IBM Software Group, TPF Support and Services

TPF User Group Conference - April 2007

Extending the LPEX editor via a Plug-in

Anthony Lawrence

TPF Toolkit Task Force

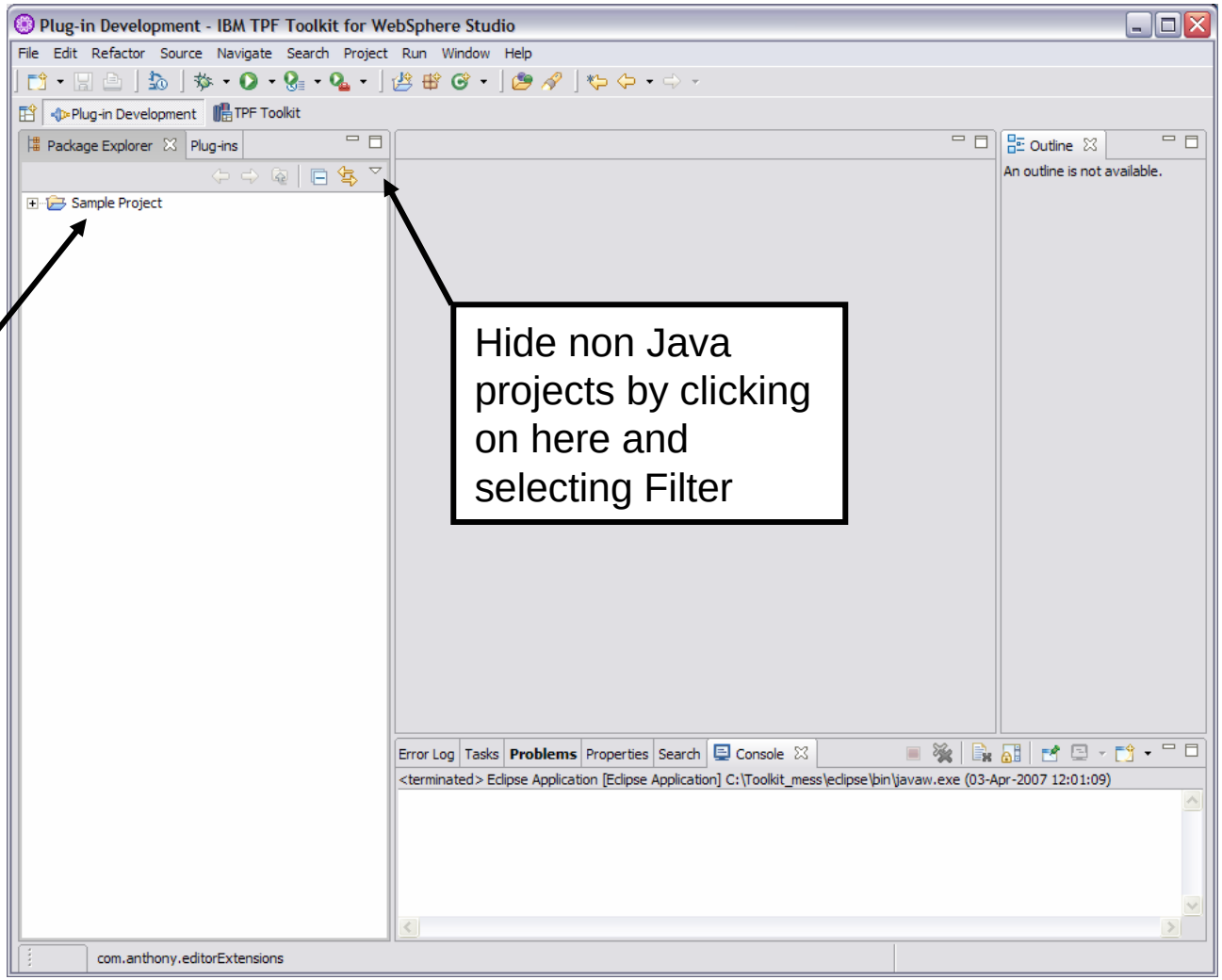
Objectives

- **How to create an eclipse Plug-in that will extend the capabilities of the LPEX editor**
- **This can be used to implement Enterprise supported actions**
- **Actions will be:**
  - Command line actions
  - Can be assigned to key sequences
  - Can appear on the popup menu

The Process

- Open the Plug-in Development perspective
- Create a new Project
- Create a new Package
- Create a new Class for each action
- Test the actions
- Package the plug-in for distribution

Plug-in Development Perspective

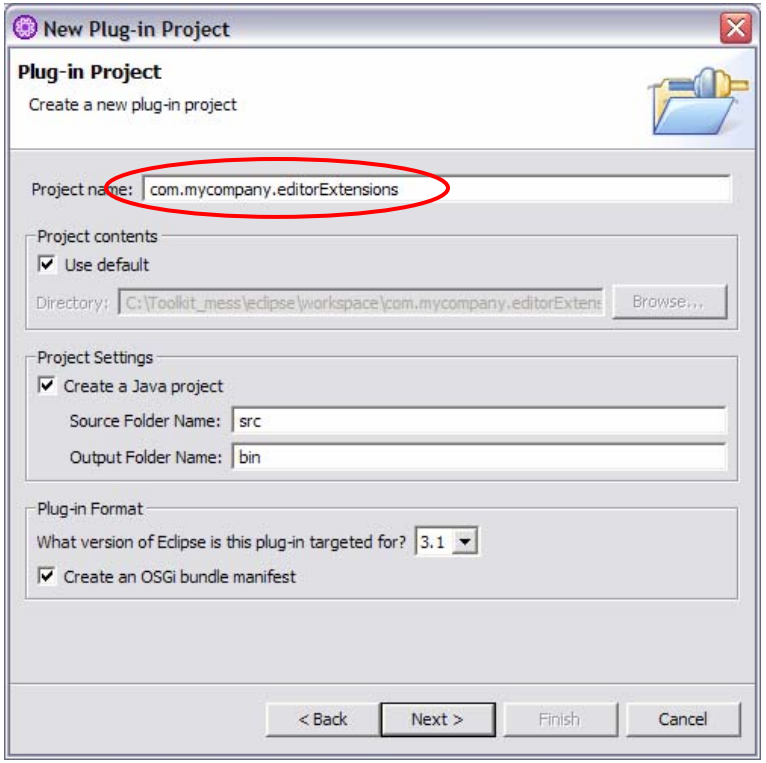


Notice: Any TPF projects you may have defined will also show up here, just ignore them or hide them

Hide non Java projects by clicking on here and selecting Filter

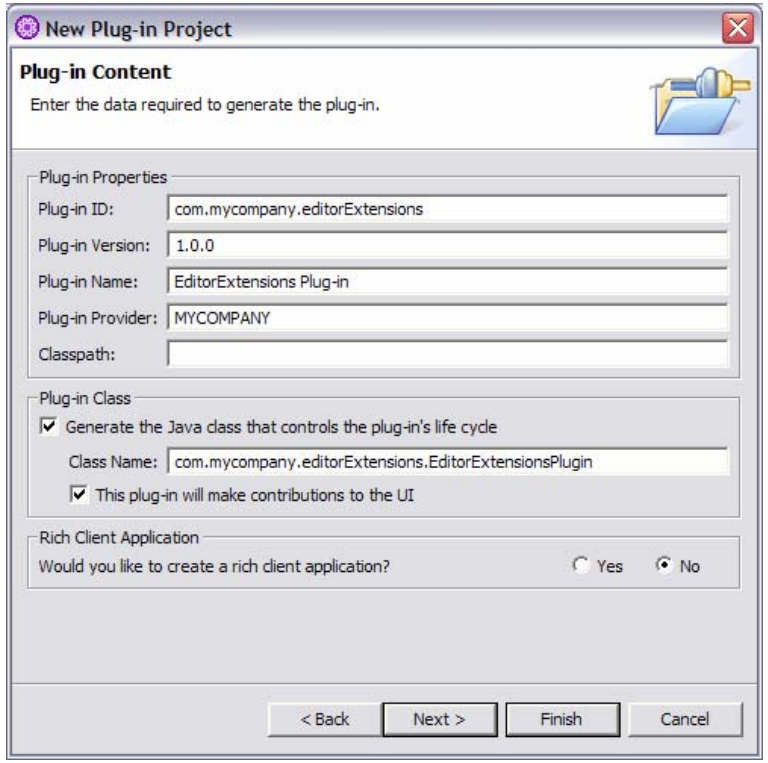
Create a new Plug-in project

- **Enter a project name**
  - This will become the name of your Plug-in
  - E.g. com.mycompany.editorExtensions



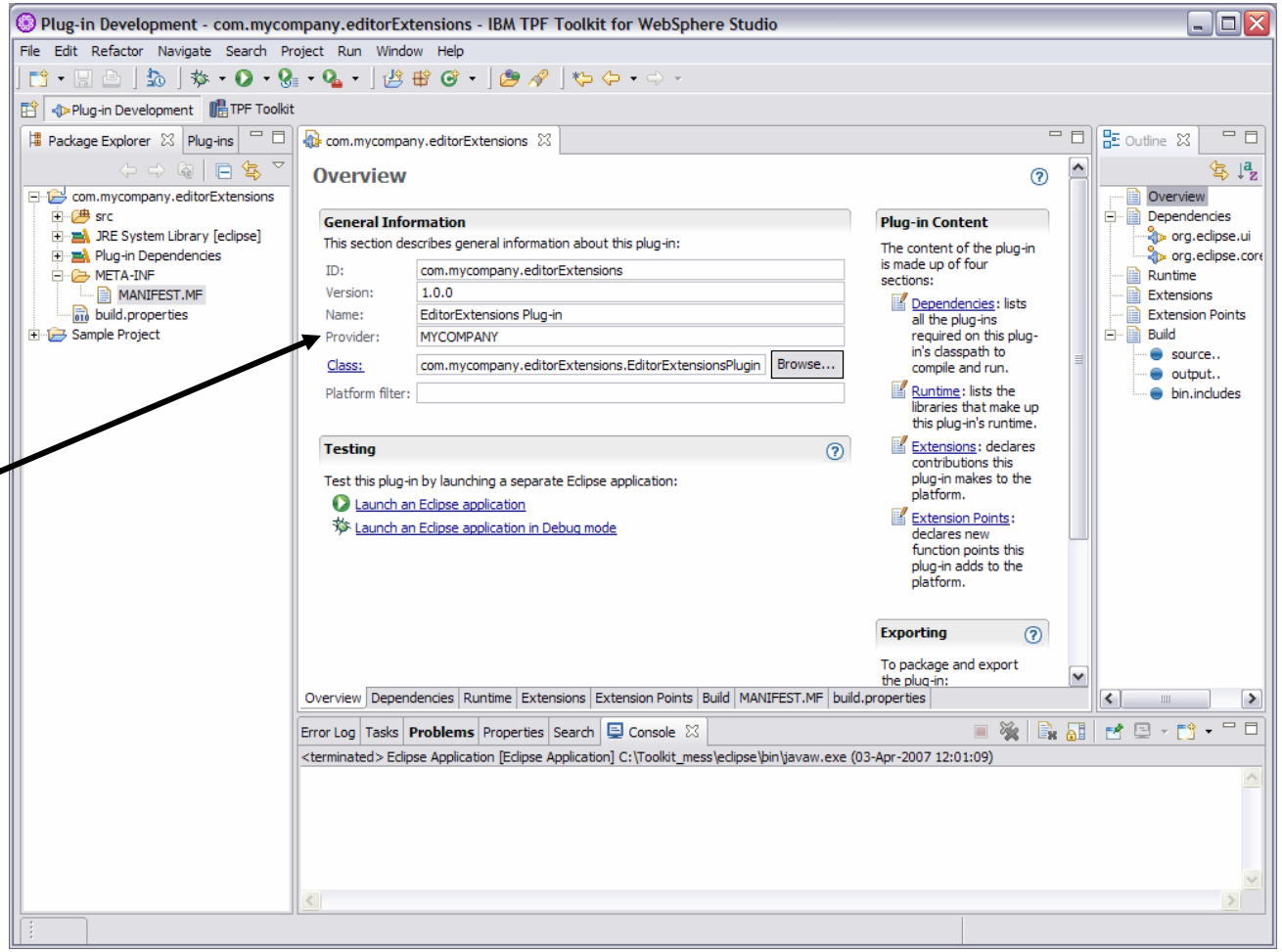
Create a new Plug-in project …

- **You can override the default Plug-in ID or version etc**
- **Make any changes you wish**



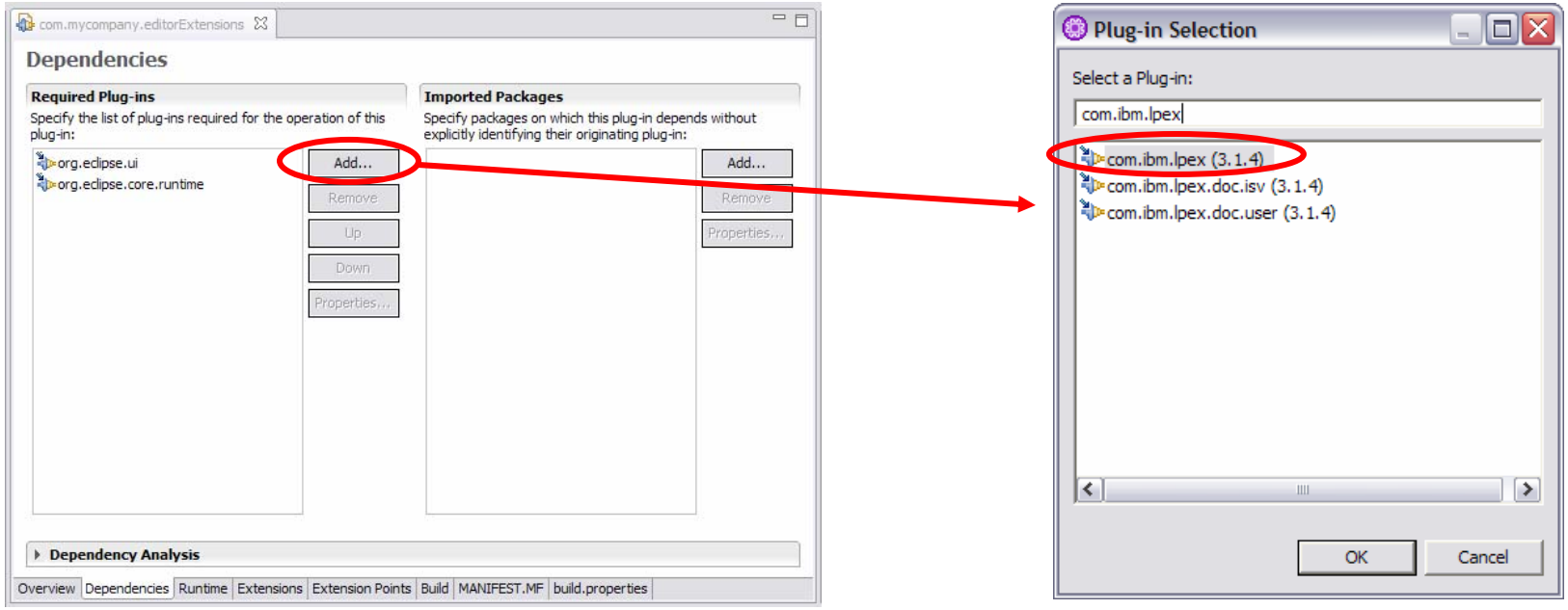
Create a new Plug-in project …

The Plug-in MANIFEST.MF file is opened in the editor when the project is created.



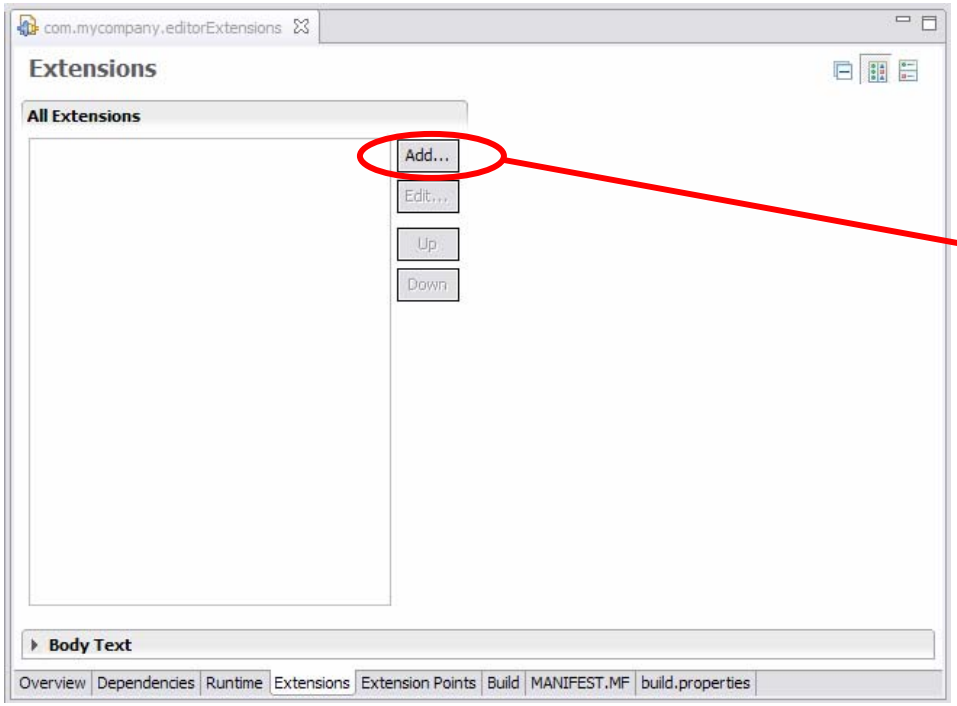
Add the Plug-in Dependencies

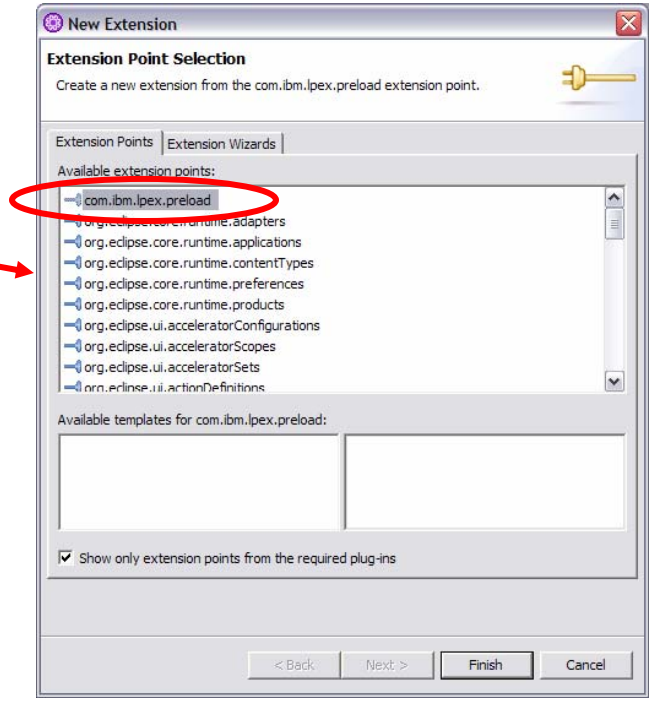
- **Need to add a dependency of the com.ibm.lpex plug-in**



Add the Plug-in Extensions

- **Need to add the extension point that we will be using com.ibm.lpex.preload**





Add the Plug-in Extensions Preload

- **Need to add the preload directive in the plugin description file**
  - Class is the full plug-in name
  - id can be anything

com.mycompany.editorExtensions

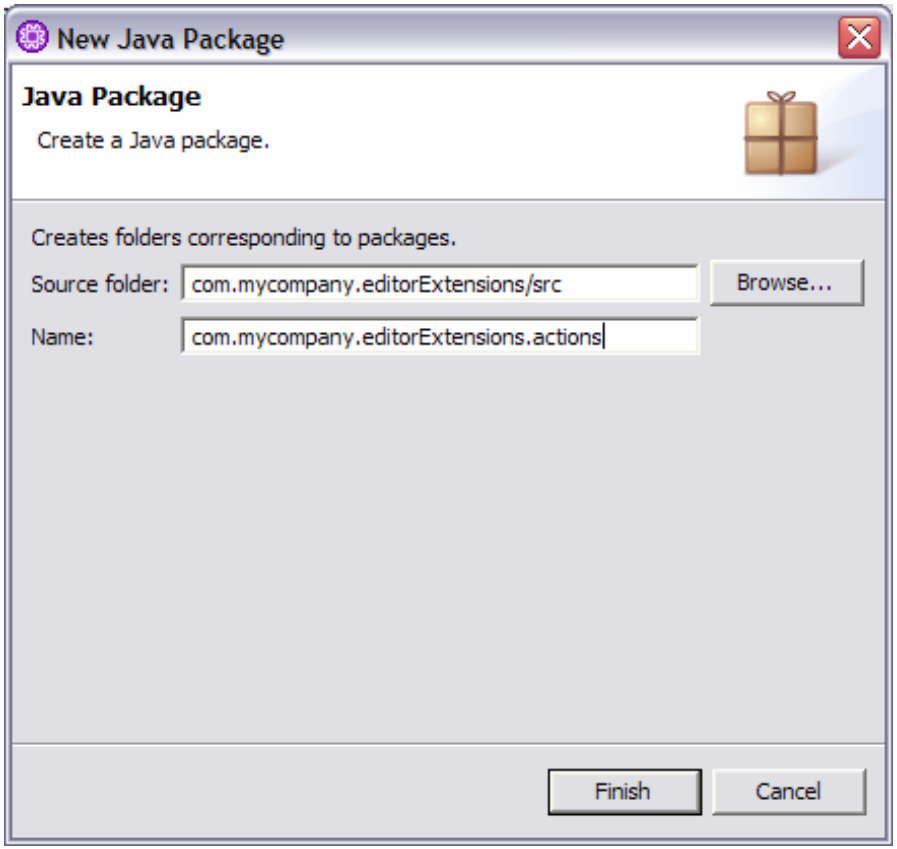
```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         point="com.ibm.lpex.preload">
      <preload
            class="com.mycompany.editorExtensions.EditorExtensionsPlugin"
            id="com.mycompany.lpexpreload"/>
   </extension>

</plugin>
```

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

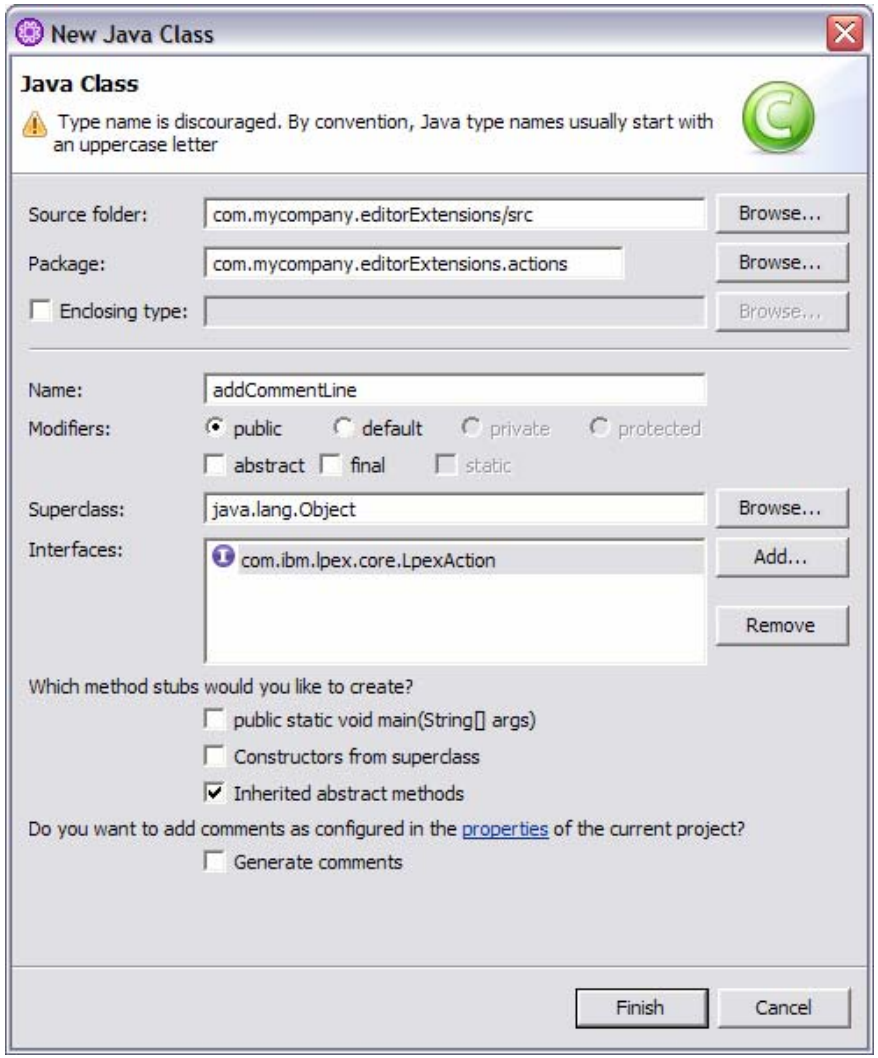
Create the Plug-in Package

- **Enter a Package name**
  - E.g. com.mycompany.editorExtensions.actions



Create the Plug-in Class (the action)

- **Enter a Class name**
  - E.g. addCommentLine
- **Add the LpexAction interface**



Code to add a comment line

Several commands have been added to the doAction section to:

- Insert a line
- Place to cursor at CC 1
- Place an asterisk at cursor location
- Then display a message

The action can run in any visible writable file

*addCommentLine.java

```
package com.mycompany.editorExtensions.actions;

import com.ibm.lpex.core.LpexAction;

public class addCommentLine implements LpexAction {
    public void doAction(LpexView view) {
        // TODO Auto-generated method stub
        // insert a new line and position the cursor at the beginning
        view.doAction(view.actionId("openLine"));
        // ensure cursor in in column 1
        view.doAction(view.actionId("home"));
        // insert asterisk at cursor location
        view.doDefaultCommand("insertText *");
        // tell the user what we have done
        view.doCommand("set messageText Comment line added");
    }

    public boolean available(LpexView view) {
        // TODO Auto-generated method stub
        // allow the action to run in any visible, writable document
        return view.currentElement() != 0 && !view.queryOn("readonly");
    }

}
```

Code the Plug-in implementation options

- **When the LPEX editor is loaded, your actions need to be made available**
- **Edit the skeleton Plug-in code**
  - In our example this is called EditorExtensionsPlugin.java

Code the Plug-in implementation options …

*EditorExtensionsPlugin.java

```
package com.mycompany.editorExtensions;

import org.eclipse.ui.plugin.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class EditorExtensionsPlugin extends AbstractUIPlugin implements LpexPreload {

    //The shared instance.
    private static EditorExtensionsPlugin plugin;

    /**
     * The constructor.
     */
    public EditorExtensionsPlugin() {
        plugin = this;
    }

    /**
     * This method is called upon plug-in activation
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
    }
```

Outline

- com.mycompany.editorEx
- import declarations
- EditorExtensionsPlugin
  - plugin : EditorExtens
  - EditorExtensionsPlug
  - start(BundleContext
  - stop(BundleContext)
  - getDefault()
  - getImageDescriptor(

- **The sample skeleton needs to be modified to tell eclipse what you are implementing**
  - Add the highlighted code
    - This will cause an error that can be resolved by clicking on the icon in the left margin and double clicking on the "Add unimplemented methods" quick fix

The added method

EditorExtensionsPlugin.java

```
    /**
     * Returns the shared instance.
     */
    public static EditorExtensionsPlugin getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path.
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return AbstractUIPlugin.imageDescriptorFromPlugin("com.mycompany.editorExtensi
    }

    public void preload() {
        // TODO Auto-generated method stub

    }
}
```

Outline

- com.mycompany.editorEx
- import declarations
- EditorExtensionsPlugin
  - plugin : EditorExtens
  - EditorExtensionsPlug
  - start(BundleContext
  - stop(BundleContext)
  - getDefault()
  - getImageDescriptor(
  - preload()

- **The highlighted code is added when you implement the method**
- **You can now code whatever you want done to implement your action(s)**
  - Suggest you add the action to
    - UserActions, UserKeyActions, and/or the Popup menu

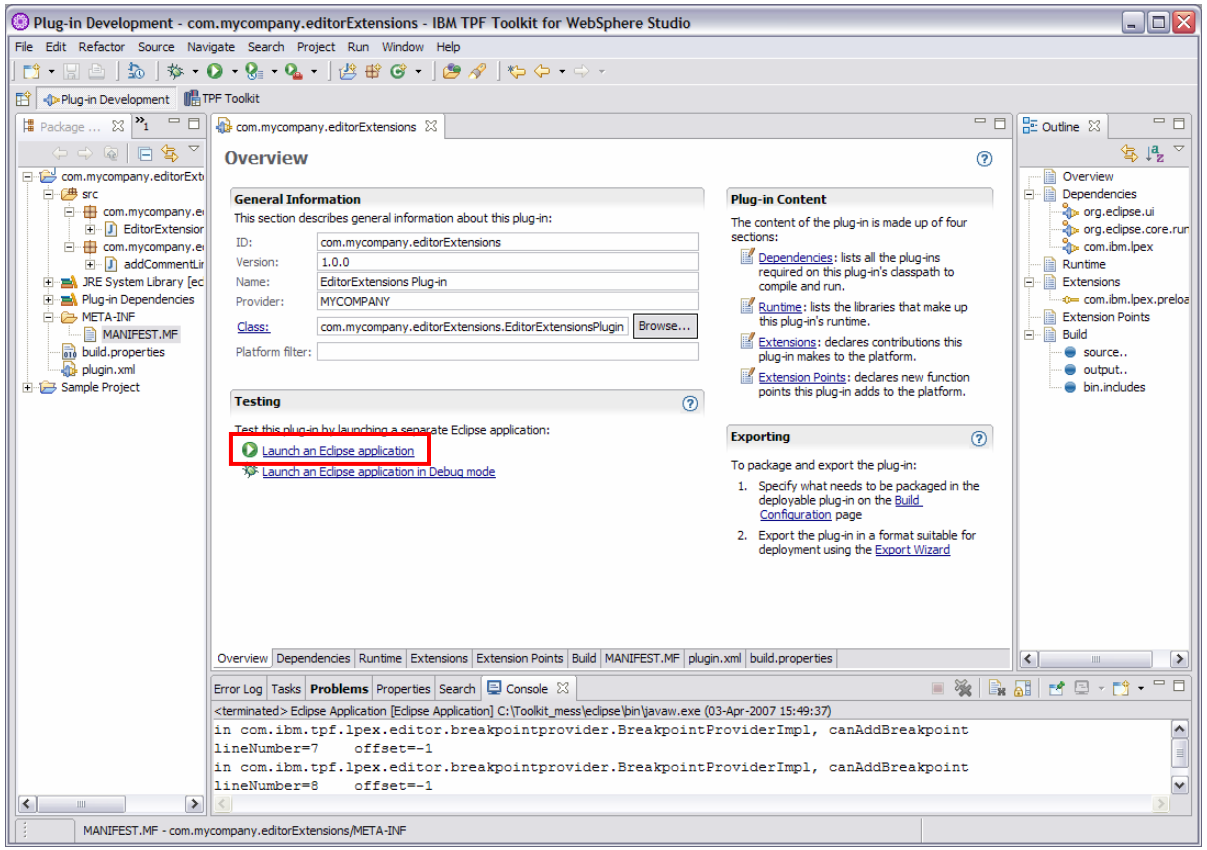
Sample implementation method

```
public void preload() {

    String newAction = "addCommentLine com.mycompany.editorExtensions.actions.addCommentLine";
    // Determine what actions have already been defined.
    String existingActions = LpexView.globalQuery("current.updateProfile.userActions");
    if (existingActions == null)
       {
           // If no actions have already been defined, set ours as the first user defined action
           LpexView.doGlobalCommand("set default.updateProfile.userActions " + newAction);
       }
       else if (existingActions.indexOf(newAction) < 0)
       {
        // If other actions have already been defined, and ours is not already in the list, add ours to the existing list
           LpexView.doGlobalCommand("set default.updateProfile.userActions " + newAction + " " + existingActions);
       }

    ...
    ...
    ...
    ...
    ...
```

Testing your Plug-in



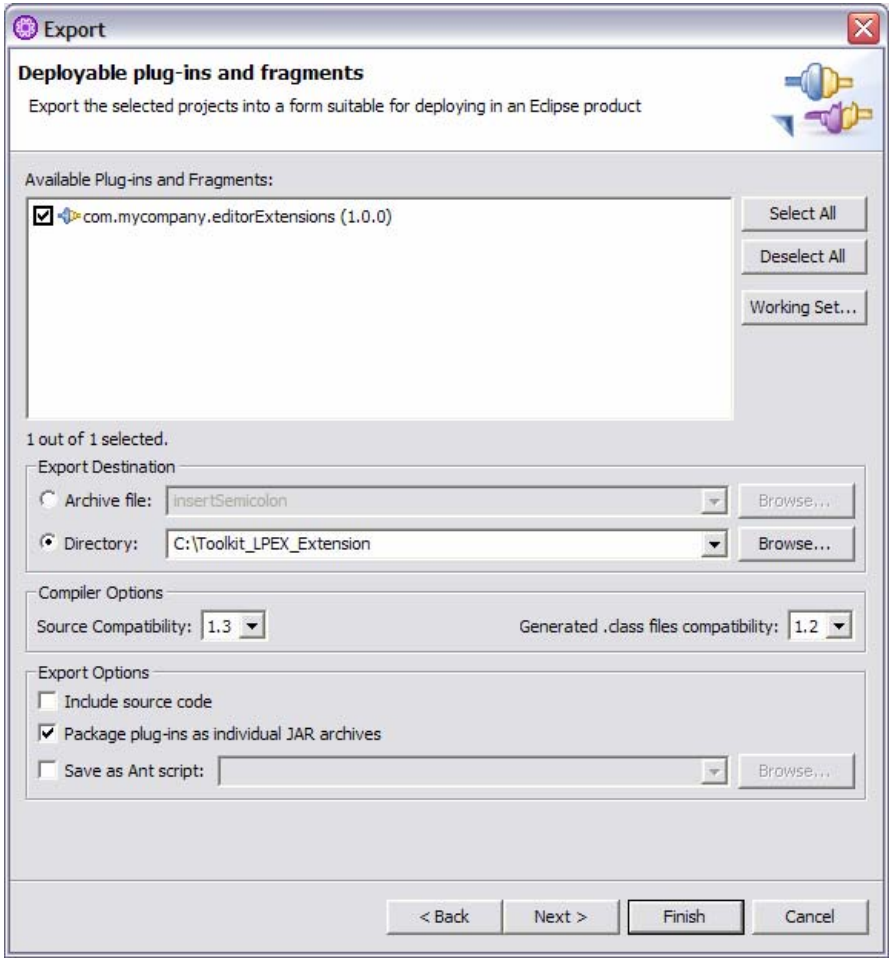
- Save all the files that have been changed
- Open the MANIFEST.MF file using the manifest editor
- Display the Overview page
- Click on the “Launch an Eclipse Application” link

Testing your Plug-in …

- **Open the TPF Toolkit perspective**
- **Define a project and filter (local files are all that is needed)**
- **Open a file in the TPF Toolkit LPEX Editor**
- **Test your actions**
  - If they fail, close the instance, open your source and set breakpoints as required then restart the testing instance using the “Launch an Eclipse Application in debug mode” link
  - When your breakpoint is encountered, the debug perspective is opened

Packaging your Plug-in for distribution …

- **Right click on your Plug-in project**
  - ➔ **Export …**
  - ➔ **Deployable Plugins and fragments**
- **Ensure your Plug-in is selected, if there are more available, deselect them as necessary**
- **Select a directory to store the plug-in**
- **A *plugin_name*.jar file will be created in the selected directory.**
- **Copy this file to the %TPFHOME%\eclipse\plugins directory**
- **Start that instance of the TPF Toolkit and test your plugin actions**



Adding more actions

- Right click on the Package created earlier and create a new Class
- Proceed as described earlier to implement your actions
- Update the Implementation options to define your actions
- (Optional) Edit the MANIFEST.MF and change the version number of your plug-in
- Test you changes
- Export the new version of your plug-in

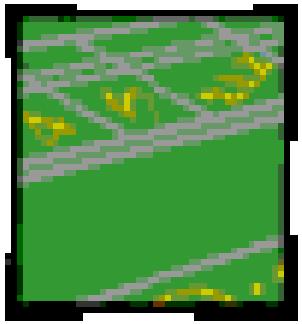
Distributing the new Plug-in

- **Copy the .jar file to the %TPFHOME%\eclipse\plugins directory of the instance you use to create your update site**
- **Modify the feature.xml file for the com.ent.customized.toolkit feature to include the new plug-in**

OR

- **Copy the .jar file to the deploy directory of your com.ent.customized.toolkit feature**
- **Update the ENT_workstation_copy_list.txt file to copy the jar file to the %TPFHOME%\eclipse\plugins directory**

Finally



How many of you have tried your luck with the chips handed out when you registered?

Did you win?

Thank You