

Eclipse Corner Article



Creating an Eclipse View

Summary

In the Eclipse Platform a view is typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. In this article the design and implementation of a view will be examined in detail. You'll learn how to create a simple view based on SWT, and a more advanced view using the JFace viewer hierarchy. We'll also look at ways to achieve good integration with many of the existing features in the workbench, such as the window menu and toolbar, view linking, workbench persistence and action extension.

By Dave Springgay, OTI

November 2, 2001

Introduction

In the Eclipse Platform the workbench contains a collection of workbench windows. Each workbench window contains one or more pages, and each page contains a collection of editors and views. An editor is typically used to edit or browse a document or input object. Modifications made in an editor follow an open-save-close lifecycle model. A view is typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. In contrast to an editor, modifications made in a view are saved immediately. The layout of editors and views within a page is controlled by the active perspective.

The workbench contains a number of standard components which demonstrate the role of a view. For instance, the Navigator view is used to display and navigate through the workspace. If you select a file in the Navigator, you can open an editor on the contents of the file. Once an editor is open, you can navigate the structure in the editor data using the Outline view, or edit the properties of the file contents using the Properties view.

In this article we'll look at how you, as a plug-in developer, can add new views to the workbench. First we'll examine the process through the creation of a simple Label view. This view just displays the words "Hello World". Then we'll design and implement a more comprehensive view, called Word view, which displays a list of words which can be modified through addition and deletion. And finally, we'll look at ways to achieve good integration with many of the existing features in the workbench.

Source Code

To run the example or view the source for code for this article you can unzip [viewArticleSrc.zip](#) into your *plugins/* subdirectory.

Adding a New View

A new view is added to the workbench using a simple three step process.

1. Create a plug-in.
2. Add a view extension to the plugin.xml file.
3. Define a view class for the extension within the plug-in.

To illustrate this process we'll define a view called "Label", which just displays the words "Hello World".

Step 1: Create a Plug-in

In step 1 we need to create a new plug-in. The process of plug-in creation is explained in detail in [Your First Plugin](#), by Jim Amsden, so we won't go into the details here. To be brief, we need to create a plugin project, and then define a plugin.xml file (shown below). This file contains a declaration of the plug-in id, name, pre-requisites, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="Views Plugin"
  id="org.eclipse.ui.articles.views"
  version="1.0.0"
  provider-name="OTI">

<requires>
  <import plugin="org.eclipse.core.boot"/>
  <import plugin="org.eclipse.core.runtime"/>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.swt"/>
  <import plugin="org.eclipse.ui"/>
</requires>

<runtime>
  <library name="views.jar"/>
</runtime>

</plugin>
```

Step 2: Add a View Extension to the plugin.xml file

Once we have a plugin, we can define a view extension. The org.eclipse.ui plug-in defines a single extension point for view contribution: org.eclipse.ui.views. In the XML below, we use this extension point to add the Label view extension. This XML is added to the plugin.xml file, after the runtime element, and contains the basic attributes for the view: id, name, icon and class.

```
<extension point="org.eclipse.ui.views">
  <view id="org.eclipse.ui.articles.views.labelview"
    name="Label View"
    class="org.eclipse.ui.articles.views.LabelView"
    icon="icons\view.gif"/>
</extension>
```

A complete description of the view extension point and the syntax are available in the developer documentation for org.eclipse.ui. The attributes are described as follows.

- **id** - a unique name that will be used to identify this view
- **name** - a translatable name that will be used in the UI for this view
- **class** - a fully qualified name of the class that implements org.eclipse.ui.IViewPart. A common practice is to subclass org.eclipse.ui.part.ViewPart in order to inherit the default functionality.
- **icon** - a relative name of the icon that will be associated with the view.

Perhaps the most important attribute is *class* (A). This attribute must contain the fully qualified name of a class that implements org.eclipse.ui.IViewPart. The IViewPart interface defines the minimal responsibilities of the

view, the chief one being to create an SWT Control within the workbench. This provides the presentation for an underlying model. In the XML above, the class for the Label view is defined as `org.eclipse.ui.articles.views.LabelView`. The implementation of `LabelView` will be examined in a few paragraphs.

The `icon` attribute (B) contains the name of an image that will be associated with the view. This image must exist within the plugin directory or one of the sub directories.

Step 3: Define a View Class for the Extension within the Plug-in

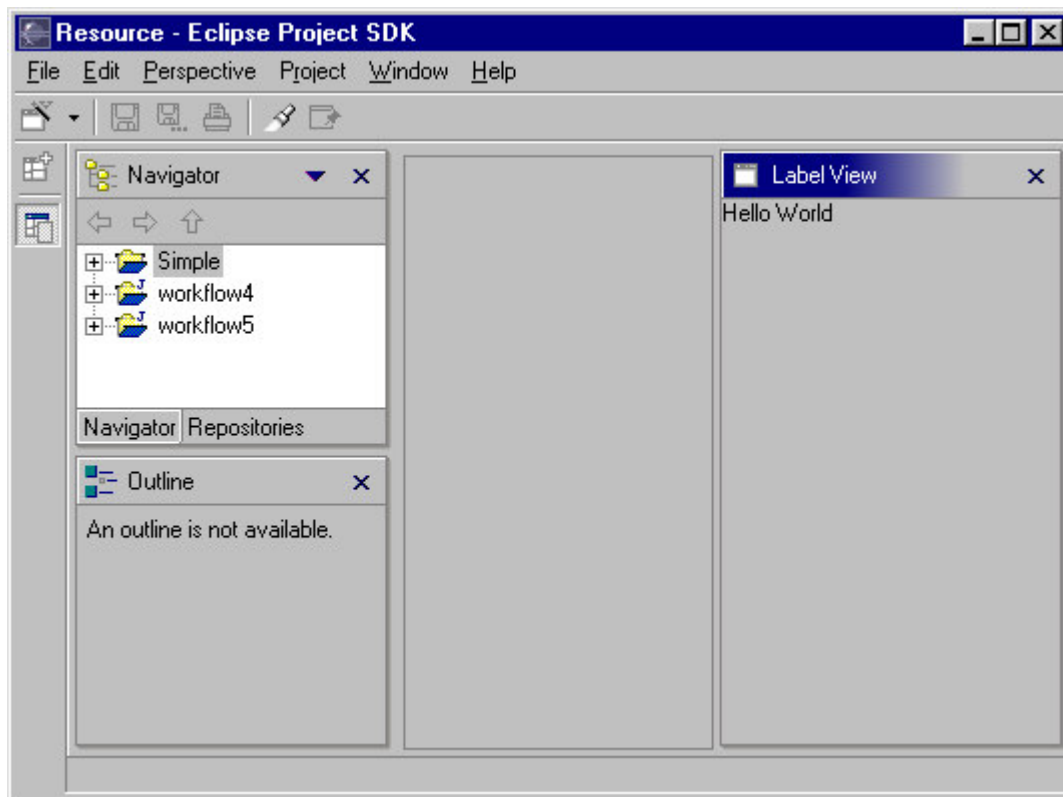
Now we need to define the view class. To help us out, the platform contains an abstract class, named `org.eclipse.ui.part.ViewPart`, which implements most of the default behavior for an `IViewPart`. By subclassing this, we inherit all of the behavior. The result is `LabelView` (shown below). The methods in this class implement the view specific presentation. The `createPartControl` method (B) creates an SWT `Label` object to display the phrase "Hello World". The `setFocus` (A) method gives focus to this control. Both of these methods will be called by the platform.

```
package org.eclipse.ui.articles.views;

import org.eclipse.swt.widgets.*;
import org.eclipse.ui.part.ViewPart;

public class LabelView extends ViewPart {
    private Label label;
    public LabelView() {
        super();
    }
    A public void setFocus() {
        label.setFocus();
    }
    B public void createPartControl(Composite parent) {
        label = new Label(parent, 0);
        label.setText("Hello World");
    }
}
```

Once the `LabelView` class has been declared and compiled, we can test the results. To do this, invoke the `Perspective > Show View > Other` menu item. A dialog will appear containing all of the view extensions: the `Label View` item will appear in the `Other` category. If you select the `Label View` item and press `OK`, the view will be instantiated and opened in the current perspective. Here is a screen snapshot of the `Label` view after it has been opened within the `Resource` perspective.



View Lifecycle

The lifecycle of a view begins when the view is added to a workbench page. This can occur during the creation of the page, or afterwards, if the user invokes Perspective > Show View. In either case the following lifecycle is performed.

1. An instance of the view class is instantiated. If the view class does not implement `IViewPart` an `PartInitException` is thrown.
2. The `IViewPart.init` method is called to initialize the context for the view. An `IViewSite` object is passed, and contains methods to get the containing page, window, and other services is passed.
3. The `IViewPart.createPartControl` method is called. Within this method the view can create any number of SWT controls within a parent `Composite`. These provide the presentation for some underlying, view specific model.

When the view is closed the lifecycle is completed.

1. The parent `Composite` passed to `createPartControl` is disposed. This children are also implicitly disposed. If you wish to run any code at this time, you must hook the control dispose event.
2. The `IViewPart.dispose` method is called to terminate the part lifecycle. This is the last method which the workbench will call on the part. It is an ideal time to release any fonts, images, etc.

View Position

As we have already seen, the user can open a view by invoking Perspective > Show View. In this scenario, the initial position of the new view is determined by the active perspective within the current page. You can think of a perspective as a layout containing views, folders, and place holders. A *folder* is a stack of views. A *place holder* marks the desired position of a view, should the user open it. When a new view is opened the platform will search the perspective for a place holder with the same `id` as the view. If such a place holder is found, it will be replaced by the new view. Otherwise, the new view will be placed on the right hand side of the page.

As a plugin developer, you may also define a new perspective which contains your view, or add your view to a

perspective which has been contributed by another plug-in. In either case, you have greater control over view position. For more information about the use of perspectives see [Using Perspectives in the Eclipse UI](#).

Can I Declare A View Through API?

Every view within the workbench must be declared in XML. There are two reasons for this.

1. Within Eclipse, XML is typically used to declare the presence of an extension and the circumstances under which it should be loaded. This information is used to load the extension and its plugin lazily for better startup performance. For views in particular, the declaration is used to populate the Show View menu item before the plugin is loaded.
2. The view declaration is also used for workbench persistence. When the workbench is closed, the id and position of each view within the workbench is saved to a file. It is possible to save the specific view class. However, the persistence of class names in general is brittle, so view id's are used instead. On startup, the view id is mapped to a view extension within the plugin registry. Then a new instance of the view class is instantiated.

The Word View

In this section we'll implement a more comprehensive view, called Word view, which displays a list of words which can be modified through addition and deletion.

Adding a View Extension to the plugin.xml file

To start out, we add another view extension to the plugin.xml (shown below). This extension has the same format as the Label view: id, name, icon and class. In addition, the extension contains a *category* element (A). A *category* is used to group a set of views within the Show View dialog. A category must be defined using a category element. Once this has been done, we can populate the category by including the category attribute in the Word view declaration (B). The category, and the Word view within it, will both be visible in the Show View dialog.

```
<extension point="org.eclipse.ui.views">
  A <category
      id="org.eclipse.ui.article"
      name="Article">
  </category>
  <view id="org.eclipse.ui.articles.views.wordview"
      name="Word View"
      icon="icons\view.gif"
      B category="org.eclipse.ui.article"
      class="org.eclipse.ui.articles.views.WordView"/>
</extension>
```

Defining a View Class for the Extension within the Plug-in

Now we need to define a view class for the Word view. For simplicity, we subclass `ViewPart` to create a `WordView` class (shown below), and then implement `setFocus` and `createPartControl`. The implementation of this class is discussed after the code.

```
public class WordView extends ViewPart
{
    WordFile input;
    ListViewer viewer;
    Action addItemAction, deleteItemAction, selectAllAction;
    IMemento memento;

    /**
     * Constructor
     */
    A public WordView() {
```

```

        super();
        input = new WordFile(new File("list.lst"));
    }

    /**
     * @see IViewPart.init(IViewSite)
     */
    B public void init(IViewSite site) throws PartInitException {
        super.init(site);
        // Normally we might do other stuff here.
    }

    /**
     * @see IWorkbenchPart#createPartControl(Composite)
     */
    C public void createPartControl(Composite parent) {
        // Create viewer.
        D viewer = new ListViewer(parent);
        viewer.setContentProvider(new WordContentProvider());
        viewer.setLabelProvider(new LabelProvider());
        viewer.setInput(input);

        // Create menu and toolbars.
        E createActions();
        createMenu();
        createToolBar();
        createContextMenu();
        hookGlobalActions();

        // Restore state from the previous session.
        F restoreState();
    }

    /**
     * @see WorkbenchPart#setFocus()
     */
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}

```

In the `WordView` constructor, the model, a `WordFile`, is created (A). The actual shape of the model is somewhat irrelevant, as we wish to focus on the details of view creation, and the appropriate model will vary with the problem domain. To be brief, a `WordFile` is simply a list of words which are stored in a file. The storage location of the `WordFile` is passed to the constructor. If this file exists the `WordFile` will read it. Otherwise it will create the file. The `WordFile` also has simple methods to add, remove, and get all of the words. Each word is stored in a `Word` object, which is just a wrapper for a `String`. The `Word` and `WordFile` classes are supplied with the source of this article.

After the view is instantiated, the `IViewPart.init` method is called with an `IViewSite` (B). The site is the primary interface between the view part and the outside world. Given the site, you can access the view menu, toolbar, status line, containing page, containing window, shell, etc. In the code above we simply call the superclass, `ViewPart`, where the site is stored in an instance variable. It can be retrieved at any time by calling `getViewSite()`.

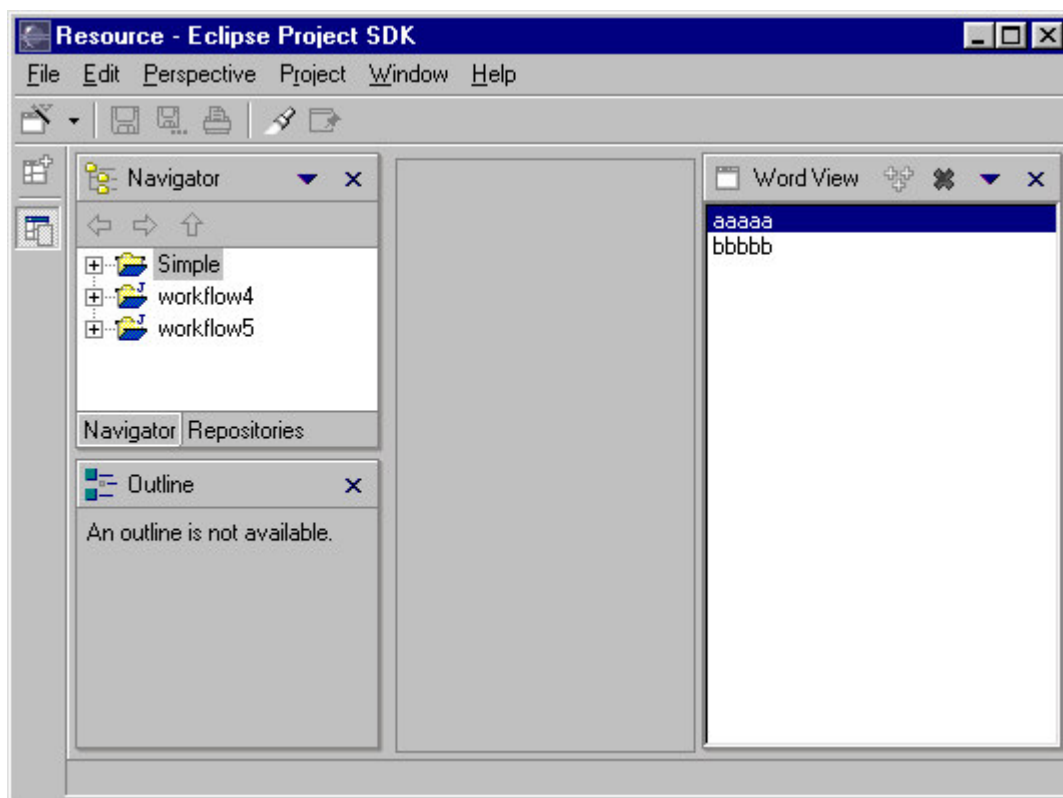
The `createPartControl` method (C) is called to create an SWT Control for the `WordFile` model. The creation of this presentation can be done with raw SWT widgets. However, the platform UI contains a class framework, known as JFace, which provides a viewer hierarchy for list, tree, table, and text widgets. A *viewer* is a wrapper for an SWT control, adding a model based interface to it. If you need to display a simple list model, tree model, or table model, use a viewer. Otherwise, it is probably easier to use raw SWT widgets. In the `Word` view the model is a simple list of words, so a `ListViewer` is used for the presentation.

On the first line of `createPartControl` the `ListViewer` is created (D). The constructor chosen here automatically creates an SWT List control in the parent. Then a *content provider* is defined. A content provider is an

adapter for a domain specific model, wrapping it in an abstract interface which the viewer invokes to get the model root and its children. If the model (the `WordFile`) changes, the content provider will also refresh the state of the `ListViewer` to make the changes visible. A *label provider* is also defined. This serves up a label and image for each object which is supplied by the content provider. And finally, we set the input for the `ListViewer`. The input is just the `WordFile` created in the `WordView` constructor.

The `createPartControl` method also contains several method calls for the creation of a menu, toolbar, context menu, and global actions (E). These methods will be examined in later sections. For now, it is sufficient to say that each view has a local menu and toolbar which appear in the view title area. The view itself may contribute menu and toolbar items to these areas. The view may define a context menu (pop-up menu) for each control it creates, or hook a global action in the parent window. A *global action* refers to those actions in the window menu and toolbar which are always visible, but delegate their implementation to the active part. For instance, the Cut, Copy and Paste actions are always visible in the Edit menu. If invoked, their implementation is delegated to the active part. The last line (E) relates to state persistence between sessions. We will examine the code for each of these features in later sections.

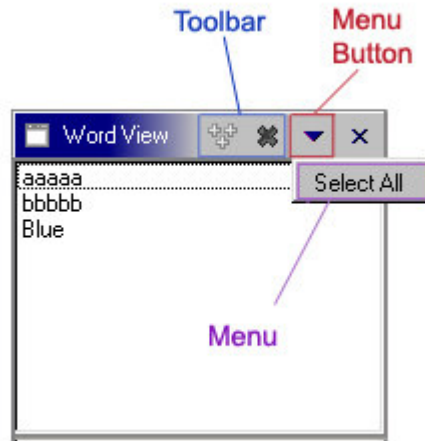
Once the Word view has been declared and compiled we can test the results. To do this, invoke `Perspective > Show View > Other`, and then select the Word View in the Show View dialog. The Word view will appear in the Resource perspective, as shown below.



Menus and Toolbars

As the Word view exists now it isn't very useful. We can browse a list of words, but we can't modify that list. In this section we'll add some menu and toolbar items to the view so the user can add and delete words within the list.

But first, some background information. Each view has a local menu and toolbar. The toolbar is located to the right of the view caption. The menu is initially invisible, but becomes visible if you click on the menu button, which is just to the left of the close button. For instance, here is a snapshot of the Word view with the menu and toolbar we are about to define.



The menu and toolbar controls for a view are initially empty and invisible. As the view adds items to each, the menu or toolbar will become visible. A view can also contribute items to the status line which appears at the bottom of the parent window. Together, the local menu, toolbar and status line are known as the *action bars*.

In code, access to the action bars is provided by the view site. The site is the primary interface between the view part and the outside world. If your view is a subclass of `ViewPart`, the site can be accessed by calling `ViewPart.getViewSite`. If not, then you must manage your own storage and retrieval of the site, which is passed to the view in the `IViewPart.init` method. Given the site, you can call `getActionBars().getMenuManager` to get an `IMenuManager`, or `getActionBars().getToolBarManager()` to get an `IToolBarManager`.

The `IMenuManager` and `IToolBarManager` interfaces are a JFace abstraction. They wrap an SWT `Menu` or `ToolBar` object so that you, the developer, can think in terms of action and action contribution. An *action* represents a command which can be triggered by the user from a menu or toolbar. It has a `run` method, plus other methods which return the menu or tool item presentation (text, image, etc.). In JFace, you create a menu or toolbar by adding instances of `org.eclipse.jface.action.IAction` to an `IMenuManager` or `IToolBarManager`.

Defining our Actions

We will contribute an "Add..." and "Delete" action to the Word view toolbar, and a "Select All" action to the view menu. In general, the initial actions within a view are contributed using Java™ code, and other plugin developers extend the menus and toolbars of a view using XML.

In the `WordView` class, the actions are created in the `createActions` method (shown below), which is called from `createPartControl`. For simplicity, each action is defined as anonymous subclass (A) of `org.eclipse.jface.actions.Action`. We override the `run` method (B) to implement the actual behavior of the action.

```

public void createActions() {
    A addItemAction = new Action("Add...") {
        B public void run() {
            addItem();
        }
    };
    C addItemAction.setImageDescriptor(getImageDescriptor("add.gif"));

    deleteItemAction = new Action("Delete") {
        public void run() {
            deleteItem();
        }
    };
    deleteItemAction.setImageDescriptor(getImageDescriptor("delete.gif"));

    selectAllAction = new Action("Select All") {
        public void run() {

```



```

        selectAll();
    }
};

// Add selection listener.
viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    public void selectionChanged(SelectionChangedEvent event) {
        updateActionEnablement();
    }
});
}

```

The label of each action is defined in the action constructor (A). In addition, an image is defined for the "Add..." and "Delete" actions (C), since they will appear in the toolbar. These images are defined by invoking `getImageDescriptor` (shown below), a method which returns an image descriptor for a file stored in the plugin directory. Within this method the `ViewsPlugin` object is retrieved, the install URL (plugin directory) is determined, and then an `ImageDescriptor` is created for a path based on this URL.

```

/**
 * Returns the image descriptor with the given relative path.
 */
private ImageDescriptor getImageDescriptor(String relativePath) {
    String iconPath = "icons/";
    try {
        ViewsPlugin plugin = ViewsPlugin.getDefault();
        URL installURL = plugin.getDescriptor().getInstallURL();
        URL url = new URL(installURL, iconPath + relativePath);
        return ImageDescriptor.createFromURL(url);
    }
    catch (MalformedURLException e) {
        // should not happen
        return ImageDescriptor.getMissingImageDescriptor();
    }
}

```

On the last line of `createActions` (D), a selection listener is created and added to the list viewer. In general, the enablement state of a menu or tool item should reflect the view selection. If a selection occurs in the Word view, the enablement state of each action will be updated in the `updateActionEnablement` method (shown below).

```

private void updateActionEnablement() {
    IStructuredSelection sel =
        (IStructuredSelection)viewer.getSelection();
    deleteItemAction.setEnabled(sel.size() > 0);
}

```

The approach taken for action enablement in the Word view is just one way to go. It is also possible to imagine an implementation where each action is a selection listener, and will enable itself to reflect the selection. This approach would make the actions reusable beyond the context of the original view.

Adding our Actions

Once the actions have been created we can add them to the existing menu and toolbar for the view. In the `WordView` class, this is done in the `createMenu` and `createToolbar` methods (shown below), which are called from `createPartControl`. In each case the view site is used to access the menu and toolbar managers, and then actions are added.

```

/**
 * Create menu.
 */
private void createMenu() {
    IMenuManager mgr = getViewSite().getActionBars().getMenuManager();
}

```

```

        mgr.add(selectAllAction);
    }

    /**
     * Create toolbar.
     */
    private void createToolbar() {
        IToolBarManager mgr = getViewSite().getActionBars().getToolBarManager();
        mgr.add(addItemAction);
        mgr.add(deleteItemAction);
    }

```

At this point the definition of the menu and toolbar are complete. If the Word view is opened in the workbench, the toolbar will be populated with the Add and Delete items. If you click the down arrow for the menu, you will see the Select All item there.

Context Menus

In Eclipse, it is very common to expose and invoke actions from the context menu. For instance, if you press and release the right mouse button over an IFile in the Navigator, a context menu appears with standard file operations like Open, Rename, and Delete. Experienced users come to expect this, so in this section we will define a context menu for the Word view.

But first, some background info. One of the primary goals for the platform UI is extensibility. In fact, it is this extensibility which gives you the freedom to add new views, editors, perspectives, actions, etc., to the platform. Of course, extensibility is a two way street. While you may wish to extend the platform, others may wish to extend your view. It is common for one plug-in to add actions to the menu, toolbar, or context menu of a view from another plugin.

Support for context menu extension is accomplished by collaboration between the view and the platform. In contrast to the local menu and toolbar for each view, which are created by the platform and populated by the view, each context menu within a view must be created by the view itself. This is because the context menus within a view are beyond the scope of platform visibility, and there may be zero, one, or many context menus within the view. Once a context menu has been created, the view must explicitly register each context menu with the platform. This makes it available for extension.

A context menu can be created using the SWT `Menu` and `MenuItem` classes directly, or it can be created using the JFace `MenuManager` class. If you define a context menu using the `MenuManager` class, the platform can extend this menu with actions. However, if you define a context menu using the SWT `Menu` and `MenuItem` classes directly, this is not possible.

Creating the Context Menu

In the `WordView` class, the context menu is created in the `createContextMenu` method (shown below), which is called from `createPartControl`. Within this method a `MenuManager` is created (A), a menu is created (B), and then the menu manager is passed to the platform for extension with other actions (C).

```

private void createContextMenu() {
    // Create menu manager.
    A MenuManager menuMgr = new MenuManager();
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager mgr) {
            fillContextMenu(mgr);
        }
    });

    // Create menu.
    B Menu menu = menuMgr.createContextMenu(viewer.getControl());
    viewer.getControl().setMenu(menu);

    // Register menu for extension.

```

```

C ▶      getSite().registerContextMenu(menuMgr, viewer);
    }

```

In the first few lines of `createContextMenu` a new `MenuManager` is created (A ▶), and it is configured for dynamic action population. In other words, whenever the menu is opened, the old actions will be removed, and new actions will be added to reflect the current selection. This behavior is a requirement for action extension, and if not implemented, the action extensions contributed by the platform will soon flood the context menu. To accomplish this, we invoke `setRemoveAllWhenShown(true)` to clear the menu when it opens. Then we add a menu listener to populate the menu when it opens. The menu listener just calls `fillContextMenu`, which will be discussed below.

After creating the `MenuManager`, we can create the menu, and then register the menu manager with the site (C ▶). Take a good look at the last line of `createContextMenu`. In this statement we pass the menu manager and the viewer to the site. In general, the platform will only add action extensions to a menu if it opens. At that time, the action extensions for the menu are determined by examining the menu id and view selection. So where is the menu id? To answer, we need to look at the definition of `IWorkbenchPartSite.registerContextMenu`. There are two `registerContextMenu` methods (shown below). If there is only one context menu in the view, we should call the first. The menu id will be derived from the part id to ensure consistency. So in this case, our menu id is "org.eclipse.ui.articles.views.wordview". If there is more than one context menu in the view, we should use the second, where the menu id is explicit.

```

// Defined on IWorkbenchPartSite.
public void registerContextMenu(MenuManager menuManager,
    ISelectionProvider selectionProvider);
public void registerContextMenu(String menuId, MenuManager menuManager,
    ISelectionProvider selectionProvider);

```

If the user opens our context menu, the `fillContextMenu` method will be called. This method adds the Add, Delete and SelectAll actions to the menu. It is important to notice that a `GroupMarker` is also added to the menu for "additions" (A ▶). This group will be used as a target for action extension by other plug-ins. If it does not exist, all of the action extensions will appear at the end of the menu.

```

private void fillContextMenu(IMenuManager mgr) {
    mgr.add(addItemAction);
A ▶    mgr.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
    mgr.add(deleteItemAction);
    mgr.add(new Separator());
    mgr.add(selectAllAction);
}

```

Supporting Extension of the Context Menu

In the last section extensibility was a prominent issue. If we want to support context menu extension in the Word view, or any other view, the following process should be followed.

1. If you create a context menu, publish the menu id in a public interface which other plug-in developers can reference.
2. Add a `GroupMarker` with `id == IWorkbenchActionConstants.MB_ADDITIONS` to each context menu. Other plug-in developers will use this as a target for menu extensions.
3. If you have additional groups in the menu, publish each id.
4. Register each context menu with your `IWorkbenchPartSite`. If you don't do this the workbench can't extend it.
5. Define an `IActionFilter` for each of the objects in your model. Within this filter, publish the filter attributes. This allows for more accurate targeting of menu extensions.

The first couple of items have already been discussed. At this point we will focus on the last point: define an `IActionFilter` for each object in your model.

The Simple Case of Action Extension

An action can be added to a context menu by defining an extension for the `org.eclipse.ui.popupMenus` extension point in the workbench plug-in. For instance, the following XML can be used to add the "Word Action" to the context menu for Word objects. Without going into too many details, the target for the action is specified by declaring an `objectContribution` (A) with an `objectClass` of `org.eclipse.ui.articles.views.Word` (B). Once this declaration is made, the action will appear within the context menu for any Word.

```
<extension point="org.eclipse.ui.popupMenus">
  A <objectContribution
      id="org.eclipse.ui.articles.pm1"
      B <objectClass="org.eclipse.ui.articles.views.Word">
        <action
            id="org.eclipse.ui.articles.a1"
            label="Word Action"
            menubarPath="additions"
            class="org.eclipse.ui.articles.views.WordActionDelegate"/>
        </objectContribution>
</extension>
```

The target for an `objectContribution` can be defined using the `objectClass` attribute and an optional `nameFilter` attribute. The `nameFilter` is used to specify objects with a specific label. Together, these attributes provide reasonable accuracy when targeting objects.

The Problem Case of Action Extension

In some situations the `objectClass` and `nameFilter` attributes are not enough to describe the intended target object. For instance, what if we wanted to add an action to a `.java` file which is read only, or a project which has the java nature? Luckily, the platform can help.

Within an `objectContribution`, a plug-in developer may define `filter` sub elements. Each `filter` describes one attribute of the target object using a name value pair. For instance, the following XML can be used to target an action at any Word object with `name = "Blue"`. Notice how the `filter` sub element (A) is used to define an attribute value pair.

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution
      id="org.eclipse.ui.articles.pm2"
      objectClass="org.eclipse.ui.articles.views.Word">
      A <filter name="name" value="Blue"/>
      <action
          id="org.eclipse.ui.articles.a2"
          label="Blue Action"
          menubarPath="additions"
          class="org.eclipse.ui.articles.views.BlueActionDelegate"/>
      </objectContribution>
</extension>
```

Now let's look at the implementation of the `filter` sub element. The attributes for an object are type specific, and beyond the domain of the workbench itself, so the workbench will delegate filtering at this level to the selection itself. To do this, the platform will test to see if the selected object implements an `org.eclipse.ui.IActionFilter`. This is a type specific filtering strategy. If the selection does not implement `IActionFilter`, the platform will ask the selection for a filter using the `IAdaptable` mechanism defined in `org.eclipse.core.runtime`. If a filter is found, the platform will pass each name value pair to the filter to determine if it matches the state of the selected object. If so, or there is no `IActionFilter`, the action will be added to the context menu for the object.

In order to support the XML above, we need to define an `IActionFilter` for the Word class. To do this, we define a class called `WordActionFilter` (shown below), which simply implements the `IActionFilter` interface (A). This class is defined as a singleton for memory efficiency (C). It also publishes the filter attributes for the Word class (B), so that they can be referenced outside the plugin.

```
A public class WordActionFilter implements IActionFilter {
```

```

B public static final String NAME = "name";
    private static WordActionFilter singleton;

C public static WordActionFilter getSingleton() {
        if (singleton == null)
            singleton = new WordActionFilter();
        return singleton;
    }

    /**
     * @see IActionFilter#testAttribute(Object, String, String)
     */
    public boolean testAttribute(Object target, String name, String value) {
        if (name.equals(NAME)) {
            Word le = (Word)target;
            return value.equals(le.toString());
        }
        return false;
    }
}

```

The action filter for the Word class will be exposed using the IAdaptable mechanism. To do this, we need to implement the IAdaptable interface on our Word object, and return the WordActionFilter singleton if the platform asks for an IActionFilter. The implementation of getAdapter is shown below.

```

public Object getAdapter(Class adapter) {
    if (adapter == IActionFilter.class) {
        return WordActionFilter.getSingleton();
    }
    return null;
}

```

At this point the WordActionFilter is complete. If we open the Word view and select a Word with *name = Blue*, the "Blue Action" will appear in the context menu. Stepping back from this example, the WordActionFilter is actually fairly useless. However, it does illustrate how precise action extension targeting should be supported by a model within a view (or editor).

In the platform, action filters already exist for markers, resources, and projects. The attribute names for each are defined in IMarkerActionFilter, IResourceActionFilter, and IProjectActionFilter within org.eclipse.ui, so other plug-in developers can reference them easily. If we look at an existing, production quality filter like IResourceActionFilter we can see that it contains matching attributes for NAME, EXTENSION, PATH, READ_ONLY and PROJECT_NATURE. You should define attributes which reflect your own model and permit precise action targeting.

Integration with the Window Menu and Toolbar

In the workbench window menu there are a number of pre-existing actions which target the active part (as indicated by a shaded title bar). For instance, if the Delete action within the Edit menu is invoked when the Navigator view is active, the Delete implementation is delegated to the Navigator view. If the Tasks view is active the Delete implementation is delegated to that view. The Delete action is just one of many actions, known as *global actions*, which are always visible within the window menu and, when invoked, delegate their implementation to the active part.

In the Word view there are three actions (Add, Delete and Select All). The complete set of global actions is declared in IWorkbenchActionConstants.GLOBAL_ACTIONS (shown below). A quick comparison will demonstrate that there is no semantic equivalent to Add, but the Delete (**A**) and Select all actions (**B**) are represented within the global actions. So the Word view will hook both of these actions.

```

/**
 * From IWorkbenchActionConstants.
 * Standard global actions in a workbench window.
 */
public static final String [] GLOBAL_ACTIONS = {
    UNDO,
    REDO,

```

```

        CUT,
        COPY,
        PASTE,
        PRINT,
        DELETE,
        FIND,
        SELECT_ALL,
        BOOKMARK
    };
}

```

In the `WordView` class, the global actions are hooked within `hookGlobalActions` (shown below), which is called from `createPartControl`. Within this method, the Word view retrieves the action bars from the workbench part site (A) and then calls `setGlobalActionHandler` for each action (B). This establishes a link between the global action in the window and the implementation within the view.

```

private void hookGlobalActions() {
    IActionBars bars = getViewSite().getActionBars();
    bars.setGlobalActionHandler(IWorkbenchActionConstants.SELECT_ALL, selectAll);
    bars.setGlobalActionHandler(IWorkbenchActionConstants.DELETE, deleteItem);
    viewer.getControl().addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent event) {
            if (event.character == SWT.DEL &&
                event.stateMask == 0 &&
                deleteItemAction.isEnabled())
            {
                deleteItemAction.run();
            }
        }
    });
}

```

On the last line of `hookGlobalActions` (C) we add a key listener to the viewer control. If the "delete" key is pressed, the Delete action should run. In the case of every other action except Delete, the accelerator is defined and implemented by the workbench, so there is no need for a key listener. In the case of Delete, however, a key listener must be defined in the part. This extra work is required because the platform cannot define an accelerator containing the delete key. In doing so, it would break any text editors where the "delete" key has two different behaviors: delete the selection, and delete the next character.

Registration of the global action handlers is complete. If the Select All or Delete action in the window is invoked, and the Word view is active, the corresponding handler within the Word view will be invoked. The approach taken here can be used for other actions, and you are encouraged to do so. For instance, most editors provide a handler for all of the global actions. The Navigator implements the Delete and Bookmark actions. The Tasks view implements the Delete and Select All actions.

Can I add new actions to the window menu or toolbar?

In general all view actions should be contributed to the local menu, toolbar, or context menu for a view. There is no way to add view specific actions to a window menu or toolbar. The justification for this is simple: the proximity of view actions to the view itself creates a stronger coupling between the two, and it also helps to reduce clutter on the window menu and toolbar.

Integration with Other Views

No view is an island. In most cases, a view co-exists with other views and selection within one view may affect the input of another. In this section we'll create a Listener view, which will listen for the selection of objects in the Word view. If a Word object is selected, the Listener view will display the word attributes. This behavior is similar to the existing Properties view in the workbench standard components.

To start out, we need to create a new Listener view. We've already done this twice, so let's skip the declaration within

the plugin.xml and concentrate on the ListenerView class (shown below). This class is very similar to LabelView. In createPartControl we create a simple SWT Label (A), where we will display the word attributes.

```
public class ListenerView extends ViewPart
    implements ISelectionListener
{
    private Label label;
    public ListenerView() {
        super();
    }
    public void setFocus() {
        label.setFocus();
    }
    A public void createPartControl(Composite parent) {
        label = new Label(parent, 0);
        label.setText("Hello World");
        B getViewSite().getPage().addSelectionListener(this);
    }

    /**
     * @see ISelectionListener#selectionChanged(IWorkbenchPart, ISelection)
     */
    C public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        if (selection instanceof IStructuredSelection) {
            Object first = ((IStructuredSelection)selection).getFirstElement();
            if (first instanceof Word) {
                label.setText(((Word)first).toString());
            }
        }
    }
}
```

Things get interesting on the last line of createPartControl (B). The Listener view exists in a page with other views. If a selection is made in any of those views, the platform will forward the selection event to all interested parties. We are an interested party. To register our interest, we get the site, get the page, and add the ListenerView as a selection listener. If a selection occurs within the page, the ListenerView.selectionChanged method will be called (C). Within this method, the selection is examined and, if it is a Word, the label text will be updated to reflect the Word name.

So far so good. The Listener view is ready to receive selection events. However, we have a problem: the Word view doesn't publish any selection events.

To reconcile our problem we add one line of code to the Word view (D). Within the createPartControl method, a selection provider is passed to the site. Fortunately, the viewer itself is an ISelectionProvider, so it is very easy to define the selection provider for the view. When the Word view is active (as indicated by shading in the title bar) the platform will redirect any selection events fired from viewer to selection listeners within the page.

```
public void createPartControl(Composite parent) {
    // Create viewer.
    viewer = new ListViewer(parent);
    viewer.setContentProvider(new ListContentProvider());
    viewer.setLabelProvider(new LabelProvider());
    viewer.setInput(input);
    D getViewSite().setSelectionProvider(viewer);

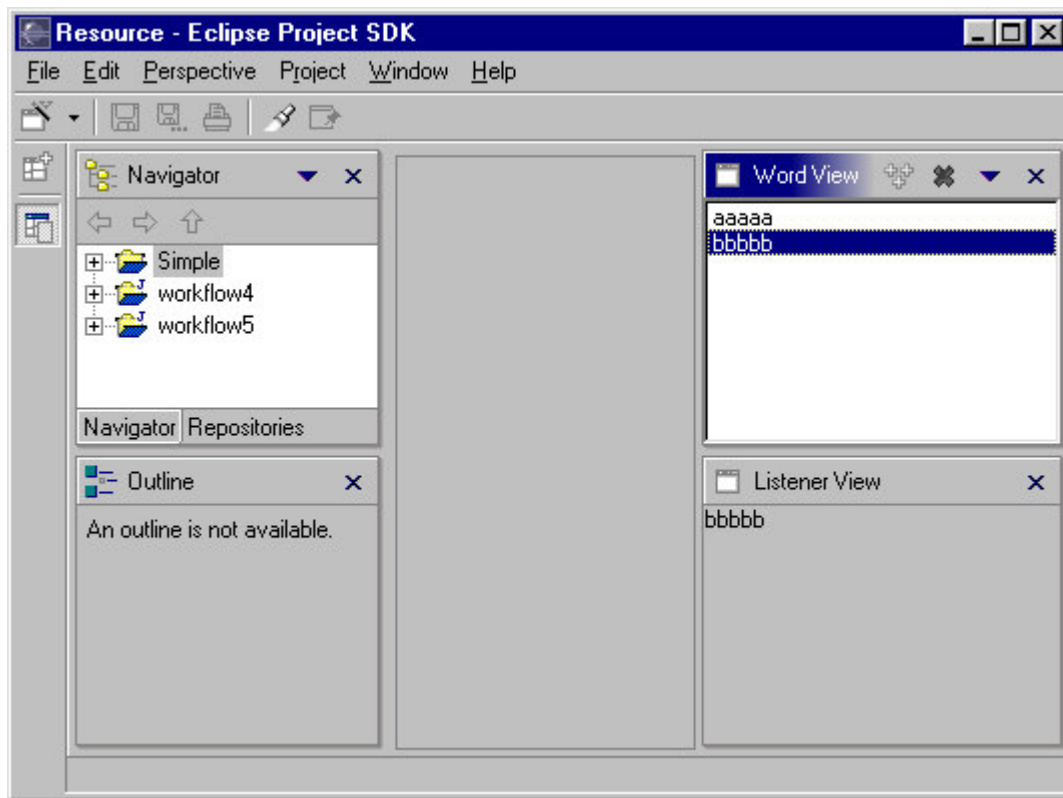
    // Create menu and toolbars.
    createActions();
    createMenu();
    createToolBar();
    createContextMenu();
    hookGlobalActions();

    // Restore state from the previous session.
    restoreState();
}
```

```
}

```

Now we can try out the Listener view. Open up the Word view and the Listener view. Add a word to the Word view and then select it. The name of the Word will be displayed in the Listener view. A snapshot of the result is shown below.



The linking approach demonstrated here can be described as *activation linking*. In activation linking, the listener receives selection from the active part. The benefit of this approach is that there is a loose coupling between the Listener view and other views within the same page. If we were to introduce another view, similar to the Word view, which published selection events for objects of type Word, these objects would also appear in the Listener view. This makes it very easy to add, remove, or replace the views in the workbench while maintaining good integration between views.

In contrast to activation linking, a view may also implement *explicit linking*. In explicit linking, the listener tracks selection within a specific source view, which is usually discovered by calling `IWorkbenchPage.findView(String id)`. There are two drawbacks to this approach. First, the listener will be disabled if the specific source view is nonexistent or closed by the user, and second, the listener will never work with additional views added by the user, or another plug-in. For this reason, the use of explicit linking is discouraged.

Integration with the WorkbenchPage Input

In a large, real world project the workspace may contain hundreds or thousands of resources. These resources are split among many projects, containing many folders, containing many files. It's a tree, and each subtree within the whole defines a physical subset of information. To avoid the information overload which can sometimes occur by looking at the whole, the user can "open a perspective" on any resource subtree within the workspace. This is done by selecting a resource in the Navigator view and invoking "Open Perspective". The result is a new workbench page where only the children of the subtree root are visible. This subtree root is known as the *input*.

In practice, the visibility of resources within a page is implemented through collaboration between the page and the views within that page. The page itself is just a visual container for views and editors. It doesn't provide any presentation for resources. That is delegated to the parts within the page. The hand off usually occurs during part creation. In the early stages of part lifecycle a part can obtain a handle to the containing `IWorkbenchPage`, and from this it can call `IWorkbenchPage.getInput()`. The result can be used as the initial input for the view. For instance, if a new perspective containing the Navigator is opened, the Navigator will use the page input as its own input. The

Packages view does the same.

The strategy should be adopted by any view which displays resources within the workspace. This will ensure a consistent navigational paradigm for users within the platform.

State Persistence

One of the primary goals for the platform UI is to provide efficient interaction with the workspace. In the platform this is promoted by saving the state of the workbench when a session ends. When a new session is started the state of the workbench is recreated. The state of each window, page, view and editor is persisted between sessions, reducing the time required for the user to get back to work. In this section we'll examine how the state of the Word view is saved.

Within any view there are at least two elements of state: the model itself and model presentation (widget state). In our Word view, the model is stored as a file in the file system, so it is a non issue. However, the widget state should be saved. To do this we need to implement the `IViewPart.init` and `IViewPart.saveState` methods (shown below) in `WordView`.

```
public void init(IViewSite site,IMemento memento) throws PartInitException;
public void saveState(IMemento memento);
```

But first, some background info. When the workbench is closed, the `IViewPart.saveState` method is called on every view. An `IMemento` is passed to the `saveState` method. This is a structured, hierarchical object where you can store integers, floats, Strings, and other `IMementos`. The platform will store the `IMemento` data in a file called `workbench.xml` within the `org.eclipse.ui` metadata directory. When a new session is started, the platform will read this file, recreate each window, page, and view, and then pass an `IMemento` to each view in the `init` method. The view can use this to recreate the state of the previous session.

Now we can do some coding. We will implement the `saveState` method first (shown below). Within this method a memento is created for the selection (A) and then one item is added to it for every word which is selected (B). In our code, the word string is used to identify the word. This may be an inaccurate way to identify words (two words may have the same string), but this is only an article. You should develop a more accurate strategy which reflects your own model.

```
public void saveState(IMemento memento){
    IStructuredSelection sel = (IStructuredSelection)viewer.getSelection();
    if (sel.isEmpty())
        return;
    A memento = memento.createChild("selection");
    Iterator iter = sel.iterator();
    while (iter.hasNext()) {
        B Word word = (Word)iter.next();
        memento.createChild("descriptor", word.toString());
    }
}
```

Next we implement the `init` method. When a new session is started, the `init` method will be called just after the part is instantiated, but before the `createPartControl` method is called. In our code all of the information within the memento must be restored to the control, so we have no choice but to hold on to the memento until `createPartControl` is called.

```
public void init(IViewSite site,IMemento memento) throws PartInitException {
    init(site);
    this.memento = memento;
}
```

Within `createPartControl`, the `restoreState` method is called. Within `restoreState` the selection memento is retrieved (A), and each descriptor within it is mapped to a real `Word` (B). The resulting array of words is used to restore the selection for the viewer (C).

```
private void restoreState() {
    if (memento == null)
        return;
    A memento = memento.getChild("selection");
```

```

    if (memento != null) {
        IMemento descriptors [] = memento.getChildren("descriptor");
        if (descriptors.length > 0) {
            ArrayList objList = new ArrayList(descriptors.length);
            for (int nX = 0; nX < descriptors.length; nX ++) {
                String id = descriptors[nX].getID();
                Word word = input.find(id);
                if (word != null)
                    objList.add(word);
            }
            viewer.setSelection(new StructuredSelection(objList));
        }
    }
    memento = null;
    updateActionEnablement();
}

```

You may notice certain limitations in the IMemento interface. For instance, it can only be used to store integers, floats, and Strings. This limitation is a reflection of the persistence requirements in the platform UI.

1. Certain objects need to be saved and restored across platform sessions.
2. When an object is restored, an appropriate class for an object might not be available. It must be possible to skip an object in this case.
3. When an object is restored, the appropriate class for the object may be different from the one when the object was originally saved. If so, the new class should still be able to read the old form of the data.

Java serialization could be used for persistence. However, serialization is only appropriate for RMI and light-weight persistence (the archival of an object for use in a later invocation of the same program). It is not considered robust for long term storage, where class names and structure may change. If any of these occur serialization will throw an exception.

State Inheritance

If we take a close look at the standard components in Eclipse an interesting pattern emerges. Most of them provide some degree of customization. For instance, in the Navigator view there are two actions in the menu, Sort and Filter, which can be used to sort or select the visible resources in the Navigator. Similar functionality exists within the Tasks view. In general, customization is a desirable feature. In this section we're not going to look at customization itself, as customization is very specific to a particular view and the underlying model. Instead, we'll look at some different strategies which you could use to share user preferences between views.

Let's start out with the assumption that the Word view has two sorting algorithms. The user may select one as the preferred algorithm. How do we share this preference with other views of the same type?

Approach #1: We don't.

The initial value of the preference will be hard coded in some way, probably as a constant in the class declaration. If the user changes this preference in one view, it will not be reflected in other views. For instance, the Navigator preference for "Sort" is a local preference, and is not shared with other Navigator views.

Approach #2: We Use a Preference Page

The initial value of the preference will be exposed in a preference page in the Workbench Preferences. If the user changes this preference, the new preference will apply to all instances of the view. For instance, the Navigator preference for "Link Navigator selection to active editor" is controlled by a check box in the Workbench Preferences. This option applies to all Navigator views.

Approach #3: Inheritance.

The initial value of the preference will be defined in the plugin metadata. If a view is opened, it will read the plugin metadata to determine the preference. If the user changes the preference in one view, it will not affect other views, but

it will be saved to the plugin metadata, so that any view created in the future will inherit the preference. For instance, if the user changes the sort preference in one Word view, it will have no affect on other Word views. However, if the user opens a new Word view it will inherit the most recent preference.

While each of these is a valid approach, the correct choice should reflect your own view and the underlying model.

Summary

In this article we have examined the design and implementation of two views, a simple Label view and a more comprehensive Word view. The features of your own view will probably vary greatly from what we have created. However, the integration of your view with the workbench will follow the same pattern demonstrated by these views. Further information on view implementation is available in the Platform Plug-in Developer Guide and in the javadoc for `org.eclipse.ui`.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.