Building a Common Navigator Framework (CNF) Viewer Part V: Action Providers

THOUGHT IS THE BLOSSOM; LANGUAGE THE BUD; ACTION THE FRUIT BEHIND IT.

-- **RALPH WALDO EMERSON**

Overview

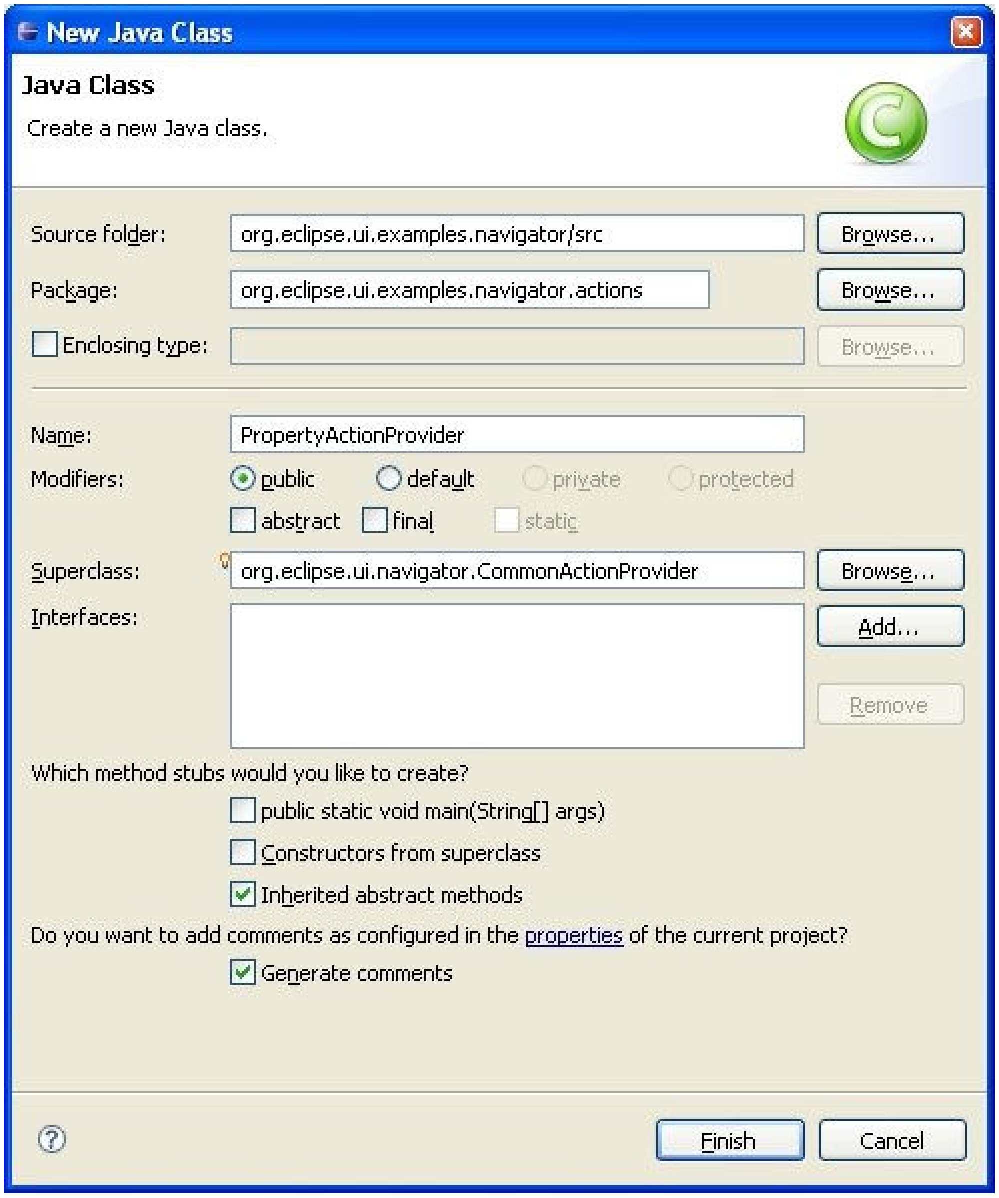
Action Providers provide a means to configure the retargetable actions and programmatically configure the context menu in a Common Navigator Framework viewer. In this article, we'll take a look at what an ActionProvider can do, and how to implement one for our Example View.

Action Providers are useful for cases where you have to perform some computation before you can decide what items to add to a menu or you need to adjust the *retargetable actions* to ensure that user keystrokes are handled properly (like Cut(Ctrl+x)/Copy(Ctrl+c)/Paste(Ctrl+p)/Delete(Delete)).

Creating our Action Provider implementation

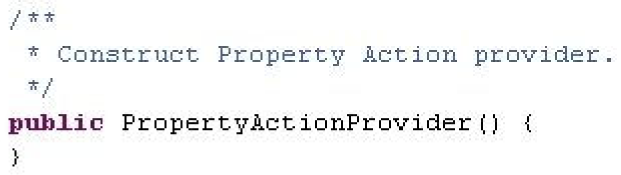
In other posts, we started with the XML configuration for the extension. For this article, we'll start from the code.

First, we'll create our Action Provider class, which must extend **org.eclipse.ui.navigator.CommonActionProvider**.



Source folder: org.eclipse.ui.examples.navigator/src
Package: org.eclipse.ui.examples.navigator.actions
[ ] Enclosing type:
Name: PropertyActionProvider
Modifiers: [✓ public] default private protected
[ ] abstract [ ] final [ ] static
Superclass: org.eclipse.ui.navigator.CommonActionProvider
Interfaces:
Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments as configured in the properties of the current project?
[✓] Generate comments

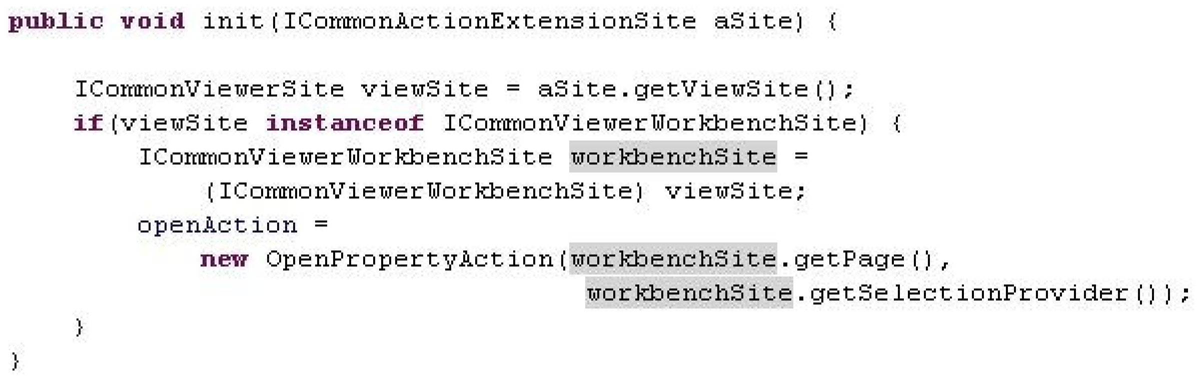
Our class must define a no-argument constructor so that instances may be created from our extension.



```
/**
 * Construct Property Action provider.
 */
public PropertyActionProvider() {
}
```

In our **init()** method, we verify that we are executing inside of a workbench part (opposed to a dialog), and then create the OpenAction, which takes the workbench page and a selection provider in its constructor. The **ICommonActionExtensionSite** provides access to several things that a **CommonActionProvider** might need. I'll refer you to the Javadoc API for more details here.

Common Navigator viewers are capable of being enclosed in dialogs. Thus, a reference that is a subclass of **ICommonViewerSite** is supplied by default from the **ICommonActionExtensionSite**. If your Action Provider is used by a viewer in a dialog, you cannot cast down to access the workbench page, since it is not available. In general, most Action Providers make assumptions that they are only used in workbench parts.



```
public void init(ICommonActionExtensionSite aSite) {

    ICommonViewerSite viewSite = aSite.getViewSite();
    if(viewSite instanceof ICommonViewerWorkbenchSite) {
        ICommonViewerWorkbenchSite workbenchSite =
            (ICommonViewerWorkbenchSite) viewSite;
        openAction =
            new OpenPropertyAction(workbenchSite.getPage(),
                                    workbenchSite.getSelectionProvider());
    }
}
```

The **fillActionBars()** method can be used to configure *retargetable actions*. A *retargetable action* is an instance of **org.eclipse.ui.actions.RetargetAction** with a unique identifier. The workbench registers many of these actions by default (see **org.eclipse.ui.IWorkbenchActionConstants**).

There is currently only one *retargetable action* defined by the Common Navigator Framework (CNF), **org.eclipse.ui.navigator.ICommonActionConstants.OPEN**. (But of course, you can register an action for any of the identifiers in

IWorkbenchActionConstants). In the CNF, a **RetargetAction** with the **ICommonActionConstants.OPEN** identifier is executed each time an open event occurs on a viewer. If no action is registered when an open event occurs on a node in the viewer, the node's expanded state is toggled.

In our example, we are going to register our OpenAction if it is enabled (we'll take a look at how this is determined shortly).

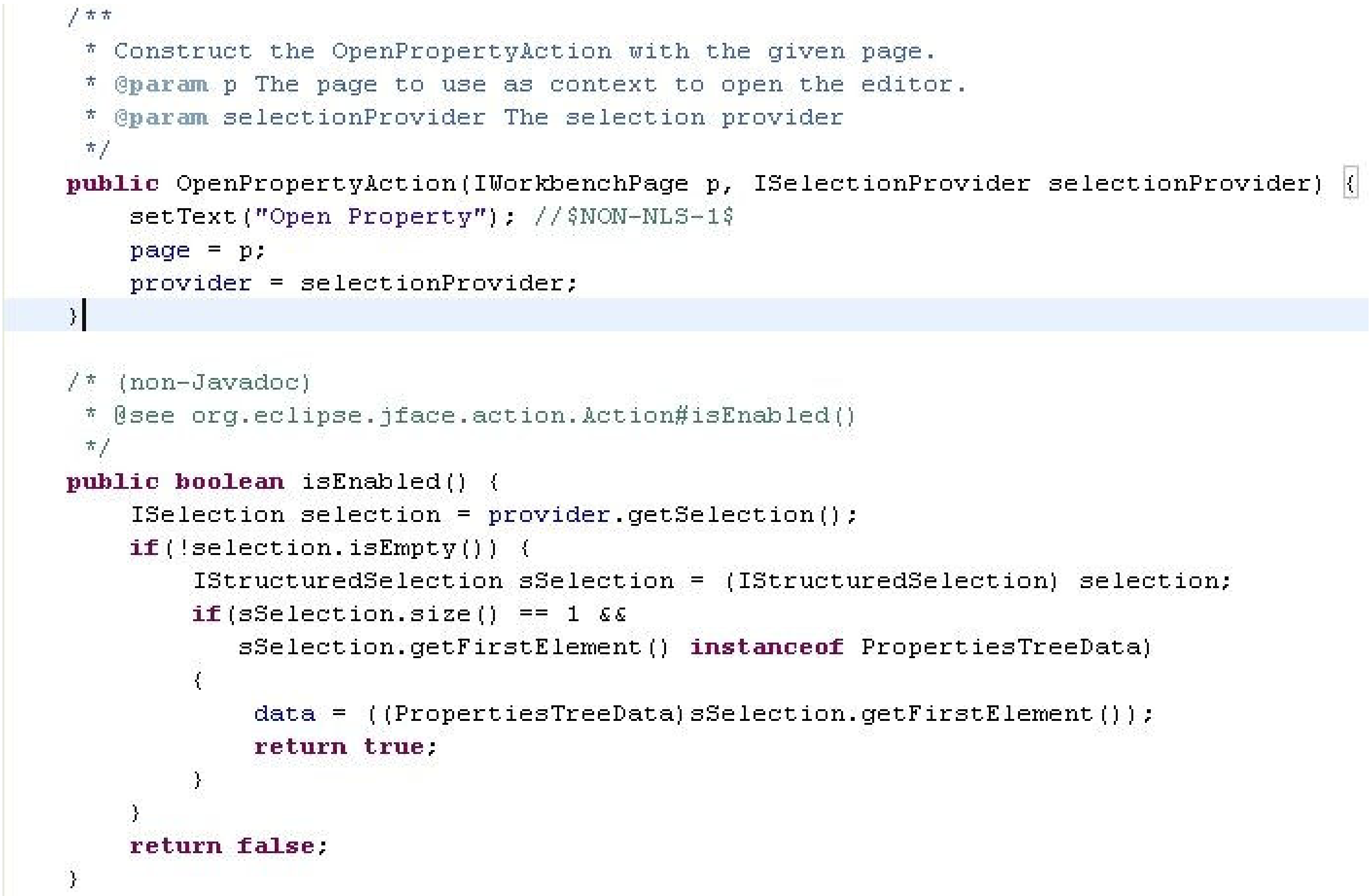
Note: If this action provider is used in a dialog context, the *openAction* would be **null** because we only initialize it if we receive an **ICommonViewerWorkbenchSite**, so if you need your Action Provider to be robust enough for either workbench parts or dialogs, be that either all of your action fields are initialized or you check for **null**.

```
public void fillActionBars(IActionBars actionBars) {
    /* Set up the property open action when enabled. */
    if(openAction.isEnabled())
        actionBars.setGlobalActionHandler(ICommonActionConstants.OPEN, openAction);
}
```

The **fillContextMenu()** method uses the **ICommonMenuConstants** to make sure that the *openAction* is positioned correctly in the menu.

```
public void fillContextMenu(IMenuManager menu) {
    if(openAction.isEnabled())
        menu.appendToGroup(ICommonMenuConstants.GROUP_OPEN, openAction);
}
```

Our *openAction* is fairly simple. The contents of the **run()** method are unimportant for this example, but you can take a look at the example in the repository. The constructor and the **isEnabled()** method are shown here. The constructor remembers the references we pull from the **ICommonViewerWorkbenchSite**.



```
/**
 * Construct the OpenPropertyAction with the given page.
 * @param p The page to use as context to open the editor.
 * @param selectionProvider The selection provider
 */
public OpenPropertyAction(IWorkbenchPage p, ISelectionProvider selectionProvider) {
    setText("Open Property"); //$NON-NLS-1$
    page = p;
    provider = selectionProvider;
}

/* (non-Javadoc)
 * @see org.eclipse.jface.action.Action#isEnabled()
 */
public boolean isEnabled() {
    ISelection selection = provider.getSelection();
    if(!selection.isEmpty()) {
        IStructuredSelection sSelection = (IStructuredSelection) selection;
        if(sSelection.size() == 1 &&
           sSelection.getFirstElement() instanceof PropertiesTreeData)
        {
            data = ((PropertiesTreeData)sSelection.getFirstElement());
            return true;
        }
    }
    return false;
}
```

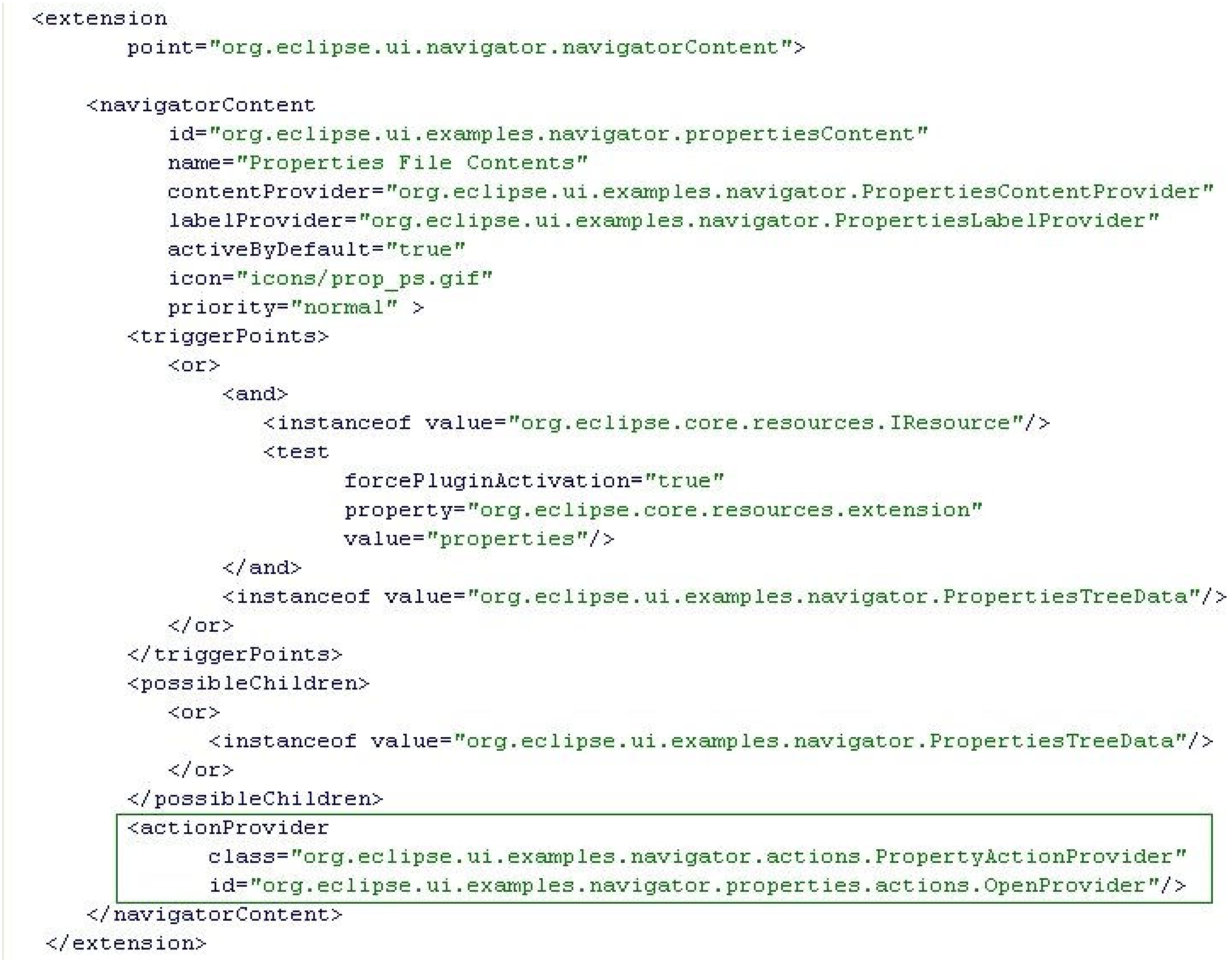
Adding our Action Provider to our viewer

Now let's take a look at our extension definition. Here we nest the **actionProvider** under our **navigatorContent** element in our **org.eclipse.ui.navigator.navigatorContent** extension. Remember that we constructed the rest of this extension in previous articles.

The **actionProvider** element specifies an identifier and our implementation class.

Since we have nested the **actionProvider** element under our **navigatorContent** extension, the existing **viewerContentBinding** that *binds* our content extension to our viewer also absorbs our action provider.

Furthermore, the menu options and *retargetable action* configuration will only be available if our extension is active. If the user turns off our extension, then they will not need these actions, the CNF will not invoke them.

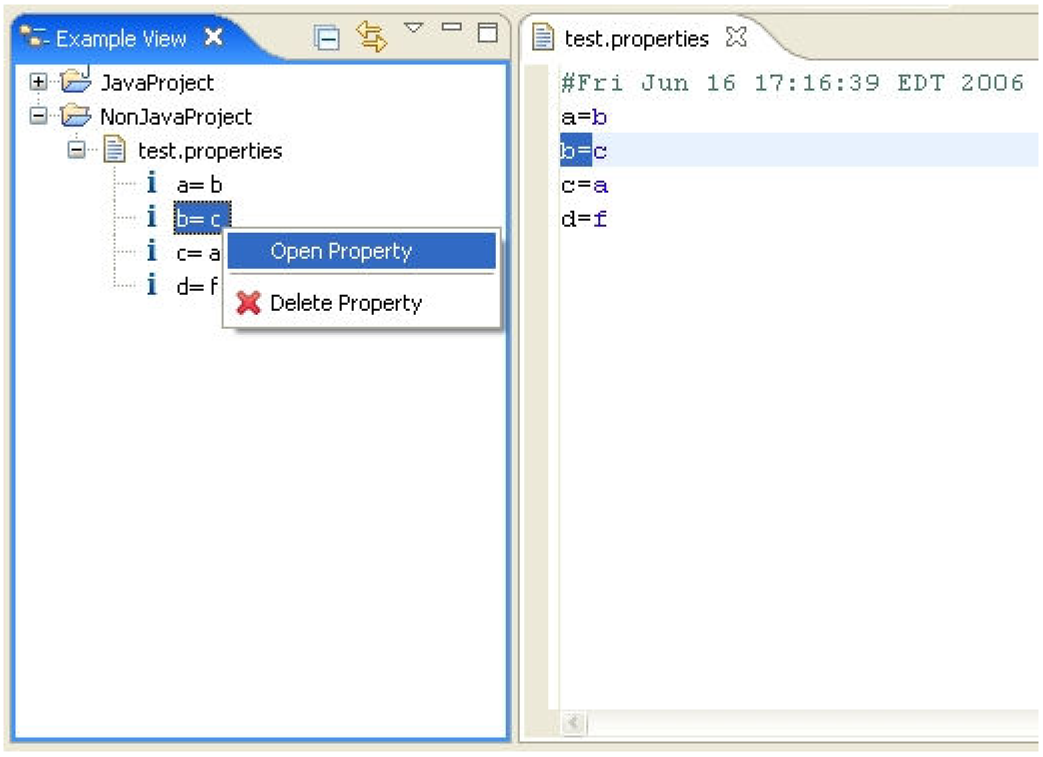


```
<extension
       point="org.eclipse.ui.navigator.navigatorContent">

    <navigatorContent
          id="org.eclipse.ui.examples.navigator.propertiesContent"
          name="Properties File Contents"
          contentProvider="org.eclipse.ui.examples.navigator.PropertiesContentProvider"
          labelProvider="org.eclipse.ui.examples.navigator.PropertiesLabelProvider"
          activeByDefault="true"
          icon="icons/prop_ps.gif"
          priority="normal" >
       <triggerPoints>
          <or>
              <and>
                 <instanceof value="org.eclipse.core.resources.IResource"/>
                 <test
                       forcePluginActivation="true"
                       property="org.eclipse.core.resources.extension"
                       value="properties"/>
              </and>
              <instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>
          </or>
       </triggerPoints>
       <possibleChildren>
          <or>
             <instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>
          </or>
       </possibleChildren>
       <actionProvider
             class="org.eclipse.ui.examples.navigator.actions.PropertyActionProvider"
             id="org.eclipse.ui.examples.navigator.properties.actions.OpenProvider"/>
    </navigatorContent>
 </extension>
```

Alternatively, we could expose our **actionProvider** independently and use a **viewerActionBinding** to *bind* the Action Provider to our viewer. In this case, the action provider will be available, even if our content extension is not.

```
<extension
      point="org.eclipse.ui.navigator.navigatorContent">
  <actionProvider
        class="org.eclipse.ui.examples.navigator.actions.PropertyActionProvider"
        id="org.eclipse.ui.examples.navigator.properties.actions.OpenProvider"/>
</extension>

<extension
      point="org.eclipse.ui.navigator.viewer">
   <viewerActionBinding viewerId="org.eclipse.ui.examples.navigator.view">
      <includes>
         <actionExtension
             pattern="org.eclipse.ui.examples.navigator.properties.actions.*"/>
      </includes>
   </viewerActionBinding>
</extension>
```

The final example view should appear like the following. When the "Open Property" action is selected, the property file will be opened and the property name will be selected.



```
#Fri Jun 16 17:16:39 EDT 2006
a=b
b=c
c=a
d=f
```

Summary

In this article, we have seen why an Action Provider does, walked through how to implement an Action Provider, and bound the Action Provider to our viewer.

In our next article, we'll take a look at sorting our properties elements.

ABOUT THE AUTHOR

MICHAEL ELDER
RESEARCH TRIANGLE PARK, NORTH CAROLINA, UNITED STATES

Michael has been a Java developer since 1999 and currently contributes to Eclipse in open source and commercial venues. Most recently, Michael has contributed the Common Navigator Framework (CNF) to Eclipse 3.2. Prior to that, he helped design several of the frameworks and API in the Web Tools Platform (WTP), and continues to contribute fixes to WTP through the open source process. Michael is currently part of the Services Oriented Architectures (SOA) Tools Team for IBM Rational® based in Research Triangle Park, NC.