Workbench Selection

Using SelectionService with a Custom View

Table of Contents

1 Introduction

In this tutorial you will:

- Create a new View
- Locate an Eclipse RCP Service
- Listen for the workbench selection
- Use “IAdaptable” to allow a single selection entry to represent multiple values

This workbook answers the bigger question of “Where to Start” when making your own application.

You will often start by contributing to the user interface - in this case we are defining a new view. This definition consists of both an XML fragment being added to the plugin.xml defining the title, icon etc... and a new class implementing the user interface. In other cases you may be defining a new menu option or a new tool.

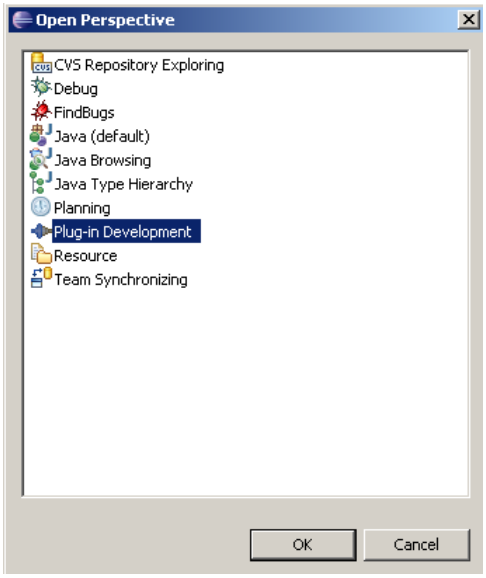
The second step will be paying attention to the what the user is up to - in this case we locate the SelectionService for the workbench window. In other cases you may be checking the current Map or the currently selected Layer.

Finally we will be acting when the user does something – in this case we are waiting for the user to select something and reporting back on what we find.

2 Defining A View

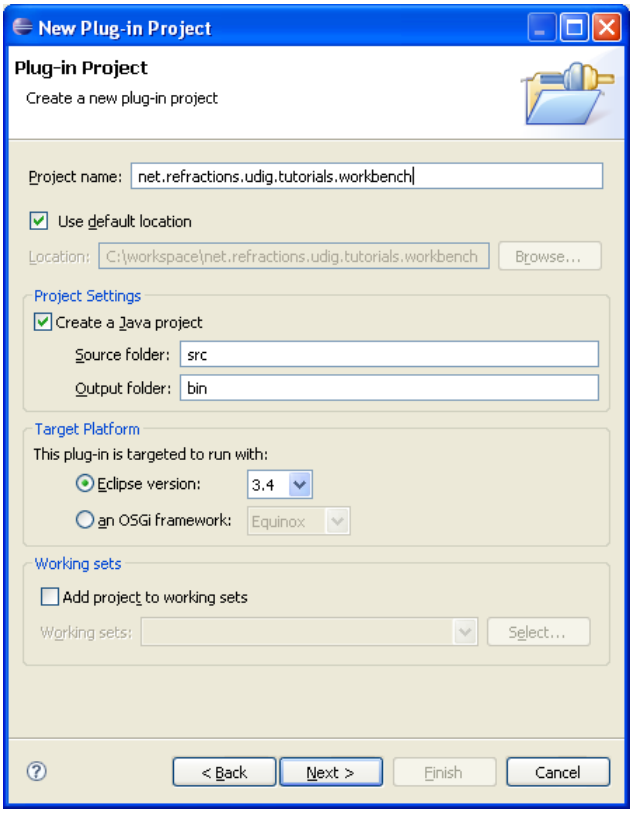
We are going to use the facilities of the **org.eclipse.ui** plug-in to define a new view.

1. Open the Plug-in Development Perspective.



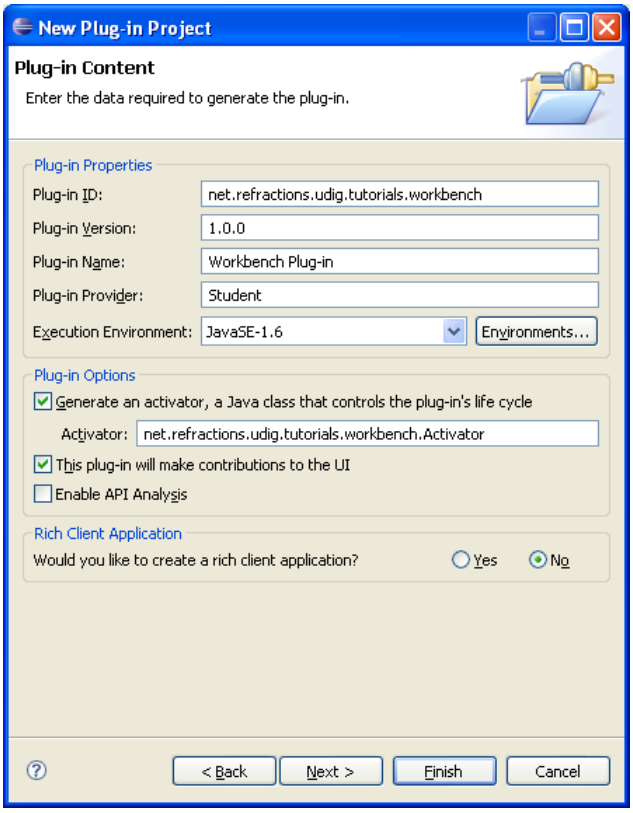
2. Create a new plug-in named **net.refractions.udig.tutorials.workbench**

Please review the earlier distance tool tutorial for detailed instructions on creating a new plug-in and adding dependencies.



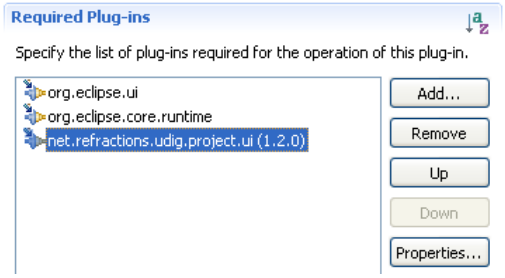
3. Press **Next** to continue

4. This time we are going to make contributions to the UI:



5. Press **Finish** to create the plug-in, the MANIEST.MF file for your new plug-in will be opened.

6. Add the following plug-in dependencies to your MANIFEST.MF:
net.refractions.udig.project



These steps make use of the Plug-in Manifest Editor; using the Dependencies and Extensions tabs

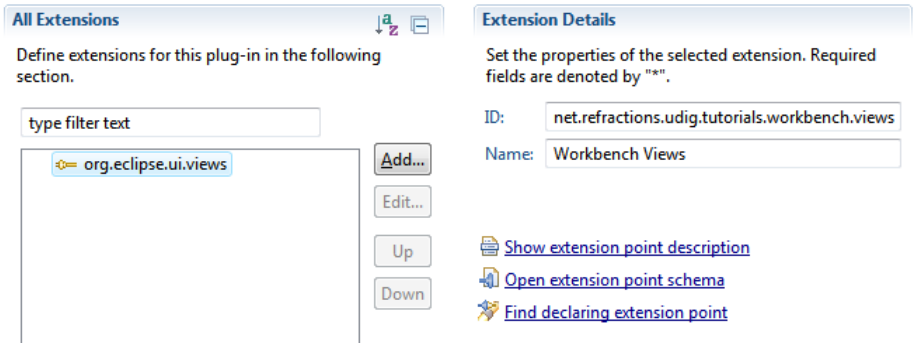
7. Save your MANIFEST.MF file, now that the plug-in knows about the dependency on **org.eclipse.ui** we are able to create an extension for the **org.eclipse.ui.views** extension point.

8. Switch to the **Extensions** tab of the Manifest editor and press the **Add** button.

9. Choose **org.eclipse.ui.views** from the wizard and press **Finish.**

The ID and Name for the extension point are mostly used when reporting exceptions in the log.

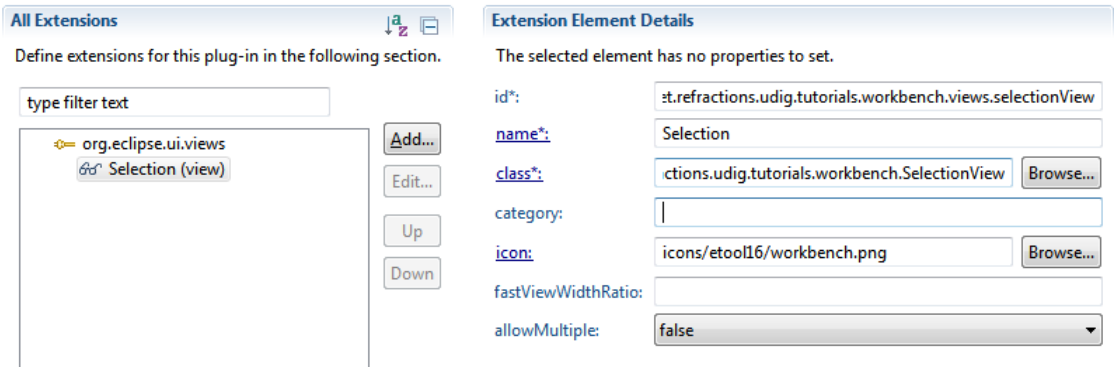
10. Select the new extension and set
    ID: net.refractions.udig.tutorials.workbench.views
    Name: Workbench Views



11. Right click on the new extension and select **New > View**

12. Take a moment to add the workbench_icons.zip to your project:
    http://udig.refractions.net/files/tutorials/workbench_icons.zip

13. Select the new view element and set:
    id: net.refractions.udig.tutorials.workbench.views.selectionView
    name: Selection
    class: net.refractions.udig.tutorials.workbench.SelectionView
    icon: icons/etool16/workbench.png
    allowMultiple: false

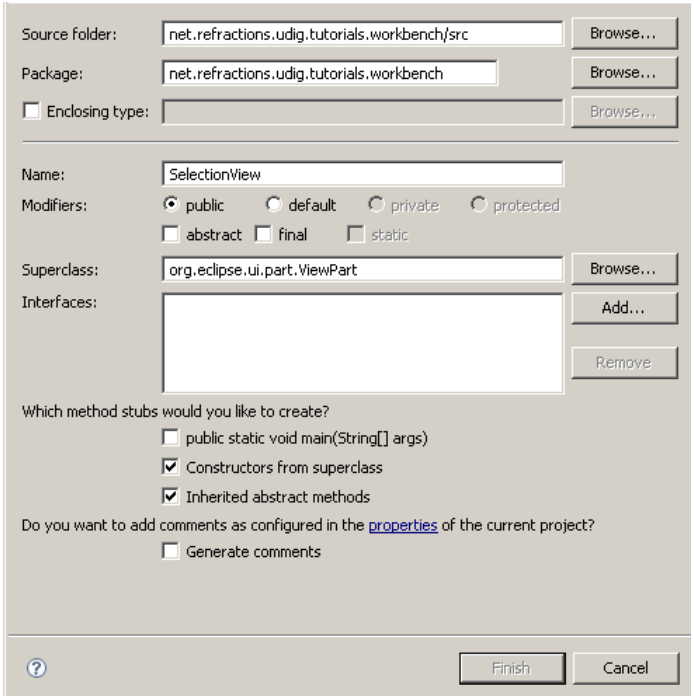


14. Save your **MANIEST.MF** file.

15. Click on the **class** link to open the New Java Class Wizard

The New Java Class wizard is aware of what you are doing and has filled in the ViewPart interface already.

16. Enter the following information:
    Name: SelectionView
    Package: net.refractions.udig.tutorials.workbench
    Superclass: org.eclipse.ui.part.ViewPart



17. Click the **Finish** button.

18. The created SelectionView class is opened for you.

```
package net.refractions.udig.tutorials.workbench;

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

public class SelectionView extends ViewPart {

	public SelectionView() {
		// TODO Auto-generated constructor stub
	}

	@Override
	public void createPartControl(Composite parent) {
		// TODO Auto-generated method stub

	}

	@Override
	public void setFocus() {
		// TODO Auto-generated method stub

	}

}
```

3 Implementing SelectionView

Now that our view has been created we can carefully implement the following methods

1. Now that you have some template code, let us start filling it in. Do not worry about the constructor. The constructor is called when your view is opened and displayed on the screen. There is no need for you to do any work inside of it.

```
public SelectionView() {
	//don't put anything in here
}
```

2. Create an init method to load in any settings from the previous run.
   You can quickly override methods by pressing **Control-Space** on any blank line and choosing the method you wish to implement from the provided list.

```
@Override
public void init( IViewSite site, IMemento memento ) throws PartInitException {
	super.init(site, memento);
}
```

3. The init method is called when your plugin is going to be used; and it provides two valuable pieces of information:

- **IViewSite:** is a utility class used to access workbench resources such as the status line or workbench window. You can retrieve the IViewSite at any point after the init method by using **`getViewSite()`**
- **IMemento:** is used to hold on to information (usually provided by the user) between runs.

1. The next method is in charge of making a user interface; the provided parent "widget" has been created for our use (we can set the layout and create any child widgets as needed).

```
private Text text;
private Text description;
@Override
public void createPartControl( Composite parent ) {
parent.setLayout( new GridLayout(2,false) );
    Label label = new Label(parent, SWT.RIGHT );
    label.setLayoutData(new GridData(SWT.RIGHT,SWT.TOP,false,false));
    label.setText("Selection:");
    text = new Text(parent, SWT.DEFAULT | SWT.READ_ONLY );
    text.setTextLimit(70);
    GridData gridData = new GridData(SWT.FILL,SWT.FILL,true,false);
    text.setLayoutData( gridData );

    label = new Label(parent, SWT.RIGHT );
    label.setLayoutData(new GridData(SWT.RIGHT,SWT.TOP,false,false));
    label.setText("Content:");
    description =
         new Text(parent, SWT.V_SCROLL | SWT.H_SCROLL | SWT.MULTI );
    gridData = new GridData(SWT.FILL,SWT.FILL,true,true );
    gridData.widthHint = 500;
    gridData.heightHint = 200;
    description.setLayoutData( gridData );
    //LISTEN TO THE WORKBENCH
}
```

2. You can use **Control-Shift-O** to sort out the required imports.

3. We are going to create a selection listener to watch the workbench for us. You can create this inner class at the top of the file before the constructor.

```
private final class WorkbenchSelectionListener implements ISelectionListener {
    public void selectionChanged( IWorkbenchPart part,
                                  ISelection selection ) {
        if( selection instanceof IStructuredSelection ){
            updateSelection( (IStructuredSelection) selection );
        }
        else {
            updateSelection( null );
        }
    }
}
private ISelectionListener selectionListener;
```

4. We can now add our code to "listen to the workbench" to the createPartControl method.

```
    // LISTEN TO THE WORKBENCH
    selectionListener = new WorkbenchSelectionListener();
    ISelectionService selectionService =
            getViewSite().getWorkbenchWindow().getSelectionService();
    selectionService.addPostSelectionListener(selectionListener);
```

5. You can see how the getViewSite() method is used to look up and make use of eclipse RCP facilities.

6. Although we do not use it in this tutorial, we could assign focus to one of the controls if we were excepting input. If we did, it would be something like: description.setFocus()

```
@Override
public void setFocus() {
       //this is where we would put stuff
}
```

7. It is good practice to clean up after any resources used (things such as widgets, colors, images and fonts). You should always be careful with null checks and never assume that createPartControl() has been called.

```
@Override
public void dispose() {
    if( selectionListener != null ){
        // if our init method failed selectionListener would be null!
        //
        ISelectionService selectionService =
            getViewSite().getWorkbenchWindow().getSelectionService();
        selectionService.removePostSelectionListener(
            selectionListener
        );
        selectionListener = null;
    }
    super.dispose();
}
```

8. Now that we know something is happening we can quickly inspect the value.

```
protected void updateSelection( IStructuredSelection selection ) {
    if( selection == null || selection.isEmpty() ){
        text.setText("(nothing is selected)");
        return;
    }
    Object object = selection.getFirstElement();
    if( object == null ){
        text.setText("(selected object is null)");
        return;
    } else {
        text.setText( object.toString() );
    }
    // DESCRIBE
}
```

9. The IStructuredSelection is used to return multiple values; on the line above we are checking if the first element is non null. When it comes time to describe the selection we will iterate over all the contents.

10. We can continue to examine the value; producing a description based on what kind of interfaces the Object supports.
    We are performing this check with a simple “instance of” that will return true if the provided object implements the interface or class mentioned.

```
    // DESCRIBE
    StringBuffer buffer = new StringBuffer();
    String separator = System.getProperty("line.separator");
    for(Iterator<?> iterator=selection.iterator();
        iterator.hasNext(); ){

        object = iterator.next();
        buffer.append( "VALUE: ");
        buffer.append( object.toString());
        buffer.append(separator);
        buffer.append("==========================");
        buffer.append(separator);
        // from net.refractions.udig.project
        if( object instanceof IMap ){
            buffer.append("instance of Map");
            buffer.append(separator);
        }
        if( object instanceof ILayer ){
            buffer.append("instance of ILayer");
            buffer.append(separator);
        }
        // from net.refractions.udig.catalog
        if( object instanceof IService ){
            buffer.append("instance of IService");
            buffer.append(separator);
        }
        if( object instanceof IGeoResource){
            buffer.append("instance of IGeoResource");
            buffer.append(separator);
        }
        // from org.geotools
        if( object instanceof Filter ){
            buffer.append("instance of Filter");
            buffer.append(separator);
        }
        if( object instanceof Feature ){
            buffer.append("instance of Feature");
            buffer.append(separator);
        }
        // IADAPTABLE
    } // NEXT
    description.setText(buffer.toString());
}
```

11. Up until this point we are working with Java objects using normal Java syntax.

    An object can support multiple interfaces (each representing a different API used to interact with the object) but the choice of what API to implement has been made at compile time.

12. Eclipse also has the facility to support additional interfaces at runtime using IAdatable.

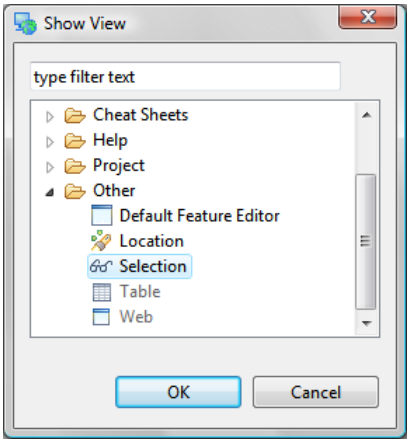
```
        // IADAPTABLE
        buffer.append("--------------------------");
        buffer.append(separator);
        if( object instanceof IAdaptable){
            // IAdaptable is a magic interface that allows
            // a single object to return multiple interfaces
            IAdaptable adaptable = (IAdaptable) object;
            if( adaptable.getAdapter(IMap.class) != null ){
                buffer.append("adapts to Map");
                buffer.append(separator);
            }
            if( adaptable.getAdapter(ILayer.class) != null ){
                buffer.append("adapts to ILayer");
                buffer.append(separator);
            }
            // from net.refractions.udig.catalog
            if( adaptable.getAdapter(IService.class) != null ){
                buffer.append("adapts to IService");
                buffer.append(separator);
            }
            if( adaptable.getAdapter(IGeoResource.class) != null){
                buffer.append("adapts to IGeoResource");
                buffer.append(separator);
            }
            // from org.opengis
            if( adaptable.getAdapter(Filter.class) != null ){
                buffer.append("adapts to Filter");
                buffer.append(separator);
            }
            if( adaptable.getAdapter(Feature.class) != null ){
                buffer.append("adapts to Feature");
                buffer.append(separator);
            }
            if( adaptable.getAdapter(URL.class) != null ){
                buffer.append("adapts to URL");
                buffer.append(separator);
            }
        }
    } // NEXT
    description.setText(buffer.toString());
}
```

13. You can see that the getAdapter works in a similar manner to an instance of check; returning non null if the object can be “adapted” into the requested interface.

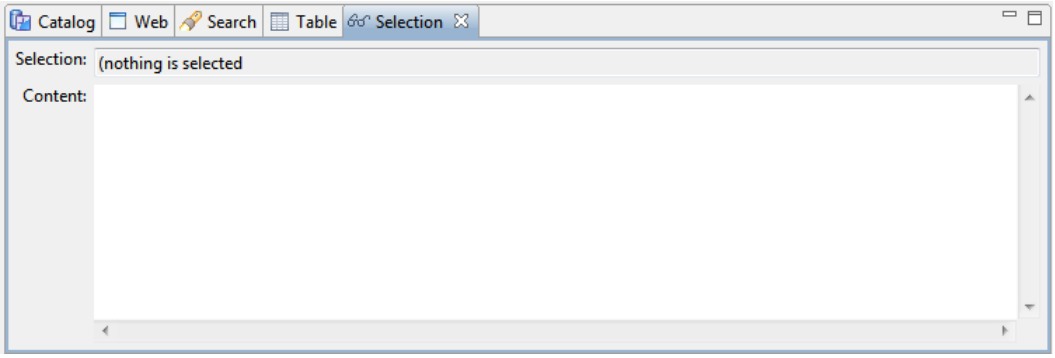
4 Running your code

We can now run uDig and try out your new plug-in:

1. Select **Run > Run Configurations ...** from the menu bar and choose the configuration you set-up in the previous tutorial. Be sure to add your new plug-in to the list.
2. Once uDig is up and running select the following from the main menu across the tope: **Window > Show View > Other...** to open the Show View dialog.

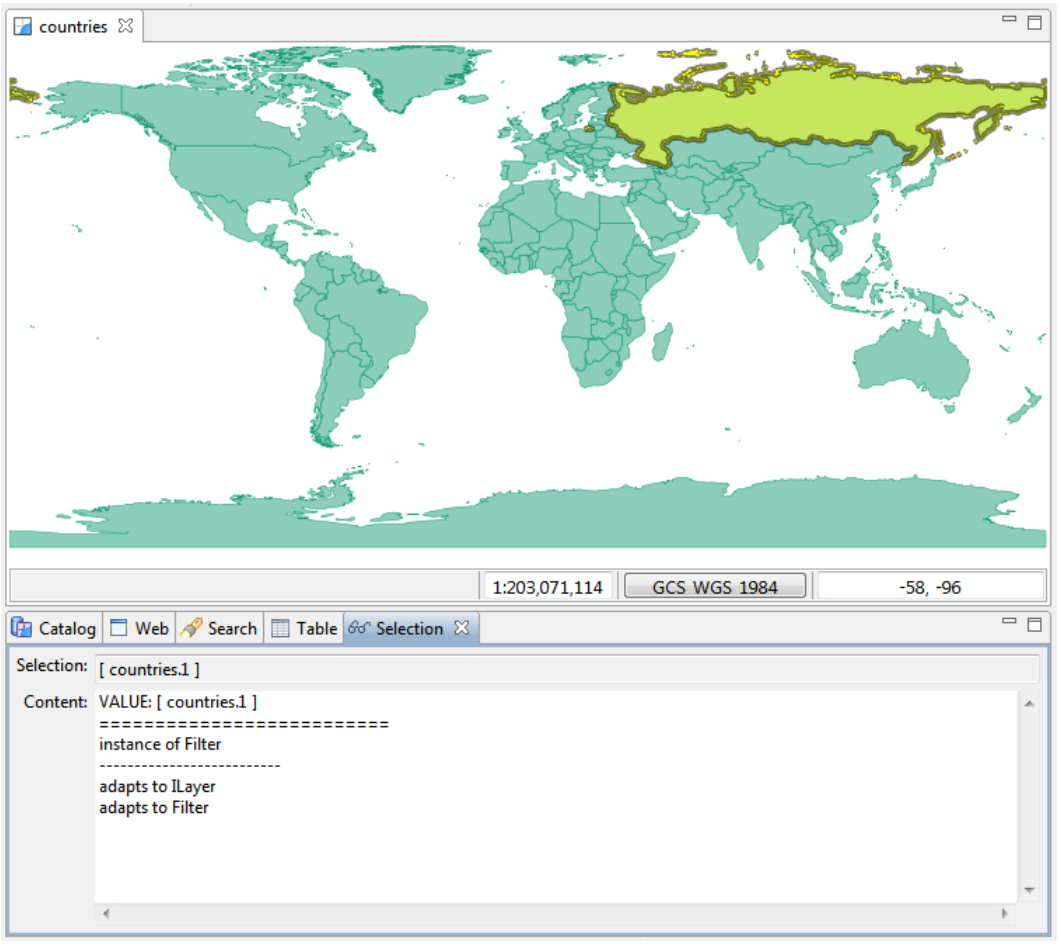


3. Expand the one folder labeled **Other** and click on **Selection** and press **OK**.
4. You should now see a view like the one below.



5. Try opening a Map, and adding several layers to it. Switch between the different Map Tools. As you try these activities watch the information displayed about the current selection.

6. The information displayed offers a breakdown of the item that you selected, any interfaces it implements, and a record on if it can be adapted into anything useful.



7. In the example above a country has been selected. A single selection is provided that is an instanceof “Filter” (a geotools class used to perform a query). This filter has the ability to adapt to a Layer is asked (presumably the layer containing the data).

   Our single selected Object capable of speaking two Java APIs; the Interfaces it was compiled with; and the Interfaces it can be adapted to.

5 What to do Next

Here are some additional challenges for you to try:

- You should have noticed that each View provides a unique selection. Did you also notice that the Map Editor will change what workbench selection it provides based on the current modal tool.

  Explore the available tools and note what content each tool thinks it is working on.

- Currently, if you want to see the workbench view you have to select **Window>Show View>Other** to open a the Show view dialog. You can then use the Show view dialog to navigate to **Other >Select View**.

  Can you use a “perspectiveExtension” make your view show up under the main view menu?

- Advanced: We have focused on listening to the workbench selection. Can you use getViewSite() to advertise an object to the workbench selection service? As a side effect, the Selection View you've just created will listen to itself

- Advanced: In this example we have checked “instance of” and “IAdaptable”. Can you extend this example to check IResolve?

  IResolve is uDig specific and represents external content. You should be very careful to read the javadocs and not call any methods from the event thread that may block while waiting for a WFS service on the other side of the work. If you make a mistake here it will look like the uDig application has “hung”.

  The uDig API very carefully throws IOExceptions when ever there is a chance of waiting for an external service. If you find yourself doing a try/catch block while in an event thread you have probably made a mistake!

Advanced: If you've done the IAdaptable workbook, you will note that your SelectionView tells you an IService is selected and gives you its URL. Similarly with an IGeoResource. However, it doesn't seem to be able to adapt them to URLs...go ahead and fix that.

Hint: you'll need to use the AdapterUtil class.