A Quick-Start Tutorial to Eclipse Plug-in Development

Anleitung zur wissenschaftlichen Arbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Balsiger

Dezember 2010

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz

Dr. David Röthlisberger

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Markus Balsiger
markus.balsiger@students.unibe.ch
http://scg.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

This document is a quick-start guide to developers which are about to create a plug-in for the Eclipse IDE. We discuss the basic project setup of the plug-in as well as the so-called extension points of Eclipse. As a quick tutorial, we create a sample plug-in with a few simple features on the fly.

Contents

List of Figures

Chapter 1

Introduction

This document is an introduction to Eclipse plug-in development. It is based on Eclipse Classic 3.6[1]. We first introduce the basic project creation and the plug-in project structure. After this short introduction, we take a closer look at the most common extension points that Eclipse offers to plug-in developers. Among other extensions, we present the user interface integration and a run configuration.

Eclipse's plug-in system is based on Equinox, which is an OSGi framework implementation. Most infrastructure and service modules of Eclipse are therefore available as bundles and can be reused or integrated. Besides the fact that modules can be installed and maintained automatically including automatic download and update of bundles on which a module depends, Equinox allows the IDE to be updated without a restart.

Often plug-ins are installed using so-called update sites. An update site typically consists of a site.xml which contains the basic information about the plug-in like its category, supported operating systems or the feature name, the artifacts.xml offering information about every OSGi bundle required by a feature, the content.xml containing more information like the license text and the jar files required by the plug-in itself. Luckily Eclipse maintains many of these informations and jar files automatically when we set up the correct plug-in structure.

[1] http://download.eclipse.org/eclipse/downloads/

Chapter 2

Preparations and Requirements

This tutorial should be feasible in under 90 minutes. First, download [1] Eclipse 3.6 Classic. Also make sure that you have a Java SDK available which is up-to-date [2].

[1] http://download.eclipse.org/eclipse/downloads/
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Chapter 3

Plug-in Project Creation

For our sample plug-in we first create a new plug-in project (Figure 3.1). Right click in the Package Explorer and select the project type as shown in Figure 3.2. Give it a name and leave the rest of the settings as proposed by Eclipse.

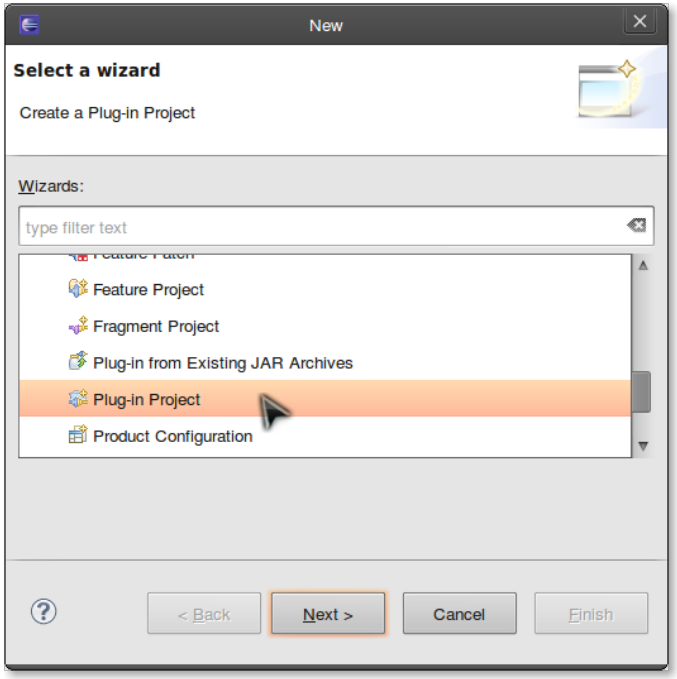

Figure 3.1: Select the plug-in project type

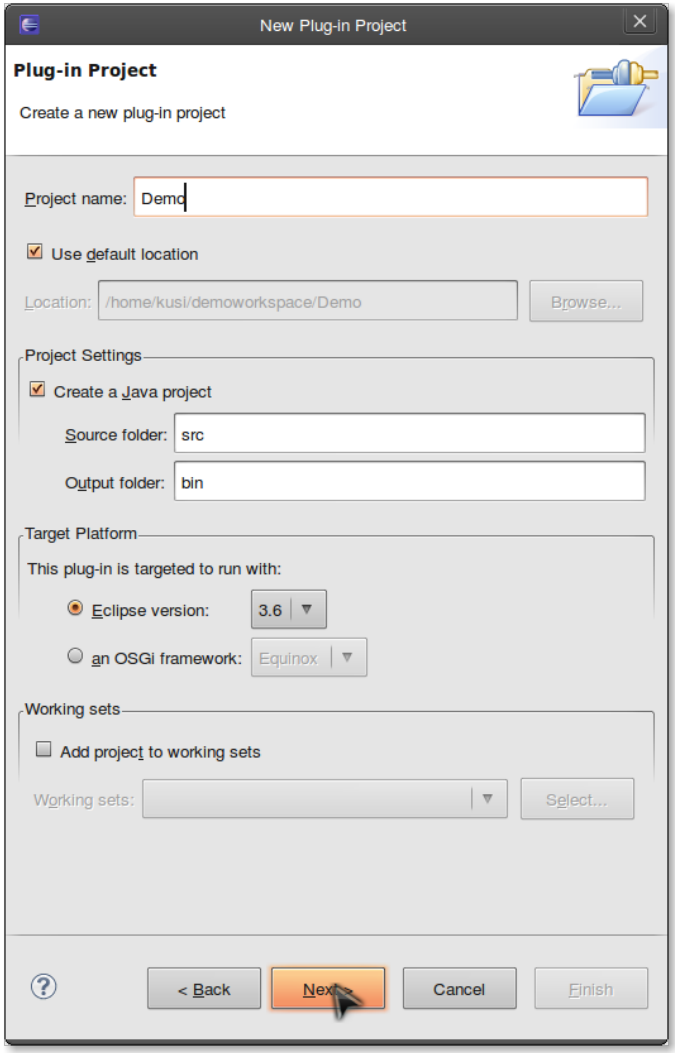

Figure 3.2: Settings for the new project

The next steps are the content settings of the plug-in. Select whatever you like to, for our purposes the provided standard properties are alright. If you like to create a complete application with your plug-in embedded inside Eclipse by default, select the „Rich Client Application" radio box. However in this tutorial, we will simply create the plug-in.

On the next screen select "Plug-in with a view". The wizard will provide a lot of completed work for our demo plug-in, for example the registration of our view. Select Next and give the

new view some properties like the name of the package in which the view should be generated into, the name of the view and so on. Again the default values are alright. Hit finish. Eclipse will ask you if you would like to change the perspective to to Plug-In Development perspective. Accept. You should now see something like the Eclipse window visible in Figure 3.3.

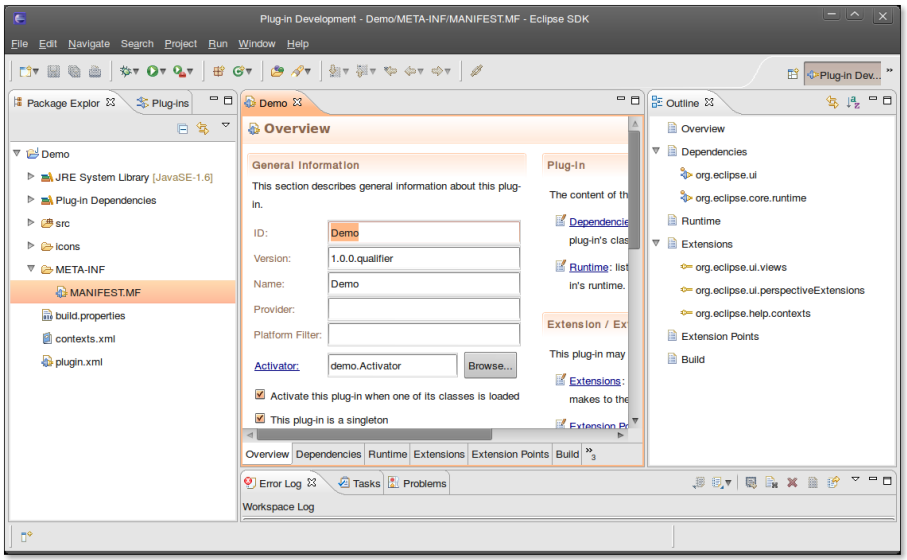

Figure 3.3: The project overview

This is it with the project creation. We now take a look at what Eclipse created for us. To start the plug-in inside a test-Eclipse environment, right-click the Project inside the package explorer and select ”Run as” - ”Eclipse Application”. A new instance of Eclipse should start. We call this instance of Eclipse, in which we test our plugin, the test-Eclipse in this paper. When the test-Eclipse has started, search for our just created view. We will not find it, because the Java perspective is selected, and the plugin’s view is not part of this perspective yet. Click on the fast-view button in the bottom-left corner (red arrow in Figure 3.4) and select ”others”. Search for the ”Sample View” inside the ”Sample Category”. Figures 3.4 and 3.5 should make the view selection clear. In Figure 3.4, we have already opened the just created view and attached it to the perspective.

As we have seen, the Eclipse IDE we just started used another workbench than our development Eclipse. By default, this workbench is located in the user home directory and is called *runtime-EclipseApplication*. This folder is automatically created when we first start our plugin as an ”Eclipse Application”. So whenever we need a fresh workspace, we delete this folder or change the workspace and eclipse environment settings in our projects run configuration under ”Eclipse Application”. We now discuss the responsibilities of the project-files Eclipse created.

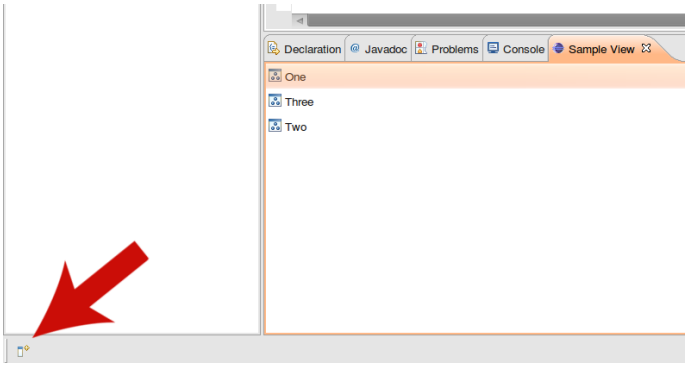

Figure 3.4: Adding our view to the Java perspective

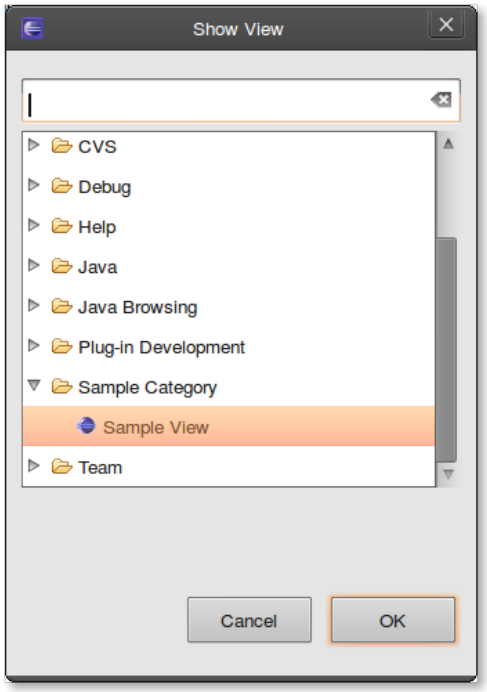

Figure 3.5: Selecting our registered view

Activator.java . The Activator class controls the plug-in's life cycle and provides a set of workbench sensitive functionalities. One of the first things you might need as a plug-in developer is the access to the preference store, Eclipse's built in property database, which is used to hold persistent settings for a plug-in.

SampleView.java . This is our generated view. As we see in Chapter 4.2 ”User Interfaces”, it is already correctly registered as an extension of Eclipse.

contexts.xml . In the contexts.xml, among other textual information, help contents of a plugin are provided.

plugin.xml . This file is the heart of our plugin. In this XML, we define dependencies to other plug-ins, bundles or packages, we register extensions and define extension points of the plugin.

Chapter 4

The Most Elementary Extension Points and Services

First things first. What is an extension point and what is a service?

Extension Points and Extensions. Eclipse provides a concept to enable plug-ins to very simply contribute functionality to other plug-ins with extension points and extensions. An extension point defines a way to contribute functionality to a plugin, while the definition of an extension inside a plugin contributes functionality. Eclipse itself already ships with a number of extension points, for example the *org.eclipse.ui.views* extension point, allowing us to add new views to the IDE.

Services. The term services is in fact not Eclipse terminiology, but used in this paper to describe the interfaces and functions Eclipse offers besides the OSGi implementation. Some of the services are called services, for example the selection listener service which we use in this paper, while others are not.

Almost every setting of our plugin.xml can be managed with Eclipse's built in plugin.xml-editor. We will now discuss the tabs or categories available in this editor.

Overview. In this tab, general information about the plugin is provided by the developer. You will see that the name of our plugin is stored here and also the Activator is registered in here. One of the most interesting settings listed is the execution attribute which enables the developer to specify the minimal execution environment for a plugin.

Dependencies. The dependencies settings define the bundles and packages a plugin depends on. The difference between the requirement of a plug-in and a package is, that the plug-in requirement defines a complete plugin including packages needed to run a plugin, while the imported packages only defines packages, leaving the Eclipse environment open from which plug-in the package will be imported.

Runtime. In this configuration tab, packages can be exposed for other plug-ins. Exposed packages can be used by other plug-ins if they, for example, define them as a package dependency.

Extensions. Inside the extensions tab, implementations that add functionalities to Eclipse or other plug-ins are defined. These extensions contribute functionalities to extension points defined by other plug-ins.

Extension Points. In the extension points tab, new extension points can be defined. New extension points can then be used by other plug-ins to add functionalities to our plug-in or to use information of our plug-in.

Build. In here we describe the build order of libraries and define the source and binary folders. A developer might already know this from the project settings of a plain Java Project.

MANIFEST.MF. In this tab we are able to view and manipulate the manifest of our plugin. Furthermore a few OSGi hacks are possible in here.

plugin.xml. The plugin.xml is the plain XML editor of the plugin.xml. Sometimes it is more comfortable to copy and paste attributes of a plugin.xml rather clicking through the editor.

build.properties. The build.properties represent the settings made in the build tab.

4.1 Our Tutorial Project

We will now start developing our Eclipse plugin. The plugin will enable us to start any application with arguments configured in the plugin view. If we have a console application which we want to test with different arguments, we can simply type in these arguments in our plugin view instead of inside the launch configuration, which takes many clicks. And to make it even better, we provide a selection listener which will enable us to use selected text from plain text files as runtime arguments. That way, we can create some kind of arguments database in a text file and test these arguments again and again without having to type a single word. The plugin every Java developer was waiting for!

4.2 User Interfaces

To introduce the user interface extensions, we take a look at the sample view that was created by the plug-in development wizard. Investigating the extensions tab in the plugin.xml editor, we see that an extension was created for *org.eclipse.ui.views*. Figure 4.1 presents the open

extension element of our sample view. The important settings are the name, the category and most of all the class.

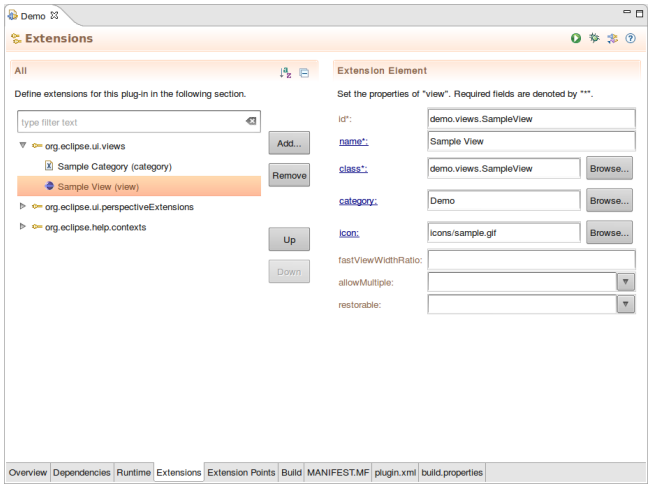

Figure 4.1: The registered view extension element of our demo project

Eclipse uses SWT for user interfaces. If you are new to SWT, there are many very nice tutorials in the Internet. If you are familiar with AWT and Swing, it should not take you too long to get used to SWT. Check out the project's website[1] which really helped us to get started, especially the widgets overview page[2].

If our class extends *org.eclipse.ui.part.ViewPart*, as the SampleView already does, we can register the extension in the plugin.xml. Our sample view is already registered, but let us take a look at how this is done:

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="Sample Category"
         id="Demo">
   </category>
   <view
         name="Sample View"
         icon="icons/sample.gif"
         category="Demo"
         class="demo.views.SampleView"
         id="demo.views.SampleView">
   </view>
</extension>
```

The important entries here are the extension points to which we deliver an extension, and the class that delivers the actual implementation. Other than that, a category, which will be visible

[1] http://www.eclipse.org/swt/
[2] http://www.eclipse.org/swt/widgets/

e.g. in the ”Show View” dialog of Eclipse, or a name can be set in here. Note that we can add as many extensions to this point as we want to. We are not limited to a single view.

To create our own view, we first remove most of the implementation that was generated by Eclipse. Actually, we remove the entire implementation inside the class body. Eclipse will now tell us that there are some methods that have to be implemented. Select the SampleView class name in the Java file and press ctrl+1 on your keyboard. A context menu showing quick-fixes will open. Select ”Add unimplemented methods” et voilà, here are our dummy methods: *createPartControl* and *setFocus*. As good developers, let's tidy up our class imports by pressing ctrl+shift+o. Done, no warnings. Beautiful. Now let us do some simple GUI work.

The following listing shows the code defining an SWT label with the text ”Program arguments:” and a multi-line text-box beneath.

```
private Text argumentsField;
private static String arguments;

@Override
public void createPartControl(Composite parent) {
        GridLayout gridLayout = new GridLayout(1, false);

        parent.setLayout(gridLayout);

        GridData gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.verticalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        gridData.grabExcessVerticalSpace = true;

        Label label = new Label(parent, SWT.NULL);
        label.setText("Program arguments: ");


        argumentsField = new Text(parent, SWT.MULTI | SWT.BORDER | SWT.V_SCROLL);
        argumentsField.setLayoutData(gridData);
        argumentsField.setText("");
        argumentsField.addListener(SWT.CHANGED, new Listener() {
                @Override
                public void handleEvent(Event event) {
                        arguments = argumentsField.getText();
                }
        });

        parent.pack();
}
```

The code produces the GUI shown in Figure 4.2.

Now we need accessors for the text inserted in the view. We will make it dirty, as this is nothing particularly interesting when focusing on plugin development. We simply create static accessors to the view's text inside the *argumentsField*. The arguments String is used to reduce the thread access on SWT's dispatch thread. Using the change listener we can simply update the String and then return it in the *getArguments()* method instead of invoking *asyncExec()* and wait for the result. We could do it without the *asyncExec()* method for sure, but actually this would be illegal thread access.

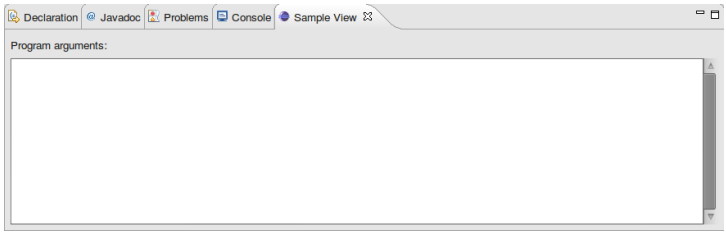

Figure 4.2: Our first GUI

```
private static SampleView view;
private static String arguments;

public SampleView(){
        view = this;
}

// ...

public static void setArguments(final String arguments) {
        Display.getDefault().asyncExec(new Runnable(){
                @Override
                public void run() {
                        view.argumentsField.setText(arguments);
                }
        });
        SampleView.arguments = arguments;
}

public static String getArguments() {
        return arguments;
}
```

4.3 Launch Configurations

A launch configuration implements a specific launching mechanism. Examples of launch configurations are the *Java Application* run configuration or the *JUnit* configuration, which we have all used already. Other examples can be found in the run configurations window of Eclipse.

To implement our own run configuration, we need the extension point provided by Eclipse that allows us to define a new run configuration. First we create a class called ”Launcher” inside the *demo.launcher* package and let it extend the *JavaLaunchDelegate*. That way we can reuse a lot of Eclipse’s standard Java launcher. Now we add the extension inside the plugin.xml. We can use the graphical editor in Eclipse’s plugin.xml editor or add the following lines manually to the DOM of the XML file:

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
     <launchConfigurationType
           delegate="demo.launcher.Launcher"
```

```
            delegateDescription="Launches the application with the arguments provided by
                the SampleView of the Demo project"
            id="Demo.launchConfigurationType1"
            modes="run"
            name="DemoLauncher"
            public="true"
            sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.
                JavaSourceLookupDirector"
            sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.
                javaSourcePathComputer">
        </launchConfigurationType>
    </extension>
```

The delegate we set as the actual delegate class must implement *ILaunchConfigurationDelegate*, which the *JavaLaunchDelegate* does. The *JavaLaunchDelegate* class is inside the *org.eclipse.jdt.launching* package. Eclipse will not be able to import the class because the project is not yet set up properly. To do so, we can try the quick fix called ”Fix project setup...”. Unfortunately, in most cases, Eclipse is unable to determine the bundle that has to be included in our plugin. This is why we do it by hand in this tutorial.

Adding a new project dependency to the project. Adding a new dependency for a class is very simple when we know in which bundle the desired class is located. Often the bundle name can be derived form the package name. In this case, it is *org.eclipse.jdt.launching*. In the plugin editor’s dependencies tab, we add the *org.eclipse.jdt.launching* plugin as shown in Figure 4.3. Also we have to make sure that the *org.eclipse.core.runtime* plugin and the *org.eclipse.core.resources* are listed in our dependencies. These two are required dependencies of the *ILaunchConfigurationDelegate* and the *CoreException* which could be thrown by the JavaLaunchDelegate’s *getProgramArguments* method.

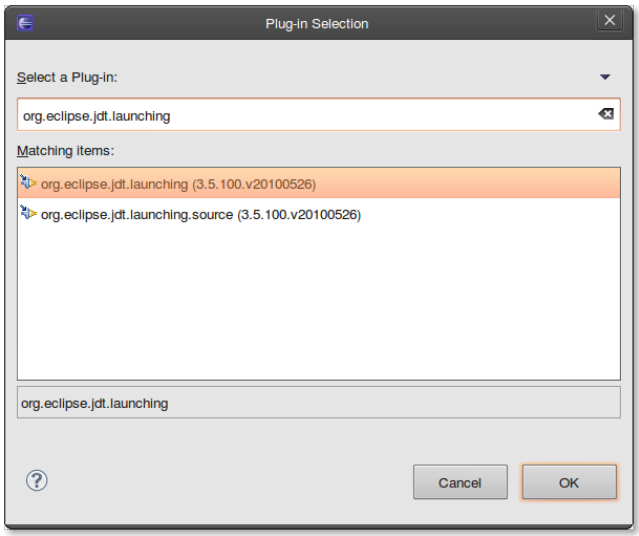

Figure 4.3: Adding *org.eclipse.jdt.launching* as required plugin of our project

After this short digression we would like to start implementing our launch configuration. As we will start the application almost like the standard Java launch config, we first take a look the source code of the original launch configuration provided by the JavaLaunchDelegate [3]. The important part for our project is the following snippet inside the launching method:

```
String pgmArgs = getProgramArguments(configuration);
```

If we override this *getProgramArguments(ILaunchConfiguration)* method, we are able to set the arguments from inside our view. We first implement the following overriding method:

```
public String getProgramArguments(ILaunchConfiguration configuration) throws
    CoreException{
        return super.getProgramArguments(configuration);
}
```

Now we add the arguments from our user interface. To make sure that general arguments can be passed by the run configuration, we will simply concatenate them with the string provided by the original *getProgramArguments(ILaunchConfiguration)* method.

As every launch configuration needs some configuration, the next step is to create a LaunchConfigurationTabGroup. See the next chapter to learn about the GUI setup of a launch configuration.

4.4 Run Configuration User Interfaces

Every launch configuration has a user interface defining properties of a specific run. Also these properties can be automatically stored inside the workbench. We can reuse already available tabs or whole groups of tabs if we like. We first create a class called *LauncherUI* inside the *demo.views* package. In this class, we add the following lines of code which we will discuss after the listing:

```
public class LauncherUI implements ILaunchConfigurationTabGroup{
        private ILaunchConfigurationTab[] tabs = new ILaunchConfigurationTab[3];

        public LauncherUI(){
                tabs[0] = new org.eclipse.jdt.debug.ui.launchConfigurations.JavaMainTab()
                    ;
                tabs[1] = new org.eclipse.jdt.debug.ui.launchConfigurations.JavaJRETab();
                tabs[2] = new org.eclipse.debug.ui.CommonTab();
        }

        @Override
        public void createTabs(ILaunchConfigurationDialog dialog, String mode) {
                dialog.setActiveTab(tabs[0]);
        }

        @Override
        public void dispose() {

        }
```

[3] http://www.java2s.com/Open-Source/Java-Document/IDE-Eclipse/jdt/org/eclipse/jdt/launching/JavaLaunchDelegate.java.htm

```
        @Override
        public ILaunchConfigurationTab[] getTabs() {
                return tabs;
        }

        @Override
        public void initializeFrom(ILaunchConfiguration configuration) {
                tabs[0].initializeFrom(configuration);
                tabs[1].initializeFrom(configuration);
                tabs[2].initializeFrom(configuration);
        }

        @Override
        public void launched(ILaunch launch) {

        }

        @Override
        public void performApply(ILaunchConfigurationWorkingCopy configuration) {
                tabs[0].performApply(configuration);
                tabs[1].performApply(configuration);
                tabs[2].performApply(configuration);
        }

        @Override
        public void setDefaults(ILaunchConfigurationWorkingCopy configuration) {
                tabs[0].setDefaults(configuration);
                tabs[1].setDefaults(configuration);
                tabs[2].setDefaults(configuration);
        }
}
```

The interfaces *ILaunchConfigurationTabGroup* and *ILaunchConfigurationTab* are located in the plugin called *org.eclipse.debug.ui*. We add it to our project's dependencies to solve the unresolved dependencies. Specificly for our implementation, we also have to make sure the *org.eclipse.jface* plugin is in your dependency list. Otherwise our plugin will not know where to take the ILaunchConfigurationDialog interface from. Other than those two, we have to add the *org.eclipse.jdt.debug.ui* package for the tabs of the JDt launcher we are reusing. These tabs are the following three:

- org.eclipse.jdt.debug.ui.launchConfigurations.JavaMainTab
- org.eclipse.jdt.debug.ui.launchConfigurations.JavaJRETab
- org.eclipse.debug.ui.CommonTab

If we wanted to add a custom tab, it must implement *ILaunchConfigurationTab* which allows to put configuration attributes on the launch configuration and so on. It is very simple once we know how the rest of Eclipse works. Also we do not have to register such a tab in the plugin.xml. We just have to instantiate it in our tab group class.

To make our *LaunchConfigurationTabGroup* usable, we have to register it in the plugin.xml. Read the following listing on how our extension to Eclipse's *org.eclipse.debug.ui.launchConfigurationTabGroups* extension point looks like:

```
<extension point="org.eclipse.debug.ui.launchConfigurationTabGroups">
  <launchConfigurationTabGroup
```

```
            class="demo.views.LauncherUI"
            description="This is the configuration tab group of the DemoLauncher"
            id="Demo.launchConfigurationTabGroup1"
            type="Demo.launchConfigurationType1">
      </launchConfigurationTabGroup>
   </extension>
```

Make sure that the type is the id of our launch configuration! This is important, as otherwise Eclipse will not be able to connect these two extensions. When we start the plugin inside the test-Eclipse and create a new *DemoLauncher* run configuration, it should look like the one on Figure 4.4 now:

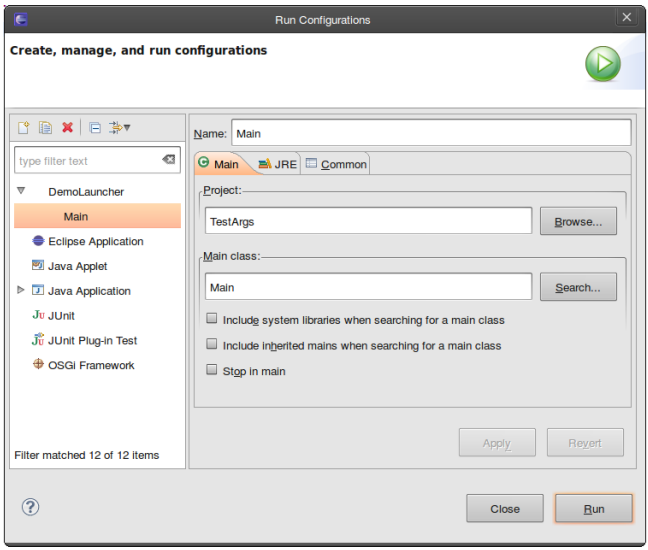

Figure 4.4: Our own launch configuration

We could have just taken the tab group of the Java launcher. But we wanted to make a quick introduction to the launch configuration tabs, and how to quickly build them up using already available tabs.

4.5 Eclipse-Wide Selection Listener

To make your plugin more useful and interactive, it might be important that the plugin is informed about what the developer selects inside the IDE other than our plugin. For example, Eclipse's Package Explorer supports linking the currently selected item in the resource tree with the currently selected editor window. To use it, simply press the toggle button with the 2-arrows-icon. This feauture is most probably implemented using the selection service of Eclipse. The selection service consists of the selection provider interface, the selection listener interface and the service classes which will inform selection listeners about the selections registered

by the selections providers. Very simple but powerful. Figure 4.6 shows the service in a diagram.

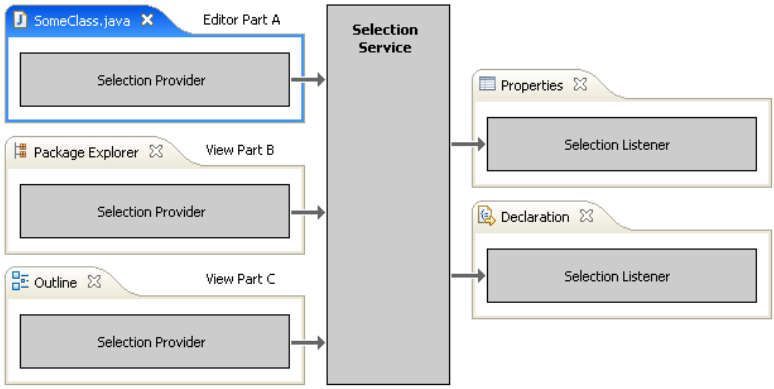

Figure 4.5: Eclipse's selection service

The *ISelectionListener* interface only holds a single Method to implement: *selectionChanged-(IWorkbenchPart, ISelection)*. We now implement this method in a way that tells us the currently selected text inside any of Eclipse's text-editors. First we create a new package called *demo.listener*. Then we create the *DemoSelectionListener* class which implements the *ISelectionListener* interface.

Because of the fact that the text-editor is part of the *org.eclipse.ui.editors* package, we need to add the *org.eclipse.ui.editors* plugin to our plugin's dependencies. For the *TextSelection*, which implements *ITextSelection* and *ISelection*, we need the *org.eclipse.jface.text* plugin. Add the dependencies and write the code that follows in your listener:

```
private static DemoSelectionListener listener;

public static DemoSelectionListener getListener(){
        if(listener==null)
                listener = new DemoSelectionListener();
        return listener;
}

private DemoSelectionListener(){

}

public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        if(part instanceof TextEditor){
                TextSelection tselection = (TextSelection)selection;
                System.out.println(tselection.getText());
        }
}
```

To add the listener to the selection service of eclipse, we have to register it. We do so inside the constructor of our user interface, because the listener will push information to it. Like that we

can make sure that no update is thrown while the user interface is not ready yet. Here's a little method registering the new listener.

```
private void registerSelectionListener() {
        new Thread(new Runnable() {
                @Override
                public void run() {
                        IWorkbenchPartSite site = null;

                        while (site == null) {
                                try {
                                        Thread.sleep(50);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                                site = getSite();
                        }
                        IWorkbenchWindow window = site.getWorkbenchWindow();
                        window.getSelectionService().addSelectionListener(
                             DemoSelectionListener.getListener());
                }
        }).start();
}
```

We add this registration to the constructor of our SampleView.

```
public SampleView(){
        registerSelectionListener();
        view = this;
}
```

To make sure the registration of the listener will not block the ui creation, we create a new thread. Also notice that we retry getting the workbench part site again and again until we really have it. When launching Eclipse, *getSite()* might return null for a few milliseconds. And to make the whole thing clean when we unload our user interface extension, we unregister the listener on dispose.

```
public void dispose(){
        getSite().getWorkbenchWindow().getSelectionService().
             removeSelectionListener(DemoSelectionListener.getListener());
        super.dispose();
}
```

Start Eclipse and open any plain text file. You will notice that your current selection is printed out to the console of the Eclipse you just launched your plugin from (Not the console of the test-Eclipse instance!).

Now to the implementation which adds useful functionality to our plugin. Change the code of the selectionChanged(IWorkbenchPart, ISelection) method to the following implementation, updating our view.

```
@Override
public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        if(part instanceof TextEditor){
                TextSelection tselection = (TextSelection)selection;
                SampleView.setArguments(tselection.getText());
        }
}
```

Now run the plugin as ”Eclipse Application” and create a project (e.g. a Java project). Inside this project create a plain text file, add text to it and select some text: The text selected will be transfered to our plugin.

To make this useful, we have to make one last change: Open the *getProgramArguments-(ILaunchConfiguration)* method of the Launcher class. Change it in a way that it adds the text of the SampleView's text box. Do not forget to convert newlines into spaces or whatever you like. Here is our solution:

```
public String getProgramArguments(ILaunchConfiguration configuration) throws
    CoreException{
        return super.getProgramArguments(configuration) + " " + SampleView.
            getArguments().replace('\n', ' ').replace('\r', ' ');
}
```

We tested the plugin with a very simple Java project containing a single main method doing nothing but printing out all arguments. We created a file called test, and stored a few arguments in it. And: when we selected text inside this file, it was inserted in our plugin automatically.

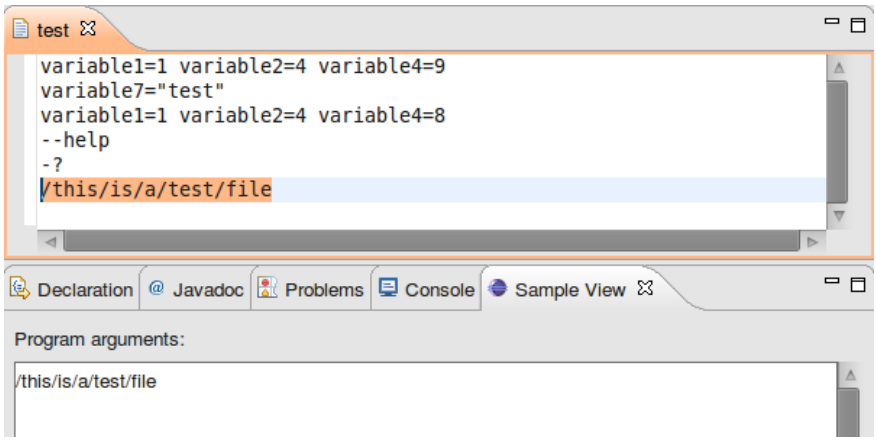

Figure 4.6: The Demo plugin in action

Chapter 5

Update Site Creation

To make our plugin available for developers which are using the Eclipse IDE, we should create an update site. These are the steps needed in order to create an update site using Eclipse:

1. Create a feature project
2. Add some information to the just created feature project
3. Create an update site project
4. Add some information to the just created update site project

Sounds easy, is even easier. Create a new feature project. The feature project is located in the Plug-In development category of the new-wizard. Give it a name, for example Demo_ Feature. Click next until you are at the ”Referenced Plug-Ins and Fragments” form of the wizard. Select ”initialize from a launch configuration” and select the one you were using when starting the plugin inside our test-Eclipse the whole time. In most cases this configuration will have the name ”Eclipse Application”. Click finish. Now fill out the feature.xml information, which is all quite trivial and can be changed later on.

The next step is to create the update site. Create a new project of the type ”Update Site Project”. Select the ”Create a web page listing all available features within the site” check-box. This is helpful in some cases, because it creates a website with a few informations. Okay, hit finish and open the site.xml. In the ”Site Map” tab, add a new category first. We will call it demo. Then add the Demo feature in this category. Done. Select ”Build All” to build the update site with the web content. The update site will create a new jar file with a new version number every time you hit ”Build All”.