

Eclipse Plug-ins

Instructor: Yongjie Zheng
February 16, 2016

CS 490MT/5555
Software Methods and Tools

Eclipse Plug-ins

- Plug-in Structure
 - *MANIFEST.MF* and *plugin.xml*
- Plug-in Manifest
 - Plug-in declaration
 - Plug-in runtime
 - Plug-in dependencies
 - Extensions and extension points
- Activator or Plug-in Class
- Development of Plug-ins

Plug-in Structure

- The installed Eclipse plug-ins can be found in the *plugins* directory of the *Eclipse* folder.
- The plug-in JAR name must be a concatenation of the plug-in identifier, and underscore, and the plug-in version in dot separated form. For example,
- `org.eclipse.ui_3.103.0.v20120521-2329.jar`

Plug-in Structure, cont.

- A plug-in JAR typically includes
 - Java class files: the implementation of the behavior of the plug-in.
 - Icons and other resources
 - META/MANIFEST.MF: a file describing the runtime aspects of the plug-in such as identifier, version, and plug-in dependencies.
 - plugin.xml: a file in XML format describing extensions and extension points.

Plug-in Structure, cont.

- Eclipse plug-ins are loaded lazily on an as-needed basis, thus reducing both the startup time and the memory usage of Eclipse.
- On startup, plug-in loader scans the *MANIFEST.MF* and *plugin.xml* files for each plug-in and then builds a structure containing the information.
- The code of an Eclipse plug-in is loaded only when the plug-in is activated (i.e. its functionality is needed at runtime).
- Eclipse plug-ins are loaded but not unloaded.

Plug-in Manifest

- There are two plug-in manifest files: *META-INF/MANIFEST.MF* and *plugin.xml*, which define how this plug-in relates to all the others in the system.
- Eclipse provides a plug-in manifest editor for developers to edit the information about
 - Plug-in declaration
 - Plug-in runtime (e.g. Export-Package)
 - Plug-in dependencies
 - Extensions and extension points

Plug-in Declaration

- Plug-in identifier: uniquely identifies the plug-in and is typically constructed using Java package naming conventions (e.g. edu.umkc.<projectName>).
- Plug-in version: three numbers separated by periods.
 - The first number indicates the major version; the second number indicates the minor version; the third number indicates the service level.
 - You can specify an optional qualifier that can include alphanumeric characters. E.g. 1.0.0.beta
 - At startup, if there are two plug-ins with the same identifier, Eclipse will choose the “latest” by comparing the major version number first, then the minor version number, and so on.

Plug-in Declaration, cont.

- Plug-in name and provider: both are human-readable text, can be anything, and are not required to be unique.
- Plug-in activator: every plug-in optionally can declare a class that represents the plug-in from a programmatic standpoint. This class is referred to as an *Activator*.

Plug-in Runtime

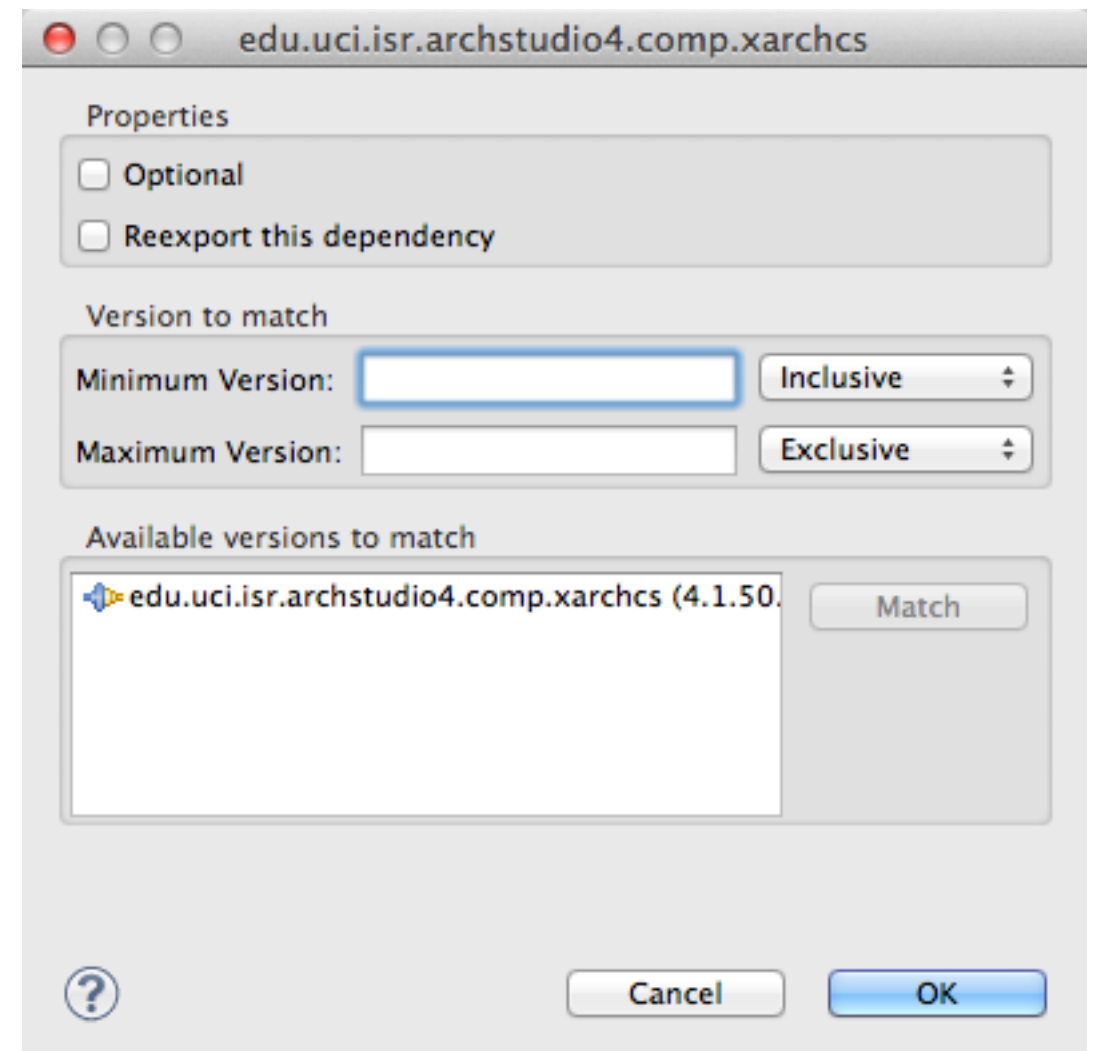
- Bundle-Classpath: this property is usually omitted.
- Export-Package: a comma-separated list indicating which packages of the plug-in are accessible to other plug-ins.
- All other packages will be hidden from clients at all times.
- This corresponds to the “export” property of the OSGi bundle.

Plug-in Dependencies

- The plug-in loader instantiates a separate class loader for each loaded plug-in, and uses *Require-Bundle* or *Import-Package* to determine which other plug-ins/classes will be needed.
- Require-Bundle: specifies the names of the required plug-ins or bundles.
 - Can be seen as required service providers.
- Import-Package: specifies the names of the packages that are required.
 - Can be seen as required services.
 - Not used as much as Require-Bundle.

Plug-in Dependencies, cont.

- If a plug-in requires not just any version of another plug-in, select that plug-in in the *Dependencies* tab of the plug-in manifest editor and click the *Properties...* button. This opens the required plug-in properties dialog where you can specify an exact version or a range of versions.



Plug-in Dependencies, cont.

- About *Java Build Path* and the *Dependencies* declaration.
- Java build path specifies the compile-time classpath of a plug-in project.
- The dependency list specifies the runtime classpath of a plug-in project.
- Any changes to the dependency list will automatically be reflected in the Java build path, but not the reverse.
- Edit the dependency list rather than the Java build path so that the two are automatically in sync.
- *NoClassDefFoundError* if they get out of sync.

Extensions and Extension Points

- A plug-in declares *extension points* so that other plug-ins can extend the functionality of the original plug-in in a controlled manner.
- This mechanism provides a layer of separation so that the original plug-in does not need to know about the existence of the extending plug-ins.
- Plug-ins declare extension points and extensions as part of their plug-in manifest. - this approach allows Eclipse to load information about the extensions declared in various plug-ins without loading the plug-ins (code) themselves.

Extensions and Extension Points, cont.

- One plug-in declares an extension point in its plug-in manifest, exposing a minimal set of interfaces and related classes for others to use.
- Other plug-ins declare extensions to that extension point, implementing the appropriate interfaces and referencing or building on the classes provided.

Extensions and Extension Points, cont.

- Each type of extension point may require different attributes to define the extension.
- You can find documentation for an existing Eclipse extension point in the Eclipse help (select **Help** > **Help Contents**, then in the **Help** dialog, select **Platform Plug-in Developer Guid** > **Reference** > **Extension Points Reference**)
- Note that to extend an extension point declared by another plug-in, you also need to include that plug-in in your dependency list.

Extensions and Extension Points, cont.

- When defining an extension point in Eclipse, you need to specify
 - Extension Point ID: [the plug-in's id].[a local identifier] (e.g. edu.umkc.myplugin.extpoint)
 - Extension Point Name: human readable text.
 - Extension Point Schema: this will be automatically populated once the above information is provided.
- In addition, you can add more element/attribute to your extension point.
 - At some point, you may need to create an attribute (e.g. named “class”) of the “java ”type for a new element.

Plug-in Related APIs

- `org.eclipse.core.runtime.Platform`
 - `getBundle(String)` - returns the bundle with the specified unique identifier.
 - `getExtensionRegistry()` - returns the extension registry of this platform.
- Plug-ins and Bundles
 - `getState()`
- Plug-in extension registry
 - `getConfigurationElementsFor (String extensionPointId)` - returns all configuration elements from all extensions configured into the identified extension point.
 - `getExtensionPoint (String extensionPointId)`

Activator or Plug-in Class

- Every plug-in optionally can have an Activator class.
- The Eclipse system always instantiates exactly one instance of an active plug-in's Activator class. Do not create instances of this class yourself.
- Startup and shutdown
 - The plug-in loader notifies the activator when the plug-in is loaded via the `start()` method and when the plug-in shuts down via the `stop()` method.
 - It is not recommend to override these two methods.

Development of Plug-ins

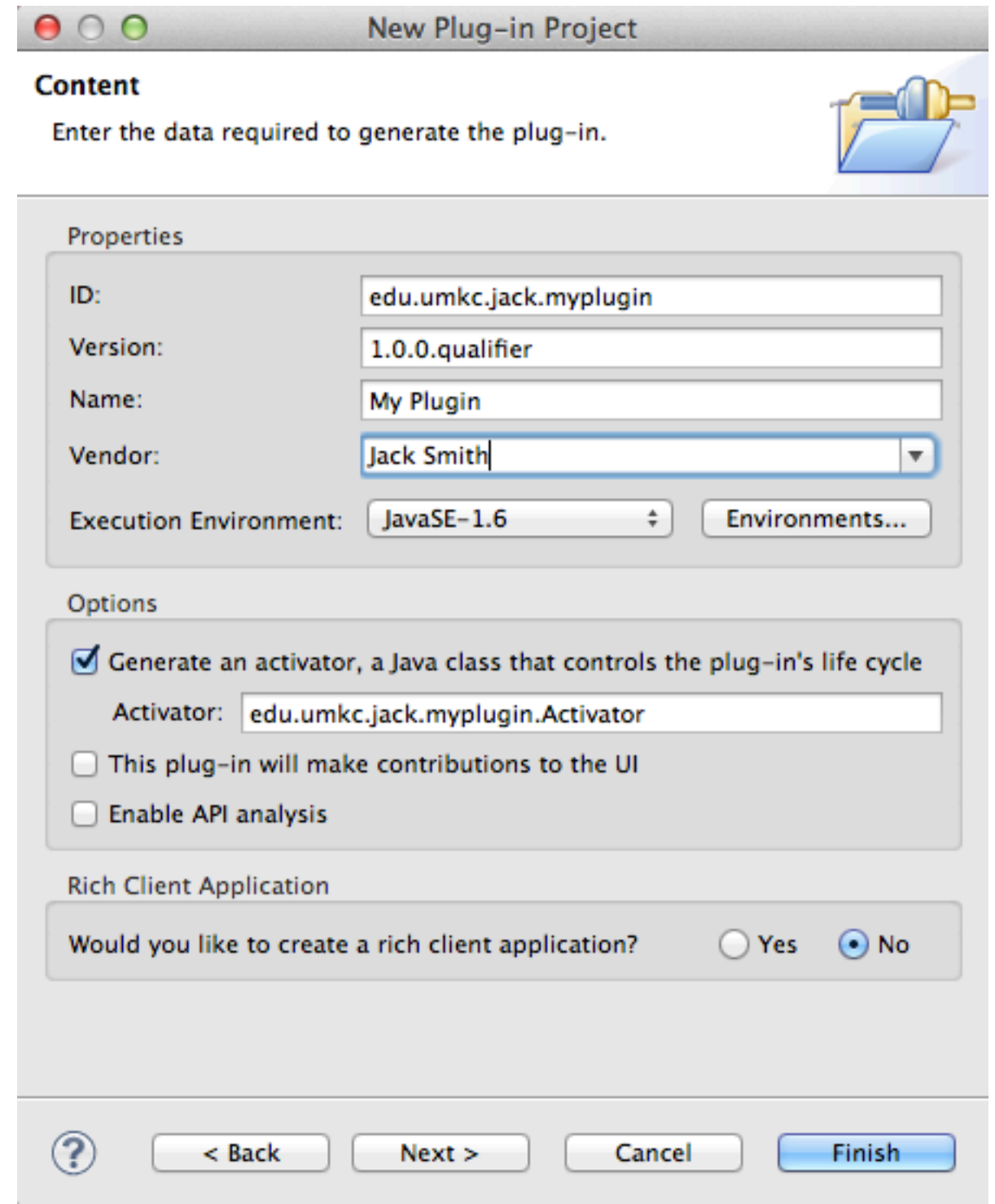
- Plug-in Development Environment (PDE) Views
- Creating a Plug-in Project
- Launching the Runtime Workbench
- Installing plug-ins
- Existing Eclipse Plug-ins

PDE Views

- In the Eclipse workbench, select Window > Show View > Other, and then expand Plug-in Development.
- The Plug-in Registration view: displays a tree view of all the installed plug-ins.
- The Plug-ins view: shows a list of external plug-ins and plug-in projects in the current workspace.
 - Double-clicking on a plug-in opens the plugin in an editor for viewing.
 - Right click a plug-in, and there are several useful context menus: *Add to Java Search*, *Open Dependencies*, *Import As*, ...
- The Plug-in Dependencies view

Creating a Plug-in Project

- Use the New Project wizard provided by Eclipse
- In Eclipse, select File > New > Plug-in Project
- In general, there are two kinds of plug-ins
 - Headless plug-ins: not contributing to the Eclipse UI
 - Eclipse UI, or IDE based plug-ins



The screenshot shows the 'New Plug-in Project' dialog box in Eclipse. The dialog has a title bar with standard window controls. Below the title bar, there is a 'Content' section with a folder icon and the text 'Enter the data required to generate the plug-in.' The main area is divided into two sections: 'Properties' and 'Options'. The 'Properties' section contains fields for 'ID' (edu.umkc.jack.myplugin), 'Version' (1.0.0.qualifier), 'Name' (My Plugin), 'Vendor' (Jack Smith), and 'Execution Environment' (JavaSE-1.6). The 'Options' section contains a checked checkbox for 'Generate an activator, a Java class that controls the plug-in's life cycle' with an 'Activator' field (edu.umkc.jack.myplugin.Activator), and two unchecked checkboxes for 'This plug-in will make contributions to the UI' and 'Enable API analysis'. At the bottom, there is a 'Rich Client Application' section with a question 'Would you like to create a rich client application?' and radio buttons for 'Yes' and 'No' (selected). The bottom of the dialog features a help icon, '< Back', 'Next >', 'Cancel', and 'Finish' buttons.

New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID: edu.umkc.jack.myplugin
Version: 1.0.0.qualifier
Name: My Plugin
Vendor: Jack Smith
Execution Environment: JavaSE-1.6

Options

☒ Generate an activator, a Java class that controls the plug-in's life cycle
Activator: edu.umkc.jack.myplugin.Activator
☐ This plug-in will make contributions to the UI
☐ Enable API analysis

Rich Client Application
Would you like to create a rich client application? ☐ Yes ☒ No

? < Back Next > Cancel Finish

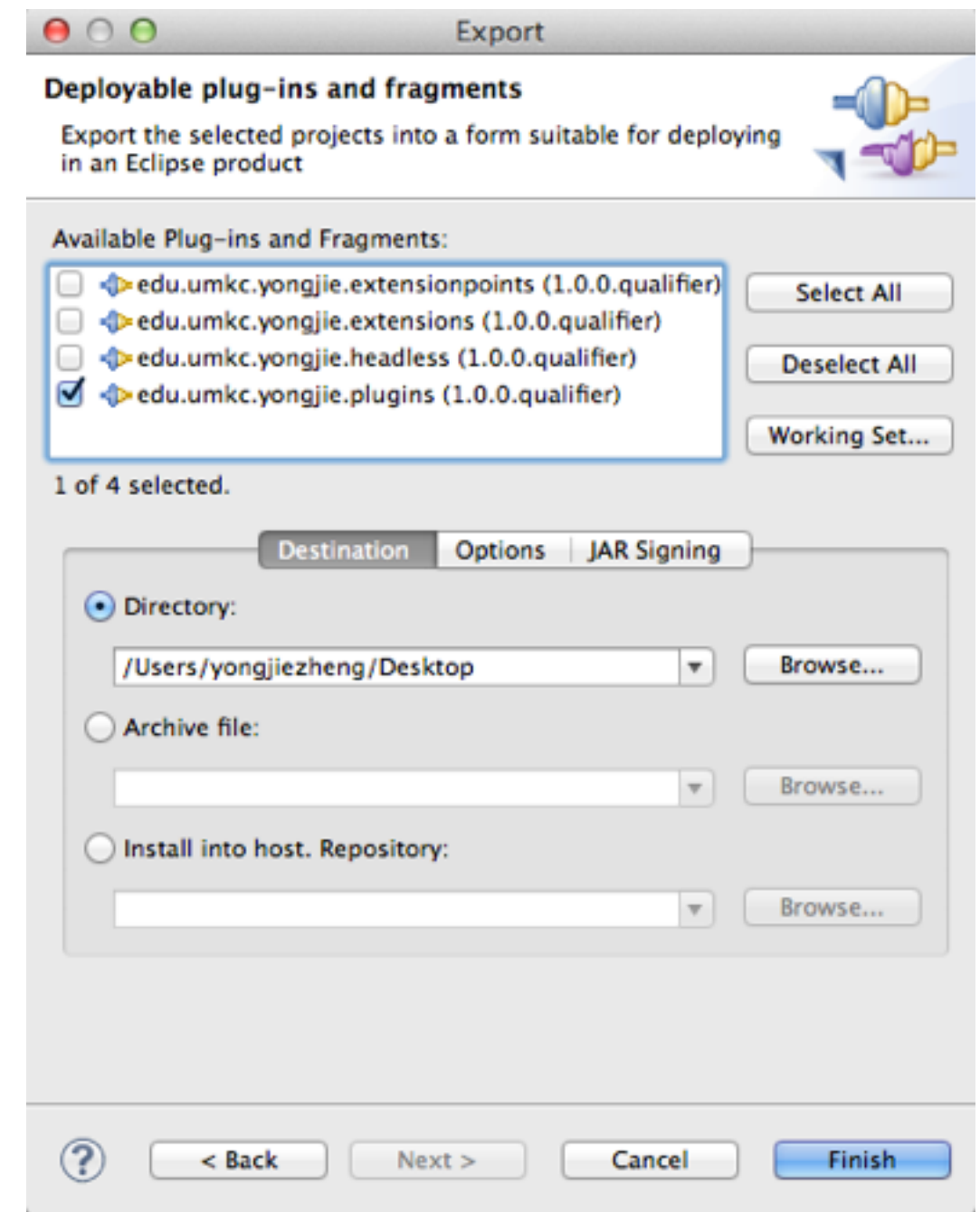
Creating a plug-in based on Eclipse
“Hello, World Command” template.

Launching the Runtime Workbench

- Once your plug-in project is finished, you can run it by selecting Run > Run As > Eclipse Application in the Eclipse workbench.
- A new Eclipse instance will be launched, and we call it the runtime workbench. Correspondingly, the original Eclipse instance is called the development workbench.
- By default, the runtime workbench will have all the development workbench's plug-ins and the plug-in projects in your workspace installed.

Installing Plug-ins

- Click on the plug-in project you want to deploy in the Eclipse workbench, and select File > Export > Plug-in Development > Deployable plug-ins and fragments.
- In the next pane, select Directory and click Finish.
- A plug-in JAR file will be generated in the directory you specified.



Installing Plug-ins, cont.

- Save the JAR file in the dropins directory of your Eclipse home folder.
- Restart your Eclipse, your plug-in will be installed.
- Alternatively, you can also deploy your developed plug-in by creating a Update Site. This is not covered in our class. In interested, please refer to our text book or some online tutorials.

Existing Eclipse Plug-ins

- Core: a general low-level group of non-UI plug-ins comprising basic services such as extension processing, creating new extension points, and so on.
- SWT: a general library of UI widgets tightly integrated with the underlying operating system, but with an OS-independent API.
- JFace: a general library of traditional UI functionality built on top of SWT.
- Workbench UI: plug-ins providing UI behavior specific to the Eclipse IDE itself, such as editors, views, perspectives, actions, and preferences.