Integration Builds

The integration build occurs automatically at 8:00 AM every Monday.

In config.xml, the CC project is ecfIntegration. The buildType is I. The build is off the HEAD.

```
<ant buildfile="cc-build.xml"
     time="0800"
     day="monday"
     target="ecf.copyIntegration" >
   <property name="mapVTag" value="HEAD" />
   <property name="feature" value="ecf.core" />
   <property name="buildIdentifier" value="false" />
   <property name="buildType" value="I" />
   <property name="genFVSuffix" value="false" />
 </ant>
```

cc-build.xml

In cc-build.xml, the target is ecf.copyIntegration, which depends on ecf.build which depends on intgration.init. The target integration.init just calles the script replaceVDATE-TIMEproperties.sh with the arument cctimestamp. cctimestamp is a property passed to cc-build.xml from Cruisecontrol. This script just edits the file integration.properties.TEMPLATE to set timestamp equal to v<*year*><*month*><*day*>-<*hour*><*minute*>. The cctimestamp gotten from Cruisecontrol has this information but not in this format.

ecf.build.xml

The first thing ecf.build does is bring in the integration.properties file, which sets ${timestamp}. It then sets forcecontextQualifier to ${timestamp}. A typical value is v20081013-0800 .

buildId is set to timestamp also. Note that in config.xml, buildIdentifier is set to false. If it were not false, buildId would be ${timestamp}-{buildIdentifier}.

Then, ecf.build calls the PDE builder. The buildfile is build.ecf.xml.

Depending on the buildType, build.ecf.xml sets either the property CVSbranch or CVShead. The integration build is done off of HEAD, and so the property CVShead is set. Based on this property, build.ecf.xml then calls either the ant target getBranch or getHead. These targets update the monitored files.

Note that in config.xml, the integration build monitors ${localcopyRelease_2_0}, and config.xml defines this property as follows.

```
<property name="localcopyRelease_2_0"
  value="/home/ted/workanonRelease_2_0/org.eclipse.ecf/framework/bundles" />
```

Then, build.ecf.xml sets the property do.P2. This property is not set for the integration build. The purpose of this property is to determine whether to call ant_p2_local_workaround.xml. What ant_p2_ocal_workaround.xml does is copy the platform stuff from /opt/build.ecf/platform_plugins to /opt/build.ecf/ecf.build/plugins so that all the references get resolved. With the integration build, we actually want to build everything so we do not set do.P2.

build.ecf.xml prints out the values of some of its properties to nohup.out as follows.

```
ecf.build:
[echo] Click on the results link to get copies of the latest Daily builds
[echo] -------------- cc-build.xml --------------
[echo] build.environment.properties: javacFailOnError = true
```

```
[echo] build.environment.properties:    buildingOSGi = true
[echo] build.environment.properties:    eclipse.home = /opt/eclipse3.4M5/eclipse
[echo] cc-build.xml                  logfile = logs/ecf.core-Iv20081013-0800.log
[echo] buildIdentifier:                   false
[echo] buildId:                         v20081013-0800
[echo] buildType:                         I
[echo] mapVTag:                           HEAD
[echo] mapVersionTag:                        HEAD
[echo] cctimestamp:                  timestamp = 20081013080042
[echo] ---------------------------------------------------------------
[echo] forceContextQualifier:       forceContextQualifier = v20081013-0800
[java] Using /opt/build.ecf/logs/ecf.core-Iv20081013-0800.log file as build log.
[echo] cc-build.xml:                buildResult = 0
[echo] cctimestamp:                  timestamp = 20081013080042
```

Then, build.ecf.xml calls build.xml, which completes the PDE build. Much of this is standard PDE build stuff, but note that we have a customTargets.xml in /opt/build.ecf/ecf.core.

customTargets.xml

The file customTargets.xml does the actual checkout of the source files. The target getMapFiles gets the map files. "Getting" here means a cvs export; export gets a copy of the files without the corresponding CVS directories. Note that it chooses what map files to get with ${mapVersionTag}. The command is actually cvs -r ${mapVersionTag}.

What is ${mapVersionTag}? If it's not defined it's the same as ${mapVTag}, which for the integration build is HEAD. When it is defined, it's p2_workaround_1. When it's defined as p2_workaround_1, then we're not using a map file that has the platform stuff in it; also, the feature.xml file does not have the platform stuff listed.

The destination for the map files is ${basedir}/maps.cvs where ${basedir} is /opt/build.ecf. The map files are then copied to ${buildDirectory}/maps where
${buildDirectory} is /opt/build.ecf/ecf.build and the token CVSTag in the map files is replaced with ${mapVTag} which in the case of the integration build is HEAD (it is defined in config.xml).

Here's an important item to mention. Consider the file
${buildDirectory}/finalFeaturesVersions.properties. Because this file is in the build directory, it is created anew with each build. We use the contents of this file to get the name of the zips we make.

First of all, where does finalfeaturesversions.properties get its values? I think from the feature.xml of releng/features/org.eclipse.ecf.core-feature. In finalfeaturesversions.properties, the property org.eclipse.ecf.core is defined.

```
<property file ="${buildDirectory}/finalFeaturesVersions.properties"/>
<property name="archiveFullPath"
  value="${buildDirectory}/${buildLabel}/org.eclipse.ecf.core-${org.eclipse.ecf.core}.zip"/>
```

For example, for an integration build, org.eclipse.ecf.core is 2.0.1.v20081013-0800.

Then, for our build, the sdk zip file is the combination of the core and examples zips.

```
<zip destfile="${zipPath}/${sdk}-${org.eclipse.ecf.core}.zip">
  <zipfileset src="${zipPath}/${core}-${org.eclipse.ecf.core}.zip" />
  <zipfileset src="${zipPath}/${examples}-${org.eclipse.ecf.examples}.zip" />
</zip>
```

Once the build is complete (that is the target ecf.build has finished), then the target ecf.copyIntegration in build.ecf.xml is called. This last target is pretty straightforward. All it does is copy the zip output to /opt/build.ecf/ecf.build/${buildType}-${timestamp} and all the update files to /opt/build.ecf/ecf.build/updateSite.

Then, control goes back to config.xml, which calls the ant publisher. In the case of integration build, the ant publisher is antint.xml with target editmappsf.

antint.xml

The purpose of antint.xml is mainly to copy over the built files to dev.eclipse.org. It also makes a map file and a psf file for the platform stuff. Note that we end up building a lot more for the integration build than is actualy necessary, but these map and psf files pull out what is necessary.

The file runs three shell scripts. These scripts create files from templates. Some specific information is encoded in these templates, and we think we can make them more general by getting information from one of the generated properties files in ecf.build.

Currently, we make ant_p2_workaround.xml, which is not used. We use the local version instead. The non-local version gets the latest platform stuff from the integration directory on dev.eclipse.org rather than from the platform_plugins directory.

Then, we just copy everything over to dev.eclipse.org. We use sshexec to make the directory that will contain our output and scp to put the output in those directories.

Finally, we call antp2tag.xml.

```
<ant antfile="antp2tag.xml">
  <property name="tagThis" value="HEAD" />
</ant>
```

antp2tag.xml

For integration builds, we just tag the HEAD with ${timestamp}. But for release builds, we want to tag the branch Release_2_0 with ${timestamp}.

antp2tag.xml does the following:

```
<cvs cvsRoot=":ext:tkubaska@dev.eclipse.org:/cvsroot/rt"
 command="rtag -r ${tagThis} ${timestamp} org.eclipse.ecf/protocols/bundles/ch.ethz.iks.r_osgi.remote
        .
        .
```