CHAPTER 5

Apache Ant Best Practices

This chapter looks in more detail at some best practices for using Ant on real projects. First I describe the use of property files to enable configuration of the build process depending on a user's role and requirements. I then describe how best to integrate Ant with IBM Rational ClearCase. Finally, I look at some general best practices for supporting the build process on large projects.

Aims of This Chapter

Apache Ant is a powerful build tool with significant built-in capabilities. However, a few capabilities and best practices stand out; they are described here. After reading this chapter, you will be able to

- Understand what Ant property files are and how they can be used to make build scripts more maintainable.
- Understand how to use Ant's capabilities to better integrate with IBM Rational ClearCase.
- Implement Ant build files that support reuse and maintainability on large projects.

This chapter assumes that you are familiar with the basic concepts of Apache Ant that were discussed in Chapter 4, "Defining Your Build and Release Scripts."

Property Files

From the perspective of Chapter 4, an Ant `build.xml` file is a single centralized build file that defines a repeatable process for bringing together an application, usually producing some form of

executable output. Although a single `build.xml` file can be enough to drive the build process, in practice it can quickly become large and unwieldy. In software development in general, it is recommended that you separate the data from the processes so that configuration changes can be made simply by focusing on and changing data alone. In Ant this is achieved through the use of property files. This section describes different uses of these property files:

- Implementing a configurable build using default and build property files. These files contain properties that define the data for your build process, such as the compiler version, optimization settings, and so on.
- Automating build numbering using build number property files. These files can be used to automate the generation of build numbers for applying labels or baselines.
- Reusing prebuilt libraries using library property files. These files can be used to specify a set of library versions to build against.

I will start by looking at default and build property files.

The Configurable Build

One of the problems of having a single, centralized build file is that there may be times when you want to change things slightly to carry out different build scenarios. For example, developers might want to override the standard optimization or debug settings. Similarly, when you change the details of an existing project or create a new project, you might have to rework a build file significantly. One of the ways to address both of these issues is to maintain property files that define default and overrideable property values. One convention for this is to create a file called `default.properties` (in the same directory as the `build.xml` file). It defines default values for the set of properties that would be required as part of a "normal" build. For example, the `default.properties` file for our RatlBankModel project might look like Listing 5.1.

Listing 5.1 default.properties File

```
# default properties
value.compile.debug     = false
value.compile.fork      = true
value.compile.optimize  = true
name.compile.compiler   = javac1.4
name.project-vob        = \\RatlBankProjects
name.project            = RatlBankModel
name.build.admin        = ccadm
name.build.branch       = RatlBankModel_Int
```

In this file you can place both project build information (compiler settings) and ClearCase-related information (the name of the UCM project, the ClearCase admin user, the build branch,

and so on). As with any other important software development asset, this file should be placed under version control in ClearCase. In the same location, you can also create a file called `build.properties`. This file should contain any of the properties from the `default.properties` file that a user wants to override. For example, a developer's `build.properties` file might look like Listing 5.2.

Listing 5.2 build.properties File

```
# build properties
value.compile.debug     = true
value.compile.optimize  = off
name.build.admin        = fred
name.build.branch       = fred_RatlBankModel_Dev
```

This file should be created by each user as and when he or she needs to override particular values. Consequently, it should *not* be placed under version control. To make use of these two property files, you have to include a reference to them in the build file. You can do this by including the following lines at the top of the `build.xml` file (usually before the definition of any targets):

```
<property file="build.properties"/>
<property file="default.properties"/>
```

The order is important here. Since in Ant all properties are immutable, they cannot be changed after they are set. Therefore, if a property value is defined in the private `build.properties` file, it cannot be changed by the value in the `default.properties` file. Given this scenario, it is therefore possible to write tasks that use these defined property values as follows:

```
<javac destdir="${dir.build}"
 debug="${value.compile.debug}"
 optimize="${value.compile.optimize}"
 fork="${value.compile.fork}"
 compiler="${value.compile.compiler}">
    <src path="${dir.src}"/>
        <classpath refid="project.classpath"/>
</javac>
```

Once these properties exist in a file, it is also possible for users to identify and override single entries from the command line. For example, to override the `value.compile.debug` setting in the preceding example, Ant could be called as follows:

```
>ant -Dvalue.compile.debug=false ...
```

Having this capability could be seen as an audit or security loophole. For example, if during a Release Build a particular property is overridden, where is the information on this override

recorded? To avoid this, I recommend that as part of your Release Build you record the values of all the customizable properties that are used. You can use the `<echoproperties>` task, as follows:

```
<echoproperties prefix="name."/>
<echoproperties prefix="value."/>
```

This command echoes all the properties that begin with the prefix `name` or `value` to the Ant log file.

> **AUTOMATICALLY PREFIXING BUILD PROPERTIES**
>
> If you want all your imported build properties to have a standard prefix, such as `bp.`, when you load the property file, you can specify the prefix at the same time. An example is `<property file="build.properties" prefix="bp"/>`. This also makes it easier to log the properties using the `echoproperties` command.

Automated Baseline Numbering

One of the most important aspects of software build automation is the reliable and consistent generation of build or release baselines. Baselines should be generated in a standard format. You also should be able to determine the baseline's quality simply by inspecting it. As I discussed in Chapter 3, "Configuring Your SCM Environment," with Base ClearCase this can be achieved via the baseline's name or through an attribute, and with UCM this can be achieved by using UCM baseline promotion levels. If these baselines are also "built" into the application, such as into a help file or `.jar` file, they can be extremely useful in debugging test or live problems.

One recommended way to automatically generate Release Build baselines is to use Ant's `<propertyfile>` task to automatically increment a build number. Ant does this by creating and updating the entries in a properties file; by convention, this particular properties file is called `buildinfo.properties`. When you start making use of this property file, I recommend creating an entry (called `name.build.info` or something similar) in the `default.properties` file to refer to it. At the same time, create an entry (called `name.build.referer` or something similar) that refers to a (single) source file or a Web page that should be updated with the automatically generated baseline. For example, the additional entries that can be added to your `default.properties` would be as follows:

```
name.build.info          = buildinfo.properties
name.build.referer       = src/com/ratlbank/model/Bank.java
name.build.prefix        = RATLBANKMODEL-
```

The `name.build.prefix` entry prefixes any labels or baselines that are created.

To insert the baseline into the file that the `name.build.referer` entry points to, some sort of marker must be placed in that file to be able to search for and replace it. If this file was a Java source file, one way to do this would be to create a Java static string called `version`:

```
private final static String version = "@(#)<label>
➥(on:<date>)@";
```

The beginning `@(#)` and ending `@` strings are simply tokens to be used for searching and replacing. This string could be displayed in a log file, in an About box, or wherever you need information about the version of the application that is running. To generate the build number, you would include a `<propertyfile>` task in the `build.xml` file:

```
<propertyfile file="${name.build.info}"
 comment="Build Information File - DO NOT CHANGE">
    <entry key="build.num"
     type="int" default="0000"
     operation="+" pattern="0000"/>
    <entry key="build.date"
     type="date"
     value="now"
     pattern="dd.MM.yyyy HH:mm"/>
</propertyfile>
```

This task writes (and increments) a build number into the `buildinfo.properties` file in the format `0000`, `0001`, and so on. It also writes the current date. Here's an example of the contents of a typical `buildinfo.properties` file:

```
#Build Information File - DO NOT CHANGE
#Wed Apr 1 17:48:22 BST 2006
build.num=0006
build.date=01.04.2006 17\:48
```

I recommend adding this file to version control and subsequently checking it in or out when it is updated. This way, the last build number is always preserved.

Once this build number has been generated, it can be used as part of a baseline to be placed across all the build files. To actually write the baseline into the Java source file containing the `version` string described earlier, you can use Ant's `<replaceregexp>` task to carry out a search and replace:

```
<replaceregexp file="${name.build.referer}"
 match="@\(#\).*@"
 replace="@(#)${name.build.prefix}-${build.num} (on:
${build.date})@"/>
```

Note that the baseline is prefixed with the `name.build.prefix` entry from the `build.properties` file, so in this case the baseline would be something like `RATL-BANKMODEL-0006`. In fact, this task searches for this expression:

```
@(#)<any text>@
```

and replaces *<any text>* with the following:

```
@(#)<build prefix>-<build number> (on: <build date>)@
```

For example:

```
@(#)RATLBANKMODEL-0006 (on: April 1st 2006)@
```

In case you are wondering why I chose the strange combination of tokens `@(#)`, this is a bit of UNIX history. On certain UNIX systems, if a static string containing these tokens was built into a program, you could use the SCCS `what` command to extract its information—for example, `what /bin/ls`. This command was always extremely useful in debugging test and live problems!

Reusing Prebuilt Libraries

In Chapter 2, "Tools of the Trade," I stated that one of the biggest issues with Java development is keeping track and control of all the third-party libraries a build requires. One of the ways to accomplish this is to maintain a library properties file. In this file, which I normally call `library.properties` (and create at the same level as the `build.xml` file), you define the exact version numbers and locations of the third-party libraries that the build requires. A sample `library.properties` file is shown in Listing 5.3.

Tool Support for Managing Library Dependencies

There are a number of additional tools that you can use to help you manage library dependencies. Apache Maven (http://maven.apache.org), an alternative build tool to Apache Ant, has a sophisticated mechanism for dealing with library dependencies. With this tool you specify the versions of third-party libraries to be built against in its project description file (the Project Object Model). Then, when you build your application, Maven resolves the dependencies by downloading them from a central repository. This repository can be hosted locally but by default is hosted on the Internet at www.ibiblio.org. Maven has not yet reached the acceptance levels of Ant but it is certainly a product for the future. A similar tool for managing Java library dependencies is Ivy (http://jayasoft.org/ivy). This is not a widely adopted tool; however, it does have the advantage of being able to work directly with Ant build scripts, and it also supports resolution from Maven's ibiblio repository.

Listing 5.3 Sample library.properties File

```
# Xalan - http://xml.apache.org/xalan-j/
xalan.version = 2.6.0
xalan.dir = ${dir.vendor.libs}
xalan.jar = ${dir.vendor.libs}/xalan-j-${xalan.version}.jar

# log4j - http://logging.apache.org/log4/
log4j.version = 1.2.9
log4j.dir = ${dir.vendor.libs}
log4j.jar = ${dir.vendor.libs}/log4j-${log4j.version}.jar
```

This example specifies the exact versions of all the third-party libraries that the build uses and where to pick them up. By default they are picked up from the JavaTools `libs` directory. To make use of this particular properties file, you include it at the top of `build.xml` at the same time you include the other property files:

```
<property file="build.properties"/>
<property file="default.properties"/>
<property file="library.properties"/>
```

Then, when specifying the `CLASSPATH` for the build process, you explicitly include the `.jar` files as follows:

```
<path id="classpath">
    ...
    <!-- include third party vendor libraries -->
    <pathelement location="${xalan.jar}"/>
    <pathelement location="${log4j.jar}"/>
    ...
</path>
```

Since the location and names of the libraries being built against are based on property values, you have the flexibility to override them as before. For example, a developer trying out some new functionality (and versions of these libraries) could create the following entries in his private `build.properties` file:

```
xalan.dir     = ${user.home}/libs
xalan.version = 2.6.1
```

Again, I recommend recording the values of all the properties used during the build with the `echoproperties` task.

STAGING LIBRARY DEPENDENCIES IN CLEARCASE

If you will version-control your library dependencies in ClearCase (either third-party or developed by you), you need to stage the objects in a suitable ClearCase repository or directory structure. Chapter 10, "The Art of Releasing," discusses the use of staged objects in more detail.

Ant ClearCase Integration

In terms of its integration with ClearCase, the Ant distribution already has a number of predefined tasks. These tasks are basically interfaces to the ClearCase `cleartool` command, with command-line options made available as task attributes. The complete list of tasks can be found online at http://ant.apache.org/manual/OptionalTasks/clearcase.html. Using these tasks to interface with ClearCase is quite straightforward. First, think of the commands you want to invoke (as `cleartool` subcommands) and the arguments that the commands need. Use the `cleartool`

man pages to help you (such as `cleartool man lock` to look up the options for the `lock` command). Then map this command and its options to the particular Ant task that implements it. For example, suppose that, as part of your build process, you want to update the build snapshot view, lock the integration branch, and check out a file (that records the build version). A target to do this might look something like this:

```
<target name="clearcase-pre" depends="init"
    description="execute ClearCase pre compile commands">
    <!-- update snapshot view -->
    <ccupdate viewpath="${user.dir}\.." graphical="false"
        overwrite="true" currenttime="true"/>
    <!-- lock the build branch -->
    <cclock objsel="brtype:project_int"
        replace="true" nusers="ccadm"/>
    <!-- checkout files to be updated -->
    <cccheckout viewpath="src/com/ratlbank/model/Bank.java"
        reserved="true" notco="false"/>
<target>
```

This set of commands is functionally equivalent to the following ClearCase `cleartool` commands:

- `cleartool update -graphical -overwrite -ctime <`*`directory_location`*`>`
- `cleartool lock brtype:project_int -replace -nusers ccadm`
- `cleartool checkout -nc -reserved src/com/ratlbank/model/Bank.java`

However, one of the problems with this set of tasks is that a Java class exists for each `cleartool` command. This sounds fine in practice and allows a degree of argument checking. However, the caveat is that whenever IBM Rational Software adds or updates a `cleartool` subcommand, this integration becomes out of date and needs to be reworked. Similarly, the existing tasks do not support some of Ant's more powerful features, such as support for the `<fileset>` element. A complementary approach is to use the third-party library *clearantlib,* which is described next.

Downloading and Installing *clearantlib*

clearantlib is a ClearCase Ant integration library in the form of an Ant *antlib*. An *antlib* is a prebuilt library that you can plug in to Ant without recompilation. It is simply a set of classes referred to by an XML namespace and then referenced just like any other Ant task or type. This makes it easy to integrate a set of tasks with a prebuilt version of Ant, such as the version of Ant included with Eclipse, Rational Application Developer, or CruiseControl. If you develop your own set of additional Ant tasks, it is definitely worth putting them in an *antlib* so that you don't have to recompile Ant each time you download and install a new version.

ANTLIB SUPPORT

Only Ant version 1.6 and later support the *antlib format,* so you need to make sure the environment or tools you are using are similarly aligned.

To install *clearantlib,* first download and extract the source distribution from the Web site: http://clearantlib.sourceforge.net. There should be a link to a page for downloading the source distribution. In this case you will download the zip file `clearantlib-src-x.x.zip` (or the latest version of *clearantlib*). Extract this file to a directory. I recommend putting it in your JavaTools `ant/src` directory. You can add this directory to source control too if you make any changes to the distribution. You should then have a structure similar to that shown in Figure 5.1.

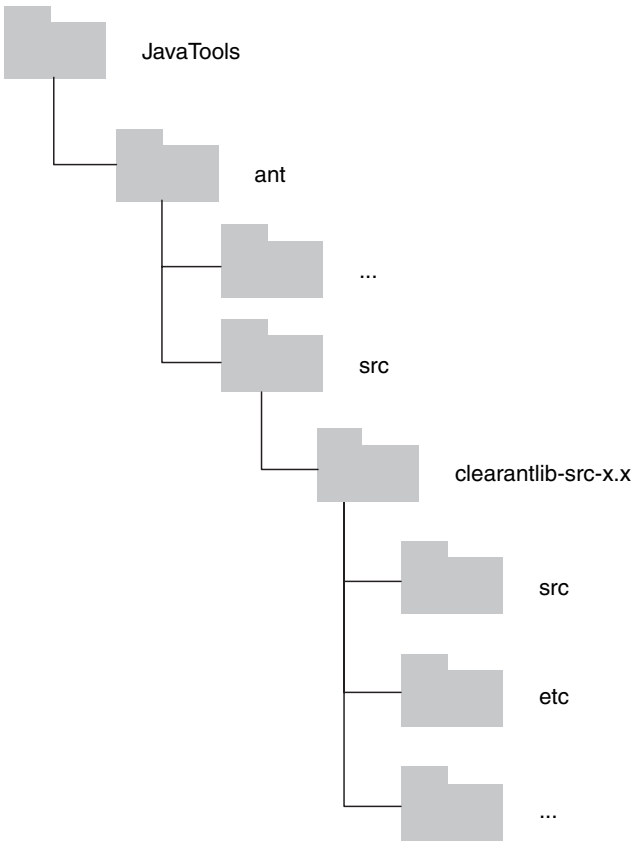

Figure 5.1 Antlib build directory structure

Once this is complete you can then build and install *clearantlib* in the standard manner. To do this, go to the command line and execute the following commands on Windows:

```
>cd C:\Views\javatools_int\JavaTools\ant\src\clearantlib-
➥src-x.x
>set ANT_HOME=C:\Views\javatools_int\JavaTools\ant
>ant install
```

or the following on Linux/UNIX:

```
>cd /Snapshot_views/javatools_int/JavaTools/ant/src/
➥clearantlib-src.x.x
>export ANT_HOME=/Snapshot_views/javatools_int/JavaTools/ant
>ant install
```

The second line is used to set the environment variable `ANT_HOME`—the root directory for your previously installed copy of Ant. The Ant build script looks for this environment variable and installs the *clearantlib* library (`clearantlib.jar`) in the Ant `lib` directory as part of the installation process. Note that these steps assume that the `javatools_int` view created in Chapter 3 is still available. Note also that the exact Linux/UNIX assignment of this variable depends on the shell being used (the preceding example is for the Bourne/Korn shell). I recommend adding the library to version control at this stage too.

Using the Library

To use the library you create a `build.xml` file for your applications as normal and also refer to *antlib*'s XML namespace. This is illustrated in the following example, which simply executes a `cleartool describe` on the file `build.properties` and redirects the output to the file `ccexec.out`:

```
<?xml version="1.0"?>
<project name="RatlBankModel" default="main"
    ➥xmlns:ca="antlib:net.sourceforge.clearantlib">
<target name="main">
    <ca:ccexec failonerror="true" output="ccexec.out">
         cleartoolpath="C:\Program Files\Rational\
             ➥ClearCase\bin">
        <arg value="describe" />
        <arg value="-long" />
        <arg value="build.properties" />
    </ca:ccexec>
</target>
```

As you can see, this new Ant task called `<ccexec>` takes any number of nested elements representing the arguments to the ClearCase `cleartool` command. The task also has an attribute called `failonerror`, which in this example is set to `true`. This specifies that Ant should terminate the build if this particular command fails. You can use the optional attribute `cleartoolPath` to specify the directory location where the ClearCase `cleartool` command can be found.

Alternatively, if you wanted to check in the complete set of Java files in the `src` directory, you could write an Ant target similar to the following:

```
<?xml version="1.0"?>
<project name="RatlBankModel" default="main"
    ➥xmlns:ca="antlib:net.sourceforge.clearantlib">
<target name="main">
    <ca:ccapply failonerror="true" maxparallel="5">
        <arg value="checkin"/>
        <arg line="-nc -ident"/>
        <fileset dir="${dir.src}">
            <include name="**/*.java"/>
        </fileset>
    </ca:ccapply>
</target>
</project>
```

In this example, the additional attribute `maxparallel` specifies how many elements should be passed to each `cleartool` invocation. For example, if 20 files needed to be checked in, four `cleartool` commands would be executed, each with five arguments. Normally, you would want as many files to be passed to a single `cleartool` command as possible. However, this might cause problems on operating systems (such as some versions of Windows) where command-line lengths are limited. `maxparallel` is a good way of circumventing this. The `<ccapply>` task has a number of additional inherited and potentially useful capabilities, such as transforming the names of the input (source) files to the output (target) files using regular expression "mappers." Some of these capabilities will be used later in this book. For more information on these capabilities, refer to the Ant `<apply>` tasks in the Ant manuals, because the `<ccapply>` task inherits all of its properties.

The `<ccexec>` and `<ccapply>` tasks are the two most basic tasks that *clearantlib* supports and that I use in this chapter. Later in the book, however, I will look at how to use more of the tasks in this library—in particular, to help with the reporting and releasing process. For more information on the tasks that are included with *clearantlib,* see the man pages that come with the distribution.

Project Support

All the Ant `build.xml` files that have been created so far have been quite simple. These files might suffice for building small Java projects, but the real world is not always so simple. You are probably wondering how Ant scales to projects with multiple components and a large code base. This section covers this issue and describes some of the capabilities and techniques that you can use for large or complicated projects.

Cumulative Targets

When creating a standard build file, you need to make sure that it can be used by all members of the development team. You don't want to have one set of build files for carrying out a Release Build or Integration Build and another set that the developers use to carry out their own Private Builds. Your aim should be to produce a single set of consolidated files that can be used by any member of the development team. However, there are often aspects of the complete build process that you might not want developers to carry out; baselining and deploying are typical examples. So how would you achieve this separation of concerns? Well, one way is to create an explicit cumulative target for carrying out Release or Integration Builds. In this target you specify how the Release or Integration Build is carried out as a sequence of individual targets.

There are two ways of creating a cumulative target. First, you can use the `antcall` task to invoke tasks in the same build file:

```
<target name="release" description="carry out Release Build">
    <antcall target="clearcase-pre" />
    <antcall target="update-buildinfo" />
    <antcall target="compile" />
    <antcall target="junit-all" />
    <antcall target="clearcase-post" />
    <antcall target="javadoc" />
</target>
```

However, the issue with this approach is that when using `antcall`, Ant reparses the build file and reruns all the targets that the called target depends on. In large projects this could slow down the build process. A better, and certainly faster, way is to use the `depends` attribute and create an empty target:

```
<target name="release" description="carry out Release Build"
    ➥depends="clearcase-pre, update-buildinfo,
    ➥compile,junit-all,clearcase-post,javadoc">
</target>
```

This example, although slightly less readable, does not reparse the complete Ant build file.

Reusable Routines

If you want to reduce the potential for duplication in your build files, you can reuse targets using the `antcall` target; you can also specify the value of attributes to be passed to the new target. For example, suppose you wanted to create a reusable target for checking ClearCase files in or out; you could create a new target called `clearcase-op`:

```
<target name="clearcase-op">
    <ca:ccexec>
        <arg value="${op.type}"/>
        <arg value="-nc"/>
```

```
            <arg value="${op.element}"/>
        </ca:ccexec>
    </target>
```

You could subsequently call this target as follows:

```
<antcall target="clearcase-op">
    <param name="op.type"    value="checkout"/>
    <param name="op.element" value="${file.build.info}"/>
</antcall>
```

This is a good start. However, beginning with Ant 1.6, a better and quicker way of creating reusable routines is to use the `macrodef` task. For example, you could create a reusable macro for any ClearCase command as follows:

```
<macrodef name="clearcase-op">
    <attribute name="type"/>
    <attribute name="element"/>
    <attribute name="extra-args"  default=""/>
    <attribute name="comment"     default="automatically
        ➥applied by clearcase-op"/>
    <sequential>
        <ca:ccexec>
            <arg value="@{type}"/>
            <arg line="-c '@{comment}' @{extra-args}"/>
            <arg value="@{element}"/>
        </ca:ccexec>
    </sequential>
</macrodef>
```

Then, to check out a file, you would simply use the following in your `build.xml` file:

```
<clearcase-op type="checkout" element="${file.build.info}"/>
```

This makes the `build.xml` file much more readable. Macros should be used in this way wherever you have a particular task being called repeatedly but with different parameters. If you wanted to, you could even create macros for every `cleartool` command, such as `checkin` or `checkout`. You might find it worthwhile to scan your build file to see if any additional opportunities exist for creating macros, typically where a set of operations is required to be carried out as an atomic operation. One example is the application of ClearCase labels or baselines. With Base ClearCase, to apply a label you create the label type, apply the label, and then promote it. A macro to achieve this could be created as follows:

```
<macrodef name="cc-apply-label">
    <attribute name="label"/>
```

```
        <attribute name="plevel"    default="BUILT"/>
        <attribute name="startloc"  default="."/>
        <sequential>
            <!-- create a new label type -->
            <ca:ccexec>
                <arg value="mklbtype"/>
                <arg value="@{label}"/>
            </ca:ccexec>
            <!-- apply the label -->
            <ca:ccexec>
                <arg value="mklabel"/>
                <arg value="-recurse"/>
                <arg value="@{label}"/>
                <arg value="@{startloc}"/>
            </ca:ccexec>
            <!-- apply the promotion level to the label -->
            <ca:ccexec>
                <arg value="mkattr"/>
                <arg value="PromotionLevel"/>
                <arg value="\"@{plevel}\""/>
                <arg value="lbtype:@{label}"/>
            </ca:ccexec>
        </sequential>
    </macrodef>
```

Then, to apply a label, you simply use the following in your `build.xml` file:

```
<cc-apply-label label="RATLBANKMODEL_01_INT" plevel="BUILT"/>
```

UCM requires fewer operations to achieve a similar effect; to apply a baseline and promote it, you could create a macro as follows:

```
<macrodef name="cc-apply-bl">
    <attribute name="basename"/>
    <attribute name="component"/>
    <attribute name="plevel"    default="BUILT"/>
    <attribute name="pvob"      default="\RatlBankProjects"/>
    <attribute name="baseline" default=
      ➥"@{component}_@{basename}"/>
    <sequential>
        <!-- create a new baseline -->
        <ca:ccexec>
```

```
                <arg value="mkbl"/>
                <arg value="@{basename}"/>
            </ca:ccexec>
            <!-- promote the baseline -->
            <ca:ccexec>
                <arg value="chbl"/>
                <arg line="-level @{plevel}"/>
                <arg value="@{baseline}@@@{pvob}"/>
            </ca:ccexec>
        </sequential>
    </macrodef>
```

Then, to apply a baseline you simply use the following in your build.xml file:

```
<cc-apply-bl basename="01_INT" plevel="BUILT"
 component="RATLBANKMODEL"/>
```

You can change these macros to conform to the exact way you want to create labels or baselines (for example, you might want to lock the label type too), but the point is that you have placed it in a macro, standardized your approach, and automated it.

Reusable Libraries

Maintaining a single large monolithic `build.xml` can be a challenge, especially with some of the eclectic syntax that Ant dictates. In most traditional programming languages, when you develop code you normally create libraries of common routines to hide some of this complexity. The preceding section mentioned macros; they obviously would be good candidates for placing in a separate library of routines. As of Ant 1.6, you can use the `import` task to include a library of routines (from the file `standard-macros.xml`), as in the following example:

```
<project name="test" default="test">
    <import file="standard-macros.xml"/>
    ...
</project>
```

One word of warning, though: the imported file must be a valid Ant build file, and it requires a `project` root element, just like any other build file. Listing 5.4 shows a common routines file containing the sample macro from the preceding section as well as the definition of some `<patternset>`s and a macro to execute a JUnit test suite.

Listing 5.4 standard-macros.xml Reusable Macros File

```
<?xml version="1.0"?>
<project name="standard-macros" description="standard reuseable macros"
    xmlns:ca="antlib:net.sourceforge.clearantlib">
```

```
...

<!-- define a patternset for Java classes -->
<patternset id="java.sources">
    <include name="**/*.java"/>
    <exclude name="**/*Test*.java"/>
</patternset>

<!-- define a patternset for Test classes -->
<patternset id="test.sources">
    <include name="**/*Test*.java"/>
</patternset>

...

<!-- macro for executing a clearcase operation -->
<macrodef name="clearcase-op">
    <attribute name="type"/>
    <attribute name="element"/>
    <attribute name="extra-args"  default=""/>
    <attribute name="comment"     default="automatically applied by
        ➥clearcase-op"/>
    <sequential>
        <ca:ccexec>
            <arg value="@{type}"/>
            <arg line="-c '@{comment}' @{extra-args}"/>
            <arg value="@{element}"/>
        </ca:ccexec>
    </sequential>
</macrodef>

...

<!-- macro for running a junit test suite -->
<macrodef name="project-junit">
    <attribute name="destdir" default="${dir.doc}"/>
    <attribute name="packagenames" default="${name.pkg.dirs}"/>
    <attribute name="classpathref"/>
    <element name="testclasses"/>
    <sequential>
```

```
        <junit printsummary="on" fork="no"
         haltonfailure="false"
         failureproperty="tests.failed"
         showoutput="true">
            <classpath refid="@{classpathref}"/>
            <formatter type="xml"/>
            <batchtest todir="@{destdir}">
                <testclasses/>
            </batchtest>
        </junit>
        <junitreport todir="@{destdir}">
            <fileset dir="@{destdir}">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="noframes" todir="@{destdir}"/>
        </junitreport>
        <fail if="tests.failed">
        One or more tests failed. Check the output...
        </fail>
    </sequential>
</macrodef>
...

</project>
```

You could also place common targets inside a similar XML file. For example, you could place standard invocations of the `init`, `update-view`, and `junit-all` targets in a file, as shown in Listing 5.5.

Listing 5.5 standard-targets.xml Reusable Targets File

```
<?xml version="1.0"?>
<project name="standard-targets" description="standard reuseable
    ➥targets" xmlns:ca="antlib:net.sourceforge.clearantlib">

<!-- create output directories -->
<target name="init" description="create directory structure">
    <mkdir dir="${dir.build}"/>
    <mkdir dir="${dir.dist}"/>
    <mkdir dir="${dir.doc}"/>
</target>
```

```
...

<!-- ClearCase snapshot view update -->
<target name="update-view" description="update ClearCase snapshot
 view">
    <ca:ccexec failonerror="false">
        <arg value="setcs"/>
        <arg value="-stream"/>
    </ca:ccexec>
    <ca:ccexec failonerror="false">
        <arg value="update"/>
        <arg value="-force"/>
        <arg line="-log NUL"/>
        <arg value=".\.."/>
    </ca:ccexec>
</target>

...

<!-- run all junit tests -->
<target name="junit-all" depends="compile"
    ➥description="run all junit tests">
    <project-junit classpathref="project.classpath"
        destdir="${dir.build}"
        packagenames="${name.pkg.dirs}">
        <testclasses>
            <fileset dir="${dir.src}">
                <patternset refid="test.sources"/>
            </fileset>
        </testclasses>
    </project-junit>
</target>
...

</project>
```

If both the `standard-macros.xml` and `standard-targets.xml` files are placed under ClearCase control and labeled or baselined, you can simply pull in the baselined version for your build process.

Using these capabilities, you can significantly reduce the content of each project's `build.xml`. For example, with targets for compilation, testing, creating distribution archives, and so on in these reusable files, the complete `build.xml` for a project might look like Listing 5.6.

Listing 5.6 build.xml Using Reusable Library Files

```
<?xml version="1.0"?>
<project name="RatlBankWeb" default="help" basedir=".">
    <property environment="env"/>
    <property name="dir.src"     value="JavaSource"/>
    <property name="dir.build"   value="WebContent/WEB-INF/classes"/>
    <property name="dir.dist"    value="dist"/>
    <property name="dir.junit"   value="build"/>
    <property name="dir.doc"     value="doc"/>
    <property name="dir.lib"     value="WebContent/WEB-INF/lib"/>

    <property file="build.properties"/>
    <property file="default.properties"/>

    <import file="${env.JAVATOOLS_HOME}/libs/standard-macros.xml"
        ➥optional="false"/>
    <import file="${env.JAVATOOLS_HOME}/libs/standard-targets.xml"
        ➥optional="false"/>

    <!-- define a classpath for use throughout the file -->
    <path id="project.classpath">
        <pathelement location="${dir.build}"/>
        <!-- include java libraries -->
        <fileset dir="${env.JAVA_HOME}/lib">
            <include name="tools.jar"/>
        </fileset>
    </path>

<!-- project override for junit test suite -->
<target name="junit-all" depends="compile"
    ➥description="compile source code">
    <project-junit classpathref="project.classpath"
     destdir="${dir.junit}">
        <testclasses>
            <fileset dir="${dir.src}">
                <patternset refid="test.sources"/>
```

```
            </fileset>
        </testclasses>
    </project-junit>
</target>

</project>
```

Notice that there is no definition for the `init`, `clean`, or `compile` targets, because they are all inherited. You will also notice that this project includes an override for the `junit-all` target. Although in Ant properties are immutable, targets can be redefined. So even if you have a standard invocation of a target in a reusable library file, you can still override it if desired, as in this case.

Build File Chaining

Large projects tend to develop build files per component or subsystem and create a top level or controlling build file to chain them together. This can be achieved using the `<subant>` task working on the results of a `<fileset>` defining which build files to call, as in the following example:

```
<target name="build-all">
    <subant target="compile">
        <fileset dir="." include="*/build.xml"/>
    </subant>
</target>
```

In this example, any file called `build.xml` in a directory below the current directory is called, and the `compile` target is invoked. One word of warning is that the `filesets` are not guaranteed to be in any particular order; this means that if there were dependencies between build components, compilation problems could occur.

Conditional Execution

Ant was not really intended as a general-purpose scripting language, so it lacks some of the capabilities found in scripting tools such as Perl, Python, and Ruby. In particular, currently it has very limited support for conditional execution or processing. However, it has some capabilities that should be sufficient in most circumstances, as I will describe here.

First, Ant allows you to conditionally execute a target depending on whether a property is set. For example, suppose you had some specific tasks that were dependent on the version of ClearCase that was installed, such as either Full ClearCase or ClearCase LT. You could then set the `cclt` property to indicate that ClearCase LT was installed (with any value, but for readability purposes I will set it to `true`). This property can then be used with the `if` or `unless` target attributes:

```
<property name="cclt" value="true"/>

<target name="cc-lt-check" if="cclt">
    <echo>ClearCase LT is installed; dynamic views not
     supported</echo>
    <!-- update snapshot view -->
</target>

<target name="cc-check" unless="cclt">
    <echo>Full ClearCase is installed; running audited
     build</echo>
    <!-- audit build -->
</target>
```

In this example the contents of the target `cc-lt-check` are executed only if the `cclt` property has been set. Likewise, the target `cc-check` is executed only if the property has not been set.

Ant also has a general `<condition>` task that can evaluate the result of grouping multiple logical conditions. This task can be used to set a property if certain conditions are true. The conditions to check are specified as nested elements. For example, the following code checks to see if the developer is carrying out a debug build:

```
<target name="debug-check">
    <condition property="debug">
        <and>
            <available file="build.properties"/>
            <istrue value="${value.compile.debug}"/>
        </and>
    </condition>
</target>

<target name="debug-build" depends="debug-check"
 if="${debug}" >
    <echo>carrying out a developer debug build</echo>
</target>
```

In this example, two conditions are checked: one using the `<available>` condition to see whether a particular file exists (in this case, `build.properties`), and another using `<istrue>` to see whether a particular property has been set to true (in this case, `value.compile.debug`—our debug switch). The `<and>` element specifies that both of the conditions must return true for the debug property to be set. There are a number of available conditions, including all the usual logical operators. For more information on evaluating conditions, see the Apache Ant Manual or Matzke [Matzke03].

The *clearantlib* library that I discussed earlier also has a ClearCase condition task that you can use in your build process. This task can be used to check for the existence of ClearCase objects, as in the following example:

```
<property name="dir.cc" value="build-results"/>

<!-- check whether ClearCase build results directory already
 exists -->
<target name="dir-check">
    <condition property="dir.ccexists">
        <ca:ccavailable objselector="${dir.cc}"/>
    </condition>
</target>

<!-- create build results directory, if it doesn't already
 exist -->
<target name="init" depends="dir-check" unless="dir.ccexists">
    <echo>creating ClearCase directory ${dir.cc}</echo>
    ...
</target>
```

In this example, executing the target `init` first executes its dependent target `dir-check`. This target uses the *clearantlib* `<ccavailable>` task to check whether a particular directory (in this case specified by the property `${dir.cc}`) exists and is under version control. If the directory does exist under version control, the property `${dir.ccexists}` is set by the `<condition>` task, and the body of the `init` task is subsequently executed. The `<ccavailable>` task can be used in a number of ways, such as to check for the existence of labels, branches, or baselines. For more information on the task, see the man page that comes with the distribution.

Groovy Ant Scripting

If you are familiar with the capabilities of scripting languages such as Perl, Python, or Ruby, you will probably find Ant's support for conditions, loops, and expressions quite limited. If you find that you can't quite do what you want with Ant's native capabilities, you can use scripting languages from inside your Ant build scripts. Since this book is about Java development, perhaps the most relevant scripting language and the one that I recommend is Groovy (http://groovy.codehaus.org/). Groovy is defined on its Web site as an "agile dynamic language for the Java 2 Platform." Basically, this means that Groovy is tightly integrated with the Java platform, and that it actually uses the Java 2 application programming interface (API) as the mainstay of its own API, rather than reinventing some new API that everyone would have to learn. Groovy has also attained Java Community Process (JCP) acceptance (as JSR #241) and therefore has been formally accepted by the Java community as the preferred language for Java-based scripting. Perhaps the most important feature of Groovy for our purposes is the fact that it understands the

Ant domain. For example, you could create the following Groovy script to compile some Java source code:

```
projSrcDir  = "src"
projDestDir = "build"
projLibDir  = "libs"
ant = new AntBuilder()
projClassPath = ant.path {
    fileset(dir: "${projLibDir}"){
        include(name: "*.jar")
    }
    pathelement(path: "${projDestDir}")
}
ant.mkdir(dir: projDestDir)
ant.echo("Classpath is ${projClassPath}")
ant.javac(srcdir: projSrcDir, destdir: projDestDir,
    ➥classpath: projClassPath)
```

This example uses Groovy's `AntBuilder` class to execute a selection of Ant tasks that should be familiar from the preceding chapter—`<mkdir>` and `<javac>`. The example also constructs an Ant `<fileset>` called `projClassPath` to be used as the Java `CLASSPATH` for building against. This is a relatively simple example, but with Groovy's support for variables, loops, and classes, you can more freely define your build script. In fact, using this capability you could actually replace all your Ant build scripts with equivalent Groovy scripts. However, the caveat here is that CruiseControl does not currently support the execution of Groovy scripts, so you would have to schedule their automation using a different tool. More importantly, however, you might already have a significant investment in your own or third-party Ant scripts that you would prefer not to rewrite. The alternative in this case is to use Groovy scripting from inside your Ant build files.

To use Groovy inside your Ant build files, you can use the Groovy Ant task (http://groovy.codehaus.org/Groovy+Ant+Task). Using this task you can place a series of Groovy statements between `<groovy>` markup tags:

```
<groovy>
  curdate = new java.util.Date()
  println("It is currently ${curdate}")
</groovy>
```

This example simply prints the current date from your Ant build file; you can see how the standard Java API call to the `Date` class is being used. As a more complete example, let's look at how to do something more useful. For example, what if you wanted to check in your build results (your `.jar` files) to a release area in ClearCase? Well, you could use Ant, Groovy, and the ClearCase `<ccexec>` task that was created earlier to achieve this using the Ant build script shown in Listing 5.7.

Listing 5.7 build.xml Groovy Scripting

```
<?xml version="1.0"?>
<project name="groovy-build" basedir="." default="main">
    <property name="dir.dist" value="dist"/>
    <property name="dir.rel"
        ➥value="C:\Views\RatlBankModel_bld\RatlBankReleases"/>
    <property name="dir.ant"
        ➥value=" C:\Views\RatlBankModel_bld\JavaTools\ant\lib"/>

    <!-- declare groovy task -->
    <taskdef name="groovy"
     classname="org.codehaus.groovy.ant.Groovy">
        <classpath>
            <path location="${dir.dist}/groovy-1.0-jsr-03.jar"/>
            <path location="${dir.dist}/asm-2.0.jar"/>
            <path location="${dir.dist}/antlr-2.7.5.jar"/>
        </classpath>
    </taskdef>

    <!-- declare ccexec task -->
    <taskdef name="ccexec"
     classname="net.sourceforge.clearantlib.ClearToolExec">
        <classpath>
            <path location="${dir.ant}/clearantlib.jar"/>
        </classpath>
    </taskdef>

<!--  execute some groovy commands -->
<target name="main">
    <groovy>
        // get the value of the Ant properties
        def dist = properties.get('dir.dist')
        def rel  = properties.get('dir.rel')

        // create a scanner of filesets for the dist directory
        scanner = ant.fileScanner {
            fileset(dir: dist) {
                include(name:"**/*.jar")
            }
        }
```

```
        // iterate over the fileset
        def dirco = false
        for (f in scanner) {
           def toFilename = rel + "\\" + f.name.tokenize("\\").pop()
           def toFile = new File (toFilename)

           if (toFile.exists()) {
              // if file already exists, update it
              ant.ccexec(failonerror:true) {
                  arg(line: "co -nc ${toFilename}")
              }
              ant.copy(todir: rel, file: f)
              ant.ccexec(failonerror:true) {
                  arg(line: "ci -nc ${toFilename}")
              }
           } else {
              // if file does not exist, check out directory
              if (!dirco) {
                  // check out directory
                  ant.ccexec(failonerror:true) {
                      arg(line: "co -nc ${rel}")
                  }
                  dirco = true
              }
              ant.copy(todir: rel, file: f)
              ant.ccexec(failonerror:true) {
                  arg(line: "mkelem -ci -nc ${toFilename}")
              }
           }

        }

        // check in directory if still checked out
        if (dirco) {
           ant.ccexec(failonerror:true) {
               arg(line: "ci -nc ${rel}")
           }
        }
    </groovy>
</target>

</project>
```

In this example, first Ant properties are defined for the build (`${dir.dist}`) and release (`${dir.rel}`) areas. Then two `<taskdef>` declarations are made—first for the `<groovy>` Ant task and then for our ClearCase `<ccexec>` task. Using the `<taskdef>` statement is simply another way of registering a prebuilt Ant Java class. Next in the `main` target, rather than using Ant tasks, Groovy script is used. First the Groovy script accesses the Ant properties defined earlier and stores them in some variables. It then constructs an Ant `<fileset>` scanner (`fileScanner`) to locate all the `.jar` files in the `${dir.dist}` directory. The script then iterates over all the entries in this `<fileset>`, copies the files to the release area, and adds or updates them in ClearCase. The interesting thing to note here is that some logic is built into the script to check whether the `.jar` file being copied already exists in ClearCase. This determines the exact ClearCase commands that will be used. For example, if the file is new, the directory needs to be checked out and a ClearCase `mkelem` command executed to create a new element. Achieving a similar result in Ant is considerably more complex.

> **AN ANT TASK FOR STAGING**
>
> Chapter 10 describes how to use the *clearantlib* task `<ccstage>` to check in a set of files to ClearCase.

In practice, I recommend using Groovy scripting only when you find that you can't do the equivalent operation well or very easily with standard Ant tasks and capabilities. It is obviously another language to learn. However, if you are interested in Java development, build processes, and scripting, I believe this will be a worthwhile exercise. The fact that Groovy is already a Java standard should also increase its visibility and popularity. For more information on the Groovy language, visit the Groovy Web site at http://groovy.codehaus.org/.

Build Script Documentation

One of the main documentation tasks for any build script is making sure that each target has an appropriate `description` attribute defined. For example, an obvious description for the `compile` target would be

```
<target name="compile" description="compile all source code">
```

The main reason for including this level of documentation is so that you can use the Ant `-projecthelp` (or `-p`) option from the command line. This option prints a list of all the targets in the build file, along with their descriptions:

```
>ant -p
Buildfile: build.xml
Main targets:
  clean          remove generated files
  compile        compile all source code
...
```

As you include more and more potential targets in your build file, this self-documenting feature can become very useful for any user who wants to execute it.

Summary

Apache Ant has been proven on many projects, from small to large. This chapter has discussed some best practices that should allow you to implement Ant successfully in your project. If you can get all members of your project team, from developers to integrators to testers, to use the same Ant build script, but with different configurations, I believe you will go a long way toward ensuring consistency and repeatability on your project.

Now that a basic definition of how to build a project is in place, I will move on to the next phase of the integrated build and release life cycle—execution. In particular, I will introduce the basics of CruiseControl and show you how you can use it to automate the execution of your Ant build scripts.