Building Eclipse-Based Products

Ingo Muschenetz, Aptana, Inc.
About Aptana

- Founded in 2005
- Large installed user base. 1.5 million downloads of Aptana Studio, the leading Ajax development environment
- Hundreds of thousands of downloads of Rails, PHP, Apple iPhone, Adobe AIR plugins for Studio
- 50%/50% split between users of the standalone product vs. Eclipse plugin users
- Recently released Jaxer, the first Ajax server
Lessons Learned

- **Speed**: Build both RCP and plugin version with one pass
- **Traceability**: Use source control changelist number and datestamp for useful build numbers
- **Size**: Post-process feature.jar files to allow for flexibility in patching plugins
- **Flexibility**: Policy files and preferences make for flexible updates
- **Scale**: Use selective Apache rewrite rules to spread the load when mirroring updates
Build both RCP and plugin version with one pass

- We use PDE build for main RCP application, PluginBuilder for add-ons.
- Installed plugins can be “packed” or “unpacked”.
- Build RCP version of application, then extract out plugins used for update site and re-jar.
- Feed .zip file into post-build machines to create installers for various platforms.
Changelist number + datestamp for useful build numbers

- Build numbers are integer.integer.integer.string
- Useful numbers include the source control changelist and the date of build.
- Since source control changelist # will grow, pad it with zeros (we pad to six digits).
- Add in date of build:

1.1.0.001234_2008010111354
Post-process features to allow for flexibility in patching plugins

- Eclipse patch process allows users to update existing features and plugins
- Strict process has little flexibility. Features cannot change what plugins they contain
- “Small updates” allow for more flexibility when deploying changes
- Reconstruct feature JAR files on the fly during post-build step to provide delta between last release and current release by SVN introspection for changes
Policy files and preferences make for flexible updates

- Policy files allow user to choose the proper release stream (nightly, release candidate, stable)
- Add user interface to automatically set policy files
- Eclipse will not update 1.X to 1.X+1 automatically ("equivalent" vs. "compatible")
- Set preference for user in UpdateCore and run your own update check
Selective Apache rewrite rules spread mirroring load

- Eclipse mirror mechanism forces user to choose mirror
- Store site.xml and feature.jar files on central machine, plugins on EC2 instances
- Apache rewrite rule redirects only plugin files to PHP script that chooses random update machine and redirects user
- Push site.xml after delay to prevent corrupt updates
Questions?
Contact me at ingo@aptana.com
Appendix

Choosing a Build Automation Tool
Update Numbering
Moving from version 1.0 to 2.0
Digest Files/Pack200
Mirroring Updates
Nested Updates
Tips and Tricks
Choosing a Build Automation Tool

- PDE is Plugin Development Environment which also contains the skeleton for an automated build process.
- PDE is good at building a whole RCP application
- Pluginbuilder does what it says it does—builds plugins
  - Installs fresh Eclipse and dependent products
  - Can’t install dependencies from a protected HTTP location
  - Can’t build platform-specific artifacts
Eclipse Update Numbering

Eclipse uses the following scheme for numbering:

```
major . minor . service . qualifier
```

or

```
integer . integer . integer . string
```

Choose the “string” portion carefully to ensure that your users will get updates. It will be based on alphabetic comparisons.
Equivalent/Compatible Updates

- 1.0.1 -> 1.0.2 “Equivalent”
- 1.0 -> 1.1 is “Compatible”
- Your users have to switch a preference to change the setting, but you can do so programmatically for them:

  Set `UpdateCore.P_UPDATE_VERSIONS` to `UpdateCore.COMPATIBLE_VALUE`

- Record the current setting, trigger your own update search, and then reset.
Upgrading from 1.0 to 2.0

• 1.0 -> 2.0 will require a new installation via the update manager.

• Eclipse will automatically disable old versions of an identically-id’d feature, so not uninstall/reinstall is required.
Site Digest Files

- Digest files are a .zip file of all the feature.xmls
- Speeds up initial display of update “install” dialog
- Cannot create a digest for a site with only a single feature (process will fail)
- When colocating different update sites at same url, the digest file must be called digest.zip, but can be in subfolder:
  - <site digestURL="site.xml.digest”>
- File will be in siteroot/site.xml.digest/digest.zip
Site Digest File Creation

Create a digest file with this command:

```xml
<!‐‐
create
site
digests
‐‐>

<mkdir dir="${output.directory}/output/update/site.xml.digest" />

<exec executable="${env.ECLIPSE_HOME}/eclipse" failonerror="true" timeout="200000">
  <arg line="-application org.eclipse.update.core.siteOptimizer -nosplash -digestBuilder -digestOutputDir=output/update/site.xml.digest/ -siteXML=output/update/site.xml" />
</exec>
```
Pack200

- Pack200 finds redundancy in the compiled Java classes and removes it to reduce file size during download. On install, the classes are “unpacked”
- Requires Java 1.5 or later on the client machine
- Update site has both packed and unpacked plugins
- If the user does not have pack200 capability, it will download the unpacked plugins
- Do not pack plugins in place—copy to new folder, run pack process and then copy only packed versions back.
Pack200 Creation

Create new directory and copy plugins to pack

<!－－ create pack200 dir －－>
<mkdir dir="output/update/pack200_source" />
<copy todir="output/update/pack200_source">
  <fileset dir="output/update/"
    <include name="/**/com.aptana.*/**/**"/>
    <include name="/**/com.aptana.*"/>
  </fileset>
</copy>

<!－－ do pack200 －－>
<exec executable="eclipse" failonerror="true">
  <arg line="-application org.eclipse.update.core.siteOptimizer -nosplash -jarProcessor -verbose -processAll -repack -pack -outputDir output/update/pack200 ${output.directory}/output/update/pack200_source" />
</exec>
Pack200 Creation Continued...

Copy packed plugins back into source directory

<!-- move new packed and gzipped JARs over (The *.jar files are packed, un-gzipped JARs. We shouldn't move those over). Don't move features, as they are very small to begin with -->

<move todir="${output.directory}/output/update/plugins">
  <fileset dir="${output.directory}/output/update/pack200/plugins">
    <include name="*.jar.pack.gz"/>
  </fileset>
</move>

<delete dir="${output.directory}/output/update/pack200"/>
Mirroring Updates

Builds are pushed to update.aptana.com and then rsync’d to each of the EC2 images. Plugin requests are round-robined to each of the individual servers.
Eclipse patches allow for upgrading plugin B or C. But what if you want to add plugin D to an existing feature?
Small Update Builder

Allows feature flexibility during updates, but ensures small downloads of only changed content
Nested Updates

For nested updates, update xml files for all ancestor features up the tree

Plugin G is new for Feature D

We rewrite feature.xml for A and D to add in new plugin D
Update ancestor feature XML

• Feature A Before:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<feature id="feature.a" version="1.0.0.001000_date">
  <includes id="feature.d" version="1.0.0.001000_date"/>
  <requires />
  <plugin id="plugin.b" version="1.0.0.001000_date" />  
  <plugin id="plugin.c" version="1.0.0.001000_date" />
</feature>
```

• Feature D Before:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<feature id="feature.a" version="1.0.0.001000_date">
  <includes id="feature.d" version="1.0.0.001000_date"/>
  <requires />
  <plugin id="plugin.f" version="1.0.0.001000_date" />
</feature>
```
Update ancestor feature XML

• Feature A Afterwards:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<feature id="feature.a" version="1.1.0.001000_date">
  <includes id="feature.d" version="1.1.0.001000_date"/>
  <requires />
  <plugin id="plugin.b" version="1.0.0.001000_date" />
  <plugin id="plugin.c" version="1.0.0.001000_date" />
</feature>
```

• Feature D Afterwards:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<feature id="feature.d" version="1.1.0.001000_date">
  <requires />
  <plugin id="plugin.g" version="1.1.0.001000_date" />
  <plugin id="plugin.f" version="1.0.0.001000_date" />
</feature>
```
Eclipse Updates

- Eclipse updates make it difficult to follow conventional build numbering strategies.
- Remember that build numbers are integer.integer.integer.string
- Last string is alphabetic comparison
- Useful build number is changelist_timestamp to allow for easy syncing with relevant changes from source control
- Pad changelist to multiple digits with leading 0’s to allow you room to grow.
- Reset update preferences and search for only your features to allow for easy upgrading.
Update Tips and Tricks

• Feature handlers can be used to run machine configuration steps

• Don’t pack feature handlers if you have an embedded jar file

• You must have a feature.properties file for localization in order to run site digest creation

• Permission management using permissions.properties file