DZone Refcardz

CONTENTS INCLUDE:

Getting Started with Eclipse RCP

By James Sugrue

ABOUT THE RICH CLIENT PLATFORM

The Eclipse Rich Client Platform (RCP) is a platform for building and deploying rich client applications. It includes Equinox, a component framework based on the OSGi standard, the ability to deploy native GUI applications to a variety of desktop operating systems, and an integrated update mechanism for deploying desktop applications from a central server. Using the RCP you can integrate with the Eclipse environment, or can deploy your own standalone rich application.

INTRODUCING THE PLUG-IN DEVELOPMENT ENVIRONMENT

To get started in developing your own plug-ins, first download a version of Eclipse including the Plug-in Development Environment (PDE). Eclipse Classic is the best distribution for this.

When developing plug-ins, you should use the Plug-in Development perspective. You'll notice this perspective provides another tab in your Project Navigator listing all the plug-ins available.

To create an RCP application, go to the **File** menu and select **New > Project** where you will be presented with the new project wizard. From here choose **Plug-in Project**.

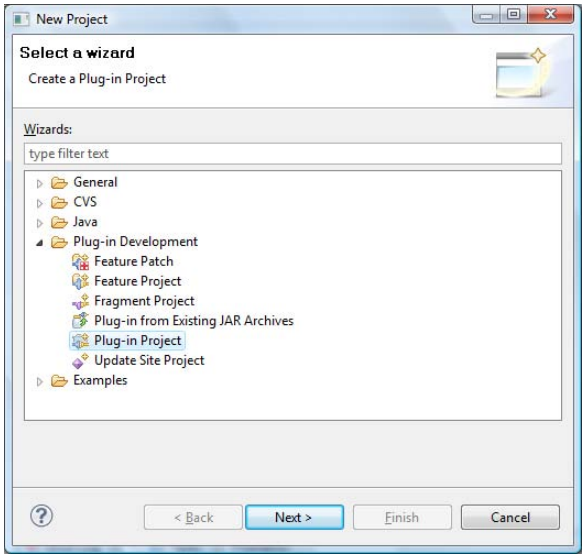

Figure 1: The New Project Wizard

The next screen allows you to assign a name to your plug-in. Usually, plug-in name follow Java's package naming conventions. RCP applications should be targeted to run on a particular version of Eclipse – here we choose to run on Eclipse 3.5.

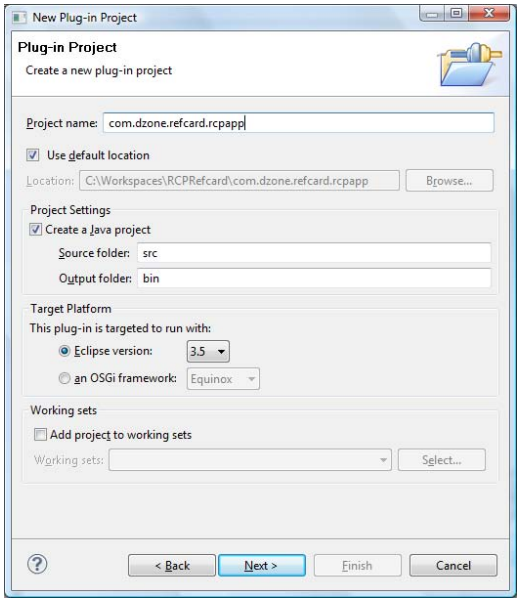

Figure 2: RCP Project Settings Page

The next page in the project wizard allows you to set some important attributes of your plug-in. This page allows you to specify whether your plug-in will make contributions to the UI. In the case of RCP plug-ins, this will usually be true. You can choose whether to create your own RCP application, or to create a plug-in that can be integrated with existing Eclipse installations.

The following table summarizes other plug-in settings and what they mean for your application. All of these settings can be changed in the generated MANIFEST.MF file for your project at any stage.

Eclipse RCP

www.dzone.com

Attribute Name	Default Value	Meaning
id	<project name>	The identifier for this RCP plug-in
Version	1.0.0.qualifier	The plug-in version. Multiple versions of any plug-in are possible in your Eclipse environment provided they have unique version numbers
Name	RCP Application	The readable name of this plug-in
Provider	The second part of your project package name.	The provider of this plug-in

Hot Tip Get started quickly with your first RCP application by using the included RCP Mail Template, available when you choose to create a standalone RCP application.

MANIFEST.MF EXPLAINED

The generated **META-INF/MANIFEST.MF** file is the centre of your RCP plug-in. Here you can define the attributes, dependencies and extension points related to your project. In addition, you may have a **plugin.xml** file. The contents of both these files are show in the plug-in manifest editor.

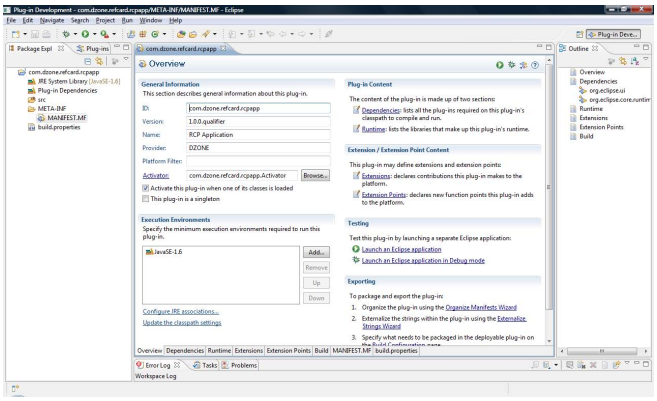

Figure 3: The plug-in manifest editor

The **Overview** tab in this editor allows you to change the settings described earlier in the new project wizard. It also provides a shortcut where you can launch an Eclipse application containing your new RCP plug-in.

The **Dependencies** tab describes how this plug-in interacts with others in the system. All plug-ins which you are dependent on will need to be added here.

The **Runtime** tab allows you to contribute packages from your own plug-in to others to use or extend. You can also add libraries that don't exist as plug-ins to your own project in the Classpath section.

Hot Tip While you may change your build path through the Dependencies or Runtime tab, changing dependent plug-ins in the Java Build Path of the project properties tab will not reflect properly in the plug-ins manifest.

The **Extensions** tab is where you go to define how this plug-in builds on the functionality of other plug-ins in the system, such as for adding menus, views or actions. We will describe these extension points in more detail in the relevant sections. The **Extension Points** tab allows you to define your own extensions for other plug-ins to use.

THE STANDARD WIDGET TOOLKIT AND JFACE

While developing UI code for your RCP application, it is important to understand the Standard Widget Toolkit (SWT). This is a layer that wraps around the platform's native controls. JFace provides viewers, in a similar way to Swing, for displaying your data in list, tables, tree and text viewers.

The UI toolkits used in Eclipse applications are a large topic, so we assume that the reader will be aware of how to program widgets in SWT and JFace.

ADDING A MENU TO YOUR PLUG-IN

One of the first things that you will want to do with your RCP plug-in is to provide a menu, establishing its existence with the Eclipse application that it is built into. To do this, as with any additions to our plug-in, we start in the Extensions tab of the plug-in manifest editor.

Up to Eclipse 3.3 Actions was the only API available to deal with menus, but since then the commands API has become available, which we will focus on here.

To add a menu in the command API you will need to follow similar steps to these:

Declare a command

To do this we use the `org.eclipse.ui.commands` extension point. Simply click on the **Add**... button in the Extensions tab and chose the relevant extension point.

First, you will need to associate this **command** with a **category**. Categories are useful for managing large numbers of commands. From the `org.eclipse.ui.commands` node, select **New>Category**. The required fields are a unique ID and a readable name.

After this right click on the node and choose **New>Command**. The important attributes for a command are listed below.

Attribute Name	Required	Use
`id`	Yes	A unique id for this command
`Name`	Yes	A readable name for the command
`Description`	No	A short description for display in the UI
`CategoryID`	No	The id of the category for this command (that you described in the previous step).
`DefaultHandler`	No	A default handler for this command. Usually you will create your own handler.

Declare a Menu Contribution for the Command

To create a menu, you will first need to add the `org.eclipse.ui.menus` extension point. From the created node in the UI, select **New>menuContribution**. The required attribute for the menu contribution is it's `locationURI`, which specifies where the menu should be placed in the UI. This URI takes the format of [scheme]:[id]?[argument-list]

An example of the more useful locationURI's in the Eclipse platform follow:

Attribute Name	Required
`menu:org.eclipse.ui.main.menu?after=window`	Insert this contribution on the main menu bar after the Window menu

`menu:file?after=additions`	Inserts contribution in the File menu after the additions group
`toolbar:org.eclipse.ui.main.toolbar?after=additions`	Insert this contribution on the main toolbar
`popup:org.eclipse.ui.popup.any?after=additions`	Adds this contribution to any popup menu in the application

Once the location of your contribution is chosen, click on **New>command** on this contribution to define the menu. The following attributes exist for each command:

Attribute	Required	Use
`commandId`	Yes	The id of the Command object to bind to this element, typically already defined, as in our earlier step. Click **Browse**... to find this
`Label`	No	The readable label to be displayed for this menu item in the user interface
`id`	No	A unique identifier fo this item. Further menu contributions can be placed under this menu item using this id in the `locationURI`
`mnemonic`	No	The Character within the `label` to be assigned as the mnemonic
`icon`	No	Relative path to the icon that will be displayed to the left of the `label`
`tooltip`	No	The tooltip to display for this menu item

Hot Tip Defining a toolbar item is a similar process. Based on a menuContribution with the correct locationURI, select New>Toolbar providing a unique id. Create a new command under the toolbar similar to the menu item approach.

Create a Handler for the Command

The final extension point required for the menu is `org.eclipse.ui.handlers`. A handler has two vital attributes. The first is the **commandId** which should be the same as the command id specified in the beginning. As you can see, this is the glue between all three parts of the menu definition.

You will also need to create a concrete class for this handler, which should implement the `org.eclipse.core.commands.IHandler` interface.

Hot Tip Clicking on the class hyperlink on the manifest editor will pop up a New Class Wizard with the fields autofilled for this.

Finally, you will need to define when this command is enabled or active. This can be done programmatically in the isEnabled() and isHandled() methods. While this is easiest, the recommended approach is to use the activeWhen and enabledWhen expressions in the plug-ins manifest editor which avoids unnecessary plug-in loading.

Hot Tip Once you have added in an extension point plugin.xml will become available. All extension points can be added through the manifest editor, or in XML format through this file.

VIEWS

In an RCP applications Views are used to present information to the user. A viewer must implement the `org.eclipse.ui.IViewPart` interface, or subclass `org.eclipse.ui.parts.ViewPart`.

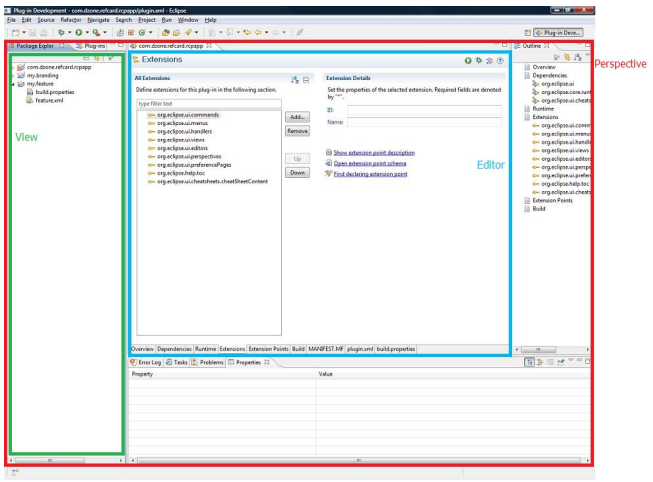

Figure 4: An illustration of the difference between perspective, editor and view.

To create a View, you will need to add the `org.eclipse.ui.views` extension point. In order to group your views, it is useful to create a category for them. Select **New>Category** from the `org.eclipse.ui.views` node to do this. The required fields are a unique ID and a readable name.

Next, choose **New>View** from the extension point node and fill in the necessary details.

Attribute	Required	Use
`id`	Yes	The unique id of the View
`name`	Yes	A readable name for this view
`class`	Yes	The class that implements the IViewPart interface
`category`	No	The id of the category that contains this view. This category should be used if you wish to group views together in the Show Views...dialog.
`icon`	No	The image to be displayed in the top left hand corner of the view
`allowMultiple`	No	Flag indicating whether multiple views can be instantiated. The default value is false.

The code behind the view is in a class that extends `org.eclipse.ui.ViewPart`. All controls are created programmatically In the `createPartControl()` method in this class.

To facilitate lazy loading, a workbench page only holds `IViewReference` objects, so that you can list out the views without loading the plug-in that contains the view definition.

When created, you will see your view in the **Window>Show View>Other**... dialog

Hot Tip It's good practice to store the view's id as a public constant in your ViewPart implementation, for easy access.

Loose Coupling

To facilitate loose coupling, your ViewPart should implement `org.eclipse.ui.ISelectionListener`. You will also need to register this as a selection listener for the entire workbench:

```
getSite().getWorkbenchWindow().getSelectionService().
addSelectionListener(this);
```

This allows your view to react to selections made outside of the view's own context.

EDITORS

An editor is used in an Eclipse RCP application when you want to create or modify files, or other resources. Eclipse already provides some basic text and Java source file editors.

In your plug-in manifest editor, add in the `org.eclipse.ui.editors` extension point, and fill in the following details

Attribute	Required	Use
`id`	Yes	The unique id of this editor
`name`	Yes	A readable name for this editor
`icon`	No	The image to be displayed in the top left hand corner of the editor when it is open
`extensions`	No	A string of comma separated file extensions that are understood by the editor
`class`	No	The class that implements the `IEditorPart` interface
`command`	No	A command to run to launch and external editor
`launcher`	No	The name of a class that implements `IEditorLauncher` to an external editor
`contributorClass`	No	A class that implements `IEditorActionBarContributor` and adds new actions to the workbench menu and toolbar which reflect the features of the editor type
`default`	No	If true this editor will be used as the default for this file type. The default value is false
`filenames`	No	A list of filenames understood by the editor. More specific than the `extensions` attribute
`matchingStrategy`	No	An implementation of `IEditorMatchingStrategy` that allows an editor to determine whether a given editor input should be opened

Editors implement the `org.eclipse.ui.IEditorPart` interface, or subclass `org.eclipse.ui.parts.EditorPart`.

Like views, to facilitate lazy loading, a workbench page only holds `IEditorReference` objects, so that you can list out the editors without loading the plug-in that contains the editor definition.

PERSPECTIVES

Perspectives are a way of grouping you views and editors together in a way that makes sense to a particular context, such as debugging. By creating your own perspective, you can hook into the **Window>Open** Perspective dialog.

To create a perspective, you need to extend the `org.eclipse.ui.perspectives` extension point.

Attribute	Required	Use
`id`	Yes	The unique id of this perspective
`name`	Yes	A readable name for the perspective
`class`	Yes	The class that implements the `IPerspectiveFactory` interface
`icon`	No	The image to be displayed related to this perspective
`Fixed`	No	Whether this perspective can be closed or not. Default is false

The class driving the perspective implements `org.eclipse.ui.IPerspectiveFactory`. This class has one method `createInitialLayout()`, within which you can use the `IPageLayout.addView()` method to add views directly to the perspective. To group many views together in a tabbed fashion, rather than side by side, `IPageLayout.createFolder()` can be used.

Hot Tip

When running your application you need to ensure that you have all required plug-ins included. Do this by checking your Run Configurations. Go to the plug-ins tab and click Validate Plug-ins. If there are errors click on Add Required Plug-ins to fix the error.

PREFERENCES

Now that you have created a perspective and a view for your RCP application, you will probably want to provide some preference pages. Your contributed preference pages will appear in the **Window>Preferences** dialog.

To provide preference pages you will need to implement the `org.eclipse.ui.preferencePages` extension in the plug-in manifest editor.

Attribute	Required	Use
`id`	Yes	The unique id of this preference page
`name`	Yes	A readable name for the preference page
`class`	Yes	The class that implements the IWorkbenchPreferencePage interface
`category`	No	Path indicating the location of the page in the preferences tree. The path may be defined using the parent preference page id or a sequence of ids separated by "/". If no category is specified, the page will appear at the top level of the preferences tree.

While the preference page class will implement `org.eclipse.ui.IWorkbenchPreferencePage`, it is useful to extend `org.eclipse.jface.preference.FieldEditorPreferencePage` as it provides `createFieldEditors()` method which is all you need to implement, along with the `init()` method in order to display a standard preference page. A complete list of FieldEditors is provided in the `org.eclipse.jface.preference` package.

Loading and Storing Preferences

Preferences for a plug-in are stored in an `org.eclipse.jface.preference.IPreferenceStore` object. You can access a plug-ins preference through the Activator, which will typically extend `org.eclipse.ui.plugin.AbstractUIPlugin`. Each preference you add to the store has to be assigned a key. Preferences are stored as String based values, but methods are provided to access the values in number of formats such as double, int and Boolean.

PROPERTY SHEETS

While preferences are used to display the overall preferences for the plug-in, property sheets are used to display the properties for views, editors or other resources in the Eclipse environments. By hooking into the Properties API, the properties for you object will appear in the Properties view (usually displayed at the bottom of your Eclipse application).

The Properties view will check if the selected object in the workspace can supports the `org.eclipse.ui.views.properties.IPropertySource` interface, either through implementation or via the `getAdapter()` method of the object. Each property gets a descriptor and a value through the `IPropertySource` interface.

HELP

All good applications should provide some level of user assistance. To add help content to the standard **Help>Help Contents** window, you can use the `org.eclipse.help.toc` extension point. Add a number of toc items to this extension point – the only mandatory attribute for each toc entry is the file that contains the table of contents definition.

Hot Tip To see a quick example of what help content should look like, choose the Help Content item from the Extension Wizards tab when adding to the plug-ins manifest.

```
<toc label="Getting Started" link_to="toc.xml#gettingstarted">
      <topic label="Main Topic"  href="html/gettingstarted/
       maintopic.html">
            <topic label="Sub Topic" href="html/gettingstarted/
             subtopic.html" />
      </topic>
      <topic label="Main Topic 2">
            <topic label="Sub Topic 2" href="html/gettingstarted/
             subtopic2.html" />
      </topic>
</toc>
```

Each topic entry should have a link to a HTML file with the full content for that topic. The above XML extract from a table of contents file illustrates this. There is also the choice to use the definition editor for help content. This will open by default in Eclipse when choosing a *toc* file.

Cheat Sheets

Another user assistance mechanism used in Eclipse is a cheat sheet, which guides the user through a series of steps to achieve a task. To create your initial cheat sheet content, use the **New>Other...>User Assistance>Cheat Sheet**. This presents you with an editor to add an Intro and a series of items, with the option to hook in commands to automate the execution of the task.

To add this cheat sheet to your plug-in manifest, the cheat sheet editor has a *Register this cheat sheet* link on the top right hand corner. When registering the cheat sheet you will need to provide it with a category and a description.

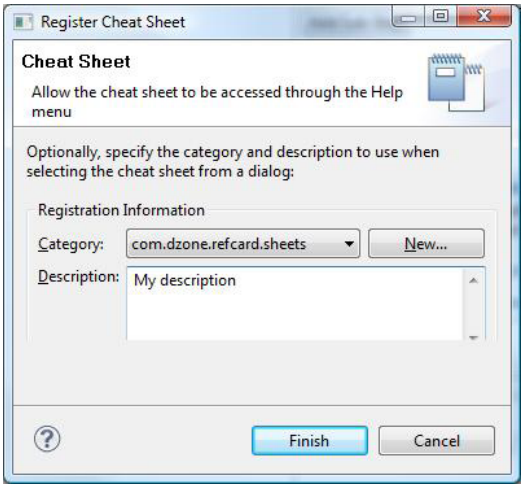

Figure 5: Cheat sheet registration dialog

Clicking finish on this dialog will add the `org.eclipse.ui.cheatsheets.cheatSheetContent` extension point to your manifest. You can modify the details of the cheat sheet from here if necessary.

FEATURES

You can help the user to load up your plug-in(s) as a single part, by combining them into one feature. Eclipse provides a wizard to create your feature through the **New Project> Plug-in Development >Feature Project** wizard.

This wizard generated a **feature.xml** file which has an editor, similar to the plug-in manifest editor, where you can change the details of your feature.

The most important section is the **Plug-ins** tab, which lists the plug-ins required for your feature. The **Included Features** tab allows you to specify sub-features to include as part of your feature. On the **Dependencies** tab, you can get all the plug-ins or features that you are dependent on by clicking on the **Compute** button.

A simple feature.xml may look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="my.feature"
      label="Feature"
      version="1.0.0.qualifier"
      provider-name="James">

   <description url="http://www.example.com/description">
      [Enter Feature Description here.]
   </description>

   <copyright url="http://www.example.com/copyright">
      [Enter Copyright Description here.]
   </copyright>

   <license url="http://www.example.com/license">
      [Enter License Description here.]
   </license>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
   </requires>

   <plugin
         id="com.dzone.refcard.rcpapp"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>

</feature>
```

BRANDING

The feature also provides a single location where you can define all the branding for your application. In the **Overview** tab, you can assign a Branding Plug-in to the feature.

The branding plug-in needs to contain the following artefacts:

Item	Purpose
`about.html`	A HTML file that will be displayed in the **Plug-in Details>More** Info dialog
`about.ini`	This file contains most of the branding information for the feature Described below
`about.properties`	Used for localisation of the strings from the **about.ini** file. The values are referenced using the %key notation

about.ini

Property	Purpose
`aboutText`	Multiline description containing name, version number and copyright information. Will appear in the **About>Feature Details>About Features** dialog.
`featureImage`	A 32x32 pixel icon representation of the feature to be used across the relevant About dialogs

All of the icons and files referenced by the about.ini file should be placed in this plug-in also.

Product Branding

A product is an entire distribution of an RCP application, rather than a feature intended to be part of an existing distribution. As such, products have additional branding requirements. To specify these extra parameters, a contribution to the `org.eclipse.core.runtime.products` extension point is required.

The **product** must be assigned the application to run, the name of the product (for the title bar) and a description. Further properties are added as name/value pairs underneath the product.

Hot Tip An application can be provided by using the org.eclipse.core.runtime.applications extension point

Property	Purpose
`windowImages`	The image used for this application, in windows and dialogs. This should be in the order of the 16x16 pixel image, followed by the 32x32
`aboutImage`	Larger image to be placed in the About dialog
`aboutText`	Multiline description containing name, version number and copyright information. Will appear in the **About>Feature Details>About Features** dialog.

You can also provide most of these details in the **Branding** tab of the generated **.product** file.

Splash Screen

The **.product** file that is generated while creating your product includes a **Splash** tab. Here you can specify the plug-in that contains the splash.bmp file for your Splash screen. Typically, this should reside in your branding plug-in. The splash screen can also be customized with templates, and can include a progress bar with messages.

ABOUT THE AUTHOR

James Sugrue is a software architect at Pilz Ireland, a company using many Eclipse technologies. James is also editor at both EclipseZone and JavaLobby. Currently he is working on TweetHub, a Twitter client based on RCP and ECF. James has also written a Refcard on EMF and has another Refcard on the way covering Eclipse Plug-ins.

RECOMMENDED BOOKS

Building on two internationally best-selling previous editions, **Eclipse Plug-ins**, Third Edition, has been fully revised to reflect the powerful new capabilities of Eclipse 3.4.

In **Eclipse Rich Client Platform**, two leaders of the Eclipse RCP project show exactly how to leverage Eclipse for rapid, efficient, cross-platform desktop development.

BUY NOW

books.dzone.com/books/eclipse-plug-ins

books.dzone.com/books/eclipse-rcp

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-75-2
ISBN-10: 1-934238-75-9
50795
9 781934 238752
$7.95