Fundamentals of Plug-in and RCP Development

Chris Aniszczyk  IBM Lotus, Austin
Brian Bauman  IBM Lotus, Austin
Wassim Melhem  IBM Rational, Toronto
Mike Pawlowski  IBM Rational, Toronto
Tutorial Outline

**The Basics**
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
- The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
The Basics
What is Eclipse?

A very popular Java™ IDE
and much more…
What is Eclipse?

An open platform for anything and nothing in particular
An Open Platform

Eclipse is designed to be easily and infinitely extensible by third parties

Diagram:

- Tool X
- Tool Y
- Product A
- Product B
- Eclipse SDK
An Open Platform for anything

IBM® Rational© Application Developer (RAD):
An Integrated Development Environment (IDE)
An Open Platform for anything

Azureus: a Java BitTorrent client
An Open Platform for anything

IBM Lotus SameTime 7.5: a chat client
An Open Platform for anything

Games!: Sudoku
An Open Platform for Nothing in Particular

- No bias in the platform toward any particular domain or discipline
- Eclipse plug-in development is a level playing field
Eclipse.org

An open source community that hosts over 60 open source projects
Downloading and Running the Eclipse SDK

1. Download & Install a Java™ Runtime Environment (JRE)
2. Download an Eclipse SDK
   ➤ http://download.eclipse.org/eclipse/downloads/
3. Unzip the Eclipse SDK archive
4. Run the Eclipse executable
Supported Platforms

- Windows™
- Solaris 8™
- Mac OSX™
- AIX™
- Linux™
- HP-UX™
Eclipse on Windows™ XP
Eclipse on Windows Vista™
Eclipse on Linux™
Eclipse on Mac OSX™
Inside the Eclipse SDK

- RCP provides the architecture and frameworks to build any rich client application
- IDE is a tools platform and a rich client application in itself
- JDT is a complete Java IDE and a platform in itself
- PDE provides all the tools necessary to develop plug-ins and RCP applications
Rich Client Platform (RCP)

- Equinox is the runtime
- Standard Widget Toolkit (SWT) is a portable and native widget toolkit for Java
- JFace is a framework for common UI programming tasks
- Generic Workbench provides the UI personality of the Eclipse platform
Integrated Development Environment (IDE)

- The IDE Workbench defines the Eclipse presentation

- IDE is an open tools platform:
  - Resource management
  - Text editing framework
  - A Language-independent debug model
  - Ant integration
  - Team repository integration
  - Help system
  - Update manager
Java Development Tools (JDT)

- JDT provides a complete Java IDE
- The compiler, which operates in incremental and batch modes, is also available as a separate download
- JDT is extensible:
  - Search and refactoring participants
  - Quick-Fix processors
  - Code Formatters
  - etc…
Plug-in Development Environment (PDE)

- PDE Does Plug-ins
- PDE Does RCP
- PDE Does Features and Update Sites
- PDE Does OSGi
- PDE Does User Assistance (as of Eclipse 3.3)
Plug-ins All the Way Down

- A plug-in is the fundamental building block of an Eclipse product
- Plug-ins build on top of and use other plug-ins
- To extend Eclipse, you must write plug-ins
- To write a rich client application, you must write plug-ins
Layout of an Eclipse product

- **eclipse**: Root installation directory, contains the eclipse executable.
- **configuration**: Contains runtime metadata read on eclipse startup.
- **features**: Contains all installed features (logical grouping of plug-ins).
- **plugins**: Contains all installed plug-ins.
- **readme**: Contains release notes.
Anatomy of a Plug-in
A Fundamental Building Block

- A plug-in is a Java Archive (JAR)

- A plug-in is self-contained
  - houses the code and resources that it needs to run

- A plug-in is self-describing
  - who it is and what it contributes to the world
  - what it requires from the world
A Tale of Two Manifest Files

**MANIFEST.MF**

- ID
- Version
- Name
- Code Location
- Dependencies
- Exports

**plugin.xml**

- Extension Points
  [ 0 or more ]
- Extensions
  [ 0 or more ]
A Mechanism for Extensibility

- Extensibility in Eclipse is achieved via loose coupling
- Plug-in A exposes an extension point (the electric outlet)
- Plug-in B extends plug-in A by providing an extension (the plug) that fits into plug-in A’s outlet
- Plug-in A knows nothing about plug-in B
If the Extension Fits…

- So many extension points…
- Each extension point is unique
- Each extension point declares a contract
- The extension point provider accepts only extensions that abide to the terms of its contract
A Declarative Approach

- Extension points and extensions are declared in the plugin.xml file

- The runtime is able to wire extensions to extension points and form an extension registry using XML markup alone
Extensibility in Pictures

1. Query the registry for registered compliant extensions
2. Present extensions based on markup
3. Load classes only when the extension is needed
Extensibility in Action

- Plug-ins may contribute preference pages
- All preference pages are assembled and categorized in the Preferences dialog
- How is the Preferences dialog created?
- How and when is a particular preference page created?
The Electric Outlet and the Plug

```xml
<extension-point
    id="preferencePages"
    name="Preference Pages"
    schema="schema/preferencePages.exsd"/>

<extension
    point="org.eclipse.ui.preferencePages">
    <page
        class="org.eclipse...MainPreferencePage"
        id="MainPreferencePage">
        name="Plug-in Development"
    </page>
...
</extension>
```
Lazy Loading

1. Create the preference page dialog
2. Retrieve all contributions for org.eclipse.ui.preferencePages
3. Generate the preference page index
4. Get the extension contributing the Plug-in Development preference page
5. Ask the extension registry to load and instantiate the class org.eclipse...MainPreferencePage
6. Generate the main PDE preference page
Create the Preferences Dialog (1/3)

- The UI plug-in provides the `org.eclipse.ui.preferencePages` extension point
- The UI plug-in first creates an empty Preferences dialog
- Now the dialog needs to be populated…
Generate the Preference Page Index (2/3)

- The UI plug-in queries the extension registry for all `org.eclipse.ui.preferencePages` extensions.

- The preference page index is then generated using the XML markup only:
  - Names for available preference pages are displayed in the tree using the `name` attribute.
  - The `category` attribute is used to categorize the pages.
Create the *Plug-in Development* Preference Page (3/3)

- When the *Plug-in Development* preference page gets selected, the UI plug-in asks the extension registry to load and instantiate the Java class specified by the `class` attribute of the corresponding extension.

- The class gets loaded and the preference page gets created.

- The plug-in providing that extension (i.e. the `org.eclipse.pde.ui` plug-in) may then get activated, if it’s not already active.
Tip of the Iceberg

- Plug-ins are connected without loading any of their code
- Code is loaded only when it is needed
- The lightweight declarative and lazy approach scales well
- An installed plug-in is not necessarily an active plug-in
A Society of Plug-ins

- An Eclipse product is the sum of its constituent plug-ins
- Plug-ins are discovered upon Eclipse startup
- Plug-ins do not know how to play and interact with each other on their own
An Ordered Society of Plug-ins

- The Eclipse runtime manages all installed plug-ins and brings order and collaboration to their society.

- A classpath for each plug-in is dynamically constructed based on the dependencies declared in its MANIFEST.MF file.

- Every plug-in gets its own classloader.
Unresolved Plug-ins

- If a plug-in has a dependency that is not met, the plug-in is deemed UNRESOLVED

- An unresolved plug-in does not get to interact with the rest of the plug-ins
A Chain Reaction
Resolving the Unresolved
Seamless Integration of Components
Presentation Outline

- The Basics
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
- The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
Exercise One: The Eclipse Browser Plug-in
Part I: Create the Eclipse Browser View

- This exercise is structured as a 5-step cheat sheet
- You use the plug-in import wizard to import the plug-in into the workspace
- You use the plug-in manifest editor to define the extension
- You use the Eclipse Application launcher to test the plug-in
Import the Eclipse Browser Plug-in

- The plug-in import wizard brings a plug-in from the file system into the workspace.
- The plug-in is converted from its deployed form (a JAR) to its development form (a workspace project).
- Choose to import the plug-in as “Project with source” if you wish to modify it.
Add a View Extension

- To create a view to the workbench, you must extend the `org.eclipse.ui.views` extension point.
Define the Eclipse Browser View

- The **name** and **icon** attributes are sufficient to put a placeholder for the view in the workbench.
- The **class** is loaded only when the view is opened by the user.
Test the Plug-in

- PDE launches a second Eclipse instance to show your plug-in in action
- Second instance uses a different workspace (i.e. a sandbox)
Part II: Extend the Eclipse Browser View

- This exercise is structured as a 4-step cheat sheet
- You create a new plug-in that extends the `org.eclipse.browser` plug-in
- You create an extension that extends the Eclipse Browser view by contributing a link to it
Create a Plug-in Project

- Recommended to use the reverse domain naming convention to name the plug-in project

- Whether or not a plug-in project is a Java project depends on whether it will contribute code

- Plug-ins contributing Help documentation, for example, do not contribute code
The Contract

- To contribute a new link to the Useful Links section of the Eclipse Browser view, you must extend and conform to the grammar of the org.eclipse.browser.usefulLinks extension point.
The Contributed Link in Action

- The new link contributed by the `org.eclipse.browser.extension` plug-in integrated seamlessly with the links provided by the `org.eclipse.browser` plug-in.
Presentation Outline

- The Basics
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
  The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
The Plug-in Manifest Editor
General Information

- A plug-in must have an ID, version and a name
- A platform filter is an optional field to specify under what conditions the plug-in should be allowed to run
- An activator controls the plug-in’s lifecycle and may do initialization upon startup and cleaning up at shutdown
Execution Environment

- An Execution Environment is the minimum JRE level required for a plug-in to run.
- If a plug-in declares a J2SE-1.5 Execution Environment and Eclipse is running using a 1.4 JRE, the plug-in gets disabled gracefully.
Dependencies

- A plug-in must list all plug-ins that it needs to compile
- The runtime and development classpaths are computed based strictly on dependencies in the MANIFEST.MF
- PDE manages and updates the development classpath for you
- All plug-in dependencies must be met before a plug-in is resolved
Exported Packages

- A plug-in may expose its code to downstream clients
- Downstream plug-ins may then make a dependency on the plug-in and use code from it
Extensions

- The Extensions section lists all the contributions the plug-in makes to Eclipse.
- The plug-in manifest editor makes creating extensions easy because it is aware of the XML schema for all available extension points.
- Hot links are available to jump back and forth between the manifest files and the source code.
Extension Points

A plug-in may contribute 0 or more extension points to the platform

The Eclipse SDK provides hundreds of extension points
Presentation Outline

- The Basics
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
- The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
Development Lifecycle of a Plug-in
From Genesis to Deployment

1. Create the plug-in project
2. Edit the manifest & write the Java code
3. Test & debug the plug-in
4. Externalize the strings
5. Clean-up the manifest
6. Configure the build content
7. Export the plug-in
Plug-in Creation

- The *New Plug-in Project* creation wizard generates a project complete with manifest files and, optionally, source code.

- The wizard also provides templates for popular extension points such as action sets, views, preference pages.

- Templates save a lot of time and allow you to create and run a plug-in in a few minutes.
Life in the Workspace

- The internal structure of a plug-in project in the workspace mirrors that of a deployed plug-in.

- Two notable differences:
  1. The code is in source folders.
  2. The plug-in project contains extra development metadata that are not part of the deployed plug-in.
Editing the Plug-in

- The plug-in manifest editor is the central place to manage your plug-in
- It provides hot links to
  - test and debug plug-ins
  - launch relevant wizards
  - quick navigation between source code and the manifest files
Testing the Plug-in
Configure the Build Content

- The plug-in project contains development-time metadata that should not be part of the deployed plug-in.

- On the Build page of the plug-in manifest editor, you check the list of files and folders that should be packaged.
Exporting the Plug-in

- The Plug-in Export wizard packages a plug-in into a deployable format
- Plug-ins can be exported en masse
- Plug-ins can be exported as an archive or as a directory structure
Externalize the Strings

- PDE provides an *Externalize Strings* wizard that extracts translatable strings and stores them in a properties file for multilanguage support.

- This allows the plug-in manifest files to remain intact, while the properties files get translated
Clean up the Manifests

- As the plug-in evolves, it may accumulate stale data

- The Organize Manifests wizard that inspects your code and manifests and removes or updates stale data
From Genesis to Deployment

1. Create the plug-in project
2. Edit the manifest & write the Java code
3. Test & debug the plug-in
4. Externalize the strings
5. Clean-up the manifest
6. Configure the build content
7. Export the plug-in
Presentation Outline

The Basics

Anatomy of a Plug-in

Exercise One: The Eclipse Browser Plug-in

The Plug-in Manifest Editor

The Development Lifecycle of a Plug-in

The Rich Client Platform

Developing for the Rich Client Platform

Exercise Two: The Eclipse Browser Product
The Rich Client Platform
Rich Client Platform (RCP)

- Equinox is the runtime
- Standard Widget Toolkit (SWT) is a portable and native widget toolkit for Java
- JFace is a framework for common UI programming tasks
- Generic Workbench provides the UI personality of the Eclipse platform
Equinox

- OSGi is a framework that manages bundles
- Plug-in == Bundle
- The extension registry manages extensions and extension points
- The concurrency infrastructure allows for running background jobs
- Other runtime facilities include tracing, logging and preferences
Standard Widget Toolkit (SWT)

- SWT is a low-level graphics library that provides UI controls such as buttons, trees, combo boxes,…
- SWT uses native widgets as much as possible
- SWT has OS-independent API and is thus portable
- SWT is independent of the Eclipse runtime
JFace

- Built on top of SWT, JFace adds the model layer to the SWT widgets, e.g. tree viewers
- JFace provides common UI constructs such as wizards and dialogs
- JFace can be used standalone without the need for the Eclipse runtime
Generic Workbench

- The Workbench defines common user-defined paradigms:
  - Views, e.g. the Package Explorer
  - Editors, e.g. the Java and plug-in manifest editors
  - Perspectives: arrangement of views and editor
Contribution-Based Extensibility
Presentation Outline

- The Basics
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
- The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
Why Use Eclipse RCP?

- An elegant plug-in architecture

- Eclipse RCP does the middleware. You do your job.

- From servers to embedded devices, RCP applications are portable

- RCP applications provide a native user experience

- JDT and PDE provide a first-class development environment
A Plug-in is a Plug-in is a Plug-in

- Developing plug-ins for a rich client application is identical to writing plug-ins for the Eclipse SDK

- Notable differences include:
  - Target Platform
  - Workbench configuration
  - Defining an application
  - Defining a product
A Smaller Target
Customizing the Generic Workbench

- When writing a plug-in for the SDK, you can extend (i.e. add) to the workbench, but you cannot remove or override.

- When writing an RCP application, you can configure every aspect of the workbench.
Applications

- An application is to Eclipse what to the `main()` method is to a regular Java program
- To run Eclipse, an application has to be specified
- When you launch Eclipse, an application supplied by the IDE is run
- In an RCP scenario, you supply your own application
Products

- A product is the Eclipse unit of branding
- Branding gives the rich client application a personality
- Branding encompasses window images, splash screen, a custom launcher
- A product is defined declaratively as an Eclipse extension
Presentation Outline

- The Basics
- Anatomy of a Plug-in
- Exercise One: The Eclipse Browser Plug-in
- The Plug-in Manifest Editor
- The Development Lifecycle of a Plug-in
- The Rich Client Platform
- Developing for the Rich Client Platform
- Exercise Two: The Eclipse Browser Product
Exercise Two: The Eclipse Browser Product
Part I: Create an Eclipse Application

- This exercise is structured as a 3-step cheat sheet
- You use the plug-in manifest editor to create an application extension
- You launch the new application and you see that the same Browser view has seamlessly integrated into a different application
Define an Application Extension

- An application is an `org.eclipse.core.runtime.applications` extension
- It specifies a class which serves as the entry point to the application
- For a typical UI application, the application creates, configures and runs a workbench
- The application exits when the workbench exits
Eclipse Browser Application

- The RCP application provides a standalone window with File and Help menus.
- The Browser plug-in integrates seamlessly with the application without any code changes.
- Don’t underestimate this minimalistic application. It can go toe-to-toe with any other RCP application. It is configurable, extensible, …
Part II: Create an Eclipse Product

- This exercise is structured as a 8-step cheat sheet

- You use the product editor to define every aspect of your product: launcher, window images, About Dialog

- You export and run a fully-branded standalone product
A New Product Configuration

- A product configuration is the central place to manage all aspects of your product.
- A product configuration is used by PDE to define and assemble a product.
- A product configuration is neither read nor interpreted by the runtime.
Product Definition

- A product is associated with an application
- A product provides branding and customization for the application
- A product name appears in the title bar of the application
- PDE uses this data to create an `org.eclipse.core.runtime.products` extension in the plug-in’s manifest file
Building Blocks

- A product must list all its constituent plug-ins. This list is only used by PDE to determine what to build and package.

- Plug-ins that are in source form in the workspace are compiled and packaged.

- Plug-ins that are already built (e.g. target plug-ins) are assembled as-is into the final product.
Window Images

- Window Images are shown in the application window, task bar, ... depending on the windowing system
- On Windows, the 16x16 GIF image is used for the task bar and the 32x32 GIF is used in the Alt-Tab application switcher
Customizing the About Dialog

About Dialog
Customize the text and image of the About dialog. The GIF image is typically located in the product's defining plug-in and its size must not exceed 500x330 pixels. The text is not shown if the image size exceeds 250x330 pixels.

Image: /org.eclipse.browser/branding/world_about.gif
Text: This is a blurb about my product.

About Eclipse Browser Product
This is a blurb about my Eclipse Browser Product
One-click Export

- PDE provides a Product Export wizard that takes a product configuration file as input
- The name of the root directory of the product is customizable
- Options to export the product as a directory structure or an archive
One-click Export to Multiple Platforms

- The RCP delta pack is a separately downloaded archive that contains all OS-specific fragments and executables.

- When present, you are able to export your plug-ins to all supported platforms in a single step.
A Finished Product

![Eclipse Browser](image)

Eclipse Browser is a finished product that demonstrates the capabilities of Eclipse in providing a browser-like experience.

- **Project Links**
  - Eclipse Project
- **Useful Links**
  - Eclipse Downloads
  - Eclipse Bugs
  - Eclipsepedia

This image illustrates how Eclipse can be used to create applications that mimic the functionality of a browser.
The End
Legal Notices

- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
- IBM, Lotus, Rational are copyrights of IBM Corporation in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others.