation to a Document
    //loaded in an Editor
    IAnnotationModel iamf = idp.getAnnotationModel(editor.getEditorInput());

    //Note: The annotation type id specify that you want to create one of your
```
//annotations
SimpleMarkerAnnotation ma = new SimpleMarkerAnnotation("com.ibm.example.myannotation", marker);

//Finally add the new annotation to the model
iamf.connect(document);
iamf.addAnnotation(ma, newPosition(selection.getOffset(), selection.getLength()));
iamf.disconnect(document);
}

Keep the marker and annotation in sync

The annotation model takes care of moving the annotation when the document is edited. It does not, however, update the marker's attributes when the annotations move. In this case, we want to update the marker's charStart and charEnd attributes. This last extension point is the marker updater. It defines a class used to update the markers when an annotation moves.

Listing 8. Marker updater definition from the plugin.xml

```xml
<extension point="org.eclipse.ui.editors.markerUpdaters">
  <updater
    id="com.ibm.example.MarkerUpdater"
    class="com.ibm.example.mymarker.MarkerUpdater"
    markerType="com.ibm.mymarkers.mymarker">
  </updater>
</extension>:
```

We use the IMarkerUpdater interface to provide the code we want to execute when our annotation is moved. The class shown in Listing 9 is our marker updater. The code we are interested in is the updateMarker method. Here we are using it to update the charStart and charEnd attributes of the marker.

Listing 9. Marker updater code

```java
public class MarkerUpdater implements IMarkerUpdater {

  /*
   * Returns the attributes for which this updater is responsible.
   * If the result is null, the updater assumes responsibility for any attributes.
   */
  @Override
  public String[] getAttribute() {
    return null;
  }

  @Override
  public String getMarkerType() {
    // returns the marker type that we are interested in updating
    return "com.ibm.mymarkers.mymarker";
  }

  @Override
  public boolean updateMarker(IMarker marker, IDocument doc, Position position) {
    try {
      int start = position.getOffset();
      int end = position.getOffset() + position.getLength();
      marker.setAttribute(IMarker.CHAR_START, start);
      marker.setAttribute(IMarker.CHAR_END, end);
      return true;
    } catch (CoreException e) {
      return false;
    }
}
```
Part 3: Use decorators to identify marked IResources

What are decorators?

Within Eclipse, decorators are used to add visual information to objects in the workbench. Usually, they display the object type and any key properties currently associated with that object. Figure 3 shows how decorators are displayed in the package explorer to the user. The decorators show which items are Java source files or packages, and shows marker icons for source and packages that contain warnings or errors. Here decorators are also used to add team details, such as whether the files are in or out of sync with the repository.

Figure 3. Packages and source decorated with symbolic icons

Define your own decorator

The first step to add our decorator is to extend the org.eclipse.ui.decorators extension point. It enables the definition of a new decorator, and selects what kind of objects it will decorate.

The important fields here are:

- **class**, — which must be the fully qualified name of a class that implements ILightweightLabelDecorator (as lightweight is set to true).
- **enablement**, — which contains the list of Eclipse objects to which the decorator applies.

Listing 10. Decorator definition from the plugin.xml

```xml
<extension point="org.eclipse.ui.decorators">
  <decorator id="com.ibm.example.filedecorator" label="MyMarker Decorator" state="true" class="com.ibm.example.mymarker.FileDecorator" adaptable="true" lightweight="true">
    <enablement>
      <objectClass name="org.eclipse.core.resources.IResource"/>
    </enablement>
  </decorator>
</extension>
```

See Related topics for more documentation on extension point from Eclipse.org.

Note: Lightweight vs. Non-Lightweight: According to the API, non-lightweight decorators may become deprecated in future versions of Eclipse.
**Our file decorator class**

We need to implement the class `FileDecorator` to determine the behavior of our decorator. This class has to implement `ILightweightLabelDecorator`. It is good idea to have it extend `LabelProvider` as this will allow us to only override the method in which we are interested, that is, `decorate()`.

A basic implementation of `decorate()` is shown in Listing 11.

**Listing 11. Basic implementation of `decorate()`**

```java
public void decorate(Object resource, IDecoration decoration) {
    decoration.addOverlay(ImageDescriptor.createFromFile(FileDecorator.class, 
        "icons/sample.gif"), IDecoration.TOP_RIGHT);
    decoration.addPrefix("My Prefix ");
    decoration.addSuffix(" My Suffix");
}
```

The `IDecoration` object also allows customization of the font and text/background color.

**Figure 4. A screenshot of the code in Listing 11 decorating IResources**

The first argument of `decorate()` can be used to filter the resources we want to decorate. If we only want to decorate resources that contain specific markers, we use code exampled in Listing 12.

**Listing 12. Decorate resources with specific markers**

```java
public void decorate(Object resource, IDecoration decoration) {
    if(resource instanceof IResource){
        List<IMarker> markers = MyMarkerFactory.findMarkers((IResource) resource);
        if (markers.size() > 0) {
            decoration.addSuffix(" Marker !!");
        }
    }
}
```

You have followed this tutorial, what's next?

Advanced improvements can include:

- **Add editable properties to markers.** Allow the user to change the state of the markers
- **Automate the creation and deletion of markers.** Use background processing jobs to create, update, and delete markers automatically.
- **Customize the marker hovers.** Use advance marker hovers to support HTML or multimedia content

In this tutorial, we used Eclipse to easily create and customize markers and perform advanced resource marking. Developers are encouraged to use this simple, yet powerful tool to perfectly
integrate their plugins into the Eclipse IDE. Keep in mind, however, that this feature can become obtrusive to the user if implemented too extensively. Furthermore, it is the responsibility of the developer to maintain the Eclipse look and feel by taking into consideration the Eclipse User Interface Guidelines, see Related topics.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code from Part 1</td>
<td>Marker.zip</td>
<td>13KB</td>
</tr>
<tr>
<td>Code from Part 2</td>
<td>Annotation.zip</td>
<td>16.3KB</td>
</tr>
<tr>
<td>Code from Part 3</td>
<td>Decorator.zip</td>
<td>18.2KB</td>
</tr>
</tbody>
</table>
Related topics

- Documentation on markers extension point from Eclipse.org
- Documentation on resource markers from Eclipse.org
- Documentation on the decorator extension point from Eclipse.org
- Documentation on the marker annotation specification extension point from Eclipse.org
- Documentation on the annotation types extension point from Eclipse.org
- Eclipse User Interface Guidelines
- developerWorks on Twitter: Follow us for the latest news.
- Eclipse Marketplace is a convenient portal where you can find open source and commercial Eclipse-related offerings. If you downloaded Indigo through a package, you'll have access to the Eclipse Marketplace client.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)