Creating standardized builds outside of the Eclipse IDE

By Steve Taylor

Today's flexible software development teams responsible for building large applications may need to step outside the box and build outside of the Eclipse IDE. Steve explains these methodologies and processes, from preprocessing all the way through postprocessing and testing.

For Single Print Only

There is a new buzz around the Eclipse community about builds. It's about time. The software compile process remains a secret art, unless of course you stay within the simple point-and-click method inside the Eclipse IDE. Agile teams building larger applications need to build outside of the IDE to support continuous builds that are command line driven.

There are options when defining the build process outside of the Eclipse IDE. The three most common methods are:

- Developing open source *ad hoc* scripts
- Developing scripts to execute the Eclipse JDT (Java Development Tool) in "headless mode"
- The use of build solutions that offer script "reuse" addressing the core problem of builds in general: the redundancies in the scripts themselves

Hundreds of scripts

The most common method of building applications outside of the Eclipse IDE is to manually develop and maintain *ad hoc* scripts using open source languages such as Ant/XML. Because these scripts are written to support each deployable object, the number of final scripts that you eventually end up with can be quite high. It is not unusual for organizations to have hundreds of these scripts. To add to the complexity, each script contains hard-coded references that require manually revisiting the scripts when changes occur, such as code or package refactoring. Listing 1 shows an example of an Ant/XML script with the hard-coded references highlighted in gray, blue, and yellow.

The second method of supporting builds outside of the Eclipse IDE is to write scripts that call Eclipse in what is called *headless* mode. Building in headless mode means that you execute a build that is similar to the method used in the manual point-and-click process, without the point and click. The benefit of building in headless mode is that you can rely on the Eclipse project files to define what is being built, versus writing a script that hard codes the reference of files, directories, and compile options. The problem with building in headless mode is that, unlike using an Ant/XML script, you cannot extend other activities into your build as you can with Ant tasks. These other tasks include checking out files, calling Junit, or deploying the objects. To do this, developers must use a combination of both Ant/XML and the headless mode process.

The final method, which is beginning to gain momentum, is to establish build reuse within the development environment.

Listing 1

```
    </target>
 – <target name="build.javamail" depends="init, javamail" if="javamail-present">
    – <javac srcdir="${java.source.dir}" destdir="${javac.dest}" deprecation="${deprecation}" includes="${stem}/net/SMTPAppender.java">
        <classpath refid="compile.classpath" />
        </javac>
    </target>
 – <target name="build.jms" depends="init, jms, jndi" if="jms-present">
    – <javac. deprecation="${deprecation}" srcdir="${java.source.dir}" destdir="${javac.dest}" includes="${stem}/net/JMS*.java">
        <classpath refid="compile.classpath" />
        </javac>
    </target>
 – <target name="build.jmx" depends="ionit, jmx, jndi" if="jmx-present">
    – <javac. deprecation="${deprecation}" srcdir="${java.source.dir}" destdir="${javac.dest}" includes="${stem}/net/jmx/*.java">
      excludes="${stem}/jmx/T.java">
        <classpath refid="compile.classpath" />
        </javac>
    </target>
```

Listing 1

VME and Critical Systems ©2007 OpenSystems Publishing. Not for Distribution.

When building an application via the Eclipse IDE by pointing and clicking, developers rely on the Eclipse IDE to perform all of the heavy lifting, including refactoring and managing compile options and dependencies. What this provides is a level of "reuse" that cannot easily be achieved when coding Ant/XML scripts for each component to be built outside of the IDE. The goal of "reuse" is to achieve the consistency of building via the point-and-click process, without needing to write hundreds of Ant/XML scripts. There are open source tools that address build reuse, such as Maven by the Apache organization. Commercial tools addressing the reuse problem are offered by OpenMake Software, (www.OpenMakeSoftware.com), Serena (www.serena.com), and Codefast (www.codefast.com).

Establishing reuse within the compile process is the ultimate method to streamline a development. Taking a "write once, use many" approach to build scripting can substantially minimize the cost of developing software by streamlining redundant development efforts. Developers can collaborate build knowledge by simply sharing the reusable script that manages the compile of a particular compile type as depicted in Figure 1.

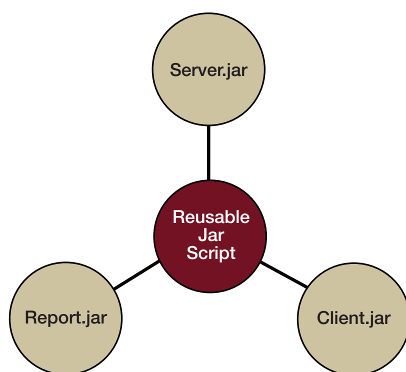

Figure 1

Developers do not have the luxury of a commercial build tool and must create a manual build process, and there are some important tips to consider. At minimum, XML build scripts should follow a basic flow with preprocessing, compile, and postprocessing steps that are consistent for every XML script written.

Preprocessing steps

The purpose of setting up preprocessing steps is to get the source code and variables organized and to perform overall housekeeping before compilation begins. The common mistake is to mix together these housekeeping steps for each call to the Java compiler or prior to calling an Ant task. Following are the recommended preprocessing steps.

CLASSPATH

The CLASSPATH identifies which Java classes are going to be used to resolve interclass dependencies. When setting the CLASSPATH, it is important to make sure that the CLASSPATH only be defined once in the process. Secondly, only the *jar* files and class directories that are used should be referenced in the CLASSPATH. Do not reference additional, unused *jar* files because it will cause the Java compiler to do more work than needed, therefore substantially slowing down the build.

Copying and renaming files

Minimize the use of copying and renaming files. Copying and renaming files makes it extremely difficult to trace backwards from the archive contents back to the original source. Instead of copying or renaming the files, place the files in correct locations from the start. Relying on the Ant script to clean up mistakes is just bad practice and can cause confusion. As an alternative to copying and renaming files, the use of the Ant Task "Zip" and its attributes, such as "dir" and "prefix," can handle obtaining source from one location and placing it in the archive in a different location.

Compile step

The compile step is, of course, the heart of the process. It will become the largest section of the XML script. The important point to remember when defining this portion of the script is dependency management.

Dependency references

With Ant, developers can explicitly define the dependencies between tasks. For example, the *jar* task can be dependent upon the javac task. Ant will also allow for multiple task dependencies to be established. Don't be seduced by this seemingly convenient Ant task. While this ability seems useful, it can turn out to be burdensome. When tracing the execution order of various Ant tasks in the Ant/XML script, it is much easier to follow the dependency chain that has only one task dependency instead of many.

Identifying source code

The best way to manage source is to define an efficient package directory structure. To do this, developers must move beyond the unique needs and address the package names at a more global level within the organization. It is best to make sure that a corporate Java package structure is agreed upon and used to minimize refactoring.

Javac –sourcepath

Consider the use of the –sourcepath javac flag. This flag allows for a directory concatenation for which to find the source code. There are two advantages in using this parameter. First, all of the code does not need to be found in the current build directory. Secondly, if the javac command is given a Java file as a parameter, it will then check using the –sourcepath parameter for additional source referenced by the original Java file and compile it as well. This process will allow for just the changed source to be passed.

Postprocessing steps

Postprocessing steps occur after the *jar, war,* or *ear* file has been built. These steps include calling the testing and deployment tools using Ant tasks.

Testing tips

Check to see if the correct deployment descriptor and properties files have been used or check the number of files in the archive to verify at a basic level if all the source was compiled. Developers can also test if the archive contains the correct manifest and directory structure.

Best practices enables reuse

By following a standard guideline, Ant/XML scripts can become more easily followed by another developer within the organization and are therefore more traceable. Traceability in the build process can only be achieved if someone other than the original developer can follow the steps of the build.

To establish solid standards in the build process, the best bet is to use solutions that create reuse within the build environment. Build solutions that offer script reuse in conjunction with an Eclipse plug-in can substantially improve the overall build

For Single Print Only

VME and Critical Systems ©2007 OpenSystems Publishing. Not for Distribution.

environment while minimizing the need for *ad hoc* scripting. Developers don't need to be confused by tools that indicate that they solve the build issue but still require developers to create scripts. As long as developers must spend time writing and debugging scripts, their process is not automated or standardized, regardless of whether the scripts are executed via a command line or by another tool that calls the script. Regardless of future build requirements, the effort in creating standards for the build is critical and well worth the effort. CS

Steve Taylor *is president and CTO of OpenMake Software. He is a senior developer with 19 years of experience in both distributed and mainframe application development. Prior to founding OpenMake Software in 1995, Steve served as a technical consultant, assisting companies with defining a solid build and release process. In this capacity, he became expert in the use of configuration management and release tools and recognized the need for a solid, reusable, and repeatable build process. During that time, he began developing the build procedures that have since become OpenMake. Steve received his BS in Computer Science/ Mathematics from the University of Illinois-CU.*

For more information, contact Steve at:

OpenMake Software
213 W. Institute Place, #404
Chicago, IL 60610
505-424-6440
steve.taylor@OpenmakeSoftware.com
www.OpenmakeSoftware.com

For Single Print Only

VME and Critical Systems ©2007 OpenSystems Publishing. Not for Distribution.