Developing Eclipse Plug-ins*

Wolfgang Emmerich
Professor of Distributed Computing
University College London
http://sse.cs.ucl.ac.uk

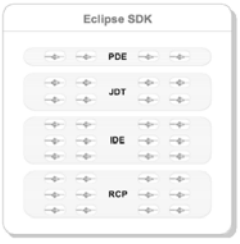
* Based on M. Pawlowski et al: Fundamentals of Eclipse Plug-in and RCP Development. EclipseCon 2007.

Learning Objectives

- Understand how to write new software development tools and integrate them into the Eclipse platform
- Understand the Eclipse extension mechanisms
- Know how to specify an OSGi manifest
- Be able to develop, test and deploy a plug-in.

2

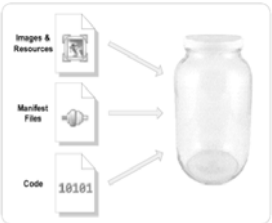
Any Eclipse product is composed of plug-ins



- 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

3

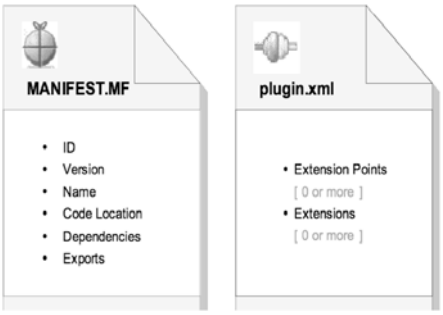
A Fundamental Building Block



- A plug-in is a **J**ava **Ar**chive (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

4

A Tale of Two Manifest Files



5

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

6

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

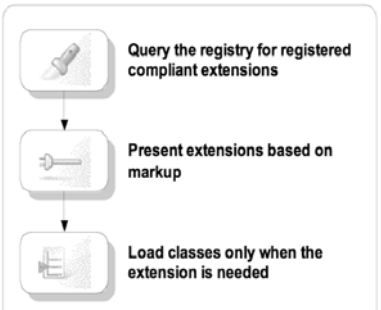
7

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

8

Extensibility in Pictures



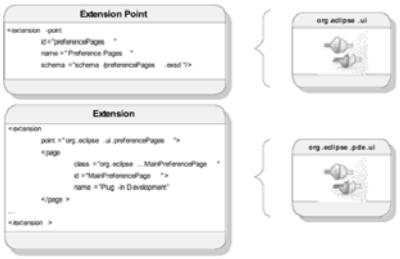
9

Example of Extensibility

- 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?

10

The Electric Outlet and the Plug



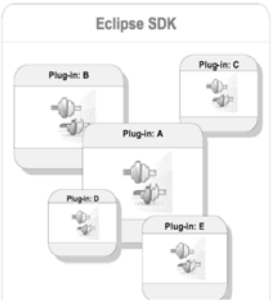
11

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

12

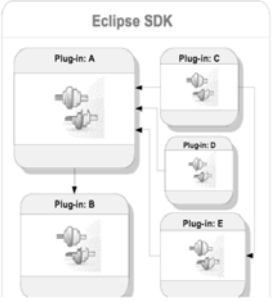
A Society of Plug-ins



- 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

13

An Ordered Society of Plug-ins



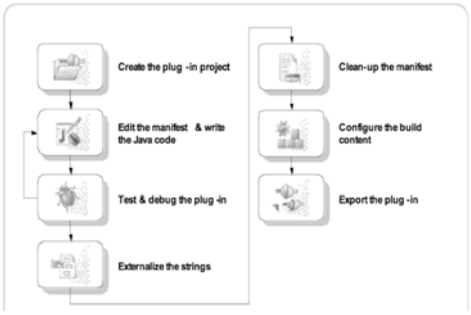
- The Eclipse runtime manages all installed plug-ins and brings order and collaboration to their society
- A classpath for each plug-in is dynamically constructed based on the dependencies declared in its MANIFEST.MF file
- Every plug-in gets its own class loader

14

Seamless Integration of Components

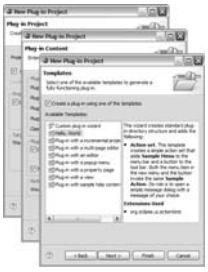
15

From Genesis to Deployment



16

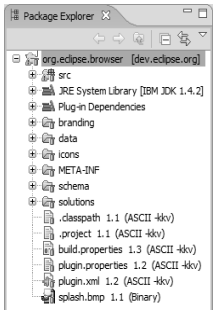
Plug-in Creation



- 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

17

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:
  - The code is in source folders
  - The plug-in project contains extra development metadata that are not part of the deployed plug-in

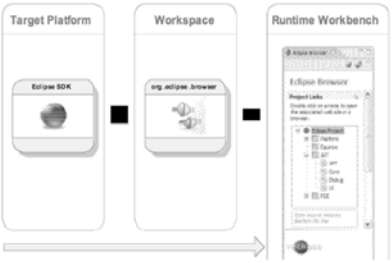
18

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

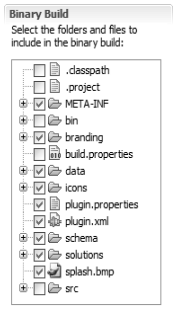
19

Testing the Plug-in



20

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

21

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

22

Externalize the Strings

- PDE provides an *Externalize Strings* wizard that extracts translatable strings and stores them in a properties file for multi-language support.
- This allows the plug-in manifest files to remain intact, while the properties files get translated

23

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

24

Key Points

- Eclipse products are composed of plug-ins
- Plug-ins use and provide extension mechanisms
- Plug-ins make contributions to all parts of the UI
- Plug-ins are insulated from each other through OSGi

25

References

- M. Pawlowski et al: Fundamentals of Eclipse Plug-in and RCP Development. EclipseCon 2007.
  http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3645

26