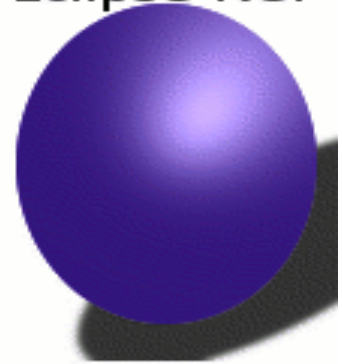


# Developing Eclipse Rich Client Applications

Eclipse RCP



## Tutorial Script

13. February 2005

**Dr. Frank Gerhardt (fg@acm.org), Software Experts Network Stuttgart**  
**Dr. Christian Wege (wege@acm.org), DaimlerChrysler AG**

Note:

Find updates to this document and the code examples at  
<http://www.eclipseteam.de>

## Inhalt

1	Introduction.....	3
1.1	Tutorial Overview.....	3
1.2	Prerequisites.....	3
2	Eclipse Installation.....	4
3	Generate the most basic rich client application.....	4
4	Starting an RCA.....	7
5	Basic elements of an RCA.....	10
5.1	RcpdemoPlug-in.....	10
5.2	RcpdemoApplication.....	10
5.3	SamplePerspective.....	10
5.4	SampleWorkbenchAdvisor.....	10
6	Deployment on RCP-distribution.....	11
6.1	Export Wizard.....	11
6.2	Configure Eclipse runtime in config.ini.....	11
7	A more complete RCA – rcpmail.....	12
7.1	Splash screen.....	12
7.2	Initial Layout.....	13
7.3	About dialog.....	13
8	Adding Help.....	13
8.1	Create Help Plug-in.....	13
8.2	Test Help Plug-in in SDK.....	13
8.3	Help UI contribution.....	14
8.4	Custom config.ini.....	17
8.5	Deployment on RCP distribution.....	18
9	Add Update-Manager to RCA.....	19
9.1	Deployment on RCP distribution.....	21
9.2	Create update site.....	22
10	PDE test.....	23
11	Build & Test Automation.....	25
11.1	Build rcpmail Distribution.....	25
11.2	Build & Perform Tests.....	30
12	References.....	33
12.1	Eclipse.....	33
12.2	Web sites.....	33
12.3	News groups.....	34
12.4	Presentations.....	34
12.5	Articles.....	34
12.6	Books.....	34

## 1 Introduction

This tutorial teaches the creation of rich client applications (abbreviated RCAs) based on the Eclipse Rich Client Platform (RCP).

Soon after its inception, Eclipse was used for building applications outside the tools domain for which Eclipse was originally designed. The Eclipse development team embraced this trend and with Eclipse 3.0 introduced the RCP, that facilitates the creation of rich client applications. In this tutorial we describe the overall architecture of an RCP-based application, its specific components, development, packaging, deployment and testing.

Participants will learn and perform the steps to build their own RCP application. We will show how to develop a minimal application plug-in, add a feature including custom branding, and package the application for deployment. We will show how to deliver updates using an update site and the update manager. We test our sample RCP application using PDE JUnit and demonstrate how test-driven development works with RCP. We cover running the JUnit tests from an Ant script as part of a nightly build.

### 1.1 Tutorial Overview

Throughout the tutorial we work with examples that come with the Eclipse distribution. Those examples are the base for adding further Eclipse components and for demonstrating the development steps.

### 1.2 Prerequisites

- Laptop with Java VM 1.4 installed.
- Current milestone build 4 of Eclipse 3.1 SDK. The 3.1 release is expected for end of July 2005.
- RCP and RCP SDK downloads available in local file system.

02.01.2005	16:38	5.274.361	eclipse-RCP-3.1M4-win32.zip
02.01.2005	18:43	20.810.672	eclipse-RCP-SDK-3.1M4-win32.zip
22.12.2004	10:46	94.433.804	eclipse-SDK-3.1M4-win32.zip
01/16/2005	18:58	242,771	eclipse-test-framework-3.1M4.zip

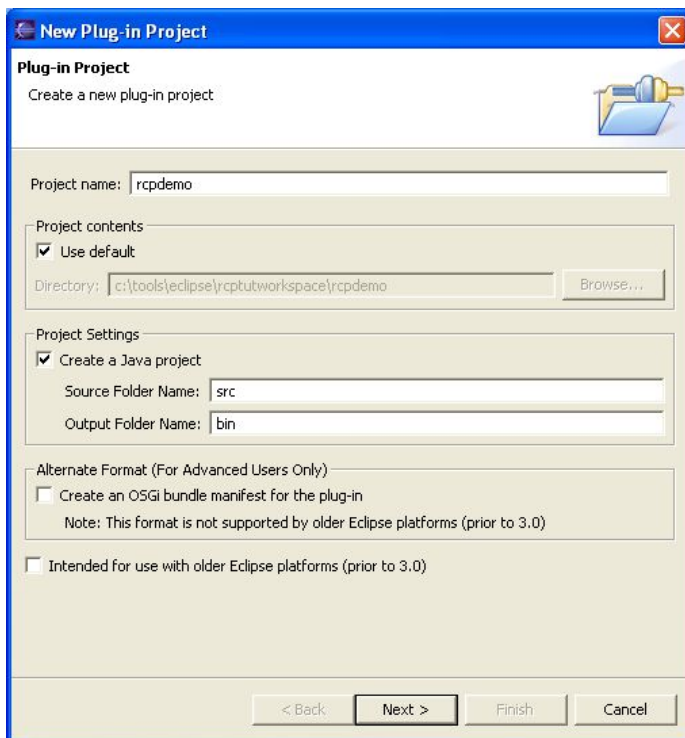
Note: This tutorial was created on a Windows XP installation. With slight modifications it should run on other Eclipse-supported platforms as well.

## 2 Eclipse Installation

We assume, Eclipse is installed in <eclipse-dir> (e.g. [C:\eclipse](#)). The workspace is created in the <workspace-dir> (e.g. C:\workspace). In later sections we will access these directories.

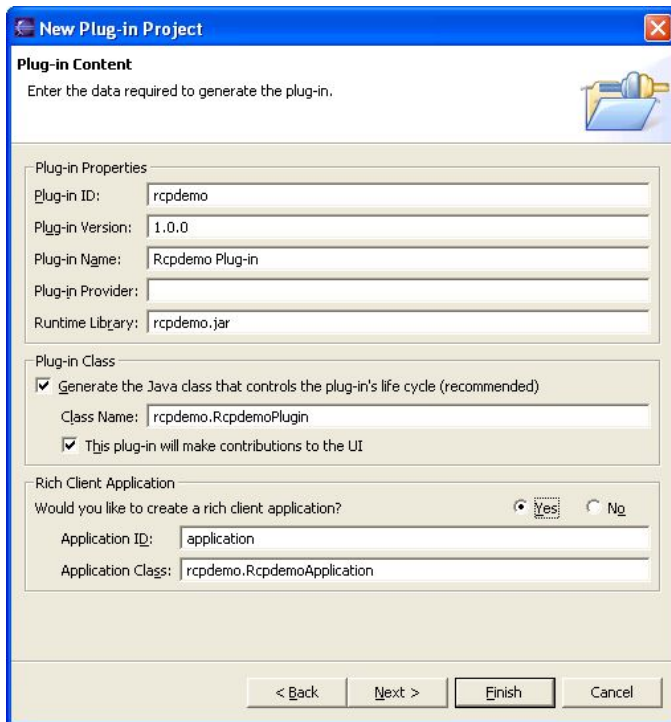
## 3 Generate the most basic rich client application

Eclipse comes with a wizard to create an RCA. This gives you a skeleton for all your own development. This wizard basically performs the manual steps that Ed Burnette describes in his RCP tutorial [EclipsePowered]. To make our life simpler, we use the wizard.



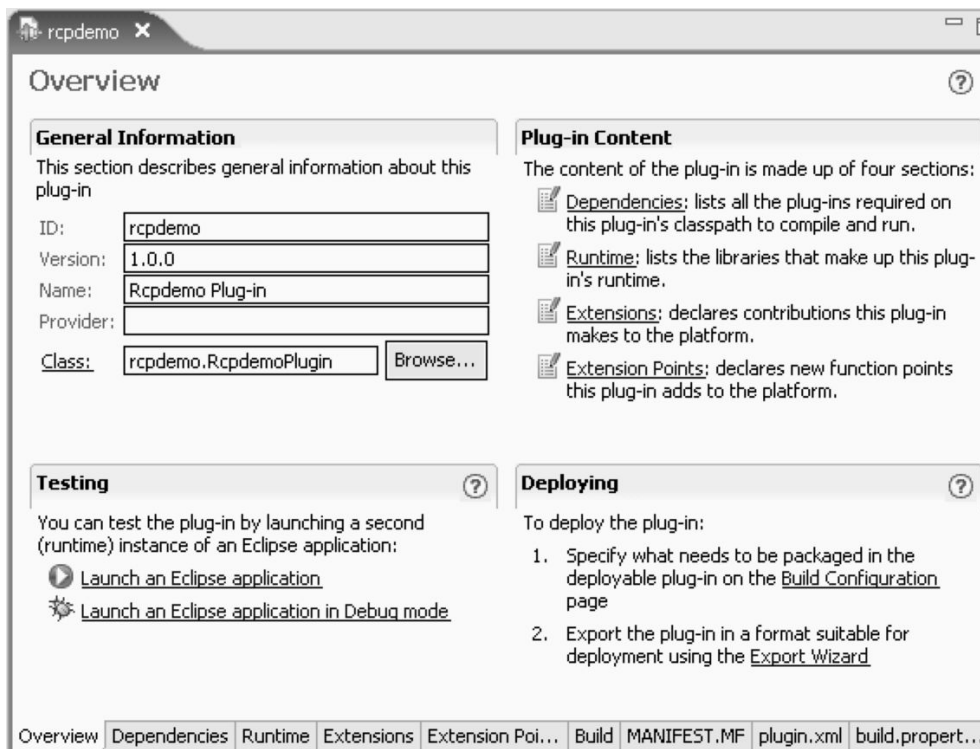
- Create a new plug-in project
- Project name: rcpdemo
- Next
- “Would you like to create a rich client application?” > Check “Yes”
- Finish

## Tutorial: Developing Eclipse Rich-Client Applications



This creates a project that contains the most basic elements of an RCA. The wizard opens the Plug-in Manifest editor. Let us quickly go through the different tabs.

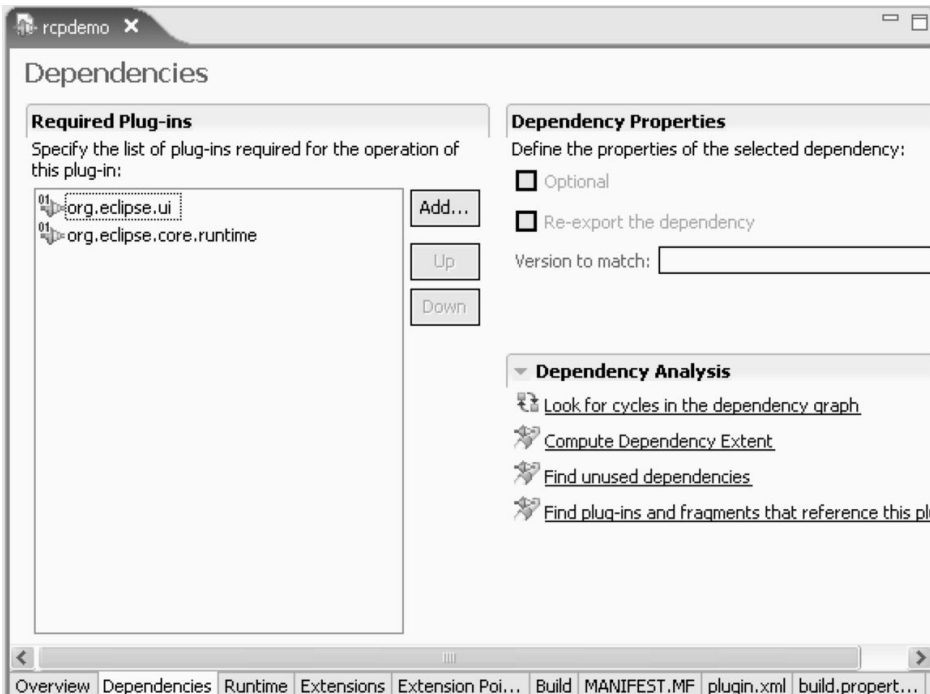
The “Overview” tab shows the plug-in ID and the name of its plug-in class.



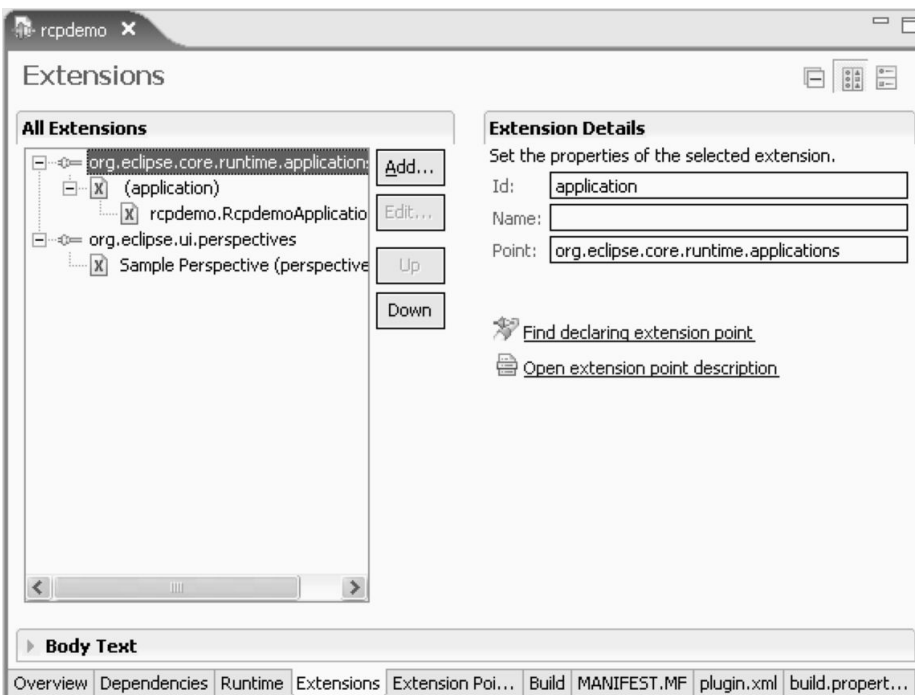
The “Dependencies” tab only shows the direct dependencies. It doesn't show indirect dependencies. For running an RCA, it needs to have all direct and

## Tutorial: Developing Eclipse Rich-Client Applications

indirect dependent plug-ins available at runtime. This extent is important for application deployment. We will see later, how the launch configuration wizard can compute the whole extent of required plug-ins based on this dependency information here.



In the “Extensions” tab we see, that our plug-in defines a new application. It also defines a new perspective. We will revisit the implementing classes in a second.



The new project wizard also created all the necessary source code for the rcpdemo application. Before we review this source code, we want to launch

the application.

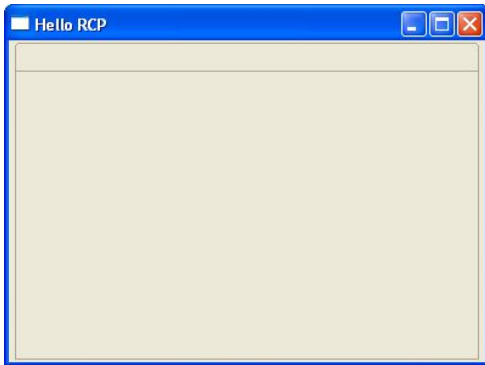
## 4Starting an RCA

A classical contribution to Eclipse consists of a set of views, editors, perspectives, etc. that extend the existing workbench. An RCA, on the other hand, replaces the well-known Eclipse workbench. In case of rcpdemo it doesn't even define a view that could be started inside the classical workbench. Therefore, to start an RCA, we need to define the set of plug-ins and point to the application that replaces the classical workbench.

This is done in the launch configuration. The plug-in manifest editor provides a convenient way to create the appropriate launch configuration (see the screenshot above).

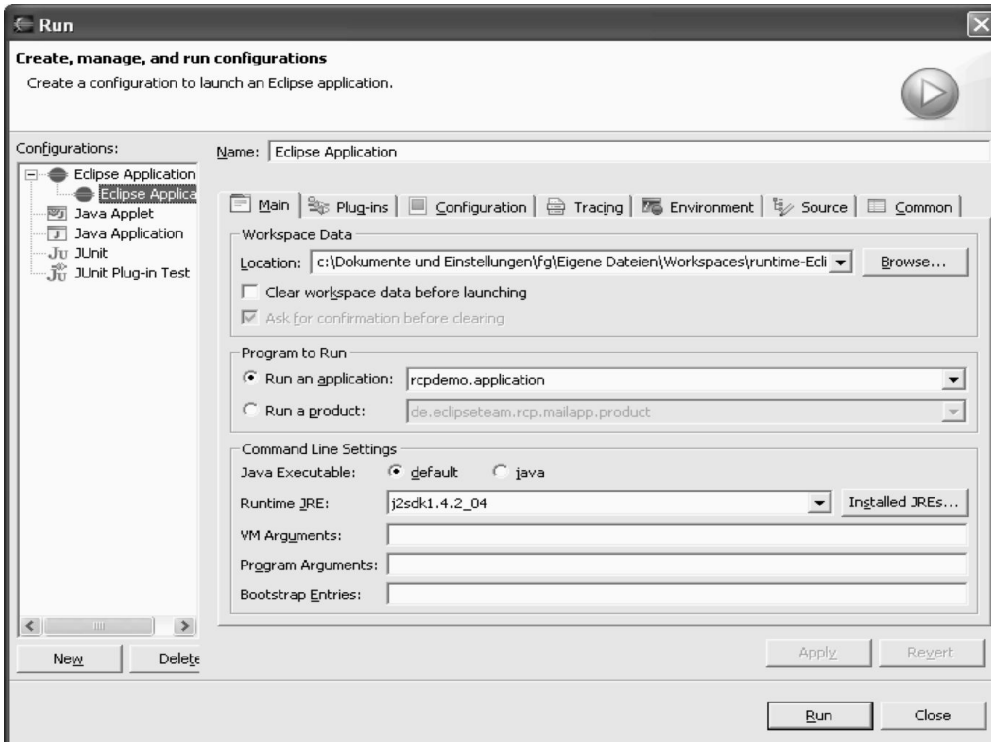
- Switch back to the “Overview” tab.
- Click on “Launch an Eclipse application”

Now you should see the most basic workbench window.



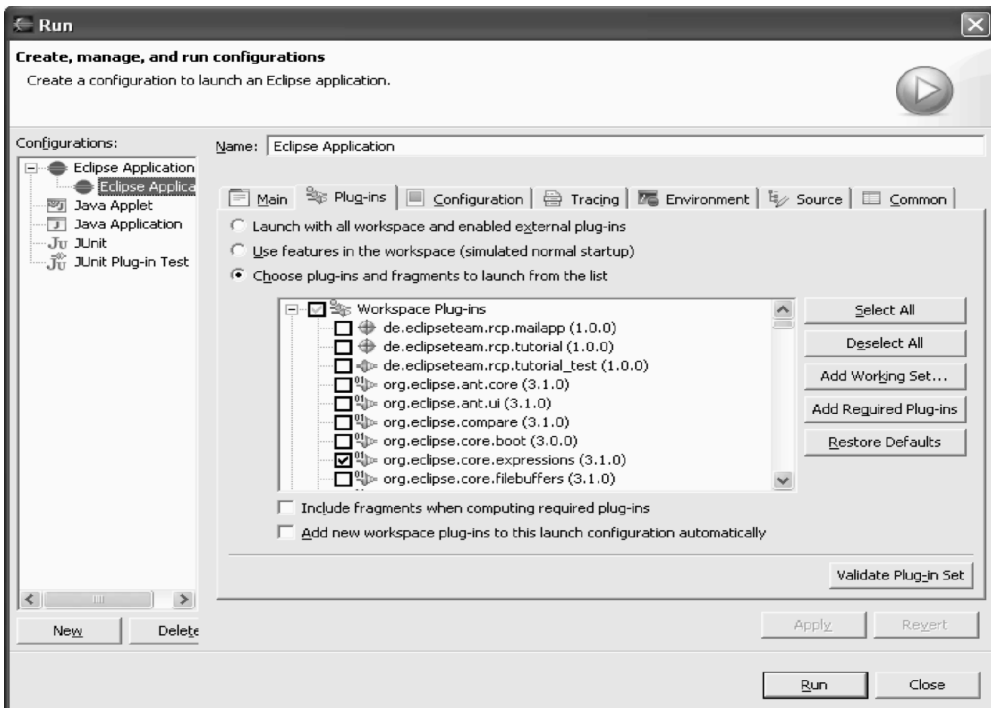
Now review the launch configuration (Run>Run...>Eclipse Application)

## Tutorial: Developing Eclipse Rich-Client Applications



The “main” tab specifies a location for workspace data. As “programm to run”, the rcpdemo.application is selected. We will discuss later, how to run the RCA as a product.

The “Plug-ins” tab lists all the necessary plug-ins. Based on the list of direct dependencies as listed in the plug-in manifest, the launch configuration lists all direct and indirect dependencies.



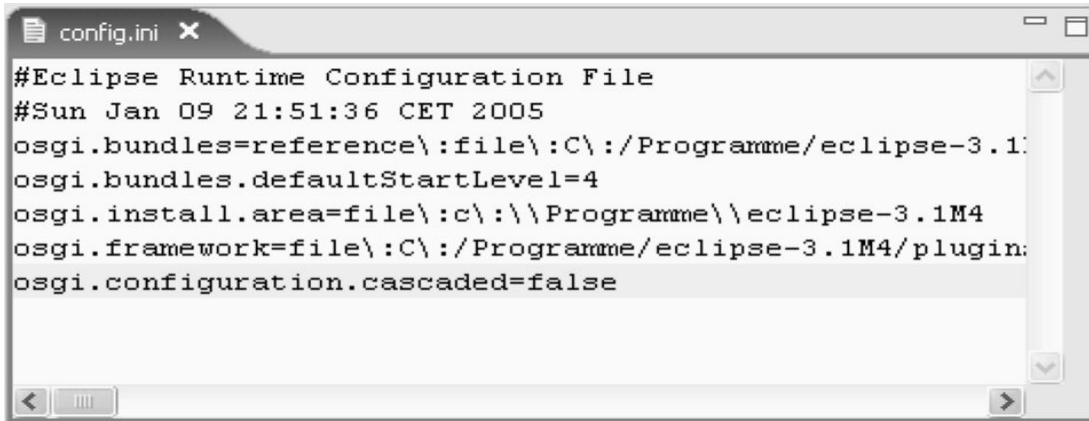
The “configuration” tab specifies the location of configuration information.



## Tutorial: Developing Eclipse Rich-Client Applications

Review the “config.ini” file that is found in that directory.

Tip: open this file inside Eclipse using File > Open external file... and navigate to “<workspace-dir>\.metadata\plugins\org.eclipse.pde.core\Eclipse Application\config.ini”.

A screenshot of a text editor window titled "config.ini" showing the contents of the Eclipse runtime configuration file. The text is as follows:

```
#Eclipse Runtime Configuration File
#Sun Jan 09 21:51:36 CET 2005
osgi.bundles=reference\file\C\:/Programme/eclipse-3.1M4
osgi.bundles.defaultStartLevel=4
osgi.install.area=file\c\:\Programme\ eclipse-3.1M4
osgi.framework=file\C\:/Programme/eclipse-3.1M4/plugin:
osgi.configuration.cascaded=false
```

The file contains a list of all plug-ins that are listed in the “plug-ins” tab and is generated from that table. The OSGi runtime reads the static list of plug-ins from this config file.

## 5 Basic elements of an RCA

Let's review the source code of the rcpdemo application. It consists of the classes.



### 5.1 RcpdemoPlug-in

Since every RCA is an Eclipse plug-in, it needs a plug-in class. The generated implementation provides access to the plug-in instance and loads the resource bundle.

### 5.2 RcpdemoApplication

This is the bootstrap class for the RCA. This class creates and runs the workbench, that contains the main event loop.

### 5.3 SamplePerspective

To arrange UI elements inside the workbench, you technically need a perspective. For now, it doesn't contain anything interesting, since the rcpdemo application doesn't contain any view or editor.

### 5.4 SampleWorkbenchAdvisor

The workbench advisor provides a number of hooks around the startup and shutdown of the workbench window. For example it allows to set the window size before it is opened.

- Change the workbench window size and start the rcpdemo application.

## 6 Deployment on RCP-distribution

For now, the RCA only runs inside the Eclipse SDK. End users want to start the application directly. Therefore we need to export the application from the development environment and deploy it on a RCP distribution (the RCP distribution contains the minimal set of plug-ins only - the RCP SDK also has the source code of those plug-ins).

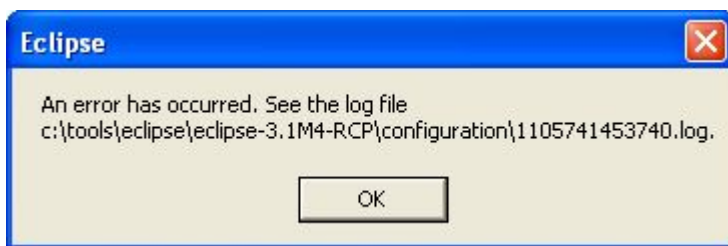
### 6.1 Export Wizard

- Create a folder <rcpdemo-dir> for your application rcpdemo.
- Unpack the RCP distribution into this folder
- In the rcpdemo plug-in manifest file open the “overview” tab.
- Click on “Export Wizard”.
- Set Export Options to “a directory structure”.
- Set destination directory to “<rcpdemo-dir>\eclipse”

The “eclipse” folder contains the RCP. There you find **eclipse.exe** to start the RCA.

- Invoke eclipse.exe

You will see a dialog box, reporting an error.



The log file tells us, that no application ID has been found. Since the RCP distribution is a platform only, it doesn't come with a preconfigured application or application ID.

Invoke eclipse.exe from the command line with the application ID

```
eclipse -application rcpdemo.application
```

You still get an error. The log file complains, that the application with the provided ID is not found. This means, the rcpdemo plug-in is not found by the Eclipse runtime. In fact, hardly any of the plug-ins is found by the runtime. We have to specify the list of available plug-ins explicitly.

### 6.2 Configure Eclipse runtime in config.ini

Besides the log file, the <rcpdemo-dir>/eclipse/configuration folder contains the config.ini file that is read by the OSGi runtime. Here you have to list the available plug-ins.

```
osgi.bundles=org.eclipse.core.runtime@2:start, org.eclipse.core.expressions,
```

```
org.eclipse.help, org.eclipse.jface, org.eclipse.osgi.services,  
org.eclipse.swt.win32, org.eclipse.swt, org.eclipse.ui.workbench,  
org.eclipse.ui, rcpdemo
```

- Start the application like above and provide the rcpdemo application.

If you don't want to specify the application every time, you can configure the default application inside config.ini.

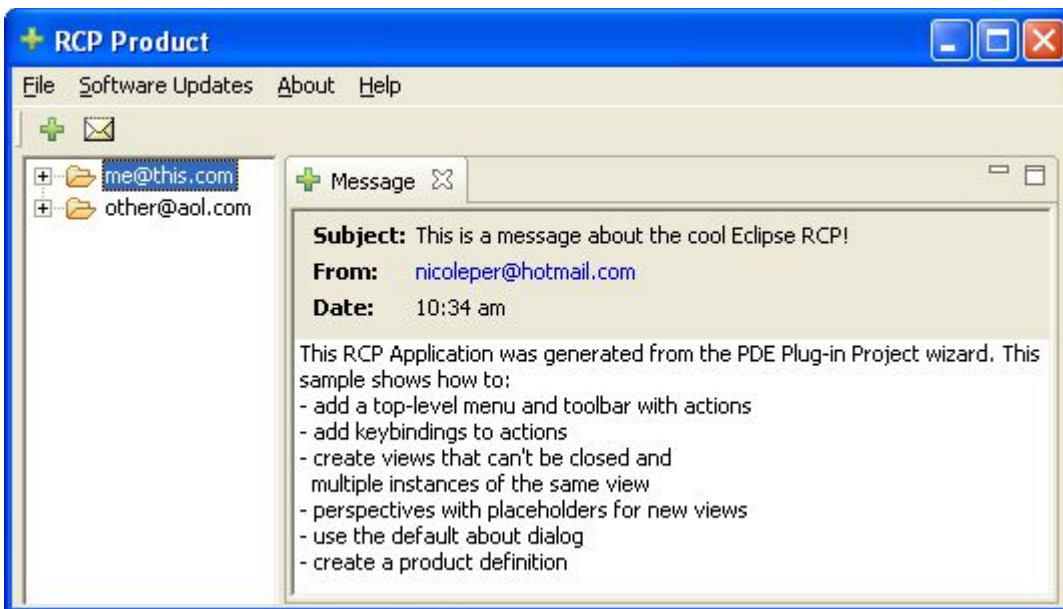
```
eclipse.application=rcpdemo.application
```

- Start the application without any parameter (e.g. double click).

## 7A more complete RCA – rcpmail

After mastering the first steps with the most simple rich client application, we want to look into a more complete example.

- In the PDE perspective, create a new plug-in project
- Name: rcpmail
- Next
- “Would you like to create a rich client application?” > Check “Yes”
- Create a plug-in using the “RCP Mail Template”. You might want to read through the short description of the template.
- Finish
- Start the new rcpmail application and observe the new features.



Review, how those features are implemented. Below, we discuss some of those features.

### 7.1 Splash screen

The splash screen is the most immediate difference. This splash screen is part

of the product branding as defined in the plug-in manifest. The name splash.bmp is the default name of application to run. In the config.ini you can define another name and path for the splash screen. We come back to this later.

Observe the “org.eclipse.core.runtime.product” extension in the extension tab of the manifest editor. It defines the window image, the about dialog image, and the about dialog text.

## 7.2 Initial Layout

The Perspective class adds the non-closable view that contains the tree navigator and one closable, stacked view. It also defines some properties of the added UI elements (like the relative size).

## 7.3 About dialog

Open the about dialog from the menu bar. It is the standard about dialog, but without the list of features. Since the RCP doesn't contain the update manager, it has no notion of Eclipse “features”. Therefore, there is no list of features in the about dialog.

This about dialog is added by the ActionBuilder class, since opening the about dialog is an action. The dialog action is instantiated through the Eclipse ActionFactory in the ActionBuilder class:

```
aboutAction = ActionFactory.ABOUT.create(window);
```

## 8 Adding Help

The help system is an optional addition to the RCP. We will add help to the existing rcpmail application.

### 8.1 Create Help Plug-in

To add Help, we first have to create a plug-in, that contains the help content.

- Create new plug-in project, name: rcphelp.
- Next until templates screen
- Check Custom plug-in wizard
- Select “Help table of contents”. Deselect everything else.
- Check “Primary” and press finish.

### 8.2 Test Help Plug-in in SDK

To test the help plug-in, we require a help system installed inside Eclipse to present the contents in the help plug-in. Since the rcpmail application does not yet contain a help system, we test the help system in a full blown Eclipse target workbench first.

- In the Overview page of the plug-in manifest editor of the help plug-in, press “Launch an Eclipse application”.
- Verify that the just created help plug-in is visible in the target workbench.

### 8.3 Help UI contribution

In order to add help to the rcpmail application, we have to contribute a menu entry, an action to invoke help, and the help system itself. In general, Eclipse provides two ways to add contributions to the UI. You can either explicitly contribute within your Java code, or you can make contributions to specific extension points. In this exercise we will use the first approach of contribution to the UI. We will cover the second approach later.

The generated rcpmail application defines some menu entries and adds some actions already. To keep it simple, extend those definitions in the `ActionBuilder` class.

Add a new field

```
private IWorkbenchAction helpAction;
```

and extend some existing methods. First, you have to add the menu entries to the UI.

```
public void populateMenuBar(IActionBarConfigurer configurer) {  
    ...  
    MenuManager helpMenu = new MenuManager("&Help", "help");  
    ...  
    mgr.add(helpMenu);  
    ...  
    // Help  
    helpMenu.add(helpAction);  
}
```

Then you need an action that is called, when the user invokes the menu entry in the UI.

```
public void makeAndPopulateActions(IWorkbenchConfigurer workbenchConfigurer,  
IActionBarConfigurer configurer) {  
    ...  
    helpAction = ActionFactory.HELP_CONTENTS.create(window);  
    ...  
}
```

Finally, clean up.

```
public void dispose() {  
    ...  
    helpAction.dispose();  
}
```

```
}
```

Now we need to run the rcpmail application in the PDE to test the extensions.

- Add the rcphelp plug-in to the dependencies of the rcpmail application.
- Open the launch configuration and add rcphelp to the plug-in list.
- Run the launch configuration

Observe that you find the added menu entries but cannot start the help system, because you are still missing some plug-ins.

The RCP contains a plug-in called org.eclipse.help. This is not the help system itself, but simply defines an extension point to add a help system. The help system itself is implemented in some plug-ins that are we can get from the Eclipse SDK. Inside the PDE this means, adding them to the launch configuration. The following plug-ins are obvious, since they indicate that they implement the help system:

- org.eclipse.help.appserver
- org.eclipse.help.base
- org.eclipse.help.ui
- org.eclipse.help.webapp

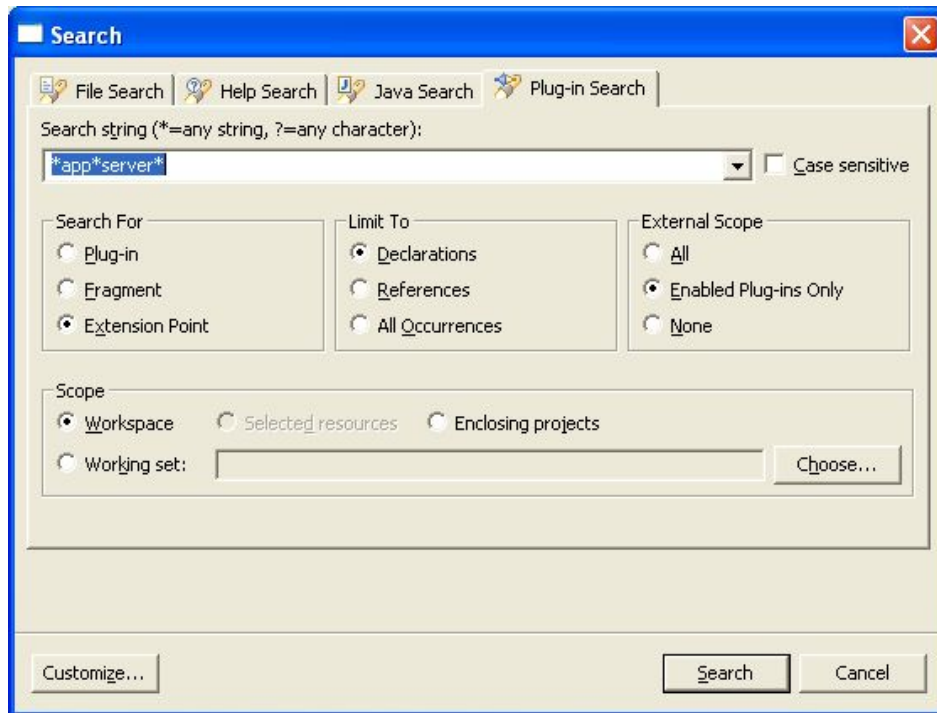
Press “Add required plug-ins”. Now you can run the modified launch configuration and see, that the help system still does not launch. Instead you get an error message to look in the log file (located in the target workspace ...\\runtime-EclipseApplication\\metadata\\log). In the log file you will find something like:

```
org.eclipse.core.runtime.CoreException: Exception occurred starting application server.  
    at org.eclipse.help.internal.appserver.AppserverPlugin.startWebappServer  
(AppserverPlugin.java:142)
```

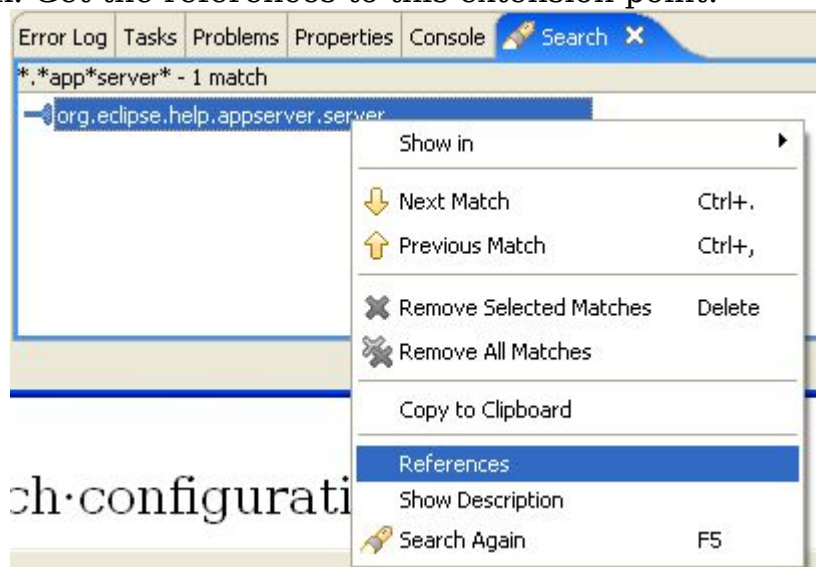
Browsing this piece of code reveals, that an extension to an extension point is missing.

```
if (appServer == null)  
    throw new CoreException(new Status(IStatus.ERROR, PLUGIN_ID,  
        IStatus.OK,  
        AppserverResources.getString("Appserver.start"), null));
```

Search for this extension point:



This search results in “org.eclipse.help.appserver.server”. It seem, we are on the right track. Get the references to this extension point:



and find

- org.eclipse.tomcat

Obviously the help system needs an application server to present its content. Eclipse uses Tomcat and for the help system to run, this plug-in has to be present in the runtime. Add it to the launch configuration, don't forget to press “Add required plug-ins”. This will add

- org.apache.ant



That should do it.

- Run the modified launch configuration.
- Start the help system through the menu entry.

This procedure described above is only one way to find the required plug-ins. Actually, finding the list of required plug-ins seem as a weak spot in Eclipse development and it appeals to your predator qualities inside the Eclipse ecosystem since those plug-ins are not an easy prey.

If you are lazy, just tick on the listed plug-ins. You can rely on this list to be stable until the next API change.

## 8.4 Custom config.ini

For the deployment of your application on an external RCP distribution, you need to modify the config.ini of that external distribution. It would be convenient to test those changes to config.ini inside the IDE.

- Copy the config.ini from a fresh RCP distribution to the rcpmail project.

If you review a full-blown Eclipse installation, you won't find a static list of plug-in in its configuration file. Instead you can install new plug-ins by dropping them into the “plug-ins” directory. Eclipse can create a list of plug-ins at startup dynamically through a so-called “configurator”. Let's switch to this type of startup of the rcpdemo application.

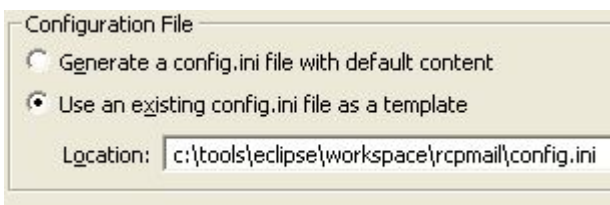
- Replace the “osgi.bundles” entry by “org.eclipse.core.runtime@2:start, [org.eclipse.update.configurator@3:start](#)” and define the product and application.

```
osgi.bundles=org.eclipse.core.runtime@2:start,  
org.eclipse.update.configurator@3:start
```

```
eclipse.product=rcpmail.product
```

```
eclipse.application=rcpmail.application
```

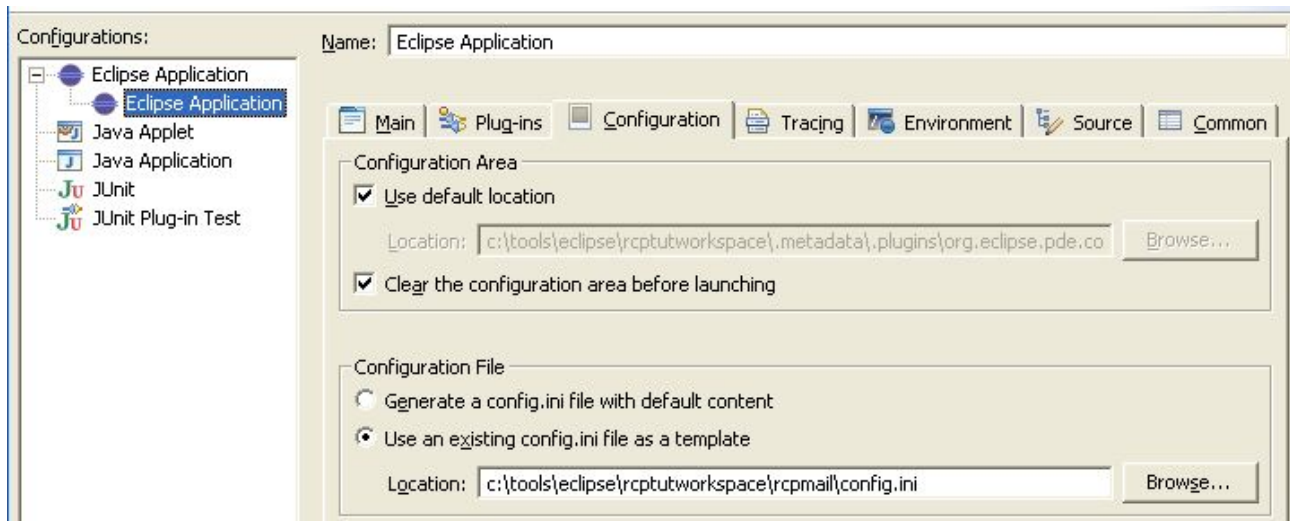
- In the “configuration” tab of the launch configuration switch to this configuration file as a template.



- In the “plug-ins” tab add “org.eclipse.update.configurator” to the plug-in list.
- Launch the new configuration. The workbench appears.

If the launch fails with an error message like, this might be caused by an old and invalid runtime configuration. Whenever the Eclipse runtime starts for the

first time, it creates a list of available plug-ins and stores this information in the so called “configuration area”. This caches information sometimes overrides the config.ini file. The PDE application launcher allows to clean this configuration area before launching the application.



Since the configurator only adds some convenience when adding new plug-ins, it is not absolutely necessary and therefore left out of the RCP distribution. We don't want to modify the config.ini every time we add a new base plug-in, so we let the configurator do the job.

## 8.5 Deployment on RCP distribution

The deployment of the rcpmail application needs a bit more work than the rcpdemo application, since it requires some Eclipse plug-ins, that are not shipped as part of the RCP distribution. Eclipse 3.1M4 does not contain an RCP export wizard that would automate the manual copying of plug-ins in the file system. Such a wizard is planned for later versions of Eclipse.

- Create a new folder <rcpmail-dir>
- Copy the RCP distribution into <rcpmail-dir>
- Copy the plug-ins listed above from the regular Eclipse distribution to <rcpmail-dir>/eclipse/plugins
- Copy the “org.eclipse.update.configurator” plug-in.
- Export the rcpmail and rcpmail projects as directory structure to “<rcpmail-dir>/eclipse”
- Copy the config.ini to <rcpmail-dir>/eclipse/configuration
- Now start the deployed application by invoking eclipse.exe.

## 9 Add Update-Manager to RCA

The update manager helps distribute updates for the application. For the RCP, this is an optional feature, that we want to add to the rcpmail application. The Update Manager requires, that we package our RCA as an Eclipse feature.

By convention, feature project names end in "...-feature". Since we create a feature for our rcpmail application, we give it the name "rcpmail-feature). The feature ID, however, must be "rcpmail" as opposed to the suggestion of the feature creation wizard. The rationale behind this is, that every feature requires a branding plug-in. By default, the ID of the branding plug-in is assumed to be equal to the feature ID. In the feature manifest you could specify a different branding plug-in ID. In our case, the existing rcpmail plug-in is the branding plug-in.

First, create the branding plug-in for the new feature that contain the Eclipse base plug-ins.

- Create a new plug-in project
- Name: rcpmailbase
- Uncheck "Create a Java project"
- Press finish

This branding plug-in is required for the following feature.

- Create an new feature project
- Name: rcpmailbase-feature
- Feature ID: rcpmailbase
- Accept the remaining defaults
- In the list of plug-ins check all plug-ins that you find in the launch configuration except "rcpmail" and "rcphelp".
- Also add the rcpmailbase plug-in to the list. (While you are at it, you might want to add it to the launch configuration for future reference.)

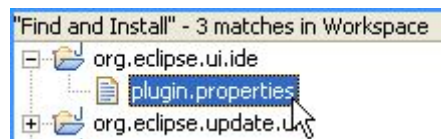
Now we can create the feature that contains our own plug-ins.

- Create an new feature project
- Name: rcpmail-feature
- Feature ID: rcpmail
- Accept the remaining defaults
- In the list of plug-ins check "rcpmail", and "rcphelp".
- Press finished
- In the manifest editor on the content tab, remove the dependencies to required plug-ins and features.
- On the advanced tab add rcpmailbase as included feature.

Like for the help system, we have to add the update manager dialog to the

menu and add some plug-ins to the launch configuration. Let's see, how the Eclipse SDK adds the menu entry (Help > Software Updates > Find and Install) to the workbench.

- Open the file search dialog
- Containing text: "Find and Install"
- File name pattern: "plugin.properties"
- Scope: Workspace (we assume that you have imported all Eclipse base plug-ins into the workspace).



- Open the plugin.properties of org.eclipse.ui.ide
- This text is defined for the variable UpdateActionSet.updates.label
- Open the manifest of that plug-in and look for this label.
- You find an action set definition that adds the menu entries.
- Copy this definition and adapt it to the rcppmail application.

```
<extension
  point="org.eclipse.ui.actionSets">
  <actionSet
    label="%UpdateActionSet.label"
    visible="true"
    id="rcppmail.softwareUpdates">
    <menu
      label="%UpdateActionSet.menu.label"
      id="rcppmail.updateMenu">
      <separator
        name="group0">
      </separator>
    </menu>
    <action
      label="%UpdateActionSet.updates.label"
      icon="icons/usearch_obj.gif"
      class="rcppmail.rcp.InstallWizardAction"
      menubarPath="rcppmail.updateMenu/group0"
```

```
        id="rcpmail.newUpdates">
    </action>
</actionSet>
</extension>
```

- Also copy the variable definitions from the plugin.properties and the icon.

The manifest of `org.eclipse.ui.ide` references the class `org.eclipse.ui.internal.ide.update.InstallWizardAction` to open the install wizard. Since this class lives in a plug-in, that we do not want to depend on, we need to create a local copy of it and reference it from the action definition. You see this reference to the local copy in the snippet above already.

Note: What? Copy and paste programming? You might think, we have lost our minds. But wait a minute. If we want to invoke the install wizard, we need to provide a action that does that job. Such an action exists in Eclipse, but inside a plug-in that we do not want to add to our requirements list (`org.eclipse.ui.ide`). Now, if you review the copied class you will find, that it contains 5 lines of code that actually do something. We balanced this code duplication against the overhead of adding the unwanted plug-in dependency and found that we did the right thing. We even can expect to get the blessings of Erich Gamma and Kent Beck, since we followed their “Monkey see, monkey do” house rule.

This copied class references some code from `org.eclipse.update.ui` plug-in.

- Add it to the dependencies list of the rcpmail application. Save the plug-in manifest.
- Open the launch configuration and in the plug-ins list press “Add required plug-ins”
- Run the launch configuration and open the Install Wizard.

## 9.1 Deployment on RCP distribution

At this point, we need to export the rcpmail feature again to the file system in order to be able to update this feature through the update manager.

To enable later updates of this feature, you need to provide an update site in the feature.xml

- On the “overview” tab, enter an update URL and a discovery URL that points to “<file://<workspace-dir>/rcpmail-site/>”. Please note the slash after “file:” and the slash at the end of the URL (we will create this update site further below).
- Add the following plug-ins to the rcpmailbase feature, since they are required by the update manager

- org.eclipse.ui.forms
- org.eclipse.update.core
- org.eclipse.update.ui
- Export the rcpmail feature. This export automatically includes the rcpmailbase feature, since it is included in the rcpmail feature.
- Run eclipse.exe
- It might be necessary to delete the configuration area, since we added new plug-ins.
- Verify that the install wizard can be started. You won't be able to install an update since there is no update site available yet. This is coming up next.
- In the About dialog verify, that both features are found.

## 9.2 Create update site

An update site contains updated or new version of existing features or completely new features. In order to be able to update the existing rcpmail feature through the update manager, we need an updated feature in the update site.

- Change some code on the rcpmail application (e.g. in `SampleWorkbenchAdvisor>preWindowOpen()` change the initial size of the workbench window).
- Increase the version number in the rcpmail plug-in manifest to 1.0.1.
- Increase the version number in the rcpmail feature manifest to 1.0.1.
- In the content tab of the rcpmail feature editor delete and add the rcpmail plug-in in order to update its version number.

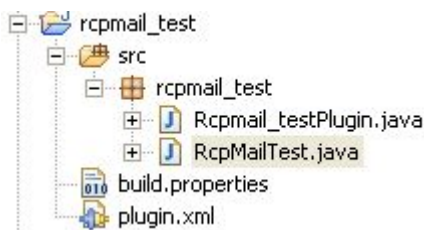
Now that you have an updated feature, you can package this as an update site. Within Eclipse, you can create an update site project for this purpose.

- Create an update site project through the project wizard.
- Name: rcpmail-site
- Add and publish the rcpmail feature
- Save
- Build all
- Start the deployed rcpmail application
- Update the feature through the install wizard
- Restart Eclipse when prompted and see the new size of the workbench window.

## 10PDE test

No application should come without an suite of automated tests. Eclipse itself is heavily covered by unit tests. The tool of choice is PDE JUnit, since it is built into Eclipse and used by the Eclipses developers to create their tests. Here, we only demonstrate a simple example of a PDE JUnit test on the Eclipse UI level.

- Create a new plug-in project
- Name: rcpmail\_test
- ID: rcpmailtest
- In plugin.xml add dependency to org.junit, rcpmail
- Create Junit test case, Name: RcpMailTest



- Create a new test method, e.g. as follows

```
public void testNewView() throws Exception {
    IWorkbenchWindow activeWorkbenchWindow = RcpmailPlugin.getDefault().
getWorkbench().getActiveWorkbenchWindow();
    IWorkbenchPage activePage = activeWorkbenchWindow.getActivePage();
    IViewReference[] viewReferences = activePage.getViewReferences();
    int before = viewReferences.length;

    new OpenViewAction(activeWorkbenchWindow, "", SampleView.ID).run();

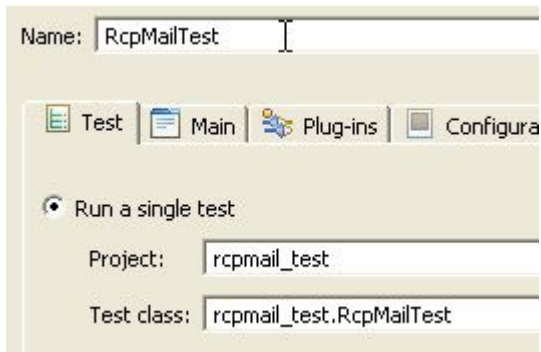
    int after = activePage.getViewReferences().length;

    assertTrue(before<after);
}
```

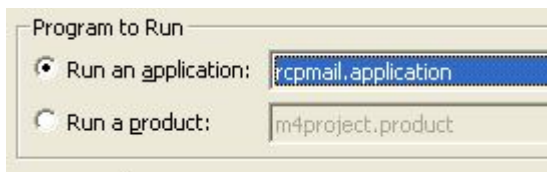
This method tests, if a new view acutally opens in the workbench. It does this by comparing the number of open views before the open attempt and after the open attempt.

- Create a new launch configuration for PDE Junit, Name: RcpMailTest
- In the Test tab, select the test project and the test class just defined

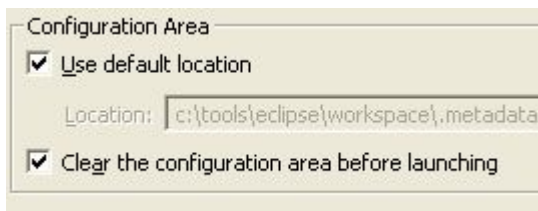
## Tutorial: Developing Eclipse Rich-Client Applications



- In the main tab select rcpmail.application

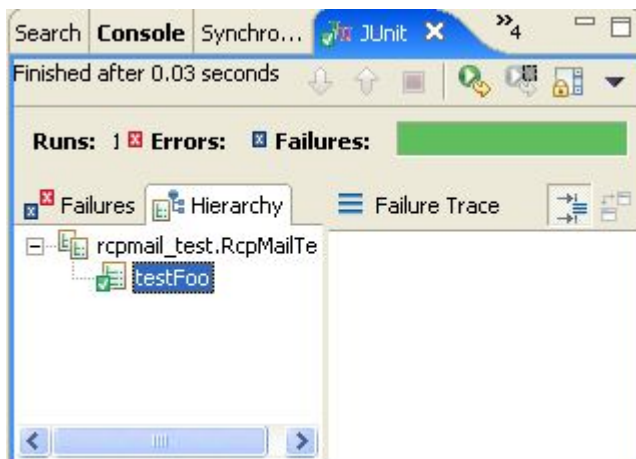


- In the plug-ins tab select “Choose plug-ins and fragments...”
- Deselect all
- Select rcpmail\_test and press “Add required”
- Select “org.eclipse.pde.junit.runtime”, “org.eclipse.core.runtime.compatibility”
- In the configuration tab, check “Clear the configuration area before launching”



- Apply and run. Observe that the rcpmail application window open very briefly.

You should get those test results.





## 11 Build & Test Automation

The goal here is to have a fully automatic build and test for the rcpmail application. Eclipse is built and tested daily and we want to reuse its functionality for that job. Eclipse uses and generates ant scripts for its build.

Some developers suggest to not use the Eclipse-generated build scripts because they don't allow to build tools for different Eclipse releases [QualityEclipse]. In case of RCP-based applications you are likely to ship the complete application instead of providing an add-on to an existing RCP installation.

### 11.1 Build rcpmail Distribution

The most obvious way to build and export an application from within Eclipse is available directly in the UI and you have used those features throughout this tutorial. This convenient functionality is provided by `org.eclipse.pde.ui`. For build automation, however, the use of this functionality is discouraged. The build processes in `org.eclipse.pde.ui` run asynchronously and therefore cannot be used in an ant build script, since it cannot join with other Eclipse processes.

Building Eclipse plug-ins and features requires to read the plug-in and feature manifests (e.g. to translate the plug-in dependencies to a comprehensive classpath). This can be done by functionality in the Eclipse runtime. Therefore, scripts that build Eclipse plug-ins and features execute inside the Eclipse runtime. To invoke this minimal runtime to execute an external Ant script, Eclipse provides the `org.eclipse.ant.core.antRunner` application. This `antRunner` is an RCP-application itself.

The real work of building Eclipse is done by `org.eclipse.pde.build`. See [PDEBuild] to understand how to automate Eclipse-based product builds with PDE build. The basics of how to export RCP applications conveniently is discussed in [ExportRCAs].

PDE Build typically does two jobs:

- Fetch source code from CVS and
- Compile and package the features and plug-ins.

For this tutorial we don't have a CVS repository available and hence we will simulate the fetch operation by copying the necessary source files from the current workspace. Since PDE Build typically performs the fetch operation, we will have to remove this step from the master build scripts simply by commenting out the line. This removal of the fetch step also frees us from some other steps like creating a map file. Please see the articles referenced above to a description of the full blown build process.

- Create a new project called "rcpmail-build" to contain all build resources.

## Tutorial: Developing Eclipse Rich-Client Applications

To modify the master build script, you need to have a copy of it inside your workspace. Import the org.eclipse.pde.build plug-in as copy.

The screenshot shows the Eclipse IDE's 'Import' dialog box. It is divided into three main sections:

- Import From:** This section is checked. It includes a 'Plug-in Location' dropdown menu set to 'c:\tools\eclipse\eclipse-3.1M4'. There are buttons for 'Target Platform...', 'Browse...', 'Source Code Locations...', and 'Environment Variables...'.
- Plug-ins and Fragments to Import:** This section has two radio buttons: 'Select from all plug-ins and fragments found at the specified location' (which is selected) and 'Import plug-ins and fragments required by existing workspace plug-ins'.
- Import As:** This section has three radio buttons: 'Binary projects' (selected), 'Binary projects with linked content', and 'Projects with source folders'.

Open the master build file at org.eclipse.pde.build/scripts/build.xml. Comment out the ant call to fetch the sources from the CVS repository.

```
<target name="main" description="the main build target">
  <antcall target="preBuild" />
<!--
  <antcall target="fetch" />
-->
  <antcall target="generate" />
  <antcall target="process" />
  <antcall target="assemble" />
  <antcall target="postBuild" />
</target>
```

In the same directory you will find template files that you have to adapt to fit your build needs. As we see later, we will require a set of build files for the regular build as well as for the building and performing the tests. Copy the template files rcpmail-build/runtime and rename them to "build.properties" and "customTargets.xml" accordingly. The "build.properties" will stay untouched from now on. In "customTargets.xml" fill out the mandatory targets.

The `allElements` defines, which element is to be built. Specify the rcpmail feature, since it references all required custom code and Eclipse base plug-ins.

```
<target name="allElements">
  <ant antfile="${genericTargets}" target="${target}" >
    <property name="type" value="feature" />
    <property name="id" value="rcpmail" />
  </ant>
</target>
```

```
</ant>
</target>
```

The second target is dependent on the operating system, the windowing system and the platform architecture. For other platforms you will have to modify this target name and some settings further below accordingly.

```
<target name="assemble.rcpmail.win32.win32.x86">
  <property name="archiveName" value="rcpmail-dist.zip"/>
  <ant antfile="${assembleScriptName}"/>
</target>
```

Remove the body of the "getMapFiles" target, since we don't need map files for this patched build process.

```
<target name="getMapFiles">
</target>
```

On the top level of the rcpmail-build project create a build.xml file that will be the rcpmail application main build script.

```
<project default="rcpmail project" default="build.dist">
</project>
```

First, we have to define some property variables:

```
<target name="init">
  <property name="buildDirectory" value="c:/tmp/rcpmail" />
  <property name="baseLocation" value="c:/tools/eclipse/eclipse-3.1M4" />
  <property name="configs" value="win32,win32,x86"/>

  <property name="fetch.dir" location="${basedir}/../"/>
  <property name="output.dir" location="${buildDirectory}/tmp/eclipse" />
  <property name="pluginsdir" value="${buildDirectory}/plugins" />
  <property name="featuresdir" value="${buildDirectory}/features" />
  <property name="testframework.dist" location="c:/software/eclipse/eclipse-
test-framework-3.1M4.zip" />
</target>
```

Modify the variables to fit your environment:

- **buildDirectory** is the folder in which the build will happen and where you will find the fully packaged application.
- **baseLocation** is the folder where you have installed your Eclipse distribution.
- **configs** is the build configuration that denotes the platform. Adapt this variable according to your platform.
- **testframework.dist** locates the Eclipse test framework that we will need to perform the automated tests.

The actual build requires a number of steps to completion. First, do some directory house keeping

```
<target name="build.dist" depends="init">
  <delete dir="${buildDirectory}"/>
  <mkdir dir="${buildDirectory}/${buildLabel}" />

  <antcall target="fetch.prod"/>
  <antcall target="build.prod"/>
  <antcall target="fetch.launcher"/>
  <antcall target="pack.dist"/>
</target>
```

Simulate the fetch of the source code from CVS repository by copying the files from the workspace

```
<target name="fetch.prod" depends="init">
  <copy todir="${pluginsdir}/rcpmail">
    <fileset dir="${fetch.dir}/rcpmail" />
  </copy>
  <copy todir="${pluginsdir}/rcpmailbase">
    <fileset dir="${fetch.dir}/rcpmailbase" />
  </copy>
  <copy todir="${pluginsdir}/rcphelp">
    <fileset dir="${fetch.dir}/rcphelp" />
  </copy>

  <copy todir="${featuresdir}/rcpmail">
    <fileset dir="${fetch.dir}/rcpmail-feature" />
  </copy>
  <copy todir="${featuresdir}/rcpmailbase">
    <fileset dir="${fetch.dir}/rcpmailbase-feature" />
  </copy>
</target>
```

Compile and package the plug-ins and features. To do this, invoke the PDE Build master build file. This takes as argument the directory where it finds additional build files – the files that you have created three pages ago.

```
<target name="build.prod" depends="init">
  <ant antfile="build.xml" dir="../org.eclipse.pde.build/scripts">
    <property name="builder" value="${basedir}/runtime" />
  </ant>
```

```
</target>
```

Copy the launcher and the config.ini to finalize the application.

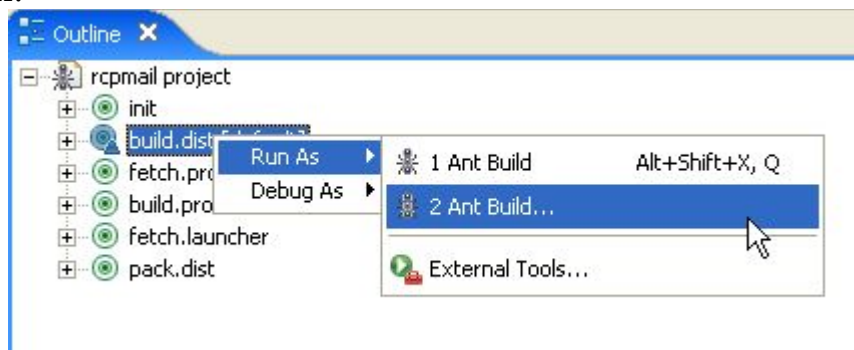
```
<target name="fetch.launcher" depends="init">
  <copy file="${buildDirectory}/plugins/rcpmail/config.ini"
todir="${output.dir}/configuration" />
  <copy file="${eclipse.home}/eclipse.exe" todir="${output.dir}" />
  <copy file="${eclipse.home}/startup.jar" todir="${output.dir}" />
</target>
```

Zip it up to be ready to go.

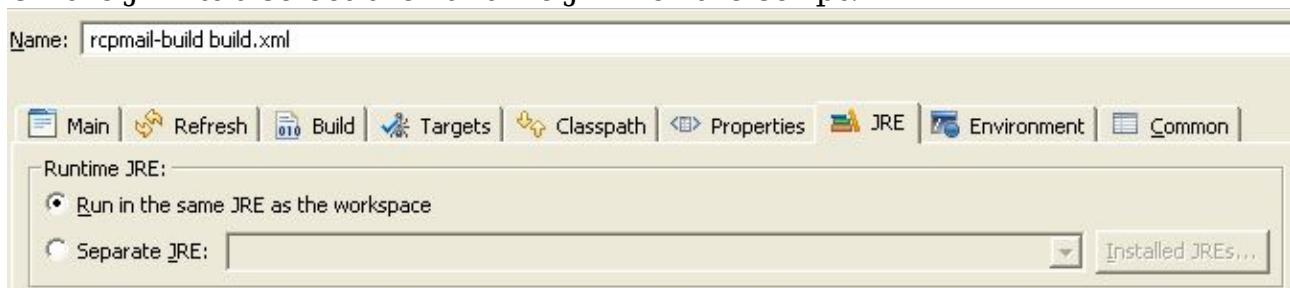
```
<target name="pack.dist" depends="init">
  <zip destfile="${buildDirectory}/rcpmail.zip" basedir="${output.
dir}"></zip>
</target>
```

For this last setp you have to provide the InfoZip zip tool somewhere on your operating system path. Get it from [InfoZip], or from the CD that was distributed during the Tutorial.

Now you can invoke your **build.xml** from the outline view. Ensure that the script runs inside the Eclipse JVM. To do that, create a new launch configuration.



On the JRE tab select the runtime JRE for the script:



Test your application inside the build directory. You will also find the rcpmail.zip file that contains your ready-to-go application.

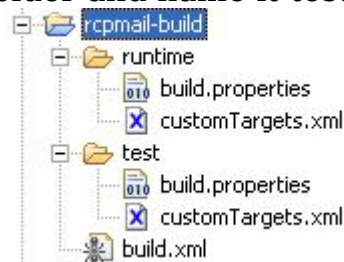
## 11.2 Build & Perform Tests

For building the test plug-in, we will wrap it into a feature similar to how we wrapped the application code.

- Create a new feature
- Name: rcpmail\_test-feature
- ID: rcpmailtest
- On the content tab enter org.apache.ant, org.junit, rcpmailtest. The rcpmail test plug-in requires those two Eclipse base plug-ins and by entering them into the feature, they will automatically be deployed during the application build.
- Empty the list of required features/plug-ins
- On the advanced tab enter rcpmail as included feature.

Now we can come back to the build script itself. In the rcpmail-build project we have a build configuration for the runtime already. For building the tests, we need another build configuration.

- Duplicate the runtime folder and name it test.



The build.properties can stay untouched. Adapt the mandatory targets in customTargets.xml. This time we want to build the test feature instead of the production feature.

```
<target name="allElements">
  <ant antfile="${genericTargets}" target="${target}" >
    <property name="type" value="feature" />
    <property name="id" value="rcpmailtest" />
  </ant>
</target>
```

The tests are still platform dependent.

```
<target name="assemble.rcpmailtest.win32.win32.x86">
  <property name="archiveName" value="rcpmailtest-dist.zip"/>
  <ant antfile="${assembleScriptName}"/>
</target>
```

For the test build we extend the already existing build.xml script. The main test target leverages the existing production build.

```
<target name="build.dist.and.test" depends="init">
  <antcall target="build.dist"/>
  <antcall target="fetch.test"/>
  <antcall target="build.test"/>
  <antcall target="fetch.launcher"/>
  <antcall target="fetch.test.framework"/>
  <antcall target="fetch.test.patches"/>
  <antcall target="test.rcpmail"/>
</target>
```

Again, the fetch from the CVS repository is simulated by plain vanilla file copy.

```
<target name="fetch.test" depends="init">
  <copy todir="${plugindir}/rcpmailtest">
    <fileset dir="${fetch.dir}/rcpmail_test" />
  </copy>

  <copy todir="${featuresdir}/rcpmailtest">
    <fileset dir="${fetch.dir}/rcpmail_test-feature" />
  </copy>
</target>
```

The test build is equal to the production build, except that it uses another test configuration.

```
<target name="build.test" depends="init">
  <ant antfile="build.xml" dir="../../org.eclipse.pde.build/scripts">
    <property name="builder" value="${basedir}/test" />
  </ant>
</target>
```

The Eclipse test framework must be unzipped into the application's eclipse folder.

```
<target name="fetch.test.framework" depends="init">
  <unzip dest="${output.dir}/../" src="${testframework.dist}"></unzip>
</target>
```

The Eclipse test framework was built to create the Eclipse distribution itself. We reuse it for doing something that is was never created for – build an RCP application. In order for it to work, we need to patch some files.

- Create a directory structure like outlined below and copy the files from the

base product locations as indicated by the folder names.



- In the library.xml file add the following line indicated in italics:

```
<arg line="-application ${application}"/>  
<arg line="-testApplication ${testApplication}"/>  
<arg line="-dev bin -data ${data-dir}"/>
```

- In the plugin.xml file add the dependency to the org.junit plug-in:

```
<requires>  
  <import plugin="org.junit"/>  
</requires>
```

Now back to the build.xml in the rcpmap-build project. We have to apply those patches to the build directory.

```
<target name="fetch.test.patches" depends="init">  
  <copy todir="${output.dir}" overwrite="true">  
    <fileset dir="patches/eclipse">  
      <include name="*" />  
    </fileset>  
  </copy>  
</target>
```

Everything is set for the test to happen. Here we go.

```
<target name="test.rcpmail" depends="init">  
  <property name="library-file" location="${output.dir}/  
plugins/org.eclipse.test_3.1.0/library.xml" />  
  
  <ant target="ui-test" antfile="${library-file}" dir="${output.dir}"  
inheritall="false">  
    <property name="os" value="win32" />  
    <property name="ws" value="win32" />  
    <property name="arch" value="x86" />  
    <property name="testApplication" value="rcpmail.application" />  
    <property name="data-dir" value="runtime-test-workspace" />  
    <property name="plugin-name" value="rcpmailtest" />  
    <property name="classname" value="rcpmail_test.RcpMailTest" />
```



```
        <property name="vmargs" value="-Dbaz=true" />
    </ant>

    <echo message="The test results can be found in ${output.dir}" />
</target>
```

This was the last step. Execute this build script as an external Ant build. You can read the test results at the indicated location and find the rcpmail distribution in the rcpmail-build project.

Alternatively, open a command shell on the directory that contains your rcpmail-build plug-in and enter

```
java -cp <eclipse-dir>\eclipse-3.1M4\startup.jar org.eclipse.core.launcher.Main
-data c:\tmp\workspace -application org.eclipse.ant.core.antRunner
build.dist.and.test
```

That's it.

## 12References

There is no book available, that specifically concentrates on the RCP. The RCP is mainly a repackaging of existing functionality. Therefore, the existing books that talk about how to extend Eclipse are still valid. There are a few publications, that deal with the RCP, though. Find them below as well as all other references from above.

### 12.1Eclipse

[RCP Help]            Online RCP help in Eclipse distribution.

[Examples]            Download from eclipse.org: eclipse-examples-3.1\*.zip

And don't forget the source code.

### 12.2Web sites

[RCP Homepage]    [http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/rcp-proposal/rich\\_client\\_platform\\_facilities.html](http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/rcp-proposal/rich_client_platform_facilities.html)

[EclipsePowered]    Eclipse Powered by Ed Burnette  
[www.eclipsepowered.org](http://www.eclipsepowered.org)

[RCP Tutorial]     RCP Tutorial by Ed Burnette  
<http://dev.eclipse.org/viewcvs/index.cgi/%7echeckout%7e/org.eclipse.ui.tutorials.rcp.part1/html/tutorial1.html>

[EclipseWiki]      Eclipse Wiki  
[eclipse-wiki.info](http://eclipse-wiki.info)

[InfoZip]           <http://www.info-zip.org/pub/infozip/>

### 12.3 News groups

- [Platform news]     General discussion about the Eclipse-Plattform  
news://news.eclipse.org:119/eclipse.platform
- [RCP news]           Discussion about the Eclipse-Plattform  
news://news.eclipse.org:119/eclipse.platform.rcp
- [Equinox news]       Discussion about OSGi based runtime  
news://news.eclipse.org:119/eclipse.technology.equinox

### 12.4 Presentations

- [Edgar]              Nick Edgar: Eclipse Rich Client Applications. Presentation  
at EclipseCon 2004  
www.eclipsecon.org

### 12.5 Articles

- [GerhardtWege]     Gerhardt, Wege: Eclipse als Basis für Rich-Client-  
Anwendungen. iX, 7/2004.
- [Williams]           Todd Williams, The Case for Using Eclipse Technology in  
General Purpose Applications  
<http://www.genuitec.com/products/eclipseapplicationframework.pdf>
- [PDEBuild]          Sonia Dimitrov and Pascal Rapicault, Automating Eclipse  
Based Products Builds with PDE Build.  
<http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/pde-build-home/articles/Automated%20Builds/article.html?rev=HEAD&content-type=text/html>
- [ExportRCAs]        Pascal Rapicault, Exporting an RCP Application.  
<http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/pde-build-home/articles/export%20rcp%20apps/article.html?rev=HEAD&content-type=text/html>

### 12.6 Books

- [DevGuide]          Sherry Shavour et al.: The Java Developer's Guide to  
Eclipse. Addison-Wesley, 2004, 2<sup>nd</sup> edition.
- [Contrib]            Erich Gamma, Kent Beck: Contributing to Eclipse.  
Addison-Wesley, 2003.
- [QualityEclipse]    Eric Clayberg, Dan Rubel: Eclipse - Building Commercial-  
Quality Plug-ins. Addison-Wesley, 2004.