Battlefield Experiences with Eclipse: Supporting Multiple Eclipse Versions Simultaneously

Eric Clayberg, VP of Product Development
Dan Rubel, Chief Technology Officer
February 4, 2004
Company Introduction

- Founded in 1997, Headquarters in Portland, OR
- Leading edge development tools for professional Java developers
  - VA Assist Enterprise (Smalltalk & Java)
  - CodePro Studio Product Line
  - Advanced Eclipse SWT Designer
- Extensive Eclipse experience
  - One of first IBM partners briefed on the technology
  - Technical development since January 2000 (>4 years)
  - First commercial Eclipse & WSAD add-on (Nov. 2001)
  - First product certified as “Ready for WebSphere Studio”
  - Eclipse Consortium board member
  - Technology PMC contributor (KOI Project)
  - Member Visual Editor Project
Who Are We

- Eric Clayberg
  - V.P. of Product Development for Instantiations
- Dan Rubel
  - Chief Technology Officer for Instantiations
- Dan & Eric
  - First used Java in 1996; Eclipse since 2000
  - Co-authors of several Eclipse articles for *WebSphere Advisor* magazine
  - Co-authors of upcoming Eclipse development book: *Eclipse: Building Commercial Quality Plug-ins*
  - Co-developers & architects of VA Assist Enterprise, CodePro Studio and over a dozen other commercial software products
Eclipse Development…

It ain’t rocket science,

but some times it feels like it. ;-)
Topics

- Challenges
  - Single product running on multiple versions
  - Using one version to build another

- Solutions
  - Solution 0: Code Branches
  - Solution 1: Façade Pattern
  - Solution 2: Ant-based Preprocessor

- Discussion
  - Version checking
  - Product Build Structure
  - Fragments
Challenges

- We want to use one version of Eclipse to develop for another

- Users span multiple versions, so we must create a product that will run on multiple versions
  - Eclipse 2.0 & 2.1 / WSAD 5.0 & 5.1 – official releases
  - Eclipse 3.0 M6 - most recent stable release

- We want to support latest and greatest version of Eclipse while retaining backward compatibility with older versions

- We want a single code base
Eclipse classpath

- ECLIPSE_HOME – always points to current Eclipse environment

- What to do when…
  - Using Eclipse 2.1 but compiling against Eclipse 3.0?
  - Different developers use different Eclipse versions?

- Use multiple classpath variables – one for each version
  - ECLIPSE20_HOME
  - ECLIPSE21_HOME
  - ECLIPSE30_HOME

- This explicitly separates the version of Eclipse being used from the version being developed
Solution 0: Code Branches – Overview

- One Eclipse project
  one branch for each version of Eclipse

  – OR –

- Multiple Eclipse projects
  one project for each version of Eclipse
Solution 0: Code Branches – Picture
Solution 0: Code Branches – Pros & Cons

- **Pros**
  - It works
  - Traditional approach

- **Cons**
  - No single code base
  - Requires merging code between streams
  - Can’t have more than one branch loaded simultaneously
Solution 1: Façade Pattern – Overview

- Main project contains common code
  - Interface hides version specific behavior

- Sub projects contain version-specific code
  - Concrete class implements interface and adapts common interface to version specific API
  - Each sub project compiled against a different Eclipse version

- Reflection used at runtime
  - Load the correct concrete class based upon the version of Eclipse being executed
Solution 1: Façade Pattern – Picture

- Primary Project
  - Subsystem 1
  - Subsystem 2
  - Subsystem 3
  - Common Access

- Sub Project 1
  - Eclipse 2.1 specific code

- Sub Project 2
  - Eclipse 3.0 specific code
Solution 1: Façade Pattern – Compilation

- Problem - Compiling against only the latest version of Eclipse is not good enough because…
  - Inheritance changes
  - Method changes
  - Visibility changes

- Solution - Compile against each supported version
  - Compiler verifies method calls against API for each supported version of Eclipse
Solution 1: Façade Pattern – Single binary

- A single binary might run correctly on different versions of Eclipse… *but are you sure?*

Your code:
```
public void yourMethod() {
    foo(0);
}
```

in the *.class file - method **selector** and **argument** signature
```
foo (I)V
```

Eclipse 3.0:
```
public void foo(int value) {
    ... some operation ...
}
```

Eclipse 2.1:
```
public void foo(long value) {
    ... some operation ...
}
```

- A binary compiled against one version may throw a runtime exception when executed on another version even if the source compiles correctly against both
Solution 1: Façade Pattern – Pros & Cons

**Pros**
- One binary runs on multiple Eclipse versions
- User can change Eclipse versions without re-installing plug-in

**Cons**
- Gets complex fast
- Compiler verifies common code against only a single version of Eclipse
- Becomes a clump of unrelated "glue" code
- Utility methods from around the system thrown into one utility class (code that really belongs elsewhere)
- Code needs to be replicated in multiple places
- Common code may throw runtime exception if executed on a different version of Eclipse
Solution 2: Preprocessor – Overview

- Single code base compiled against each Eclipse version
- Version specific code in special preprocessor comments
- Ant-based build process includes
  - `eclipsertools.preprocessor` Ant task to generate source specific to each supported Eclipse version
  - `eclipsertools.getclasspath` Ant task to generate classpath specific to each supported Eclipse version
  - One binary for each supported Eclipse version with all code compiled against that version’s API
- We use this approach to support
  - last two major releases (Eclipse 2.0 & 2.1, WSAD 5.0 & 5.1)
  - the latest stable build (Eclipse 3.0 M6)
Solution 2: Preprocessor – Picture

![Diagram showing the preprocessor and its interaction with different subsystems and binary versions of Eclipse.](image-url)
Solution 2: Preprocessor – Source conversion

Preprocessor Ant task

```
<eclipsertools.preprocessor
  sourceVersion="3.0"
  targetVersion="2.1"
  dir="src"
  todir="src21"/>
```

Original code

```java
public static Map getSomeInfo() {
  if (cachedInfo != null)
    return cachedInfo;
  /*$if version >= 3.0 $*/
  cachedInfo = call some Eclipse 3.0 API
  /*$elseif version < 2.1 $*/
  cachedInfo = call some Eclipse 2.1 API
  /*$endif $*/
  return cachedInfo;
}
```

Derived code

```java
public static Map getSomeInfo() {
  if (cachedInfo != null)
    return cachedInfo;
  /*$if version >= 3.0 $*/
  cachedInfo = call some Eclipse 3.0 API
  /*$elseif version < 2.1 $*/
  cachedInfo = call some Eclipse 2.1 API
  /*$endif $*/
  return cachedInfo;
}
```
Solution 2: Preprocessor – XML conversion

Preprocessor Ant task

```
<eclipsertools.preprocessor
  sourceVersion="3.0"
  targetVersion="2.1"
  dir="src"
  todir="src21"/>
```

Original code

```
<!-- $if version >= 3.0 $ -->
<extension point="org.eclipse.ui.ide.resourceFilters">
  <filter pattern="*.class" selected="true"/>
</extension>
<!-- $elseif version < 3.0 $ -->
<extension point="org.eclipse.ui.resourceFilters">
  <filter pattern="*.class" selected="true"/>
</extension>
<!-- $endif $ -->
```

Derived code

```
<!-- $if version >= 3.0 $ -->
<extension point="org.eclipse.ui.ide.resourceFilters">
  <filter pattern="*.class" selected="true"/>
</extension>
<!-- $elseif version < 3.0 $ -->
<extension point="org.eclipse.ui.resourceFilters">
  <filter pattern="*.class" selected="true"/>
</extension>
<!-- $endif $ -->
```
Solution 2: Preprocessor – Classpath

- New `eclipsertools.getclasspath` Ant task

- In most cases, differences are minor
  - `org.eclipse.ui_2.1.0` becomes `org.eclipse.ui_3.0.0`

- More difficult cases involve
  - Functionality moved from one plug-in to another
  - A single plug-in split into multiple new plugins
## Solution 2: Preprocessor – Classpath

<table>
<thead>
<tr>
<th>Build script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;eclipsetools_classpath_modifications id=&quot;classpath.mods.21&quot;&gt;</code></td>
<td>#1 identifies modifications</td>
</tr>
<tr>
<td><code>&lt;eclipsetools_classpath_variable name=&quot;ECLIPSE30_HOME&quot; path=&quot;ECLIPSE21_HOME&quot;/&gt;</code></td>
<td>#2 translate ECLIPSE30_HOME to ECLIPSE21_HOME</td>
</tr>
<tr>
<td><code>&lt;eclipsetools_classpath_modify path=&quot;ECLIPSE30_HOME/plugins/org.eclipse.ui.ide_3.0.0/ide.jar&quot;&gt;</code></td>
<td>#3 translate “ide” plugin to “ui” plugin</td>
</tr>
<tr>
<td><code>&lt;eclipsetools_classpath_entry path=&quot;ECLIPSE21_HOME/plugins/org.eclipse.ui_2.1.1/ui.jar&quot;/&gt;</code></td>
<td>#4 replace “bin” with “bin21”</td>
</tr>
<tr>
<td><code>&lt;eclipsetools_classpath_modifications refid=&quot;classpath.mods.21&quot;/&gt;</code></td>
<td>#5 find plugins with same id but different version</td>
</tr>
<tr>
<td><code>&lt;eclipsetools.getClasspath</code></td>
<td>#6 identifies modifications to use</td>
</tr>
<tr>
<td><code>binPath=&quot;bin21&quot; resolve=&quot;search&quot; propertyName=&quot;classpath21&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;eclipsetools_classpath_modifications refid=&quot;classpath.mods.21&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/eclipsetools.getClasspath&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
Solution 2: Preprocessor – Classpath

Original 3.0 classpath \(\rightarrow\) Converted 2.1 classpath

#1 \(\rightarrow\) ECLIPSE30_HOME/plugins/org.eclipse.ui.ide_3.0.0/ide.jar
\(\rightarrow\) ECLIPSE21_HOME/plugins/org.eclipse.ui_2.1.1/ui.jar

#2 \(\rightarrow\) ECLIPSE30_HOME/plugins/org.eclipse.ui.views_3.0.0/views.jar
\(\rightarrow\) ECLIPSE21_HOME/plugins/org.eclipse.ui.views_2.1.0/views.jar

#3 \(\rightarrow\) ECLIPSE30_HOME/plugins/org.eclipse.jface.text_3.0.0/jfacetext.jar
\(\rightarrow\) ECLIPSE21_HOME/plugins/org.eclipse.jface.text_2.1.0/jfacetext.jar

… etc …
Solution 2: Preprocessor – Pros & Cons

- **Pros**
  - Simple to maintain
  - All algorithm code in one place
  - Doesn’t need extra projects and “glue” code
  - Provides compiler verification against each Eclipse version

- **Cons**
  - Separate binary required for each Eclipse version
  - Must check that product is running on correct version
  - User needs to re-install when changing major versions

- **Available soon from…**
  - [http://www.qualityeclipse.com/ant/tools](http://www.qualityeclipse.com/ant/tools)
Version Checking

- Problem: What happens if the user installs the wrong binary?
  - Product may fail in unpredictable ways
  - Can compromise the integrity of the IDE itself

- Solution: Add version checking to plug-in startup method
  - Compile *expected* Eclipse version into plug-in
  - Query *actual* Eclipse version at startup
  - Check current version against complied-in version
  - Issue a warning & disable the plug-in as necessary
Version Checking Code

// Version of Eclipse this product is compiled against
public static final PluginVersionIdentifier VERSION_EXPECTED =
/* $if version == 3.0 */
    new PluginVersionIdentifier(3, 0, 0);
/* $elseif version == 2.1 */
    new PluginVersionIdentifier(2, 1, 0);
$elseif version == 2.0 $
    new PluginVersionIdentifier(2, 0, 0);
$endif */

// Version of Eclipse executing this product
public static final PluginVersionIdentifier VERSION_ACTUAL =
    ResourcesPlugin.getPlugin().getDescriptor().getVersionIdentifier();

// Is this plug-in compiled for this version of Eclipse ?
public boolean isValidProductVersionForIDE() {
    return VERSION_EXPECTED.getMajorVersion() == VERSION_ACTUAL.getMajorVersion() &&
        VERSION_EXPECTED.getMinorVersion() == VERSION_ACTUAL.getMinorVersion();
}
Build Structure: What is Ant?

- Apache tool for automating your build process - http://ant.apache.org/
- Ships “in the box” as part of Eclipse

```xml
<target
  name="product_30"
  description="Build the Eclipse 3.0 specific product"
  depends="build_common, build_30">
  <!-- Build the product from the temp directory -->
  <mkdir dir="${product_dir}/"/>
  <zip zipfile="${product_dir}/QualityEclipseTools_v${product_version}_for_Eclipse3.0.zip">
    <zipfileset
      dir="${temp_common}" prefix="${product_prefix}"/>
    <zipfileset
      dir="${temp_30}" prefix="${product_prefix}"/>
  </zip>
</target>
```

- Both procedural and declarative
Build Structure: Ant **depends** – how you use it

- We all get “procedural”… what’s “declarative”?
- The Ant **depends** declaration

```xml
<target
    name="product_30"
    description="Build the Eclipse 3.0 specific product"
    depends="build_common, build_30">```

- `<antcall>` is like a method call – executes each time
- Targets specified using **depends** are only executed once
Build Structure: One or More Binaries

- Targets to build a single binary
  - `product_30`
  - `product_21`

- Or to build all binaries at once
  - `product`

- Structured using `depends`

- Scales to support additional versions of Eclipse
Fragments

- Problems
  - Public API in newer version of Eclipse, but same functionality is inaccessible in the older version
  - New functionality added in newer version of Eclipse, but does not exist in the older version

- One Solution – Fragments
  - Inserts new code into existing plug-in – cannot override existing classes
  - Originally for Internationalization
  - Can be used to access internal code and backport functionality
New Eclipse Plug-In Development Book

- Eclipse: Building Commercial Quality Plug-ins
  - Addison Wesley, 2Q2004
  - [http://www.qualityeclipse.com/](http://www.qualityeclipse.com/)
  - ~600 pages
  - Authors:
    - Eric Clayberg
    - Dan Rubel
  - Series Editors:
    - Erich Gamma
    - Lee Nackman
    - John Wiegand
  - Forwards by:
    - Skip McGaughey
    - Simon Archer
Blatant Commercial Plug – SWT Designer

- Use SWT Designer to quickly and painlessly construct your SWT & JFace user interfaces
  - Free version of Designer supports all SWT widgets
  - Professional version of Designer adds JFace, Menu Editing, FormLayout & StackLayout support, Morphing, Templates, etc.

Swing support coming soon!
Blatant Commercial Plug – CodePro Studio

- CodePro Studio adds >300 enhancements to the Eclipse IDE
  - 425+ Audit Rules & Metrics, Design Patterns, Javadoc Repair, Dependency Analysis, Task & Build Automation, etc.
  - NC (Non-Commercial) version available for $99
Visit our website at:

- **www.instantiations.com**

Extreme Productivity.
The Ultimate Developer Experience.