

Architecture Patterns and Code Templates for Enterprise RCP Applications

Labs

The four labs in this tutorial will lead you through the process of creating an RCP application using architecture patterns relevant to enterprise development. Throughout the labs, the emphasis is on high-level architecture and design patterns rather than on lower-level coding details.

The pre-requisites for this lab are:

- Eclipse Classic, RCP and RAP Developers, or Modeling 3.6.*. Eclipse 3.7 milestones are ok as well.
- Target platform should be set to the default (your Eclipse installation). If you don't know what this means, then your setup is most likely correct.

For each lab there will be a folder containing a starting point for that lab. There is also a `lab-final` folder containing the end result of all four labs. At the beginning of each lab you should point Eclipse at the appropriate folder and import the projects.

Please begin each lab by importing the appropriate projects. Even if you completed the previous lab, there are additions in each project set that you will need.

Lab 1 - Creating a properly shaped RCP application

The goal of this lab is to create a properly shaped base RCP application into which additional functionality can be installed.

You will start with a generic Eclipse RCP application containing no perspectives, views, etc. The goal is to make this application self-updating: so that while it contains nothing, it can update itself on the fly to be anything you want it to be. We also will demonstrate how to unit test parts of an OSGi application separately from the whole.

Here is how we will do this:

1. Import this generic “bootstrapper” application's projects into workspace.
2. Modify the bootstrapper to load a configurable perspective.
3. Configure the product to be feature based.
4. Create a test fragment containing a unit test.

Import projects into workspace.

For the labs to function properly, you will need to point Eclipse at the appropriate lab directory. Creating a workspace someplace else and importing the projects there will *not* work.

The lab folders are located under the tutorial home directory in the `webapps/root/labs` directory.

1. Open Eclipse and select the `lab-1` folder as the workspace.
2. Right-click in the **Navigator** view and select **Import...** from the context menu.
3. Select **General > Existing Projects into Workspace** from the wizard selection dialog. Click **Next**.
4. On the second page, make sure that the **Select root directory** radio button is selected.
5. Click **Browse** and the `lab-1` folder should already be selected. If it is not, browse to the `lab-1` folder. A set of projects should show up in the **Projects** list. They should already be checked.
6. Click **Finish** and the projects should appear in the **Navigator** view.

Modify the bootstrapper to load perspectives.

Your workspace should now contain bundle, feature and product projects representing a simple “Hello World” RCP application. We’ll now add to this application.

1. Open the `ApplicationWorkbenchAdvisor` class in the `com.example.app.bootstrapper` bundle.
2. In the `getInitialWindowPerspectiveId` method, remove `return null` and add the following line:

```
return System.getProperty("initialWindowPerspectiveId");
```

This will allow us to configure the bootstrapper later on with a perspective by passing the perspective id on the command line or through an INI file.

Configure the product to be feature based.

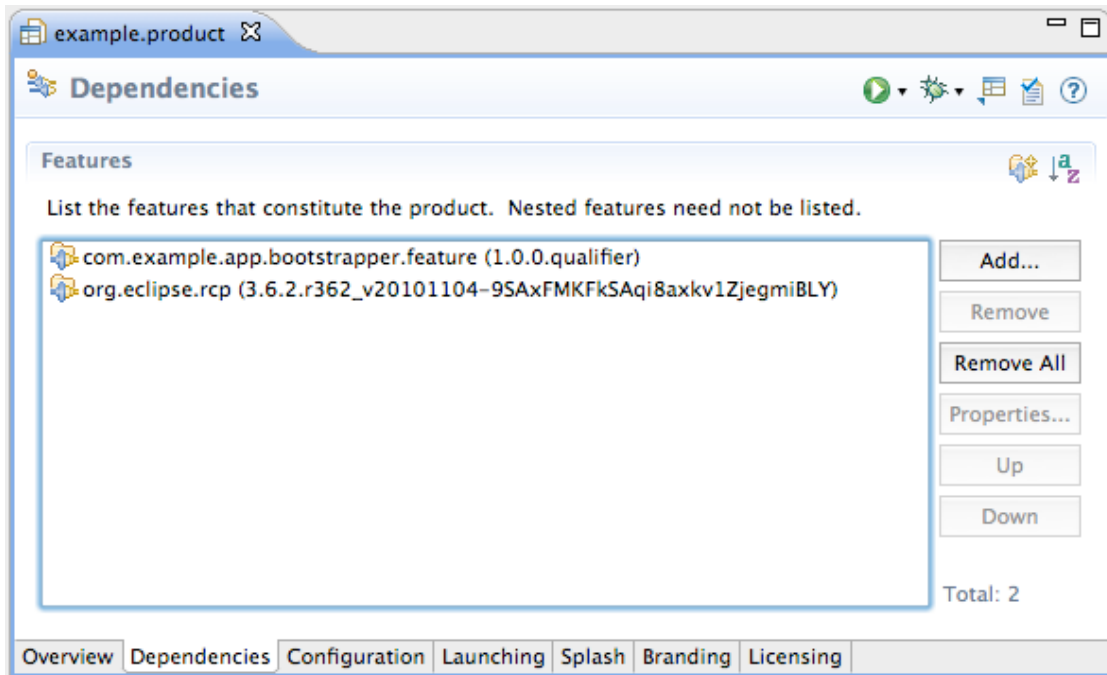
A properly shaped RCP application utilizes a *Product Configuration* file to manage branding elements and the way in which the product should be built.

The Product Configuration file is placed into its own project because a core bootstrapper bundle is potentially reusable by multiple products. Placing the configuration in a separate project is also necessary to perform a Maven Tycho build.

A properly shaped RCP application also utilizes features to group related functionality. A product configuration will then be defined in terms of these features.

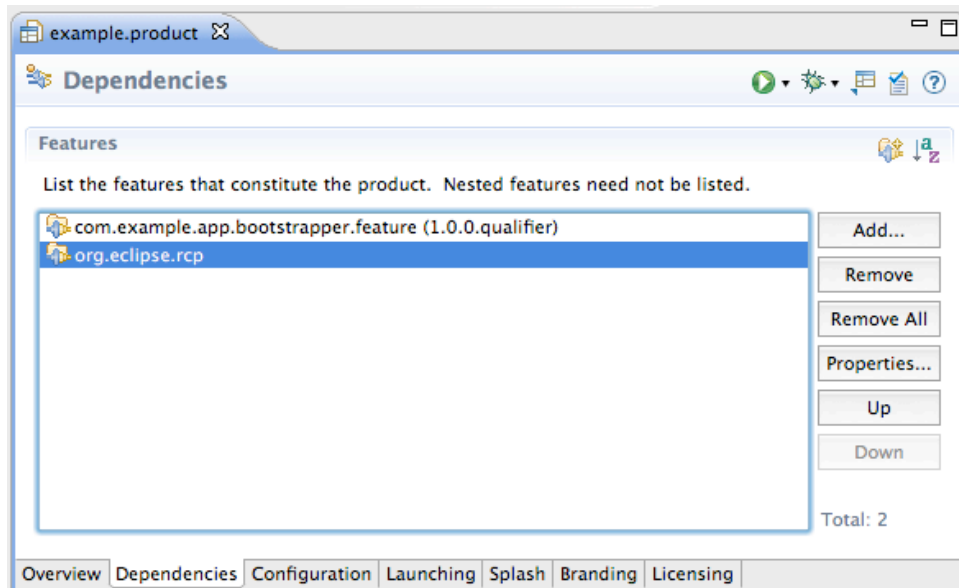
1. Open the `example.product` file in the `com.example.app.bootstrapper.product` project.
2. The file should have opened with the **Overview** tab showing. Verify that the product runs by clicking on the **Launch an Eclipse application** link in the **Testing** section. Then shut down the launched application.
3. Make this product a feature based product. Again on the **Overview** tab, select the **features** radio button in the **Product Definition** section. This will allow us to configure our product as a set of features instead of a set of bundles.

4. Switch to the **Dependencies** tab. Add `com.example.app.bootstrapper.feature` along with `org.eclipse.rcp`. Save the file and run the application again by clicking the link on the **Overview** tab. Clicking the link is important because it will regenerate your run configuration based on the listed features.



If you see the bootstrapper application appear, your product configuration is valid. Shut down the launched application.

4. Generally we don't want to limit the RCP feature to a specific version. On the **Dependencies** tab, select the `org.eclipse.rcp` feature and click **Properties**. In the dialog, enter `0.0.0` in the **Version** field. Note that no version is now displayed next to the feature.

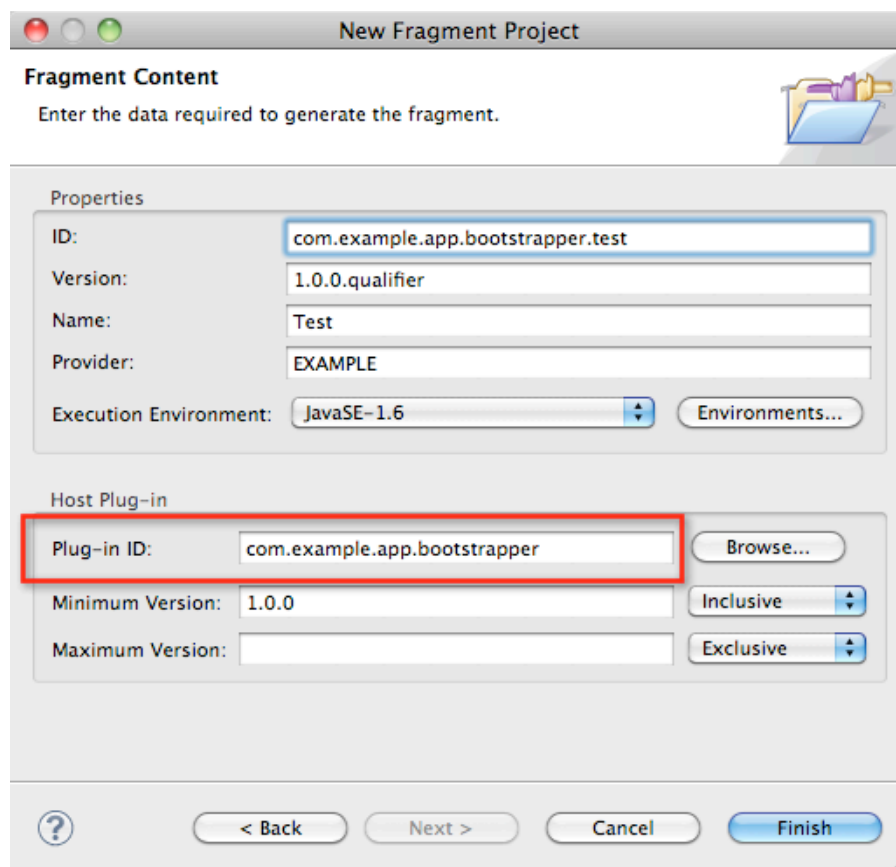


Verify that the product runs by clicking on the **Launch an Eclipse application** link on the **Overview** tab. Shut down the launched application.

Create a test fragment containing a unit test.

Create the test fragment project.

1. Select **File > New > Project...** from the main menu. Select **Plug-in Development > Fragment Project** from the list of wizards and click **Next**.
2. On the first page of the wizard, enter (or browse to) `com.example.app.bootstrapper.test` as the **Project name**. Leave everything else as is and click **Next**.
3. On the second page, enter `com.example.app.bootstrapper` in the **Host Plug-in > Plug-in ID** field. This is the bundle that contains the code that we will be testing. Click **Finish**.



New Fragment Project

Fragment Content
Enter the data required to generate the fragment.

Properties

ID:

Version:

Name:

Provider:

Execution Environment:

Host Plug-in

Plug-in ID:

Minimum Version:

Maximum Version:

Create a unit test that exercises a method in the bootstrapper bundle.

A simple non-public method called `getMessage` has been added to the `Activator` class in the `com.example.app.bootstrapper` bundle. The method returns the string "Hello!".

We'll now write a unit test to exercise this method. Of course we would normally write the unit test *before* the method.

1. In the `com.example.app.bootstrapper.test` fragment, create a package called `com.example.app.bootstrapper`.

Note that it's important that this package have the same name as the package containing the class to test. At runtime, the test will then live in the same package and have access to non-public class members.

2. Right-click on the new package and select **New > JUnit Test Case** from the context menu. If this menu is not there, make sure you are in the Java Perspective. Create a unit test class called `ActivatorTest`.

Also make sure that **New JUnit 4 test** is selected in the dialog.

When you click **Finish**, you may see a dialog box asking if you want to add `org.junit` to your build path. If you get this dialog box, say **OK**.

3. Create a test method called `testGetMessage` and write an assertion verifying the expected string value.

```
public class ActivatorTest {  
    @Test  
    public void testGetMessage() throws Exception {  
        assertEquals("Hello!",  
            Activator.getDefault().getMessage());  
    }  
}
```

Note that you will need to run an Organize Imports to bring in the import

4. Right-click on the test case and select **Run As > JUnit Plug-in Test** from the context menu. Lab 1 is now complete.

