

Apache Ant

The Buildmeister's Guide

Apache Ant

The Buildmeister's Guide

Integrating and Applying Apache Ant

Kevin A. Lee

Second Edition

Copyright

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers. Inquiries concerning reproduction outside those terms should be sent to the publishers.

© Kevin A. Lee, 2007.

Published by Lulu.com.

The use of registered names, trademarks etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

www.lulu.com/buildmeisterbooks

About the Author

Kevin A. Lee has over 15 years of experience in implementing software build and release processes on projects of various sizes. He currently works as a Technical Consultant specialising in Java Development, Lifecycle Management and Enterprise Integration. Kevin has worked for IBM and Rational Software, helping customers in the adoption of new build tools and processes. He has also held various development roles at large telecommunications and financial services organizations.

Kevin is the author of *ClearCase, Ant and CruiseControl – The Java Developer's Guide to Accelerating and Automating the Build Process* and *The Buildmeister's Guide*. He is also the author of many other articles and tutorials that have been published on the Internet. He contributes to open source projects such as Apache Ant and CruiseControl, and maintains a web portal dedicated to the build process (www.buildmeister.com).

Kevin holds a B.S. in computer science from the University of East Anglia (UEA) and is based in the UK; where he lives with his family in the Buckinghamshire countryside. Kevin can be contacted at: **kevin.lee@buildmeister.com**

For Mum and Dad.

Thank you for making me happy, healthy and somewhat wise.

Table of Contents

Preface.....	15
Why this book came about.....	15
What this book covers.....	16
How to read this book.....	18
Example scenario.....	18
Introduction	23
What is Apache Ant?	23
The history of Ant	24
Why use Ant?	24
The purpose of Ant.....	25
Getting started with Ant.....	27
Installing Ant.....	27
The Ant build script.....	30
Typical Ant project directory structure	31
An example project.....	32
Summary	36
Fundamental Concepts	39
Properties	39
Built-in properties.....	40
Property files	41
Targets and tasks.....	42
Types.....	45
Common issues with Ant.....	48
Maintaining library dependencies	48
Portability	49
Debugging	50

Build avoidance	52
Defensive programming	53
XML syntax	54
Properties	55
Summary	55
In Practice	57
Creating configurable builds	57
Automating build numbering	59
Orchestrating builds	60
Cumulative targets	60
Chaining build scripts	61
Reusing build scripts	62
Reuseable macros	62
Pre-defined tasks	63
Importing build scripts	64
Implementing conditional logic	66
Extending Ant via scripting	68
Implementing library management	70
Installing the Ant tasks for Apache Maven	70
Using Apache Maven repositories	71
Implementing your own repository	75
Resolving Java libraries	76
Publishing Java libraries	77
Summary	79
Release Packaging and Deployment	81
Java EE Packaging	81
Java EE modules	83
Release packaging with Ant	87
Prompting for a release number	88
Creating manifest files	89
Creating Java archives	90
Creating Web and Enterprise archives	90

Signing and checksumming your archives	92
Release deployment with Ant	93
Summary	96
Integrating with Version Control Tools	97
Requirements for version control integration	97
Ant and Subversion	99
Installing SvnAnt	100
Using SvnAnt – creating or updating a workspace	101
Using SvnAnt – committing changes	103
Using SvnAnt – creating a baseline	104
Installing StatSvn	106
Using StatSvn	107
Ant and ClearCase	109
Installing ClearAntLib	109
Using ClearAntLib – creating or updating a workspace ...	110
Using ClearAntLib – committing changes	111
Using ClearAntLib – creating a baseline	112
Using ClearAntLib – creating a baseline report	114
Using the Ant build auditing listener	116
Summary	119
Integrating with Build Control Tools	121
CruiseControl	121
Installing CruiseControl	122
The CruiseControl configuration file	124
The CruiseControl workspace	126
CruiseControl Listeners	127
CruiseControl Bootstrappers	127
CruiseControl SourceControls	128
CruiseControl Builders	129
CruiseControl log results	129
CruiseControl Publishers	130

The CruiseControl Dashboard.....	131
Ant and IBM Rational Build Forge	132
Installing Build Forge.....	134
The Build Forge workspace	135
Build Forge Environment Groups	136
Build Forge Projects.....	137
Build Forge Adaptors and Filters	139
Build Forge scheduling	141
Summary	142
Extending Ant.....	145
Why extend Ant?	145
An example task	146
Implementing a new task	147
Create a new Java class	148
Write attribute setter methods	149
Add support for nested elements and character data.....	150
Supporting character data.....	152
Write an execute method.....	152
Register the task	153
Re-compile the source code	154
Final source code.....	154
Summary	155
Building the Reference Application.....	157

Preface

This preface discusses my motivations and experience for writing this book. I describe the framework of the book and how best to read it depending on your role, interests and experience.

Why this book came about

I have been working in the field of software build and release management for many years – either directly on projects or in a consulting capacity. In the last few years I have been working mainly with Java projects – implementing build processes based on the Apache Ant build scripting tool. As a result, I have been fortunate to gain a lot of practical experience. This has helped me to understand Ant’s main concepts and issues, good and bad practices, how to integrate Ant with other tools, and finally how to scale its implementation to make it work on large projects.

In order to “giveback” some of this information, over the last few years I have been publishing Apache Ant based material on *The Buildmeister* portal (www.buildmeister.com) – a community website I founded. *The Buildmeister* is dedicated to discussing what I call **Agile Software Delivery**: Agile development practices and their intersection with building, releasing and deploying software. For those of you that don’t already know, a *Buildmeister* is a name for the individual who is tasked with defining and implementing the technical build process for a project.

The Buildmeister website has been well received, with many subscribers and active participants. Although there is now a significant amount of information on this site, I would be the first to admit that it is not always as consumable and as easy to reference as you might want it to be. To help solve this, I decided to collect together some of the more significant content regarding Apache Ant, combine it with some new unpublished information and publish the consolidated manuscript which you are now reading.

The purpose of this book is not to describe all the details and intricacies of Apache Ant – there are plenty of other good books on this topic¹. Although I will take some time to discuss the fundamental concepts and terminology of Ant, this book is more concerned with how to integrate and apply Ant into an end-to-end build and release process. I will therefore be looking at issues such as how to scale Ant, how to integrate it to version control tools and also how to execute it from higher level build control tools.

What this book covers

The chapters in the book cover the following topics:

Chapter 1, “Introduction”

This chapter introduces Apache Ant and how it fits within the build process. It describes how to setup a working Ant environment and gives an example of how a simple project can be built by Ant.

Chapter 2, “Fundamental Concepts”

There are a number of fundamental underlying concepts in Apache Ant that you should understand in order to be able to

get the most out of it. This chapter looks at these in detail. It also looks at some of the common issues that users of Ant might come across and gives some suggestions on how they can be mitigated.

Chapter 3, “In Practice”

This chapter looks in detail at how Ant is implemented on real world projects. More specifically, it illustrates how some of Ant’s capabilities can be used to support the needs of large projects and how Ant can be used across multiple projects.

Chapter 4, “Release Packaging and Deployment”

Apache Ant is not just used for building (or compiling) software. You can also use it to package Java applications and physically deploy them to their run-time environments. This chapter looks at how Java applications are typically packaged and some of the capabilities of Ant that are available to support release packaging and deployment.

Chapter 5, “Integrating with Version Control tools”

Apache Ant is not implemented in isolation - it will need to work with your existing tools. This chapter looks at how Ant can be integrated with version control tools – in particular it looks in detail at how Ant can be integrated with the popular open source Subversion and commercial IBM Rational ClearCase tools.

Chapter 6, “Integrating with Build Control tools”

Although you can implement a complete end-to-end build process in Apache Ant, implementing repeatable build and deployment processes for multiple projects typically requires

¹ My preferred book is: “Ant: The Definitive Guide, 2nd Edition by Steve Holzner”

some form of build control framework. This chapter looks at how Ant can be integrated with the popular open source CruiseControl and commercial IBM Rational Build Forge tools.

Appendix A, “Extending Ant”

One of the more useful features of Apache Ant is that it can be easily extended by writing new Java classes. This appendix looks at why and how you might want to extend Ant and gives an example of how to achieve it.

Appendix B, “Building the Reference Application”

This appendix describes how you can download, execute and examine the Ant build process for the reference application.

How to read this book

This book can be read by many different members of the software development team. Developers and Architects should read at least chapters 1 and 2 in order to gain an understanding of how Apache Ant works. Release Engineers (Deployment Engineers) should read chapters 1 to 3 to understand how Ant can be used to support the release engineering and deployment process, whilst Build Engineers should obviously read the whole of the book.

Example scenario

To illustrate how Ant can be used in a typical Enterprise project, I will be putting together the framework of a build and release process for a working scenario. Although the project used in this scenario is not especially large or complex, it is based on real world examples and should give you an idea of how Ant can be successfully implemented.

Throughout the book, you will see reference to *Hotel de Java*. This is a hotel booking and reservations system built using open source technologies and frameworks. An example screenshot is illustrated in Figure 1.



Figure 1 - Hotel de Java application

Hotel de Java uses a standard Java EE (Java Enterprise Edition) system architecture as illustrated in Figure 2.

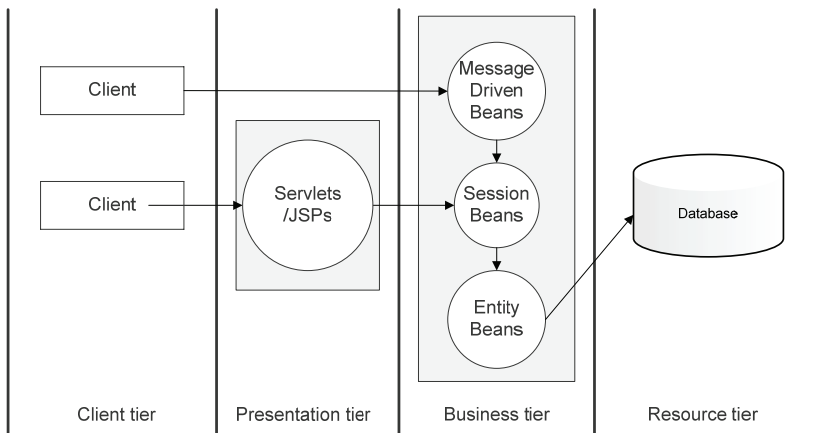


Figure 2 - Hotel de Java architecture

This architecture is relatively simple and can be described as follows:

- A backend database is used to persist hotel reservations and customers.
- Container managed Entity beans are used to access this data.
- Session beans are used as a facade to implement business logic and coordination.
- Message driven beans are used to manage reservations from external systems (for example. third party travel agencies).
- Servlets and JSPs (Java Server Pages) are used for web interface presentation.
- Users and Clients can access the system via a web interface or external messaging system.

If you are a Java programmer then this will all make perfect sense. If you are still getting to grips with Java and this has gone over your

head do not worry – you do not have to understand the code to be able to build the application.

The application is J2EE 1.4 compliant and can be executed in any run-time container that meets this standard. Apache Geronimo (<http://geronimo.apache.org>) was used to develop the application. The complete application is described in more detail at <http://hoteldejava.sourceforge.net>. Appendix B describes how to download and build it for yourself.

CHAPTER 1

Introduction

“The days just prior to marriage are like a snappy introduction to a tedious book”.
Wilson Mizner

This chapter introduces Apache Ant and how it fits within the build process. It describes how to setup a working Ant environment and gives an example of how a simple project can be built by Ant.

What is Apache Ant?

Apache Ant is a popular and widely used build scripting tool for Java development environments. Created by James Duncan Davidson while working on an open source project (which ultimately became Apache Tomcat); Ant was developed as a platform neutral build scripting tool. Unlike build scripts written in variants of *make*¹, Ant scripts are inherently portable. The same build script can be written and executed

¹ www.gnu.org/software/make

on Windows, and work without change on Linux or UNIX environments².

The history of Ant

Ant was officially released as a standalone product in 2000 and is now the de-facto standard for building Java projects. Lately, Apache Maven (<http://maven.apache.org>) has been developed as an alternative build scripting tool for Java. Maven introduces capability above and beyond Ant, including an embedded project lifecycle to limit the amount of potential scripting that is required. However, Ant is still the most widely used Java build scripting tool³ - especially in commercial development environments - and is supported, enhanced and delivered by industry vendors such as IBM, BEA and Sun.

Why use Ant?

If you are developing a Java project, the main reasons to use Ant (other than its cross-platform capabilities) is that it understands the Java domain. It has built-in tasks to define Java Classpaths, call the Java compiler and create Java distribution packages. This makes it natural to implement Java build processes with Ant – the tools works work with (not against you) to do so. Ant is also easily extendable; each of its tasks is a Java object. Therefore if you want to extend Ant, you can write a new Java class (building on the framework classes that Ant provides). I will be looking at how to extend Ant in Appendix A.

Ant has been around for a number of years now and is therefore well integrated into other development tools. Practically every Java

² This is not quite true, you might need to do some work to translate file system path references – I will discuss this in chapter 2.

³ I have no hard statistics to back this up other than empirical observations of many Java projects

IDE (Integrated Development Environment) has been integrated to Ant – examples include Eclipse, NetBeans and IntelliJ IDEA. There are also integrations to most version control tools (from CVS to ClearCase and everything in between). Collectively, this means that if you invest in Ant then you will be well supported, both by the collective knowledge of information that is available and the environments that it will integrate with.

The purpose of Ant

In its purest sense, Ant allows you to specify a build tree – a collection of related build targets which execute various individual tasks, for example. compile, test, package and so on. A single target can be called to invoke the entire build tree. This can significantly simplify the process of constructing an end-to-end build process. Note that Ant is not a compiler rather it calls your compiler. Java source code is compiled via the Ant `<javac>` task - which in turn calls your JDK (Java Development Kit) compiler.

Although, Ant can be used to coordinate an end-to-end build process, it is not a replacement for a complete control and execution framework, nor is it an infrastructure management tool that monitors, manages and makes use of a collective set of build servers. However there are a number of tools that integrate with Ant to achieve this. I call these **build control** tools as they manage and control the overall process. Each of these tools describe themselves slightly differently, however they all carry out similar functions. Examples of such tools include:

- *CruiseControl*
(<http://cruisecontrol.sourceforge.net>)
an open source Continuous Integration toolkit.
- *Anthill OS and Pro*
(www.anthillpro.com/html/default.html)
an open source and commercial Build

Management Server.

- *IBM Rational Build Forge* (www.ibm.com/software/awdtools/buildforge)
a commercial build and release framework.
- *ElectricCloud ElectricCommander* (www.electric-cloud.com/products/electriccommander.php)
a commercial software production automation tool.

There are many other tools that are available, however I have had personal experience of successfully integrating these particular ones with Ant. A diagram illustrating where Ant fits in to a complete build and release process is illustrated in Figure 3.

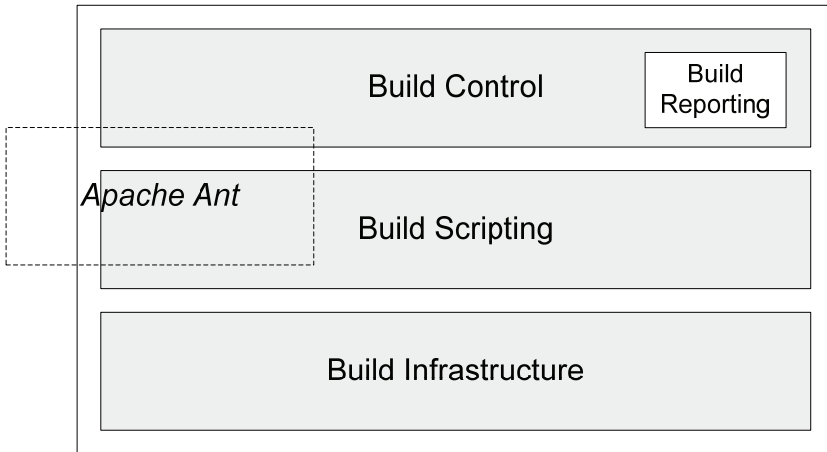


Figure 3 - Build capability architecture

This diagram is taken from the book *The Buildmeister's Guide*⁴.

⁴ www.lulu.com/content/409652

Getting started with Ant

The first question to ask before you start using Ant is have you got a copy of already? If you use an IDE, for example, Eclipse, then the chances are that you have. Ant is an open source product so anyone can distribute it with their application. The first decision you therefore need to make is whether to stay with the version that is integrated and delivered with your IDE or to maintain your own copy. I prefer to maintain my own copy so that Ant can be controlled and executed outside of the IDE. Most IDE's can also be pointed at different Ant installations. This lets you upgrade to a new release of Ant, and eliminate inconsistencies between the command-line and IDE executed versions of Ant.

You can download your own copy of Ant from <http://ant.apache.org>. There are two versions: binary and source. I recommend the source version since you can configure it with support for additional third party libraries.

Installing Ant

Once you have downloaded Ant, extract it to an area on your file system. In this chapter I will assume you have downloaded the source version and extracted it to the following directory on Windows:

```
c:\temp\apache-ant-1.7.0
```

Or the following directory on Linux or UNIX:

```
/tmp/apache-ant-1.7.0.
```

Next decide where you want to install Ant after it has been built; in this chapter I will assume you want to install it to the following directory on Windows:

```
C:\JavaTools\ant-1.7.0
```

Or the following directory on Linux or UNIX:

```
/opt/javatools/ant-1.7.0
```

Next, download any third party libraries you want to use with Ant and install them into the `lib` directory at the top level of the extracted source directory. Third party libraries typically add additional support to Ant⁵. Examples include support for JUnit, Ruby scripting and the FTP tasks. The FTP tasks are particularly useful and I will be using them in chapter 4. Figure 4 illustrates how the Ant source and installed binary directory structures will look.

To execute the installation process, start up a command prompt and navigate to the source directory; on Windows you can achieve this using the following commands:

```
cd C:\temp\apache-ant-1.7.0
set ANT_HOME=C:\JavaTools\ant-1.7.0
set CLASSPATH=%CLASSPATH%;.\lib
build.bat install
```

Whilst on Linux or UNIX:

```
cd /tmp/apache-ant-1.7.0
export ANT_HOME=/opt/javatools/ant-1.7.0
export CLASSPATH=$CLASSPATH:./lib
build.sh install
```

Note that the Linux/UNIX example assumes the use of the Korn shell or equivalent

⁵ For more information on the third-party libraries that can be downloaded and installed visit: <http://ant.apache.org/manual/install.html#librarydependencies>.

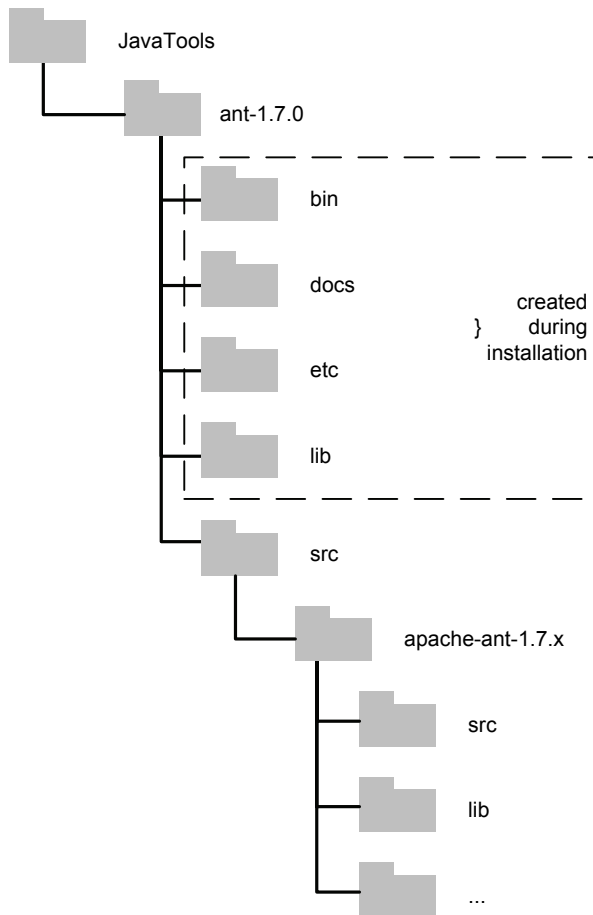


Figure 4 - Example Ant source and binary directories

Ant will now be compiled and installed in your chosen destination directory. Next you should set the `ANT_HOME` environment variable permanently. Either use the Windows "My Computer" applet or set it in your login shell (Linux or UNIX). Also, add the `%ANT_HOME%\bin` directory (`$ANT_HOME/bin` on Linux or UNIX) to your path so you can execute Ant from the command line. Finally, test that Ant is installed by entering:

```
ant -v
```

If successful, the version of Ant that is installed should be displayed, similar to the following:

```
Apache Ant version 1.7.0 compiled on December 13...  
Buildfile: build.xml does not exist!  
Build failed
```

Note that to make use of a common version of Ant in a team environment, you can either install Ant to a shared directory, or add it in to your version control repository.

The Ant build script

The input to Ant is an XML build script. By default it is called `build.xml`. Although any filename can be used it is best to keep the “.xml” extension. A complete application can be built using one or many build scripts. If there are multiple scripts, then they are commonly located at different component directory levels. Each build script describes a set of **targets**, which are the core parts of the composite build process, for example, compile source code, run unit tests, create a Java archive and so on.

Each of the targets can make reference to any of Ant's built-in **tasks**. A task is basically an interface to a Java object written to carry out a predefined operation, for example, compile a set of source files, create a java archive, delete a directory structure and so on. To define the tasks exact invocation, parameters can be specified as **attributes**, for example, the location of the Java archive on the file system. There are a large number of built-in tasks⁶, however since the tasks are interfaces to Java objects, you can quite easily extend Ant by creating

⁶ See <http://ant.apache.org/manual/> for the complete set.

new tasks yourselves. I will discuss how to achieve this in Appendix A.

As an example, calling Ant from the command line to execute a "release" target could be carried out as follows:

```
ant -f hoteldejava.xml release
```

Here the "-f" option is used to specify the name of the build script to find (`hoteldejava.xml`) and the target which will be executed is the "release" target. On start-up, Ant parses the build script to:

- Discover any errors in its construction.
- Determine the target that will be executed.
- Evaluate the dependencies on "other targets" that need to be executed too.

Although there is a lot more that happens behind the scene this is basic premise of Ant – along with many other build scripting tools too! However remember, Ant is not a compiler. It does not understand about underlying program structure or source code dependencies and it relies on the version of Java JDK compiler that you have installed.

Typical Ant project directory structure

Figure 5 shows a typical directory structure for a Java project to be built using Ant. The `build.xml` file is at the top level, as is the `src` directory containing all of the source code. The `build` and `dist` directories are “intermediate” directories whose contents are generated as part of the build process. The `test` directory contains the JUnit source code. Your own directory structure can be different, but this is a well known and common project directory structure.

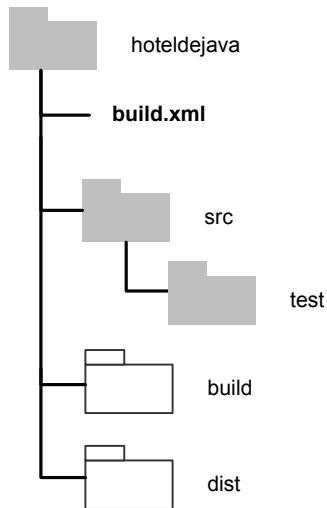


Figure 5 - Typical Ant directory structure

An example project

A working example of an Ant build script to compile and test a set of Java classes (for the *Hotel de Java* application) and produce a releasable Java archive is given below.

```
<?xml version="1.0">
<project name="hoteldejava" default="compile">

  <property name="dir.src"      value="src"/>
  <property name="dir.build"    value="build"/>
  <property name="dir.dist"     value="dist"/>
  <property name="dir.libs"     value="c:\libs"/>

  <!-- define classpath to be used -->
  <path id="classpath">
    <pathelement location="$dir.build"/>
```



```
<!-- include third party libraries -->
<fileset dir="${dir.libs}">
    <include name="*.jar"/>
</fileset>
</path>

<!-- initialise directory structure -->
<target name="init"
    description="initialise directory structure">
    <mkdir dir="${dir.build}"/>
    <mkdir dir="${dir.dist}"/>
</target>

<!-- remove generated files -->
<target name="clean" description="remove generated files">
    <delete dir="${dir.build}"/>
    <delete dir="${dir.dist}"/>
</target>

<!-- compile source code -->
<target name="compile" depends="init"
    description="compile source code">
    <javac destdir="${dir.build}" >
        <src path="${dir.src}"/>
        <classpath refid="classpath"/>
    </javac>
</target>

<!-- test source code -->
<target name="junit-all" description="run junit tests">
    <junit printsummary="on" fork="no" showoutput="true"
        haltonfailure="false" failureproperty="tests.failed">
        <classpath refid="classpath"/>
        <formatter type="plain"/>
        <batchtest todir="${dir.build}">
```

```
<fileset dir="${dir.src}">
    <include name="**/Test*.java"/>
</fileset>
</batchtest>
</junit>
<fail if="tests.failed">
    One or more tests failed. Check the output...
</fail>
</target>

<!-- create distribution archive -->
<target name="dist" depends="compile"
    description="create distribution archive">
    <jar jarfile="${dir.dist}/${ant.project.name}.jar"
        basedir="${dir.build}" includes="**/*.class">
</target>

</project>
```

The first thing that you will notice about this example is that the Ant build script is in valid XML format. This has both advantages and disadvantages. The advantages are that you can use XML editing tools and technologies to edit and transform the build scripts. For example you can quite easily turn the build script into HTML via XSLT⁷ technology. The main disadvantage however is that XML is not a natural human language. You will need to get used to defining both start tags (e.g. “<target>”) and end tags (e.g. “</target>”). It is very common to forget to close tags properly in XML. To mitigate this you can use an editor (such as Eclipse) that understands XML and offers you syntax checking and code completion capabilities.

⁷ <http://en.wikipedia.org/wiki/XSLT>

If this build script was used to build the *Hotel de Java* application then the output result would look similar to the following:

```
>ant -f hoteldejava.xml clean
Buildfile: hoteldejava.xml

clean:

BUILD SUCCESSFUL
Total time: 0 seconds

>ant -f hoteldejava.xml dist
Buildfile hoteldejava.xml

init:
    [mkdir] Created dir:C:\Build\hoteldejava\build
    [mkdir] Created dir:C:\Build\hoteldejava\dist

compile:
    [javac] Compiling 20 source files to
    C:\Build\hoteldejava\build

dist:
    [jar] Building jar:
    C:\Build\hoteldejava\dist\hoteldejava.war

BUILD SUCCESSFUL
Total Time: 4 seconds
```

In the build script a Java archive called `hoteldejava.jar` is being built. There are five targets: `init`, `clean`, `compile`, `junit-all` and `dist`. Sometimes a target will have a dependency on another target, for example before you create a distribution Java archive, you would need to compile the source code. These dependencies are easily specified via a **depends** attribute in the target. You can see this with

the `dist` target (where the target requires the `compile` target to be executed first). Ant will automatically execute any dependencies.

You will see that the first few lines of the build script make reference to **property** values. In Ant, properties are equivalent to most programming language's concept of variable constants; that is to say they are immutable and once properties have been assigned a value they cannot be changed. Here they are being used to define directory names. You make references to properties by using the `${property_name}` syntax.

If you are compiling source code in Java you will need to define a **Classpath**. This is the combination of libraries containing pre-built classes that you need to build against. In this example a Classpath is defined which looks at the `build` directory as well as the set of third party Java archives held at the location `c:\libs`. Note for simplicity, in this example I have not enforced the execution of the JUnit tests. However in a real world scenario, you should make sure that the `junit-all` target was also a dependency of the `dist` target.

One final point worthy of note is the use of Ant's built in variables. In this example, I am referencing the variable `${ant.project.name}` when creating a Java archive, this refers to the project name `hoteldejava` as defined at the project level. Consequently, the name of Java file produced will be `hoteldejava.jar`.

Summary

In this chapter, I have introduced some of the background and concepts of Ant. I have also looked at how to setup a working Ant environment and an example of an Ant build script. In the next chapter I will delve more deeply into Ant in order to explain its core terminology and concepts.

CHAPTER 2

Fundamental Concepts

*Learn the fundamentals of the game and stick to them.
Band-Aid remedies never last.
Jack Nicklaus*

There are a number of fundamental underlying concepts in Apache Ant that you should understand in order to be able to get the most out of it. This chapter looks at these in detail. It also looks at some of the common issues that users of Ant might come across and gives some suggestions on how they can be mitigated.

Properties

Using properties, you can configure Ant to execute projects for different scenarios and environments. Ant properties are essentially name/value pairs, defined in the following format:

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

Properties are case-sensitive and once defined they can be referred to using the “dollar/bracket” syntax, i.e. `${name}`. For example to create a property to set a “debug” flag you could define the following:

```
<property name="debug" value="false"/>
```

You would then refer to the value of this property in your build script as `${debug}`.

As I discussed in chapter 1, the most important point to remember about properties is that they are immutable – that is once they have been defined their value cannot be changed. For example, although it is legal syntax to include another definition of `debug` further down in your build script, its value will not be changed from its first definition. Although this might seem limiting, you can still override properties from the command line. For example, if you wanted to override the `debug` property, you could use the following invocation:

```
ant -f hoteldejava.xml compile -Ddebug=true
```

Having immutable properties also allows you to implement user and project specific property files. I will discuss why and how to achieve this in detail in chapter 3.

Built-in properties

As well as defining your own properties, Ant also has a set of built-in properties that you can make use of in your build script and that are listed in Table 1.

Property	Description
<code>basedir</code>	The absolute path of the project's base directory (as set with the <code>basedir</code> attribute of <code><project></code>).

<code>ant.file</code>	The absolute path of the build file.
<code>ant.version</code>	The version of Ant.
<code>ant.project.name</code>	The name of the project that is currently executing; it is set in the name attribute of <code><project></code> .
<code>ant.java.version</code>	The JVM version Ant detected; it can hold the values “1.1” to “1.6”

Table 1 - Ant built-in properties

A larger number of properties are also inherited via the Java system (such as “user.home”, “os.name”, “java.class.path”, “java.library.path” and so on). You can make use of these directly in your build script. In order to see what these properties have been set to in your environment, you can execute the “ant -diagnostics” command. You can also access any operating system environment variable by setting an environment prefix, as in the following:

```
<property environment="env"/>
<echo message="ANT_HOME is set to = ${env.ANT_HOME}"/>
```

This example creates a prefix called `env` which you can use to reference any of your operating system environment variables as Ant properties.

Property files

As well as defining properties directly within your build script, or via the command line, you can also create specific property files. These are text files which use the “name=value” syntax to define the properties. For example, you could create a property file containing the following:

```
dir.build = bin
debug = false
name.javac = javac1.4
```

By convention, these files are usually given the extension “.properties”. For example this file might be called “default.properties”. Once you have defined a property file, you can then import it using the `<property>` task as follows:

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

You can also specify a prefix to be placed before all imported properties if required, for example:

```
<property file="default.properties" prefix="dp"/>
<echo>${dp.debug}</echo>
```

This can be useful for debugging purposes if you have different “types” of properties. Property files can be very powerful if used correctly and will be discussed in more detail in chapter 3.

Targets and tasks

Targets and tasks implement the core functionality of your build script. They are typically defined in the following format:

```
<target name="target" depends="target2">
    <task attr1="attr1"/>
</target>
```

A target is a set of tasks that you want to execute, for example, create a set of initial directories. A task is a piece of Java code that can be executed to fulfil a build operation, for example if the target is to create the set of initial directories, then the individual tasks will be

calls to the Ant `<mkdir>` task. Tasks can take any number of attributes as parameters. For example the attributes to the `<mkdir>` task would obviously include the name of the directory to create. There are many built in tasks in Ant, however it is also easy to create your own. I will be discussing how to achieve this in appendix A.

Each project build script usually defines a default target that is to be executed, for example:

```
<project name="hoteldejava" default="compile">
```

If you type in "ant" and don't specify a target, it will default to the target specified as "default" in the project element – in this case the default target is "compile".

To execute a target other than the default one, specify its name on the command line, for example to execute "target2" you would invoke Ant as follows:

```
ant target2
```

If you do not specify a build script, by default Ant will look for a file called "build.xml" in the current directory. To use another file you can use the "-buildfile" or "-f" option. For example to use the "mybuild.xml" build script, you would invoke Ant as follows:

```
ant -f mybuild.xml
```

To control the flow of your build script, targets can optionally have dependencies on other targets. Target dependencies are executed in the order that they are specified; for example given the following extracts from a build script:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="A"/>
<target name="D" depends="C"/>
```

Executing target B would execute A first then B.

Executing target D would execute A first then C, then finally D.

You can also carry out basic logical operations with targets. To achieve this you use properties and the target “if” and “unless” attributes. For example, give the following extracts from a build script:

```
<target name="A" if="propertyA"/>
<target name="B" unless="propertyA"/>
```

Target A will only be executed if `propertyA` has been assigned a value. Conversely, if `propertyA` has been assigned a value then target B will not be executed. One way you could use this capability is to have a “dump” target which printed out values only if the `debug` property was defined, for example:

```
<target name="dump" if="debug">
    <echo>dir.build = ${dir.build}</echo>
    <echo>dir.dist = ${dir.dist}</echo>
    ...
</echo>
<target name="init" depends="dump">
    ...
</target>
```

Consequently, when you executed the Ant command:

```
ant init -Ddebug=true
```

The value of your properties would be output using the Ant `<echo>` task. For this target to work, you would need to make sure that the `debug` property was only ever defined on the command line and not in the build script or in an external property file.

Types

Whereas Ant tasks are used to execute specific functionality, Ant types are typically used to group or manipulate resources such as files, directories, permissions or properties. There are a large number of types but they can more or less be broken down into the following categories:

- *Path-like structures* – used to refer to collections of files or directories as operating system paths or Java Classpaths.
- *Resource collections* – used to define collections of files or directories for inclusion in the path-like structures of supported tasks.
- *Mappers* – used to map file or directory names from one format to another.
- *Redirectors* – used to redirect the output of supported tasks.
- *Filters* – used to replace tokens from files with new values.

As with tasks you can implement your own types; however you are probably less likely to need to implement your own “type” functionality. In this section I will concentrate on describing the most commonly used types: path-like structure and resource collections.

When you are first learning to use Ant, its path-like structures and resource collections will probably be the capability which will (initially at least) be the hardest to learn and understand. Path-like structures are used for referring to many things: a collective set of source files, a Java Classpath, or a set of libraries to build against. They can be specified directly using standard operating system ":" or ";" separators (for example “a.jar;b.jar;c.jar”). But more likely they are specified as nested elements, as in the following:

```
<classpath>
  <pathelement location="lib/mylib.jar"/>
  <pathelement location="build/myotherlib.jar"/>
</classpath>
```

This example creates a new Java Classpath that refers to two physical Java archives.

Resource collections are used to specify the physical “collection” of files that path-like structures refer to. There are a number of resource collection types including `<fileset>`, `<dirset>`, and `<filelist>`. For example, the following `<fileset>` will match all Java source files in the `src` directory (but not unit test files, i.e. those whose names start with "Test"):

```
<fileset dir="${dir.src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/Test*" />
</fileset>
```

Note how regular expressions are used to “collect” the set of files together, rather than you having to specify each entry them manually yourself.

Resource collections can either be used inside path-like structures, as in the following example:

```
<classpath>
  <fileset dir="${dir.lib}">
    <include name="**/*.jar"/>
  </fileset>
</classpath>
```

Or within tasks which support them, for example the Ant `<copy>` task:

```
<copy todir="../dest/dir">
  <fileset dir="${dir.dist}">
    <exclude name="**/*.jar"/>
  </fileset>
</copy>
```

This particular example will copy all the built Java archives to another directory - possibly for sharing and reuse.

Rather than specifying the same resource collection each time you can make use of previously defined collections using the “id” and “refid” attributes, for example:

```
<path id="base.path">
  <fileset dir="${dir.lib}">
    <include name="**/*.jar"/>
  </fileset>
</path>

<path id="tests.path">
  <path refid="base.path"/>
  <pathelement location="testclasses"/>
</path>
```

This example creates a path-like structure called “base.path” which refers to all the Java archives located in the `${dir.lib}` directory. However, it also creates a second path-like structure called “tests.path” which reuses the first definition but also includes the directory “testclasses”.

A refinement on this practice is make use of the `<patternset>` type which is used to group sets of files together that can be referenced later via its “id” attribute. For example, you could create a reusable reference to match all "releaseable" Java archives (but excluding stubbed interfaces) as follows:

```
<patternset id="release.libs">
  <include name="**/*.jar"/>
  <exclude name="**/*stub*.jar"/>
</patternset>
```

This `<patternset>` could subsequently be reused as follows:

```
<fileset dir="${dir.lib}">
  <patternset refid="release.libs"/>
</fileset>
```

The advantage of `<patternset>` types is that they can be defined at the top level of your build script for reuse (or even in external files as I will discuss in chapter 3).

Common issues with Ant

Although I believe Ant is a very good build scripting tool, like all tools it is not perfect. In this section I will discuss some of the common issues that users of Ant typically come across and give some suggestions of how to work around them.

Maintaining library dependencies

If you took a typical enterprise Java application and decomposed it down into the libraries that were developed or reused to implement it, you could quite easily find upwards of 100 different libraries (Java archives). A number of these libraries might be open source implementations, such as the Jakarta commons libraries¹; whilst others might be commercial business components that you have sourced from a third party. Unfortunately, since many commercial components also

¹ <http://jakarta.apache.org/commons>

make extensive use of open source components themselves, you can quite easily find yourself with duplicate sets of libraries at different versions levels!

Similarly, when you are developing your own common libraries, you will often be creating multiple versions of them and will probably need to support different versions for different projects (or for projects in different phases). Following on from this, library dependencies can often have dependencies on other libraries themselves (which is called **transitive dependencies**).

If this all sounds like a nightmare problem, then it is. The typical way that organizations solve this problem is to create and/or baseline a single shared repository of libraries - sometimes in a version control system, sometimes on a networked file system. This approach can work but requires a significant amount of administration to keep it up to date. This is especially true for transitive dependencies where you would need to somehow store associative metadata alongside the components.

Fortunately, there are now a number of Java tools that can help you solve this problem. These include Apache Maven (in the form of Ant tasks) and Apache Ivy (<http://incubator.apache.org/ivy>). Both can be made to work with Ant, however since Maven could be seen as the next generation of Java build tool, I recommend looking at its implementation first – doing so will give you a possible future upgrade path. In chapter 3 I will describe in detail how to implement library management with the Ant tasks for Apache Maven.

Portability

Although Ant is written in Java and can produce portable build scripts you still need to be careful when platform-dependent issues arise. A typical example is the difference between paths on the various operating systems, for example `C:\build\mydir` on Windows or `/build/mydir` on Linux or UNIX. If you define relative paths in

your build scripts (such as “../release”) then Ant will successfully translate these for you, however if you define absolute paths it will have problems.

To work around this you can make use of the `<pathconvert>` task. For example, suppose that you had a deployment tool that required the fully qualified (absolute) path of a file as one of its inputs. To make your build script portable, you could use `<pathconvert>` to format the input as follows:

```
<pathconvert targetos="windows" property="absEarPath">
  <path>
    <pathelement
      location="${dir.dist}/${ant.project.name}.ear"/>
    </path>
  </pathconvert>
```

This example will convert the path location of the local “.ear” file into Windows format and place the output in the `absEarPath` property. To convert to UNIX format, you would similarly set the “targetos” attribute to “unix”. Since you can retrieve the version of the operating system from the Java environment “os.name” property, you could use target “if/unless” attributes to call `<pathconvert>` correctly for either environment.

Debugging

Ant is not the easiest tool to debug. Unlike a traditional programming language there is no debugger that you can easily step through. Since Java classes are being called, if an error occurs you might even end up with a complete and complex stack trace. This might be useful to the developer of the class but not to someone who is writing a build script that uses it. Debugging Ant build scripts is therefore not trivial but there are a few practices you can use.

To get Ant to display verbose output you can use the “-v” command line option. Similarly, to display even more “debug” type output you can use the “-d” option. For example, to use both verbose and debug options you would invoke Ant as follows:

```
ant -d -v target
```

Note that turning both of these options on can produce a significant amount of output, it might therefore be better to redirect this output to a file so you can search through it. To display information about the version of Ant that is running and its environment, you can also use the “-diagnostics” option as discussed earlier.

To display information about what is happening in your build script you can use the `<echo>` task to output debug text and property values; for example:

```
<echo>checking in file ${name.idfile}</echo>
```

You can also use the `<echoproperties>` task to output all of the defined properties and their values. Note that a good way of using this task is to dump (and possibly version control the property settings) for each of your builds as follows:

```
<echoproperties destfile="audit.properties"/>
```

In this way you have an audit of the property values in case you need to understand the environment in which a previous build was executed.

Perhaps the best way to debug your Ant build scripts however, is to execute them from within an IDE, as most add some additional capabilities for logging Ant output and controlling its execution. Eclipse² for example, has an excellent Ant editor and launcher which allows you to log and pinpoint build errors more efficiently than the command line.

² www.eclipse.org

Build avoidance

By default, Ant uses a simple build avoidance mechanism; it works out what to compile based on basic out-of-date rules, that is to say only “.java” source files that have no corresponding “.class” file or where the “.class” file is older than the “.java” file will be compiled. It will not however, rebuild classes where the files that they depend upon change, such as a parent class, an imported class or a change in interface. In some cases this can lead to runtime errors!

Most projects understand and accept this situation and subsequently create and make judicious use of an Ant “clean” target to remove all the “.class” files before compilation. In this case every build is a full re-build, however Java build times are typically not that long (less than an hour) and therefore this is acceptable. However, if this is not acceptable then an alternative is to use the Ant <depend> task to carry out some basic dependency checking.

The <depend> task works by determining which classes are out of date with respect to their source files, and then parsing the source code to remove the “.class” files of any other classes which depend on the out-of-date classes. To enable dependency checking, simply pass the “.java” source and “.class” build directories to the <depend> task before your <javac> task as in the following:

```
<target name="clean">
    <delete dir="${dir.build}"/>
    <delete dir="depcache"/>
</target>

<target name="compile" depends="init">
    <depend srcdir="${dir.src}" destdir="${dir.build}"
        cache="depcache">
        <include name="**/*.java"/>
    </depend>
```

```
<javac destdir="${dir.build}">
    <src path="${dir.src}"/>
</javac>
</target>
```

This example references a cache called `depcache`; this is used to cache the dependency information. The “`clean`” target has also been updated to remove the dependency cache; however you would obviously not want to call the “`clean`” target prior to each build.

This might all sound positive, but in practice dependency checking is not infallible because it does not understand all the nuances of Java coding. Therefore if you do use it, I would still recommend executing full “`clean`” rebuilds on a periodic basis – maybe on a nightly or weekly basis - so that you can uncover any possible issues.

Defensive programming

When programming code you should always try and foresee any problems that might occur, catch them and recover gracefully. This is called defensive programming. When you are writing build scripts (which can often end up quite long and complicated) you should always try and follow similar defensive programming practices. It is not always that easy to do so in Ant but there are a number of ways you can achieve it.

One way of achieving this is make use of the `<fail>` task to check to see that properties have been set. For example, suppose your build script depended on the property `vc.provider` being set (either on the command line or in a property file). To halt the execution of the build script with an appropriate message, if the property was not set, you could use the following:

```
<fail unless="vc.provider"
    message="Error: vc.provider property has not been set"/>
```

Note it is also possible to make the `<fail>` task set an exit code number so that you could use it in any scripts that call Ant.

Since Ant is extensible, there are lots of add-on tasks and libraries that are available to use with it. If you make use of any of these add-on libraries, you should always check that the relevant Ant task, script or macro is available. To achieve this you can use the `<typefound>` condition as in the following:

```
<fail message="Error: cannot find ccexec task">
  <condition>
    <not><typefound name="ccexec"/></not>
  </condition>
</fail>
```

This example checks for the existence of the `<ccexec>` task (which is a task I will develop in appendix A) and if it is not available it will use the `<fail>` task to halt the build script with an appropriate message. Note that this example makes use of the `<condition>` task which is a powerful way of executing Boolean logic functions in Ant and which I will describe in detail in chapter 3.

XML syntax

One of the major issues with Ant is that its build scripts need to be expressed in XML. This is not a natural language for humans. XML was also not really designed to be concise - you can quite easily end up with large and complex build scripts. It is difficult to express conditional (if/else) logic in XML and so Ant's capabilities in this area are somewhat lacking. However there are a number of different conditions with Boolean support available and I will discuss them in more detail in chapter 3.

Properties

If you remember one thing from the first few chapters it is the fact that properties in Ant are immutable! They are not variables as per your usual programming language. However, one other issue is that you cannot use one property to hold the "name" of another property and resolve it at reference time; for example the following will not work as you might expect:

```
<property name="propB"      value="${propA} jnr"/>
<property name="propA"      value="fred"/>
<echo>propA = ${propA}; propB = ${propB}</echo>
```

Instead of setting `propB` to “fred jnr” it would set it to “`${propA} jnr`” because the value of the `propA` at first reference time was not set!

Summary

In this chapter I have introduced a number of the most important concepts and terminology of Ant and some of the issues that you might come across. Hopefully by now you understand that it is quite easy to get up and running with Ant. However, from personal experience it is also quite easy to create an unmanageable mess – particular for large enterprise projects. In the next chapter therefore, I will look at how to use some of Ant’s capabilities “in practice”, so that you can manage build scripts for large projects successfully. I will also look at how you can potentially reuse build scripts across projects.

CHAPTER 3

In Practice

*In theory, there is no difference between theory and practice;
In practice, there is.*
Chuck Reid

This chapter looks in detail at how Ant is implemented on real world projects. More specifically, it illustrates how some of Ant's capabilities can be used to support the needs of large projects and how Ant can be used across multiple projects.

Creating configurable builds

Rather specifying properties individually in your build script, you can move their definition out to specific property files. This effectively separates your control logic from your data, applying a Model-View-Controller¹ (MVC) like design pattern. This will ultimately make your builds scripts more readable and reusable. One common convention is

¹ <http://java.sun.com/blueprints/patterns/MVC.html>

to create a file called `default.properties` at the same level as your `build.xml` file. This file could contain properties such as:

```
# default properties
compile.debug = false
compile.compiler = javac1.4
repository = HotelDeJava
build.admin = kevin
```

You could then import the contents of this file, using the `<property>` task at the top of your build file, as follows:

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

Remember that properties defined in such files can always be overridden from the command line, for example to override the `compile.debug` property you would use:

```
ant -Dcompile.debug=true
```

However, this would be inefficient if developers wanted to override a number of properties. A better mechanism therefore is to allow developers to override "system-level" properties by using a shadow properties file, usually called `build.properties`. The `build.xml` file should then import these two files in the following order:

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

As properties are immutable any entry in the `build.properties` file would retain its value and not be overwritten by a corresponding entry in the `default.properties` file.

To make this approach work, you should ensure that the `default.properties` file is held under version control so that every developer picks it up. However, the `build.properties` is user

specific; it should be created on demand and should therefore never be placed under version control. To enforce this, you might want to add it to your version control tools list of “ignored” files.

Automating build numbering

You can use Ant’s `<propertyfile>` task to automatically create and update a property file. Consequently, this task can be used to create a property file that auto-increments a build number and/or date-time stamp. The values in this property file can then be used as part of your version control baselining or packaging process². To achieve this, create a target in your build script similar to the following:

```
<target name="update-buildinfo">
  <propertyfile file="buildinfo.properties"
    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>
</target>
```

If this target is executed, it will create a property file called `buildinfo.properties` with contents similar to the following:

```
#Build Information File - DO NOT CHANGE
#Wed Apr 1 17:48:22 BST 2007
```

```
build.num=0006  
build.date=01.04.2007 17\ :48
```

You can then import and use these properties using the `<property>` task as follows:

```
<property file="buildinfo.properties">  
<makebaseline baselineselector="${build.num}" ...>
```

Note the `<makebaseline>` task is not an existing task it is just used as an example of how you would call a baseline command for a version control tools. I will discuss how to actually achieve this for a number of version control tools in chapter 5.

Orchestrating builds

Although I believe Ant is not a complete build and release framework, it can still be used to orchestrate an end-to-end build process for individual projects. Such a process could either encompass the execution of a number of distinct tasks for a single project, or it could encompass the coordination of tasks across a number of related project components. In this section I will discuss how both of these can be achieved.

Cumulative targets

A cumulative target is a single Ant target which orchestrates an end-to-end build by calling a number of other dependent targets. There are two ways to achieve this in Ant. You can achieve it via the `<antcall>` task as follows:

² I will also look at a way to manually prompt for and verify a release number in chapter 4.

```
<target name="release">
  <antcall target="update-buildinfo"/>
  <antcall target="compile">
    <param name="param1" value="value"/>
  </antcall>
  <antcall target="junit-all"/>
</target>
```

Or via dependencies, as in the following:

```
<target name="release" depends="update-buildinfo",
"compile", "junit-all", ... />
```

Although `<antcall>` is useful for redefining the value of properties at call time, it is not recommended for use with internal targets³ as the complete build file is reparsed and all the targets that the called target depends on are re-run!

Chaining build scripts

Large projects tend to develop build scripts per component and then create a top level, orchestrating build script to chain them together. In Ant this can be achieved via the `<subant>` task as in the following:

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

This example will invoke the target `compile` in each `build.xml` file it finds from the current directory down. One word of warning though,

³ Internal targets are targets defined in the same build script

with the `<fileset>` task there is no guarantee on the order that the build scripts will be found and called. If you require your scripts to be executed in a specific order you should list all the `<subant>` tasks for each directory, in build order.

Reusing build scripts

One of the most important practices in software development is reuse⁴. This is particularly true with build processes, where you are effectively implementing the same or similar processes over and over again. Making build scripts reuseable not only allows you to use the scripts across projects, it can also cut down on the amount of scripting that is required – significantly increasing readability and maintainability. In this section I will discuss some of the capabilities of Ant that can enable reuse.

Reuseable macros

To reduce the potential for duplication you can create reusable routines using the `<macrodef>` task, which as its name suggests creates a “macro” routine. For example to create a macro for simplifying the steps needed to execute a version control tool command, you could implement the following macro:

```
<macrodef name="vc-op">
  <attribute name="file"/>
  <attribute name="command"/>
  <attribute name="comment" default="vc-op"/>
  <sequential>
    <exec>
      <arg value="vc_tool.exe"/>
```

```
        <arg line="-c '{@comment}' '{@command}'"/>
        <arg value="@{file}"/>
    <exec>
</sequential>
</macrodef>
```

In this example you can see that a new macro called “vc-op” is being created. It takes three parameters (or attributes): the name of the file to execute the command on, the command to execute (for example “checkout”) and the comment to apply. These attributes are then used in the `<exec>` task to execute the command line program “vc_tool.exe” with the parameters supplied. Note that the `<exec>` task is executed exactly the same as it would be outside of a macro; however inside a macro it always needs to be wrapped inside a `<sequential>` task to serialize its execution. Also, instead of referring to properties using the `${name}` syntax, attributes are referred to using the `@{name}` syntax.

In order to execute this macro you could then add the following line to your build script:

```
<vc-op command="checkout" file="${build.info}"/>
```

Note that the “comment” attribute has not been supplied because it will be automatically set by the macro.

Pre-defined tasks

As well as creating macros, you can also create pre-defined definitions of a task and its related attributes using the `<presetdef>` task. For example to define standard settings for the `<javac>` task, you could use the following:

⁴ Sometimes known as “steal-with-pride”!

```
<presetdef name="std.javac">
  <javac debug="${compile.debug}"
    deprecation="${compile.deprecation}"
    srcdir="${dir.src}"
    destdir="${dir.build}"/>
</presetdef>
```

In this example a new preset call “std.javac” has been created with four of its typical attributes. In order to execute this macro and override just the debug setting, you could then use:

```
<std.javac debug="true" classpath="my.classpath"/>
```

Note that none of the `deprecation`, `srcdir` or `destdir` attributes need to be supplied, and that additional attributes – in this case `classpath` – can be supplied as well.

Importing build scripts

Once you have started to build up a library of macros and presets you can place them in a separate file to be imported. In the same file you could also place a standard set of targets. I like to create two files:

- `common-macros.xml` - which contains common macros and preset definitions, as well as reusable `<patternset>` definitions.
- `common-targets.xml` - which contains common targets.

You can then import the contents of these files using the Ant `<import>` task. For example, if you create an Ant build script called `common-targets.xml` which contains the following:

```
<project name="common-targets">
```

```

<target name="compile" depends="init"
  description="compile source code">
  <javac destdir="${dir.build}" >
    <src path="${dir.src}"/>
    <classpath refid="classpath"/>
  </javac>
</target>

<target name="dist" depends="compile"
  description="create distribution archive">
  <jar jarfile="${dir.dist}/${ant.project.name}.jar"
    basedir="${dir.build}" includes="**/*.class">
  </target>

</project>

```

Then you could import the tasks from this file into a project build script as follows:

```

<project name="HotelDeJavaWeb" default="compile">

  <property name="dir.common" value="../common"/>

  <import file="${dir.common}/common-targets.xml"
    optional="false"/>
  <import file="${dir.common}/common-macros.xml"
    optional="false"/>
  ...
  <!-- override of dist target -->
  <target name="dist">
    <war warfile="${dir.dist}/${ant.project.name}.war"
      webxml="WebContent/WEB-INF/web.xml"
      manifest="${dir.build}/manifest.mf">
    <fileset dir="WebContent">
      <patternset refid="web.sources"/>
    </fileset>
  </target>

```

```
<zipfileset dir="WebContent/images" prefix="images"/>
<classes dir="${dir.build}">
    <include name="com/ratlbank/**/*.class"/>
    <include name="org/apache/jsp/**/*.class"/>
</classes>
<lib dir="${dir.weblib}"/>
</war>
</dist>

</project>
```

In this example, the `<import>` task has set the `optional` attribute to “false”, this means that the build script will fail if this file is not located. Also, note that although there is a definition for the `dist` target in the `common-targets.xml` file, you also have the option of overriding it in the project’s build script – as in this case where I am creating a web archive (“.war” file) rather than a Java archive (“.jar” file).

Implementing conditional logic

Ant was not really developed as a general purpose scripting language; consequently it has limited support for conditional execution. However, it is still possible to implement quite a few logical scenarios using Ant’s limited functionality. If you require more functionality then additional scripting integration or third-party libraries can be used.

In chapter 2 I discussed the fact that targets can implement a basic form of conditional logic using properties, for example:

```
<target name="verification" if="release">
```

In this case the contents of the target “verification” would only be executed if the property “release” had been set to a value. In

practice, this form of conditional logic is rarely used in isolation as it gives you limited flow of control. However, it is often used with other condition logic.

Ant has a general `<condition>` task that can evaluate the grouping of multiple logical conditions. Rather than simply looking at property values, the `<condition>` task can group together individual conditions that query your environment. Some of the individual conditions that can be carried out include:

- test for the existence of files and directories
- test whether the operating system is of a give type
- test whether a specific web (HTTP) page exists
- test if a string matches a specified regular expression

As an example, suppose that you wanted to execute a release deployment only if the environment was completely setup, you could create targets similar to the following:

```
<target name="release-check">
  <condition property="release">
    <and>
      <available file="release.properties"/>
      <isset value="${deploy.password}"/>
    </and>
  </condition>
</target>

<target name="deploy" depends="release-check"
  if="${deploy.password}">
  <echo>carrying out a release deployment</echo>
  ...
</target>
```

This example would look to see if both a property file called “release.properties” existed and a property called “deploy.password” had been set to a value (probably passed in from the command line). It groups these two individual conditions together in an `<and>` element, and if both conditions are met then the property `release` is given a value. You can specify what the value is if you need to but the default is “true”. This property is then used in the “deploy” target via the “if” attribute.

As well as the `<and>` element, a number of the usual logical operators are also available, including `<not>`, `<or>` and `<xor>`. Although quite a few conditions are supplied (and more are added as Ant evolves), if you find that you need to test for another type of condition, you can always write your own.

Extending Ant via scripting

Although in general Ant is not a scripting language, it can call languages that are via the `<script>` task. This allows you to implement significantly more complicated logic in your build scripts. Supported languages include JavaScript, Ruby, Groovy and Python. For example, to execute JavaScript in your build you could use the following:

```
<project name="scriptdemo">

<target name="main">
  <script language="javascript"> <![CDATA[
    for (i=1; i<=10; i++) {
      echo = scriptdemo.createTask("echo");
      echo.setMessage("Testing" + i);
      echo.perform();
    } ]]>
</script>
```

```
</target>
```

```
</project>
```

This example uses a JavaScript “for” loop to create a set of Ant tasks and print out a simple message. Note that all of Ant’s core elements (tasks, targets, and so on) are accessible from the script, using either their `name` or `id` attributes.

Using the scripting capabilities, you can also define new tasks whose implementation is a language specific script. This is achieved via use of the `<scriptdef>` task. For example, to create a new task (this time using Ruby⁵) that will construct a test file of a specific byte size, you could implement the following script:

```
<scriptdef name="mktestfile" language="ruby">
  <attribute name="filename"/>
  <attribute name="filesize"/>
  attr=$bsf.lookupBean("attributes")
  filename=attr.get("filename");
  filesize=attr.get("filesize");
  print "creating file #{filename}\n"
  f = File.new(filename, "w")
  1.upto(filesize.to_i) {f.putc("X")}
  f.close
</scriptdef>
```

Note that the `<scriptdef>` task is similar to the `<macrodef>` task I discussed earlier and takes attributes as its parameters. To execute this task to create a file of say 100 bytes, you would use the following:

```
<mktestfile filename="test.dat" filesize="100"/>
```

⁵ <http://sourceforge.net/projects/ruby>

Scripting can be a powerful additional “weapon” in Ant. If you need to implement some additional functionality then you can do everything in scripting that you could do with a new task written in Java. Personally, I prefer to implement new tasks, conditions and so on in Java as that makes them more reusable across implementations. However if you are better at language scripting than coding Java then this approach makes a viable alternative.

Note that if you are going to be using scripting, you will definitely need to recompile Ant. This is because scripting support (and your chosen scripting language) is not automatically compiled in. To do this, examine the library dependencies for your chosen scripting language⁶, then recompile Ant as described in chapter 1.

Implementing library management

In chapter 2 I discussed the issues that Java projects can face when managing large numbers of Java libraries and their dependencies. In this section I will describe how the Ant tasks for Apache Maven (<http://maven.apache.org/ant-tasks.html>) can be implemented to help solve this issue.

Installing the Ant tasks for Apache Maven

The Ant tasks for Apache Maven are available as a Java library from the Apache Maven website. You can download the latest version of the tasks from <http://maven.apache.org/download.html>. There should be a single Java library. You can either install this library to your Ant `lib` directory (for example, `%ANT_HOME%\lib`) or to a directory alongside your top-level `build.xml` file. I prefer to install the Java

⁶ <http://ant.apache.org/manual/install.html#librarydependencies>

library into a directory called `lib` alongside the `build.xml` file, and then place it under version control.

Once you have done this you will need to change your `build.xml` to reference the tasks “namespace”. You can achieve this by adding the following to your top-level `<project>` element:

```
<project ... xmlns:artifact="urn:maven-artifact-ant">
```

Namespaces allow you to reference third party libraries and their tasks. In this case the namespace being created is called “artifact” and refers to the Uniform Resource Name (URN) “maven-artifact-ant”. In order for Ant to be able to locate this URN, you will also need to define a Classpath where Ant can locate the library as follows:

```
<typedef uri="urn:maven-artifact-ant"
  resource="org/apache/maven/artifact/ant/antlib.xml">
  <classpath>
    <pathelement location="${dir.lib}/
      maven-artifact-ant-2.x.x-dep.jar" />
  </classpath>
</typedef>
```

This `<typedef>` element references the Ant tasks for the Apache Maven Java library that you have downloaded and installed – in this case in the `${dir.lib}` directory.

Using Apache Maven repositories

Apache Maven uses file system repositories to store and organize the Java libraries that you will be using as part of your build process. Alongside the Java libraries themselves, it also stores metadata representing their versions and dependencies. By default this repository is located on the Internet at <http://ibiblio.org/maven2> (and is mirrored around the world), however you also can implement internal

repositories for your own organization. One obvious reason for wanting to do this is for security purposes – you probably would not want to publish your companies internally developed libraries on the Internet.

The basic structure of any Maven repository is relatively simple and is illustrated in Figure 6.

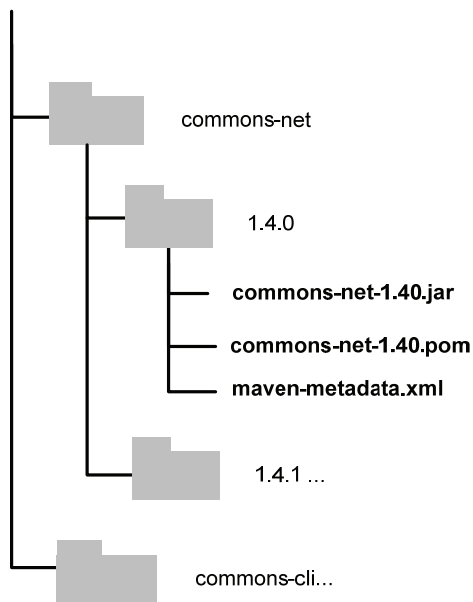


Figure 6 - Example Maven repository

In this example I am focusing on a specific directory in the Maven `ibiblio` repository for the Jakarta `commons-net` component⁷. This is a popular component which contains a collection of network utilities and protocol implementations such as FTP, SMTP and Telnet. Inside the `commons-net` directory of the Maven repository is a sub-directory

for every version of `commons-net` that has been released. Inside these sub-directories are three types of files:

- The Java libraries which contain the implemented functionality. In Figure 6 there is only a single library called `commons-net-1.4.0.jar` (although the directory could contain several libraries).
- The Project Object Model (POM) file for the component, in this case `commons-net-1.4.0.pom`.
- The `maven-metadata.xml` which is an internally managed file used to mark out the fact that this is a Maven repository

There will also be a number of checksum files but these are managed by Maven as well.

Maven uniquely identifies each version of a library via its filename, for example, `commons-net-1.4.0.jar`. However, it also allows you to manage snapshots, which are the latest but unreleased versions of libraries. Snapshots are marked via a “-SNAPSHOT” identifier in the version name. Although Maven will help you manage these files in its repository, creating checksums, directory structures and so on, it does not automatically know about dependencies between libraries. This is where the POM file comes in.

In the full version of Apache Maven, POM files replace `build.xml` files as the build scripts - defining how to build a project or component. Included in this file is the list of dependent libraries that are required for the build to be successful. For example, if you looked

⁷ <http://jakarta.apache.org/commons/net>

at this file for `commons-net` you would see that it contains the following lines:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>commons-net</groupId>
  <artifactId>commons-net</artifactId>
  <name>Jakarta Commons Net</name>
  <version>1.4.0</version>
  ...
  <dependencies>
    <dependency>
      <groupId>oro</groupId>
      <artifactId>oro</artifactId>
      <version>2.0.8</version>
    </dependency>
  </dependencies>
  ...
</project>
```

In this example you can see that the initial elements define information about the library and how to access it from within the repository - via its “`groupId`” and “`artifiactId`”. You can also see the set of additional libraries that this library is dependent on; in this case it is the Jakarta ORO (regular expression) library version 2.0.8. You only really need to be concerned with the content of POM files if you are going to be creating your own Java libraries and publishing them to the Maven repository. If this is the case then you will need to create POM files for each component that you wish to publish. I will discuss how to achieve this later.

The way that the Apache Maven dependency management feature works is that when you carry out a build, it cycles through all these dependencies and downloads the appropriate versions from the site repository to a user’s local repository (by default this repository is

called “.m2” and is located in your home directory). The local repository is essentially your own personal cache – and is automatically maintained – you do not need to do anything to update or refresh it.

Implementing your own repository

If you want to make use of the Internet repository then you need to do nothing to specify the location of the library repository. However, if you wish to implement your own site based repository, along side or instead of the Internet repository, then you need to specify the location of this repository in your `build.xml` file. You can achieve this by including a line similar to the following:

```
<artifact:remoteRepository id="internal.repository"
    url="http://myhost/maven2"/>
```

In this example an internal repository is being used which is accessible via a Web server on the local Intranet at `http://myhost.maven2`. This repository is going to be referenced using the id “internal.repository”. If you do not have a Web server you can still specify a URL for a file system location, for example, `file:///shared/maven2`.

You can initialize your own repository by simply creating an empty directory on a file system. If you have an existing set of internally developed Java libraries then you can import them into your repository using the Apache Maven `deploy` command. For example, to install the Java library `HotelDJEJBClient.jar` at version 1.0.1 you would use the following command:

```
mvn deploy:deploy-file -DgroupId=hoteldejava -
-DartifactId=hoteldejavaejbclient -Dversion=1.0.1 -
-Dpackaging=jar -Dfile=./HotelDJEJBClient.jar -
-DrepositoryId=repository -Durl=file:///shared/maven2
```

In order to use this command you will need to download Apache Maven itself since it requires the “mvn” binary. One particular point to consider is that only open source libraries are contained in the ibiblio repository, not those with click-through licenses like Sun’s. Therefore if you are going to build anything which is based on or uses Sun’s libraries then you will need to install the relevant Sun Java libraries⁸ using this installation process.

Resolving Java libraries

Once you have installed the Ant tasks for Apache Maven, in order to specify the versions of libraries you wish to build against I recommend creating a new properties file called `library.properties`. You can then populate this file with the versions of libraries that you wish to build against, for example:

```
ver.hoteldejava.ejbclient = 1.1
ver.commons-net = 1.4.0
```

Since you are going to compile against these libraries, you will need to setup a Java Classpath to contain them. Fortunately, the Ant tasks for Apache Maven can do this for you automatically. Therefore, to specify and resolve the Java libraries that you need and to automatically create a new Classpath you would include the following in your `build.xml`:

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

<artifact:dependencies pathId="maven.classpath">
  <remoteRepository refid="internal.repository"/>
  <dependency groupId="hoteldejava"
```

⁸ See <http://maven.apache.org/guides/mini/guide-coping-with-sun-jars.html>

```
artifactId="ejbclient"
version="${ver.hoteldejava.ejbclient}"/>
<dependency groupId="commons-net"
artifactId="commons-net"
version="${ver.commons-net}"/>
</artifact:dependencies>
```

In this case I have specified that the Java libraries are to be downloaded from the internal repository. The Ant tasks for Apache Maven will therefore download the versions of the libraries from this repository to the “.m2” directory in your home directory – if they do not exist there already. If the libraries that you require have dependencies on other libraries, these will be downloaded too – this is the transitive dependencies feature in action. You can then use the generated `maven.classpath` in your Ant build script, for example to compile your project you could use a line similar to the following:

```
<javac classpathref="maven.classpath" .../>
```

If at a later date you require a different version of a library then you can simply override the relevant entry in the `library.properties` file. Additionally, if as a developer you were doing some testing and did not want to update this file, you could override the version on the command line (for example, using “`ant -Dver.commons.cli=1.4.1`”) or make use of your own `build.properties` file.

Publishing Java libraries

As well as downloading pre-built libraries to build against, you can also publish your own libraries to an Apache Maven repository. This allows you create a single consistent area for re-using libraries, and to take advantage of Apache Maven’s dependency management capabilities. In order to publish libraries, you will first need to create a

pom.xml file. For example, for the *Hotel de Java* EJBClient project this file would contain the following:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>hoteldejava</groupId>
  <artifactId>ejbclient</artifactId>
  <version>1.0.1</version>
</project>
```

These are the only lines that you will need to be able to publish a new library. However if applicable, you can also add dependencies to the POM file similar to that for `commons-cli` component described above.

To deploy your built library to the Maven repository, you would then include a target in your `build.xml` file similar to the following:

```
<target name="maven-deploy"
  description="deploy to maven repository">
  <artifact:pom id="maven.project" file="pom.xml" />
  <artifact:deploy file="dist/HotelDJEJBClient-
    ${rel.num}.jar">
    <remoteRepository refid="internal.repository"/>
    <pom refid="maven.project"/>
  </artifact:deploy>
</target>
```

In this example the library version is referenced by the `${rel.num}` property and it is being published to the repository referenced by “`internal.repository`”.

In practice, this is all you will need to start using Apache Maven repositories to implement Java library re-use and dependency management. However there are additional settings and configurations

that you can make (such as specifying the location of your own local repository). These are described in more detail on the Apache Ant tasks for Apache Maven website⁹.

Summary

In this chapter I have described how you can take advantage of a number of Ant's capabilities in order to keep your build scripts maintainable and reusable. In your own build scripts you might have different or related issues to solve. Hopefully however, by describing how Ant's individual capabilities can be brought together to implement quite complex solutions, you should have an idea on how to solve these too. At some stage in addition to building your Java application you will also need to package and deploy it to your development, test or production environments. In the next chapter I will therefore look at how Java applications are typically packaged and deployed and what support Ant has for automating this process.

⁹ <http://maven.apache.org/ant-tasks.html>

CHAPTER 4

Release Packaging and Deployment

*We all have the extraordinary coded within us,
waiting to be released.*

Jean Houston

Apache Ant is not just used for building (or compiling) software. You can also use it to package Java applications and physically deploy them to their run-time environments. This chapter looks at how Java applications are typically packaged and some of the capabilities of Ant that are available to support packaging and deployment.

Java EE Packaging

Java Platform, Enterprise Edition or Java EE, is a programming platform for developing and executing distributed, multi-tier applications. The Java EE platform is made up of a whole host of APIs

(with a wealth of acronyms such as JDBC, RMI, JMS) defined by Java community specifications¹. The purpose of this section is not to discuss the entire myriad of technologies at play in a Java EE application but rather to look at what information someone involved in the build and deployment of such applications would typically need to know.

Java EE applications are designed to execute on specific software infrastructure (or middleware) components; an example of a typical infrastructure is illustrated in Figure 7

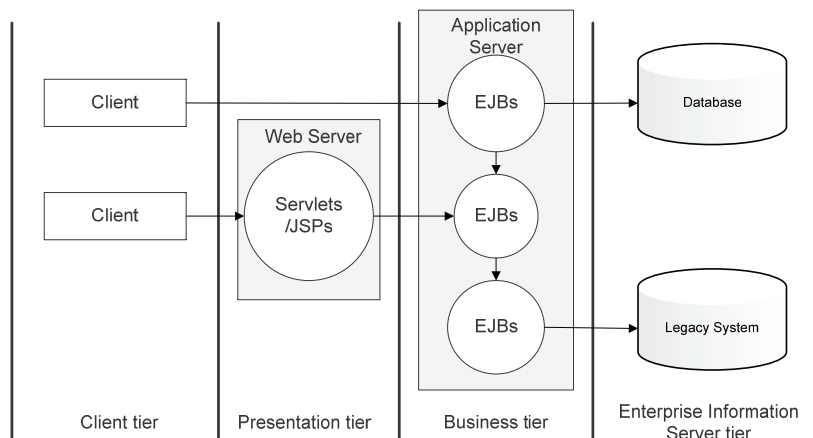


Figure 7 - Typical Java EE infrastructure

In this example a multi-tiered application is illustrated, the application makes use of (or connects with) the following components:

- A Database to store and serve data.
- An existing Legacy System.

¹ <http://jcp.org>

- An Application Server to execute the core business logic. Examples of such a server include IBM WebSphere, BEA WebLogic or Apache Geronimo.
- A Web Server to serve requests and execute presentations functions. Examples include Apache Tomcat or the IBM HTTP Server. Note that the Web Server can also be "embedded" in the Application Server for smaller applications.
- The various Clients (Internet browsers, desktop applications) that users make use of to operate the application.

There are two important points to realize about such an infrastructure. Firstly, that a Java EE application consists of multiple modules that need to be deployed to each of these infrastructure components. Secondly, that there are many different vendor supplied infrastructure components that the application could be deployed to.

The first point should naturally get you thinking about complexity - Java EE packaging and deployment is not simple - whilst the second point leads us on to the fact that although Java EE applications are based on standards, there are still vendor specific packaging and deployment steps that might have to be carried out. Given this type of infrastructure, I will look next at how Java EE applications are typically packaged.

Java EE modules

All Java packaging is carried out using variations of the Java ARchive (JAR) format; this is a Zip compatible format that allows pre-built Java classes to be bundled together into a module. An optional Manifest file can also be included in the Java archive that contains

meta-data to additionally describe the contents. An example of a manifest file for the application would be as follows:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.4.2
Built-By: Kevin Lee
Main-Class: net.sourceforge.hoteldejava.about
Name: net/sourceforge/hoteldejava
Specification-Title: "Hotel de Java Classes"
Specification-Version: "1.0"
Specification-Vendor: "The Buildmeister".
Implementation-Title: "net.sourceforge.hoteldejava"
Implementation-Version: "HOTELDEJAVA_02_REL"
Implementation-Vendor: "The Buildmeister"
```

The manifest file is defined according to a standard and allows you to inject custom information that defines more detail about the archive and could potentially be used at a later date - for debugging as an example².

Java EE applications are typically packaged up into an Enterprise ARchive (or EAR file) for deployment. An Enterprise archive is a composite module that is made up of all the modules that need to be deployed to the different tiers. It is still a valid Java archive (and the same tools can be used to create and extract it), however it has additional content and it is best to think of an EAR file as a “JAR of JARs”. A logical example of a typical Enterprise archive is illustrated in Figure 8.

² For more information on the manifest format see
<http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html#JAR%20Manifest>

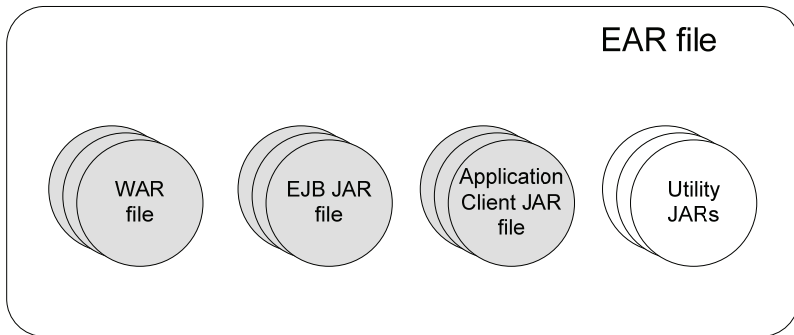


Figure 8 – Enterprise archive logical example

In this example, the Enterprise archive file is made up of modules for each of the tiers as follows:

- Web ARchives (or WAR files), that are to be installed into the Web Server.
- Enterprise Java Bean JAR files, that are to be installed into the Application Server.
- Application Client JAR files, that are to be executed on the Client.
- Utility and Resource JARs, which are standard Java Archives that the application will make use of.

Note that each of these individual modules can be deployed separately, for example a simple Web application might only consist of a Web archive; however complex Java EE applications typically contain the complete set of modules.

In order to describe how each of these modules should be deployed on the target infrastructure, each of them has a deployment descriptor. This is an XML file stored in the module that describes the contents of the module, its components and relationships. The standard

name for each of these deployment descriptors is shown in the grey boxes in Figure 9 .

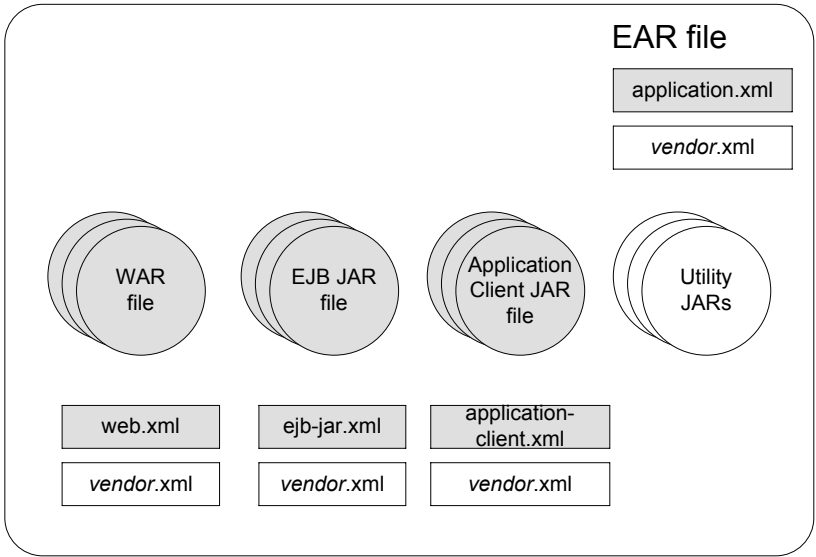


Figure 9 - EAR deployment descriptors

In this example, as well as the standard deployment descriptors you will notice that there are placeholders for each of the vendor specific deployment descriptors. These have different names from vendor to vendor, however an example of the names used by Apache Geronimo and IBM WebSphere are given in Table 2.

Module	Standard	Vendor Specific
EAR file	application.xml	Apache Gernimo: geronimo-application.xml IBM WebSphere: ibm-application-

		bnd.xml ibm-application- ext.xmi
EJB-JAR file	ejb-jar.xml	Apache Geronimo: openejb-jar.xml IBM WebSphere: ibm-ejb-jar-bnd.xmi ibm-ejb-jar-ext.xmi
WAR file	web.xml	Apache Geronimo: geronimo-web.xml IBM WebSphere: ibm-web-bnd.xmi ibm-web-ext.xmi
Application Client	application-client.xml	Apache Geronimo: geronimo-application-client.xml IBM WebSphere: ibm-application-client-bnd.xmi

Table 2 - Example deployment descriptors

In this table each module has additional vendor specific deployment descriptors (called deployment plans in Apache Geronimo's case). IBM WebSphere has two for the majority of modules, a bindings (“bnd”) file that binds the abstract contents of the Java EE deployment descriptor to a concrete WebSphere runtime component, and an extension (“ext”) file that contains extensions that WebSphere provides above and beyond the Java EE standard.

Release packaging with Ant

As part of your end-to-end build process, at some stage you will want to package up the files that you have built so that they are ready to be

deployed. There are a number of areas where Ant can help with this as follows:

- Creating manifest files.
- Packaging your Java classes into Java Archives, Web Archives or Enterprise archives.
- Signing and checksumming your archives.

In this section I will describe how Ant can help you automate all of these, but first I will look at an additional approach for obtaining a release number.

Prompting for a release number

In chapter 2 I described how to automate the creation of a `release.properties` files containing a release number to be used throughout your build script. In practice I have seen many people manually enter release number information - as the format and number often change frequently. An alternate way to prompt for a release number is to use the Ant `<input>` task and then verify the format of the entry using the `<matches>` task. For example, suppose your organization defined release numbers in the following format:

major.minor.maintenance[-rc][-alpha][-beta]x

For example 1.0.0, 1.0.1-rc1 or 2.10.1beta2, then a target to request and check for a valid release number could be implemented as follows:

```
<target name="get-relnum">
  <input message="Enter release number:"
    addproperty="rel.num"/>
  <condition property="legal-relnum">
```



```
        <matches pattern="^\d\.\d\.\d(-rc\d|-alpha\d|-
beta\d)*$" string="${rel.num}"/>
      </condition>
      <fail message="Error: the release number is not
formatted correctly" unless="legal-relnum"/>
      <propertyfile file="release.properties">
        <entry key="rel.num" value="${rel.num}"/>
      </propertyfile>
    </target>
```

In this example, the property `rel.num` is used to store the release number. It is checked for validity using a regular expression pattern and the `<matches>` task. If it is formatted correctly, it is then stored in the `release.properties` file for further use.

Creating manifest files

A manifest file contains information about an archive, for example the author, version, Classpath and so on. A template manifest is generated automatically for you as part of Ant's `<jar>`, `<war>` and `<ear>` tasks – however you can also add your own information to the manifest. To generate your own manifest file you can use the Ant `<manifest>` task as follows:

```
<manifest file="${dir.build}/manifest.mf">
  <attribute name="Implementation-Title"
    value="HotelDeJava"/>
  <attribute name="Implementation-Version"
    value="HOTELDEJAVA-${rel.num}"/>
</manifest>
```

This example creates a manifest file called `manifest.mf` with two attributes. You can either specify values for attributes as recommended

by the Java archive specification³ (which is recommended) or create your own attributes.

Creating Java archives

The Java archive “.jar” format is a standard way for distributing Java archives. Java archive files use the same compression format as Zip files so can be opened and extracted by utilities such as WinZip. To create a Java archive you could use the Ant `<jar>` task as follows:

```
<jar destfile="${dir.dist}/HotelDeJava.jar"
    manifest="${dir.build}/manifest.mf"
    includes="**/*.class"
    excludes="**/Test*.class"/>
```

This example creates a Java archive called `HotelDeJava.jar`; it includes a manifest file called `manifest.mf` and all of the built “.class” files but excludes test classes.

Creating Web and Enterprise archives

Web and Enterprise archives are similar to a Java archive but they are packaged with special content. Web archives require a deployment descriptor called `web.xml` plus any other vendor-dependent files. Enterprise archives require a deployment descriptor called `application.xml` plus any other vendor-dependent files. An Enterprise archive is a collection of related Java and Web archives.

To create a Web archive you could use the `<war>` task similar to the following:

```
<war warfile="${dir.dist}/HotelDeJava.war"
```

³ <http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html>

```
webxml="web/Web-INF/web.xml"
manifest="${dir.build}/manifest.mf">
<fileset dir="WebContent"/>
    <exclude name="web.xml"/>
</fileset>
<classes dir="${dir.build}">
    <include name="**/*.class"/>
    <exclude name="**/Test*.class"/>
</classes>
</war>
```

This example creates a Web archive called `HotelDeJava.war` with the manifest file `manifest.mf` and the deployment descriptor `web.xml`; it also includes all of the web (i.e. HTML/JSP) files from the `WebContent` directory and all of the pre-compiled “.class” files

To create an Enterprise archive you could use the Ant `<ear>` task similar to the following:

```
<patternset id="dist.sources">
    <include name="**/*.jar"/>
    <include name="**/*.war"/>
    <include name="**/*.ear"/>
</patternset>

<ear destfile="${dir.dist}/HotelDeJava.ear"
    appxml="META-INF/application.xml"
    <fileset dir="${dir.dist}">
        <patternset refid="dist.sources">
            <exclude name="**/*.ear"/>
        </patternset>
    </fileset>
</ear>
```

This example creates an Enterprise archive called `HotelDeJava.ear` with the deployment descriptor `application.xml`; it also uses a

<patternset> to include all the pre-packaged Java and Web archives that were built for it.

Signing and checksumming your archives

As part of your release process you can also sign and checksum your archives for additional security and integrity. Signing Java archives, records signature information in the archive which can be retrieved to ascertain if the archive is from a know source. Whilst, check summing creates MD5 checksums of individual files to guarantee that they have been delivered correctly.

Singing Java archives allows users who verify your signature to grant your archive software security privileges that it wouldn't ordinarily have⁴. Signing is typically used for Java applications that users execute or install from a web browser, for example Java applets or Java Web Start⁵ applications. However, you can also sign your archives as part of your build process to ensure traceability and auditability. To sign a Java archive you can use the Ant <signjar> task as follows:

```
<signjar jar="${dir.dist}/HotelDeJava.jar
  keystore="hdj.ks"
  alias="kevin" storepass="password"/>
```

This example signs the Java archive `HotelDeJava.jar` using the security certificate contained in the keystore `hdj.ks`.

Creating checksums on files allows you to guarantee that they have not become corrupt in anyway, for example when downloading

⁴ See <http://java.sun.com/docs/books/tutorial/deployment/jar/signindex.html> for more information on Java signing

⁵ <http://java.sun.com/products/javawebstart>

them over a network connection. To create a checksum of a Java archive you can use the `<checksum>` task as follows:

```
<checksum>
  <fileset dir="${dir.dist}">
    <includes name="**/*.jar"/>
  </fileset>
</checksum>
```

This example will create a checksum file (a file with the “.MD5” extension) for all Java archives in the `dir.dist` directory. To verify the integrity of all of the delivered Java archives you can then re-use the `<checksum>` task as in the following:

```
<condition property="isChecksumOK">
  <checksum>
    <fileset dir="downloaded">
      <include name="**/*.jar"/>
    </fileset>
  </checksum>
</condition>
```

This example will checksum all of the files in the `downloaded` directory and set the property `isChecksumOK` only if all of them match. You would typically use this mechanism when you supply an Ant install script to deploy your application. A typically example is when you handover your developed application to your Operations team to be installed on your production (customer facing) servers.

Release deployment with Ant

Once you have packaged up your application, you will probably want to deploy it. Although Ant is not really a deployment tool it has access to tasks that can be used for deployment. For example, you can deploy

to file systems using the `<ftp>` or `<scp>` tasks and you can execute vendor specific deployment tools from the command line using Ant's `<exec>` task.

An example of using the Ant `<ftp>` task to deploy or copy a file to a remote location would be as follows:

```
<ftp server="ftp.localhost.com" remote dir="install"
  userid="kevin" password="password" verbose="yes">
  <fileset dir="${dir.dist}">
    <include name="**/*.ear"/>
  </fileset>
</ftp>
```

This example would copy all of the Enterprise archives in the `dir.dist` directory to the FTP server `ftp.localhost.com`. Once the Enterprise archives had been copied then an installation script could be executed to install it.

Every Web or Application Server has its own mechanism for deploying an application. If you are lucky, for example with Apache Tomcat, then there will be a set of Ant tasks which you can use for deployment. If not there will typically be a command line that you can integrate with. For example you could write a macro to integrate with the Apache Geronimo command line deployment tool as follows:

```
<macrodef name="geronimo-deploy">
  <attribute name="username" default="system"/>
  <attribute name="password" default="manager"/>
  <attribute name="filename"/>
  <attribute name="home"/>
  <attribute name="host" default="localhost"/>
  <attribute name="port" default="8080"/>
  <attribute name="action" default="deploy"/>
  <sequential>
```

```
<exec dir="@{home}"
    executable="@{home}/bin/deploy.bat">
    <arg line="--user @{username} --password
@{password} @{action} --host @{host} --port @{port}
@{filename}"/>
    </exec>
</sequential>
</macrodef>
```

You could then use this macro as follows:

```
<geronimo-deploy
    filename="${dir.dist}/HotelDeJava.ear"
    home="C:\JavaTools\geronimo-1.1.1"/>
```

In this example the Enterprise archive `HotelDeJava.ear` would be deployed to the default server “localhost:8080”.

In practice, Ant is not an Enterprise deployment tool as it does not have capabilities such as server discovery, rollback and failover. For deployment to development and test environments, Ant targets are usually called by build control tools such as CruiseControl or IBM Rational Build Forge as I will discuss in the chapter 6. However for deployment to production and maybe final test environments, where there can be many servers and resources that need to be configured, additional scripts are typically written to achieve this. These can be additional Ant scripts but more likely they are written in other scripting tools that the deployment servers support. A typical example is the Jython⁶ scripting language which is supported by IBM WebSphere and BEA Weblogic.

⁶ www.jython.org

Summary

In this chapter I have described how Java applications are packaged and how you can use Ant to automate their packaging and deployment. I believe it is good practice to continually package and deploy your application during development - exactly the same as you would when you would deploy to your production environment. This ensures that your deployment process does not fail at this critical stage. One way to achieve this continual deployment process is to schedule and automate it using a build control tool as I will discuss in chapter 6.

Of course, Ant is not the only tool that plays an important part in your build and release process, your version control tool is also critical for ensuring repeatability and auditability. In the next chapter I will therefore look in detail at how Ant can be integrated with a number of popular version control tools.

CHAPTER 5

Integrating with Version Control Tools

*History is the version of past events that people
have decided to agree upon.*
Napolean

Apache Ant is not implemented in isolation - it will need to work with your existing tools. This chapter looks at how Ant can be integrated with version control tools – in particular it looks in detail at how Ant can be integrated with the popular open source Subversion and commercial IBM Rational ClearCase tools.

Requirements for version control integration

Apache Ant is delivered with tasks to integrate with a number of version control tools, including but not limited to CVS, PVCS, Visual SourceSafe, Perforce and IBM Rational ClearCase. The functionality

of these integrations varies but at the very least they usually provide basic access to a version control tools command line. You will typically not need to execute every single version control command from your build script; however there are usually a minimum required set of operations that most builds scripts implement and that can be summarized as follows:

- The ability to create and update a version control workspace.
- The ability to check-in or commit files changed during the build back to the repository.
- The ability to create a tag or baseline in the repository of all the files that have gone into the build.
- The ability to create a report on the set of changes that have gone into the build.

One of the main issues of integrating your build process with version control is when do you create the workspace? If you create an Ant target to create a new workspace how do you get access to it? The build script containing the target should be in version control. Therefore you might have to create a workspace to get the script to create a workspace! There are essentially two ways to solve this problem:

- Create the new workspace outside of your build script – usually in a build control tool. A workspace is typically called a **working copy** in Subversion or a **view** in ClearCase.
- Create a workspace specifically for building in and reuse it – updating the workspace with the latest changes as the first part of the build process.

In the remainder of this chapter I will look at how these requirements can be implemented when using either Subversion or IBM Rational ClearCase as your version control provider.

Ant and Subversion

Subversion is an open source version control tool and was designed as a replacement for CVS. There are a number of integration libraries that can be used to integrate Ant and Subversion and which are summarised as follows:

- *The “official” Subversion Ant library* – the default integration library with support for Subversion via an “antlib” library that wraps the `svn` command line executable. The library is available in Ant's own Subversion source repository¹, and works with Ant 1.7. At the time of writing you still need to check out and build this library for yourself.
- *SvnAnt* - The SvnAnt project at Tigris² is an Ant task that integrates to Subversion. It uses a JNI (Java Native Interface) binding to Ant if possible for speed, falling back to the Subversion command line if not.
- *StatSvn* – The StatSvn project at SourceForge³ is a library that generates a set of statistical reports and graphs from Subversion metadata. An Ant task is available for integrating the execution of StatSvn into your build process.

¹ <http://ant.apache.org/antlibs/svn/index.html>

² <http://subclipse.tigris.org/svnant.html>

³ <http://sourceforge.net/projects/svnstat>

Currently, I recommend the use of SvnAnt and StatSvn. In this section I will therefore be describing how to use them as part of your Ant build process.

Installing SvnAnt

Although SvnAnt is available to download as a binary, at the time of writing a new binary has not been released for a while and is significantly out of date with respect to the source code. I therefore recommend that you rebuild this project from the source code to get the most up-to-date features. Although this is obviously a risk, the library is small enough that the risk is limited.

To download the source you will need a working Subversion command line (the `svn` command). I will assume that you have already set this up, if not visit the Subversion website⁴ for instructions on how to download Subversion for your environment. The Subversion command you need to execute to initiate the download is as follows:

```
svn co http://subclipse.tigris.org/svn/subclipse/ ↵  
trunk/svnant/ svnant --username guest --password ""
```

This will create a `svnant` directory in your current directory, navigate into this directory and start the build, specifying the version of the Java Development Kit (JDK) that you are running. For example, if you are running JDK 1.4 then you would execute the following:

```
cd svnant  
ant -DtargetJvm=1.4
```

Once the build has completed you should copy the following files to a directory inside your project:

⁴ <http://subversion.tigris.org>

```
build/lib/svnant.jar
lib/svnClientAdapter.jar
lib/svnjavahl.jar
lib/svnkit.jar
```

I typically create a `lib` directory at the top-level of my project's build directory to contain libraries such as these. However you could also copy them to your Ant home directory (`%ANT_HOME%\lib`) or even install them into your Apache Maven repository.

Using SvnAnt – creating or updating a workspace

SvnAnt executes most Subversion commands via a single Ant task. However, before you can execute this task you need to make sure that it can be located from within your build script. To achieve this, you can add the following lines to your build script:

```
<taskdef name="svn"
    classname="org.tigris.subversion.svnant.SvnTask">
    <classpath>
        <fileset dir="${dir.lib}">
            <include name="**/svn*.jar"/>
        </fileset>
    </classpath>
</taskdef>
```

This example creates a new task called `<svn>` which can be used to access the library.

Now that this task is available, to create a new workspace you could implement a target similar to the following:

```
<property name="svn.url-base" value="https://hoteldejava.
svn.sourceforge.net/svnroot/hoteldejava"/>
<property name="svn.url" value="${svn.url-base}/trunk/">
<property name="svn.dir" value="hoteldejava"/>
```

```
<target name="svn-checkout" description="checkout a
project from Subversion">
    <svn username="${svn.username}"
        password="${svn.password}">
        <checkout url="${svn.url}"
            destPath="${svn.dir}"/>
    </svn>
</target>
```

This example will create a new workspace (a Subversion working copy) in the directory `hoteldejava`. It will download the latest files from the URL specified by the `svn.url` property and using the login credentials specified by the `${svn.username}` and `${svn.password}` properties. Login credentials are not always required to checkout a project, however if you need to supply them then the best approach to use is to add the properties to your personal `build.properties` file, for example:

```
svn.username=guest
svn.password=password
```

To update an existing workspace you could implement a target similar to the following:

```
<target name="svn-update" description="update a project
from Subversion">
    <svn username="${svn.username}"
        password="${svn.password}">
        <update dir="${svn.dir}"/>
    </svn>
</target>
```

Remember, the update approach is really only acceptable if you create a workspace specifically for building in or releasing from.

Using SvnAnt – committing changes

In controlled build environments, it is likely that files will get changed or generated as part of the build process and which you want committed to version control. Although I don't recommend adding intermediate files such as Java classes to version control, you might want to add built libraries for sharing and reuse. If you need to add new files into the repository then you could implement a target similar to the following:

```
<target name="svn-add-dist">
  <svn username="${svn.username}"
        password="${svn.password}">
    <add dir="${dir.dist}" force="true">
      <fileset>
        <include name="**/*.jar"/>
      </fileset>
    </add>
  </svn>
</target>
```

This example will add the contents of Java libraries in the `dir.dist` directory into the repository.

If you have added files to the repository or are updating files already in version control then you could implement a target similar to the following:

```
<target name="svn-commit">
  <svn username="${svn.username}"
        password="${svn.password}">
    <commit dir="${svn.dir}"
```

```

        message="${svn.message}"/>
    <info target="${svn.dir}"/>
</svn>
    <echo>Last Revision = ${svn.info.lastRev} by
${svn.info.author} on ${svn.info.lastDate}</echo>
</target>

```

In this example you would need to set `svn.message` to an appropriate value as it will be recorded in the repository. The example also runs the `<info>` task to get the latest information about the repository. Information from this command is placed in Ant properties, such as `svn.info.lastRevision` for the last repository revision. This information can be used and recorded for auditing purposes.

Using SvnAnt – creating a baseline

In Subversion, baselines are usually created by copying a repository directory on your `trunk` to the `tags` directory. This assumes that you are following the well known convention of creating a `trunk`, `branches` and `tags` directory at the top level of your repository as illustrated in Figure 10. If so then to effectively create a new baseline for your build, you could implement a target similar to the following:

```

<target name="svn-copy">
    <fail message="Error: rel.num has not been set"
        unless="rel.num"/>
    <property name="svn.url-rel"
        value="${svn.url-base}/tags/${rel.num}"/>
    <svn username="${svn.username}"
        password="${svn.password}">
        <copy srcurl="${svn.url}"
            desturl="${svn.url-rel}"
            message="${svn.message}"/>
    </svn>

```

`</target>`

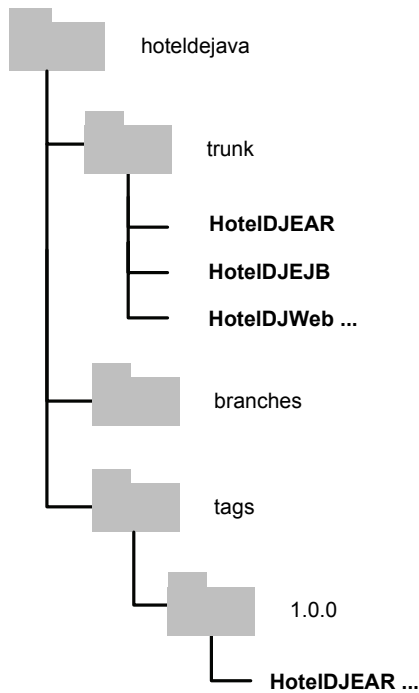


Figure 10 - Example Subversion directory structure

This example requires the property `rel.num` to be set with an appropriate release number (for example 1.0.0). A check using the `<fail>` task has been implemented to ensure this is the case as you do not really want to risk creating random directories in your Subversion repository. Again, you would need to set `svn.message` to an appropriate value as it will be recorded in the repository.

Installing StatSvn

StatSvn is a metrics analysis tool for Subversion repositories. It can automatically generate a number of useful reports including:

- Source Lines of Code timelines
- Source Lines of Code for each developer
- Developer activity
- Developer activity per module
- Developer most recent commits with links to ViewVc⁵ for web browsing

An example of a Source Lines of Code timeline is illustrated in Figure 11.

To install StatSvn simply download the binary Zip file from the SourceForge project website⁶. Extract the Zip file and copy the Java library `statsvn.jar` to your `lib` directory.

⁵ www.viewvc.org

⁶ <http://sourceforge.net/projects/statsvn>

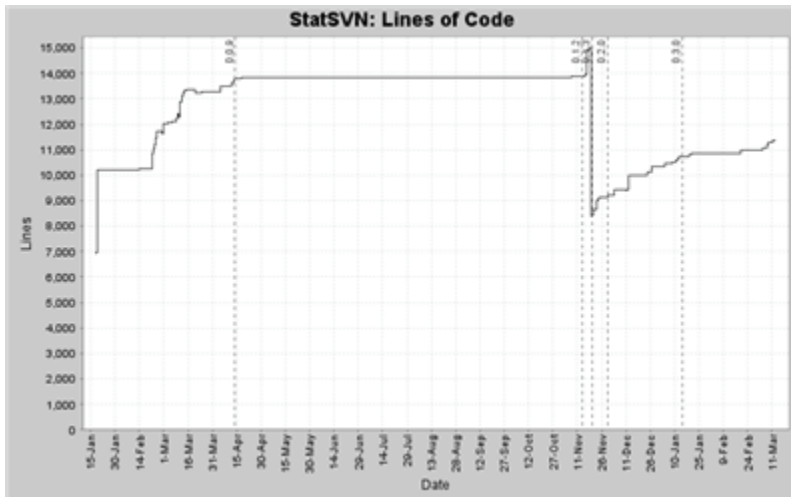


Figure 11 - StatSVN report

Using StatSvn

StatSvn can be executed via a single Ant task. However, before you can execute this task you need to make sure that it can be located from within your build script. To achieve this, you can add the following lines to your build script:

```
<taskdef name="statsvn"
    classname=" net.sf.statsvn.ant.StatSvnTask">
    <classpath>
        <fileset dir="${dir.lib}">
            <include name="**/statsvn.jar"/>
        </fileset>
    </classpath>
</taskdef>
```

This example creates a new task called `<statsvn>` which can be used to access the library.

StatSvn relies on the execution of the Subversion `log` command to output information on the history of the repository and its changes. You therefore need to execute this command before executing the `<statsvn>` task. Unfortunately, the `log` command is not currently supported by SvnAnt so you will have to call the Subversion command line. To run both of these commands you could implement a target similar to the following:

```
<target name="svn-statreport">
  <property name="svnstat.log"    value="svn.log"/>
  <property name="svnstat.dir"    value="statsvn"/>
  <property name="svnstat.title"
    value="${ant.project.name} Report"/>
  <exec executable="svn" output="${svnstat.log}">
    <arg line="log ${svn.url} --xml -v"/>
  </exec>
  <statsvn path="." log="${svnstat.log}"
    outputDir="${svnstat.dir}"
    title="${svnstat.title}"/>
</target>
```

In this example a Subversion log file called `svn.log` is first created using the Ant `<exec>` task to call the Subversion command line. The `<statsvn>` task is then executed to turn this log file into more useful metrics and graphs. The output is placed in the `statsvn` directory as HTML and PNG files. There will be an `index.html` file that you can open to navigate between all of the reports.

In practice, you should execute this task on a regular basis, maybe as part of a nightly build. You can then publish the results to an internal project Intranet. This information can then be used by all members of your project team.

Ant and ClearCase

IBM Rational ClearCase is a commercial version control tool and has been deployed across a large number of organizations. There are a number of integration libraries that can be used to integrate Ant and ClearCase and which are summarised as follows:

- *The built-in ClearCase tasks* – by default Ant is delivered with a number of tasks for integrating with ClearCase. These are typically wrappers around the ClearCase `cleartool` command line.
- *ClearAntLib* – ClearAntLib is a ClearCase Ant integration library developed by myself. It provides a number of “value-add” tasks such as staging built files, applying commands to Ant FileSets and producing baseline reports.
- *Ant build auditing listener* – ClearCase version 7 delivers an integration library for executing an audited build using Ant. This allows you to automatically record all the versions of objects and the environment that have been included in a build.

Currently, I recommend the use of ClearAntLib and potentially the Ant build auditing library. In this section I will therefore be describing how to use them as part of your Ant build process.

Installing ClearAntLib

The ClearAntLib library is available in the form of an Ant **antlib**. An antlib is a pre-built library that you can plugin to Ant without recompilation. It is simply a set of classes referred to by an XML namespace and then referenced just as any other Ant task or type. This makes it easy to integrate a set of tasks with a pre-built version of Ant.

To install ClearAntLib simply download the binary Zip file from the SourceForge project website⁷. Extract the Zip file and copy the Java library `clearantlib.1.x.x.jar` to your `Ant lib` directory.

Using ClearAntLib – creating or updating a workspace

ClearAntLib can execute any ClearCase `cleartool` commands via its `<ccexec>` Ant task. However, before you can execute this task you need to reference the antlib to make sure that it can be located from within your build script. To achieve this, you can reference its namespace at the top of your build script as follows:

```
<project name="..." ...  
  xmlns:ca="antlib:net.sourceforge.clearantlib">
```

This example creates a new namespace called “ca” which can be used to reference each of the libraries tasks.

Now that these tasks are available, to create a new workspace you could implement a target similar to the following:

```
<property name="cc.viewtag" value="hoteldejava_int"/>  
<property name="cc.viewloc"  
  value="C:\\Views\\hoteldejava_int"/>  
<property name="cc.vob" value="HotelDeJava"/>  
  
<target name="cc-mkview" description="create a view from  
ClearCase">  
  <ca:ccexec>  
    <arg value="mkview"/>  
    <arg value="-snapshot"/>  
    <arg value="-tag ${cc.viewtag}"/>
```

⁷ <http://sourceforge.net/projects/clearantlib>

```

        <arg value="\${cc.viewloc}"/>
    </ca:ccexec>
    <ca:ccexec dir="\${cc.viewloc}">
        <arg value="update"/>
        <arg value="-add_loadrules \${cc.vob}"/>
    </ca:ccexec>
</target>

```

This example will create a new workspace (in this case a ClearCase snapshot view) called `hoteldejava` in the directory `C:\Views\hoteldejava_int`. It will then update the load rules of this view to look at the repository `HotelDeJava` and download its latest files.

To update an existing workspace you could implement a target similar to the following:

```

<target name="cc-update" description="update a view
from ClearCase">
    <ca:ccexec dir="\${cc.viewloc}">
        <arg value="update"/>
    </ca:ccexec>
</target>

```

Remember, the update approach is really only acceptable if you create a workspace specifically for building in or releasing from.

Using ClearAntlib – committing changes

If you need to add new files into the repository then you can use the ClearAntLib `<ccstage>` task to implement a target similar to the following:

```

<target name="cc-add-dist">
    <ca:ccstage todir="\${dir.stage}"

```

```
<fileset dir="${dir.dist}">
  <include name="**/*.jar"/>
</fileset>
</ca:ccstage>
</target>
```

This example will add the contents of Java libraries in the `dir.dist` directory into the `${dir.stage}` directory of the repository. The `<ccstage>` task checks to see if files are already under version control, and adds directories too if required.

If you are updating files already in version control then you could implement a target similar to the following:

```
<target name="cc-checkin">
  <ca:ccapply>
    <arg value="checkin"/>
    <arg line="-nc -ident"/>
    <fileset dir=".">
      <include name="**/*.properties"/>
    </fileset>
  </ca:ccapply>
</target>
```

In this example any Ant property files that have been checked out (during the build) will be checked back into version control.

Using ClearAntLib – creating a baseline

In ClearCase baselines are created by either applying a Base ClearCase label or by applying a UCM baseline. With Base ClearCase, to apply a label you would first have to create the label type, then apply the label. As there are a number of operations it is therefore worth creating a macro to achieve this. An example of how you do this is as follows:

```

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

```

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

```
<cc-apply-label label="HOTELDEJAVA_01_INT"/>
```

With UCM you do not need to create the baseline type first, but you still might want to promote the baseline and therefore create a macro as in the following:

```

<macrodef name="cc-apply-bl">
  <attribute name="basename"/>
  <attribute name="component"/>
  <attribute name="plevel" default="BUILT"/>
  <attribute name="pvob"/>
  <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 would simply use the following in your `build.xml` file:

```

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

```

You can change these macros to conform to any policies that you might want to implement, for example you might want to lock the label type too.

Using ClearAntLib – creating a baseline report

ClearAntLib has a number of tasks for reporting on ClearCase labels and baselines. There are two tasks (`<cclabelreport>` and `<ccbldreport>`) to generate a report on the content of a label or baseline, as well as two tasks (`<ccdifflabelreport>` and `<ccdiffbldreport>`) to generate a report on the difference (in terms of activities or versions) between labels or baselines. For example, if you wanted to produce a report on the contents of the Base ClearCase label `HOTELDEJAVA_02_INT` and send the output to the file

cclabelreport.xml, you could write an Ant target similar to the following:

```
<target name="label-report">
    <ca:cclabelreport
        labelselector="RATLBANKMODEL_02_INT"
        outfile="cclabelreport.xml"/>
    <style in="cclabelreport.xml"
        out="cclabelreport.html"
        style="cclabelreport.xsl"/>
</target>
```

As you can see from this example `<ccblreport>` takes two attributes: the name of the label to report on via the `labelselector` attribute, and the XML output file to create via the `outfile` attribute. The task generates a valid XML file listing the elements to which the label has been applied. Using XML stylesheet language translation (XSLT) technology you can also convert the XML file into a HTML for presentation. An example stylesheet called `cclabelreport.xsl` is delivered with ClearAntLib.

Reporting on the content of a ClearCase baseline is achieved in a similar way. For example, if you wanted to produce a baseline report on the UCM baseline `HOTELDEJAVA_02_INT` and send the output to the file `ccbaselinereport.xml`, you could write an Ant target similar to the following:

```
<target name="baseline-report">
    <ca:ccblreport
        baselineselector="HOTELDEJAVA_02_INT@
        \HotelDeJavaProjects"
        outfile="ccbaselinereport.xml"/>
    <style in="ccbaselinereport.xml"
        out="ccbaselinereport.html"
        style="ccblreport.xsl"/>
</target>
```

</target>

As you can see this example is very similar to the first, however rather than a `labelselector` attribute there is a `baselineselector` attribute and a corresponding output file. An example of the HTML output that this task can produce is illustrated in Figure 12.

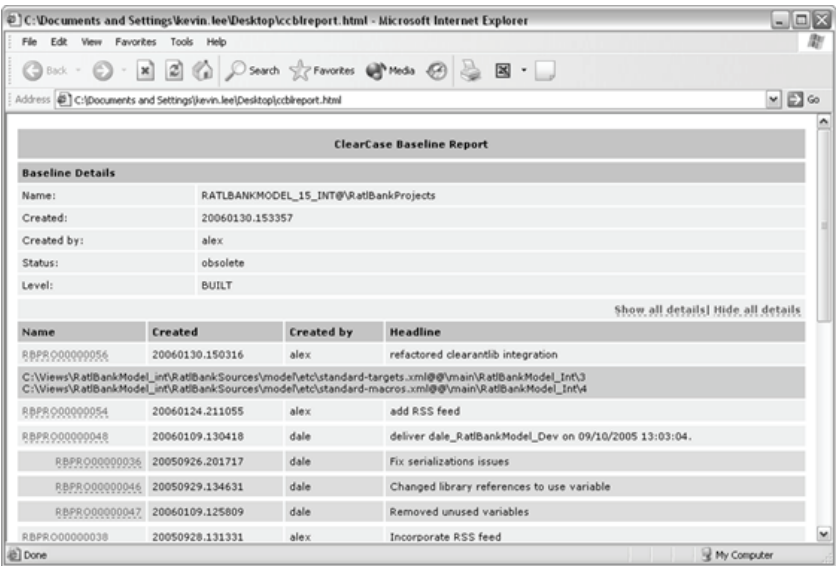


Figure 12 - Example ClearCase baseline report

Note that this example is based on the supplied stylesheets; you can change these stylesheets to easily customize the look and feel of this report for your organization.

Using the Ant build auditing listener

In ClearCase 7.0 a new Ant listener is available which allows you to audit an Ant build. This allows you to capture all of the element versions and environment that have gone into a build. The only caveat

is that in order to use the Ant listener you must execute your build in a ClearCase dynamic view. To make use of the listener, you will first need to set your Java Classpath to point to its library. On Windows you can achieve this by setting your Classpath to the following:

```
CLASSPATH=%CLASSPATH%;C:\Program Files\Rational\ClearCase\bin\CCAudits.jar
```

Whilst on Linux or UNIX you can carry out the following:

```
CLASSPATH=$CLASSPATH: "/opt/rational/clearcase/java/lib/CCAudits.jar"
```

Note that both of these examples assume that ClearCase is installed in its default location.

To execute an audited Ant build, you can then simply navigate into a ClearCase dynamic view and execute Ant as in the following

```
ant -listener CCAudits dist
```

In this example, the project's `dist` target is being executed. When a build is executed in a dynamic view using the listener, every build output (for example, Java `.class` file) is recorded as a **derived object**. A number of additional derived objects are also created in the `CCAudits` directory where the build was executed from. This directory contains a set of derived objects named after your project and audited targets.

For each build audit ClearCase creates a single **configuration record**, recording all the source versions used. All files read during execution of the audited shell are listed as inputs to the build and all of the files created become derived objects, associated with the single configuration record. What this means is that after the build has finished, if you carried out a listing of the derived objects using the `cleartool lsdo` command you would see output similar to the following:

```
cleartool lsdo -recurse
--12-29T15:04 "CCAudits\clean@@--12-29T15:04.2147483695"
--01-03T13:20 "CCAudits\compile@@--01-03T13:20.
2147483716"
--01-03T13:20 "CCAudits\init@@--01-03T13:20.2147483684"
--12-29T15:04 "CCAudits\javadoc@@--12-29T15:04.
2147483734"
--01-03T13:20 "CCAudits\HotelDeJava@@--01-03T13:20.
2147483694"
--01-03T13:20 "build\net\sourceforge\hoteldj\
CustomerBean.class@@--01-03T13:20.2147483772"
--01-03T13:20 "build\net\sourceforge\hoteldj\
CustomerCMP.class@@--01-03T13:20.2147483744"
--01-03T13:20 " build\net\sourceforge\hoteldj\ejb\
CustomerData.class@@--01-03T13:20.2147483743"
--01-03T13:20 " build\net\sourceforge\hoteldj\ejb\
CustomerLocal.class@@--01-03T13:20.2147483741"
...
```

This is essentially a listing of all the files that the build produced. If you then executed a `cleartool catcr -union` on the built Java archive, you would see the versions of source elements that went into the build as follows:

```
cleartool catcr -union dist\HotelDeJava.jar@@--01-  ↵
03T13:20.2147483681
-----
MVFS objects:
-----
      2 \build\net\sourceforge\hoteldj\ejb\
CustomerBean.class@@--01-03T13:20.2147483772
...
      1 \src\net\sourceforge\hoteldj\ejb\CustomerBean.java@@
\main\HotelDeJava_Int\2 <2005-12-29T13:34:52Z>
```

```
1 \ src\net\sourceforge\hoteldj\ejb\CustomerCMP.java@@  
\main\HotelDeJava_Int\1 <2005-11-26T01:37:50Z>
```

...

Using this information you can trace a particular version of a library back to its source file versions. To preserve this information you can also check the configuration record into version control⁸.

Summary

In this chapter I have described how you can integrate Ant with both Subversion and IBM Rational ClearCase to ensure traceability and auditability in your build process. In some cases the integration of these two types of tools might be sufficient, especially if you are initiating builds infrequently or manually. However, if you want to implement more regular builds, and extend your build process to incorporate testing and deployment then you will probably be better integrating Ant with a build control tool. In the next chapter I will therefore look in detail at how Ant can be integrated with a number of popular build control tools.

⁸ See the Building Software with ClearCase manual for more information on how to achieve this.

CHAPTER 6

Integrating with Build Control Tools

*“If everything seems under control,
you're not going fast enough”*

Mario Andretti

Although you can implement a complete end-to-end build process in Apache Ant, implementing repeatable build and deployment processes for multiple projects typically requires some form of build control framework. This chapter looks at how Ant can be integrated with the popular open source CruiseControl and commercial IBM Rational Build Forge tools.

CruiseControl

CruiseControl (<http://cruisecontrol.sourceforge.net>) is a popular continuous integration framework. It was originally developed by ThoughtWorks and like Ant is available as open source. In essence, CruiseControl is a wrapper around Ant for automating and controlling

builds, however over time it has developed a large amount of additional functionality. Using CruiseControl in combination with Ant you can carry out the following activities:

- Automating the execution of builds on a schedule, for example, nightly builds.
- Automating the execution of builds on a change in the repository, for Continuous Integration.
- Automating email notification on build success or failure.
- Publishing the results of your build in a variety of ways, for example via FTP, RSS or via an Ant target.
- Viewing the current and historical build results on a web dashboard.
- Executing builds and collating the build results for many different projects.

Note that there are additional features in CruiseControl, this is just the most commonly used set. CruiseControl is typically installed onto a central **build server** which orchestrates the end-to-end build process for a number of related or independent projects.

Installing CruiseControl

You can download the latest version of CruiseControl from its SourceForge website¹. There are three types of downloads:

- The Windows self installing executable.
- The binary Zip file.

¹ <http://cruisecontrol.sourceforge.net/download.html>

- The source Zip file.

If you run your build processes on Windows then I recommend the self installing executable as it simplifies the installation process. If you are running Linux or UNIX then download the binary Zip file. If you choose the Windows installer then by default, CruiseControl will be installed to the directory `C:\Program Files\CruiseControl`, however you can select an alternate installation directory. If you choose the binary Zip file then simply extract the contents to you chosen installation directory. In this section I will assume you install CruiseControl to the following directory on Windows:

```
C:\JavaTools\cruisecontrol-bin-2.7
```

Or the following directory on Linux or UNIX:

```
/opt/javatools/cruisecontrol-bin-2.7
```

If you have chosen the Windows installer then CruiseControl is installed as a service, to start it for the first time, enter the following from the command line:

```
net start cruisecontrol
```

If you have chosen the binary Zip version then navigate to the install directory and execute the CruiseControl start-up script. On Windows this would be:

```
cd C:\JavaTools\cruisecontrol-bin-2.7\bin
cruisecontrol.bat
```

Or on Linux or UNIX:

```
cd /opt/javatools/cruisecontrol-bin-2.7/bin
cruisecontrol.sh
```

CruiseControl should start up successfully and build its sample project “connectfour”. You can examine the build results by pointing your

browser at the CruiseControl dashboard – the default URL is `http://localhost:8080/dashboard`. You can also examine the CruiseControl log file (`cruisecontrol.log`) to see if there have been any errors.

The CruiseControl configuration file

Like Ant, the input to CruiseControl is also an XML configuration file. By default it is called `config.xml` and is located in the CruiseControl home directory. If you examine this file you will see that it is configured to build a single sample project called “connectfour”. CruiseControl is designed with a small core and a very high-level controlling implementation; most of the implementation details are delegated to plugins. There are a large number of existing plugins - for most scenarios - however if you cannot find the one that you require, you can always write a Java class to implement your own.

The configuration file can be amended to build many related or independent projects. Its general multi-project format is as follows:

```
<cruisecontrol>
  <property name=...>
  <plugin name=...>

  <project name="project1">
    <property name=...>
    <listeners/>
    <bootstrappers/>
    <modificationset/>
    <schedule/>
    <log/>
    <publishers/>
  </project>

  <project name="project2">
```

```
...
</project>

...
</cruisecontrol>
```

Every CruiseControl plugin takes a number of attributes; some in fact can take a significant amount. Fortunately, you can pre-configure plugins with common attributes, and then override (re-configure) them at the project level if necessary. As an example to pre-configure the `<htmlmail>` plugin you could place the following definition at the top of your configuration file – outside of any `<project>` elements:

```
<plugin name="htmlmail"
  mailhost="smtp.mailhost.com"
  returnaddress="bldadmin"
  xsldir="${dir.javatools}/cruisecontrol-bin-2.7/xsl"
  css="${dir.javatools}/cruisecontrol-bin-2.7/css/
    cruisecontrol.css"
  subjectprefix="[CruiseControl]: "/>
```

Whenever you wanted to call this plugin from within a project you could then simply include:

```
<htmlmail/>
```

However there are probably additional project specific attributes which you would set too and which for the `<htmlmail>` plugin I will describe later.

CruiseControl properties are specified and used in the same way as Ant properties; however the main difference is that they are not completely immutable. Rather, whoever sets a property last freezes its value within the scope in which it was set, for example you can override global properties with project specific properties. As an example, to create properties for the location of an installed version of

Ant, you could create the following properties at the top level of your build script:

```
<property name="dir.javatools" value="C:\JavaTools"/>
<property name="dir.ant" value="${dir.javatools}\apache-
ant-1.7.0"/>
```

Implementing these two simple practices - pre-configuring plug-ins and re-using properties - can significantly reduce the size of your CruiseControl configuration file. Although you will undoubtedly not need to use every single CruiseControl plugin there are a common set that most projects will use. I will discuss these plugins in the sections which follow, however before I do so you need to understand how CruiseControl defines its build workspace.

The CruiseControl workspace

CruiseControl is designed to execute a build from an existing version control workspace. Although you could configure CruiseControl to create a new workspace, I recommend that that you create a CruiseControl specific build workspace and reuse it to build in. In the Windows installer and binary versions of CruiseControl, build workspaces are typically held in its `projects` directory. Therefore, to create a new build workspace for Subversion in this workspace, you could execute a command similar to the following:

```
cd c:\JavaTools\cruisecontrol-bin-2.7\projects
svn co https://hoteldejava.svn.sourceforge.net/
svnroot/hoteldejava/trunk hoteldejava
```

This will create a workspace called `hoteldejava` containing the latest files from the Subversion trunk.

CruiseControl Listeners

CruiseControl Listeners are notified on every project event; however most are designed to handle a subset of events. Typically the only listener you will use - and which is more or less required - is the `<currentbuildstatuslistener>` plugin. This plugin is used to specify the file that records details about a project as it is being built. As an example, to log the build details to the file `buildstatus.txt` you would add the following:

```
<listeners>
  <currentbuildstatuslistener
    file="logs/${project.name}/buildstatus.txt"/>
</listeners>
```

In this example `${project.name}` is a built-in property referring to the name of the current project, so the file that would be created is `logs/hoteldejava/buildstatus.txt`.

CruiseControl Bootstrappers

CruiseControl Bootstrappers are run before a build takes place, regardless of whether the build is necessary. In the previous chapter I discussed the issue of when an Ant build script contains a task to create or update a workspace and that it potentially has to update itself. Bootstrappers can be used to solve this problem by ensuring that your build workspace has the latest build script – before the script is called. As an example to ensure that the Ant files `build.xml` and `default.properties` files are up to date out of Subversion, you could add the following:

```
<bootstrappers>
  <svnbootstrapper
    file="projects/${project.name}/build.xml"/>
</bootstrappers>
```

```
<svnbootstrapper  
  file="projects/${project.name}/default.properties"/>  
</bootstrappers>
```

You could similarly add any additional files which are required to be up to date before Ant is called, for example the `common-macros.xml` and `common-targets.xml` files.

CruiseControl SourceControls

CruiseControl was designed with continuous integration in mind; its default way of working is therefore to look for changes in a repository and initiate a build when files are updated. This set of changes is called a modification set in CruiseControl. However, even if you are not implementing continuous integration, modification sets are still useful as they tell you the changes that have been made since the last build.

The criteria for the files to check for are specified via a combination of the `<modificationset>` plugin and the relevant source control plugin. As an example to check a Subversion workspace to see if changes have been made you could add the following.

```
<modificationset quietperiod="30">  
  <svn localWorkingCopy="projects/${project.name}"/>  
</modificationset>
```

The `quietperiod` attribute is used to prevent modifications from being found while changes are being actively committed. This is less of a problem for Subversion as commits are atomic, but for CVS you could be half way through a commit process when the build is initiated. With a `quietperiod` set to 30, if commits are found within the last 30 seconds, CruiseControl waits and then checks again.

CruiseControl Builders

CruiseControl Builders schedule when the build is to be carried out and what to do. You can define a schedule on an interval, for example, every 10 minutes or at a specific time, for example, 20:00 hrs. Since this book is about Ant, the builder that you will probably use is the `<ant>` plugin, however there are other builders too, for example to call Apache Maven or a command line executable. As an example to (potentially) execute the Ant target “dist” every 20 minutes (1200 seconds) you could add the following:

```
<schedule interval="1200">
  <ant antscript="${dir.ant}/bin/ant.bat"
      antworkingdir="projects/${project.name}"
      buildfile="build.xml"
      target="build"/>
</schedule>
```

In this example the `build.xml` file located in the workspace directory is executed with the target “build”.

CruiseControl log results

As part of any Java build process you should run JUnit unit tests. The output of a JUnit test run is typically an file. To present the unit test information in the CruiseControl dashboard application, you need to merge the JUnit test results into CruiseControl’s build log. You can achieve this via the `<merge>` plugin, which can be implemented as follows:

```
<log>
  <merge file="projects/${project.name}/build/TESTS-
      TestSuites.xml"/>
</log>
```

By default, the Ant `<junit>` task creates an output file called `TEST-TestSuites.xml`, which is what this plugin merges into the CruiseControl log file.

CruiseControl Publishers

CruiseControl Publishers are executed after every build has completed. There are many publishers, to carry out actions such as sending emails, copying files or executing a target in an Ant build script. The publishers you will probably use most are the `<artifactspublisher>` plugin which copies build outputs to a specific location and the `<htmlemail>` plugin which sends an email with the build results to a set of designated users. They can be implemented as follows:

```
<publishers>
  <onsuccess>
    <artifactspublisher
      dest="artifacts/${project.name}"
      file="projects/${project.name}/
        dist/HotelDJEAR.ear"/>
    </onsuccess>
    <htmlemail
      buildresultsurl="http://localhost:8080/
        dashboard/build/detail/${project.name}">
      <always address="integrators@mailhost.com"/>
      <failure address="developers@mailhost.com"/>
    </htmlemail>
  </publishers>
```

Notice that the `<onsuccess>` plugin is used to wrap the `<artifactpublisher>` plugin so that it is only executed if the build is successful. Also note that a number of the attributes for the `<htmlemail>` plugin will be inherited from the top level definition of

`<htmlemail>`. This particular definition simply adds project specific definitions.

The CruiseControl Dashboard

As well as sending email notifications on build success or failure CruiseControl also has a web based dashboard where you can monitor the success or failure of all of the projects that are being built using CruiseControl. The default URL for the dashboard application is `http://localhost:8080/dashboard`. An example of the dashboard is illustrated in Figure 13.

From the dashboard you can get a quick indication of the status of all of your projects, drill down for further information and also “force” builds to happen. You can also edit the configuration file from the interface to make minor changes.

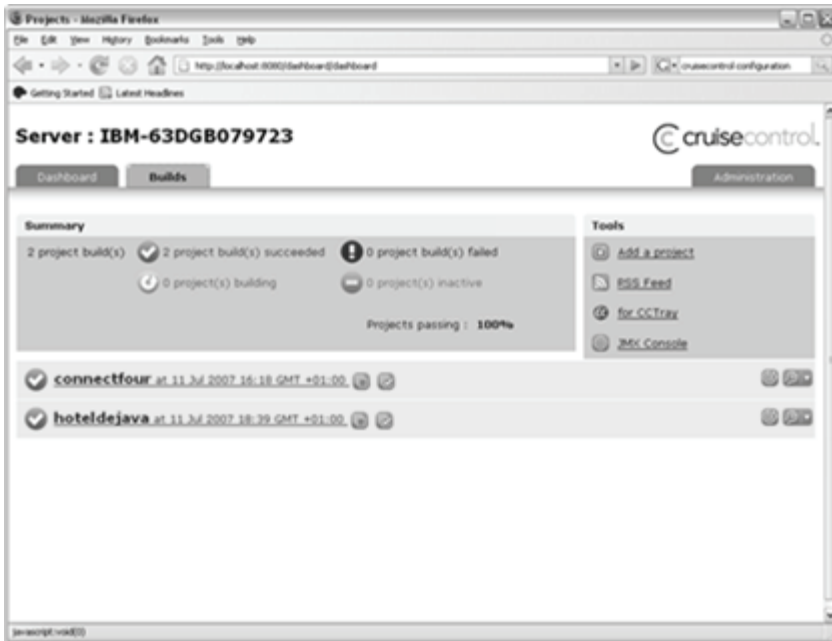


Figure 13 - CruiseControl dashboard

Ant and IBM Rational Build Forge

Build Forge (www.ibm.com/software/awdtools/buildforge) is an Enterprise build and release execution framework that allows development organizations to standardize and automate repetitive tasks. It works in combination with tools such as Ant to "orchestrate" an end-to-end build, testing and release process. Build Forge has been designed to be technology and platform neutral. It is open enough to be able to execute, manage and report on any process consisting of a sequence of command line steps and/or scripts. It also integrates with a number of version control tools to report on and monitor the changes that go into builds.

IBM Rational Build Forge has a three-tier system architecture as illustrated in Figure 14. Unlike CruiseControl, Build Forge uses a relational database to persistently store build projects and executions, and enable complete security and permissions. By default Build Forge comes with a supported DB2 express database; however it can be made to work with many other commercial databases including MySQL, Sybase, Microsoft SQL Server and Oracle.

The Build Forge **Server** is the heart of the system and oversees the activities that are executed on your build and test machines, including execution, logging, email notifications and auditing. Build Forge **Agents** are installed on the machines to be controlled. The Server sends commands to the Agents for execution; the Agents execute then, and then return the results to the Server for diagnosis.

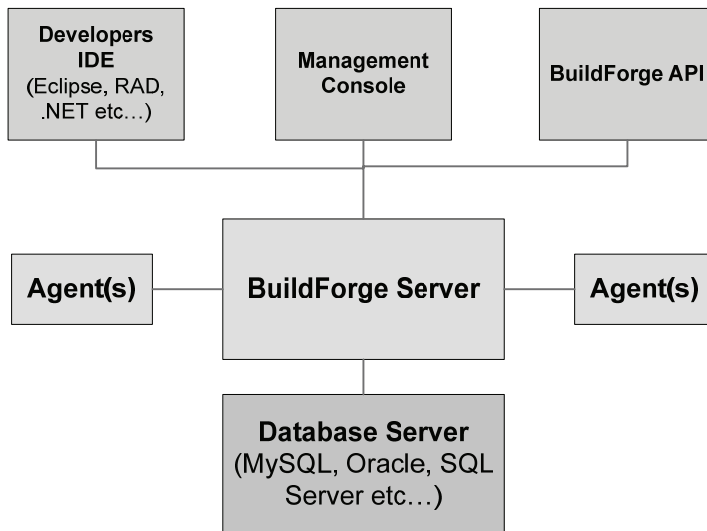


Figure 14 - Build Forge Architecture

The **Management Console** is the main control user interface for the system. Through it you organize the individual scripts or command

line executions that are part of your build process into Build Forge projects and manage the servers and environments that they will be executed on. An additional interface is provided for **Developer Self Service**. This interface integrates part of the Management Console into the developers IDE (which can be either Eclipse or Visual Studio.NET based). It allows developers to execute their own private builds using the Build Forge infrastructure and also to view build projects, status and results without leaving their IDE. There is also a unique capability which allows the developer to selectively choose local file changes from within the IDE and "reflect" them into a controlled Build Forge build. Using this capability, developers can gain confidence that the changes they are making will not break the team's integration build when they commit them into the repository.

Installing Build Forge

The first decision to make with regards to a Build Forge installation is which database server technology to use. Build Forge 7.0 and onwards can be installed automatically with a copy of DB2 Express. However if you have a preferred or existing database vendor, then you can use them instead. Once you have decided on your database, the general installation process is as follows:

- Install the IBM Rational FlexLM License Server (if not already installed).
- Request and install your Build Forge license keys from the IBM Rational Licensing Center².
- Create a database instance and database owner for Build Forge to make use of.

² <https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=rational>

- Install the Build Forge Management Console on a specific server and specific port.
- Install the latest Build Forge interim fixes from the IBM support web site³.
- Install the Build Forge Agents on the machines you wish to execute your commands on, for example, your build, version control or test servers.

The exact setup for each database will vary; refer to the latest Build Forge release notes for further information.

Build Forge has a graphically driven user interface for defining build and release processes. There are also many Enterprise capabilities that you can take of when defining these process. Therefore, rather than describing how a complete end-to-end process can be implemented⁴, I will instead concentrate on how Apache Ant can be best integrated and invoked as part of a complete Build Forge build process.

The Build Forge workspace

For every project to be executed by Build Forge, you can either create a specific build workspace or configure a step in the Build Forge project to create a new workspace for you. Creating a new workspace on each build is the recommended approach. This is made practical with Build Forge because you can either create an additional “cleanup” step at the end of the project, or even create a “cleanup” project that is automatically invoked after a certain number of builds. Creating a new workspace on each build helps ensure a clean and

³ www-306.ibm.com/software/awdtools/buildforge/support

⁴ For a walkthrough of how to setup a complete build process in Build Forge see www.buildmeister.com/modules/content/index.php?id=35

repeatable build process and also allows developers to make use of the self service "reflection" capability. In this section I will define a Build Forge project that creates and removes its own build workspace.

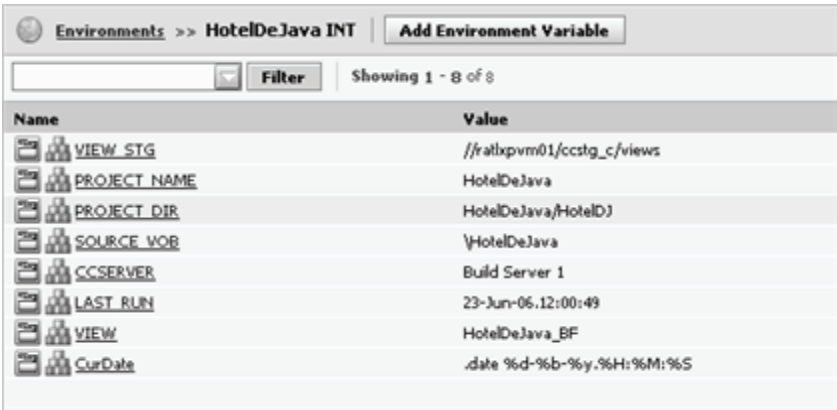
Build Forge Environment Groups

A Build Forge Environment Group is a collection of environment variables, for example the `PATH` or `JAVA_HOME` setting. Build Forge maintains these separately so that they can be assigned to servers, projects and even individual project steps as and when needed. As this approach doesn't rely on the values you last set in your login shell or system, this can help make your build process more repeatable. As well as operating system specific environment variables you can also create internal Build Forge environment variables which can then be used to configure the flow and execution of your project.

For building projects using Apache Ant, I recommend creating at least two Environment Groups for your build process as follows:

- A server based Environment Group – which defines variables to refer to the locations of your build tools, for example your Ant home directory and the system path.
- A project based Environment Group – which defines variables specific to your build project, for example the name of your project in its version control repository.

An example of the Environment Group that could be used for building the *Hotel de Java* project is illustrated in Figure 15.



Name	Value
VIEW_STG	//ratlxpvm01/ccstg_c/views
PROJECT_NAME	HotelDeJava
PROJECT_DIR	HotelDeJava/HotelDJ
SOURCE_VOB	\\HotelDeJava
CCSERVER	Build Server 1
LAST_RUN	23-Jun-06.12:00:49
VIEW	HotelDeJava_BF
CurDate	.date %d-%b-%y.%H:%M:%S

Figure 15 - Build Forge project environment

Build Forge Projects

A Build Forge **project** is the name for the overall process flow that is to be executed. Build Forge **steps** are the various tasks that are performed within the project. Each step is basically one or more command line statements that are to be executed on an Agent. A Build Forge project is typically created for each application you wish to build. In our example I will create a single Build Forge project for executing the *Hotel de Java* integration build.

As an example, the complete set of Build Forge steps required to compile, test and deploy (to a development server) the *Hotel de Java* application could be defined as listed in Table 3.

Step	Command
Create Workspace	<code>cleartool mkview -snapshot -tag "\${PROJECT_NAME}_\$_BF_TAG" -stgloc -auto</code>
Update Workspace	<code>cleartool update -add_loadrules \$SOURCE_VOB</code>

Clean Workspace	<code>ant clean</code>
Static Analysis	<code>.test "CheckStyle IF" CheckStyleAll</code>
Compile Source Code	<code>ant compile</code>
Execute Unit Tests	<code>.test "JUnit IF" JUnitAll</code>
Create Distribution Archive	<code>ant dist</code>
Create Java Documentation	<code>ant javadoc</code>
Stage Archive	<code>ant -Dbuild.num=\${BF_TAG} stage</code>
Create Baseline	<code>cleartool mklbtype -nc \${BF_TAG}_INT cleartool mklabel -recurse \${BF_TAG}_INT .</code>
Deploy Archive	<code>ant deploy</code>
Remove Workspace	<code>cleartool rmview -f \$BF_SERVER_ROOT\\${PROJECT_NAME}_\${BF_TAG}</code>

Table 3 – Hotel de Java project steps

You will notice that the majority of the steps in this project are calls to Ant targets. Although it would be possible to call a single Ant target, for example, “`ant release`”, to execute all of these targets as dependencies, this would not take advantage of the capabilities of Build Forge. By breaking them down as discrete steps, if the build fails at a particular step, you can potentially fix the problem and restart the build at that step. You can also thread a number of these steps and execute them in parallel to save total build time.

As well as calls to Ant, this example also contains steps to execute ClearCase commands using its command line tool “cleartool”. In this example Base ClearCase is being used to create a new workspace (ClearCase view), create a baseline and remove the workspace when the build has completed. Finally you will also see steps that call Build Forge DOT commands - in this case the “.test” interface to call CheckStyle and JUnit Build Forge adaptors.

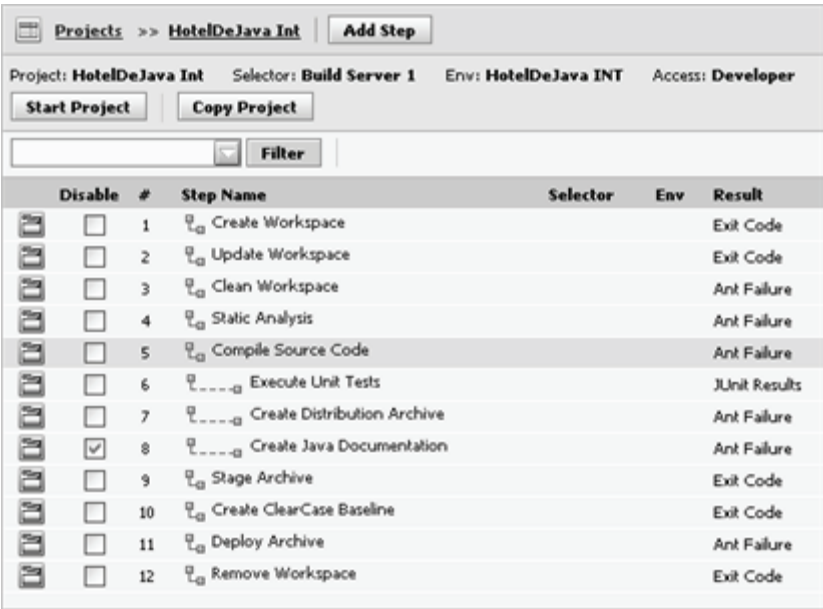
Build Forge Adaptors and Filters

In CruiseControl everything is a plugin, if you wanted to implement an integration to a new tool you would need to write the source code for the plugin. Build Forge uses a different and potentially more scaleable mechanism. Rather than write plugins for integrations, you can take advantage of Build Forge’s extension capabilities, in particular **Filters** and **Adaptors**.

Not every command you wish to execute from Build Forge will always terminate with an appropriate exit code. One such example is the Windows Ant batch file which terminates with a successful exit code irrespective of whether the build passed or failed. In this case rather than determining success or failure of the step based on exit code you can create a filter to parse the output of the command by searching for appropriate text in the build log. When an Ant build fails the build output always includes the message “BUILD FAILED”. It is therefore safe to create a Build Forge filter which searches for this string. Once you have created the filter, rather than selecting “Exit Code” as the determinant of success or failure, you would select the name of your Ant filter. An example of how the Filters could be used for the *Hotel de Java* project is illustrated in Figure 16.

Adaptors are how you interface Build Forge with version control, defect, test and many other kinds of tools. Basically, the adaptor is the application of an interface to a project. The interface is an XML file which executes the command line functionality available via the

version control, defect or test system and manipulates its output. There are a number of out-of-the-box interfaces delivered with Build Forge, including integrations for most version control tools that report on the baselines and changes that have gone into builds.

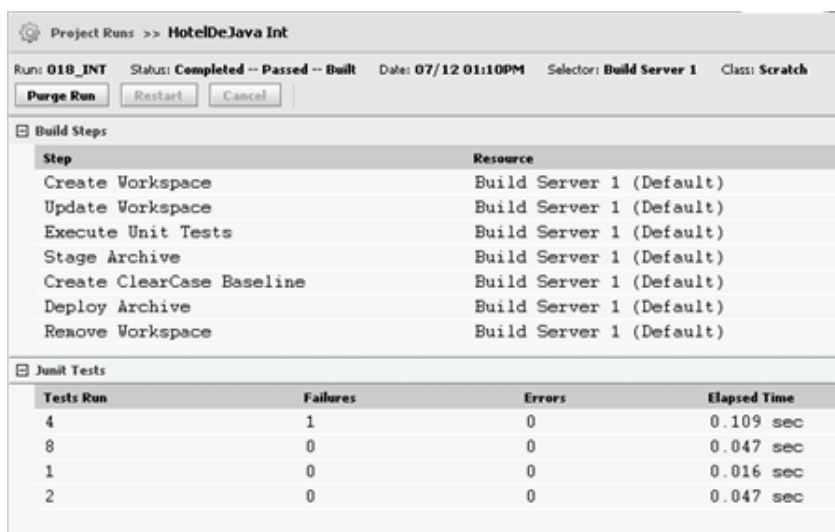


The screenshot shows the Build Forge web interface for a project named 'HotelDeJava Int'. At the top, there are tabs for 'Projects' and '>> HotelDeJava Int', along with an 'Add Step' button. Below this, a header bar displays 'Project: HotelDeJava Int', 'Selector: Build Server 1', 'Env: HotelDeJava INT', and 'Access: Developer'. There are two buttons: 'Start Project' and 'Copy Project'. A 'Filter' input field is also present. The main area is a table with 12 rows, each representing a build step. The table has columns for 'Disable' (checkbox), '#', 'Step Name', 'Selector', 'Env', and 'Result'. Steps 1 through 12 are listed, with step 5 'Compile Source Code' highlighted. The results for steps 4, 5, 7, 8, and 11 are 'Ant Failure', while others are 'Exit Code' or 'JUnit Results'.

Disable	#	Step Name	Selector	Env	Result
<input type="checkbox"/>	1	Create Workspace			Exit Code
<input type="checkbox"/>	2	Update Workspace			Exit Code
<input type="checkbox"/>	3	Clean Workspace			Ant Failure
<input type="checkbox"/>	4	Static Analysis			Ant Failure
<input type="checkbox"/>	5	Compile Source Code			Ant Failure
<input type="checkbox"/>	6	Execute Unit Tests			JUnit Results
<input type="checkbox"/>	7	Create Distribution Archive			Ant Failure
<input checked="" type="checkbox"/>	8	Create Java Documentation			Ant Failure
<input type="checkbox"/>	9	Stage Archive			Exit Code
<input type="checkbox"/>	10	Create ClearCase Baseline			Exit Code
<input type="checkbox"/>	11	Deploy Archive			Ant Failure
<input type="checkbox"/>	12	Remove Workspace			Exit Code

Figure 16 - Build Forge project steps

You can of course also implement your own interfaces. In this example I have implemented two “test” interfaces that basically call CheckStyle and JUnit via their relevant target in Ant, for example “ant checkstyle” or “ant junit”. They then parse the build output to determine success or failure and then write the results to the Build Forge Bill of Materials (BOM). An example of a BOM for a completed execution of the *Hotel de Java* project is illustrated in Figure 17.



Project Runs >> HotelDeJava Int

Runs: 018_INT Status: Completed -- Passed -- Built Date: 07/12 01:10PM Selector: Build Server 1 Class: Scratch

Purge Run Restart Cancel

Build Steps

Step	Resource
Create Workspace	Build Server 1 (Default)
Update Workspace	Build Server 1 (Default)
Execute Unit Tests	Build Server 1 (Default)
Stage Archive	Build Server 1 (Default)
Create ClearCase Baseline	Build Server 1 (Default)
Deploy Archive	Build Server 1 (Default)
Remove Workspace	Build Server 1 (Default)

JUnit Tests

Tests Run	Failures	Errors	Elapsed Time
4	1	0	0.109 sec
8	0	0	0.047 sec
1	0	0	0.016 sec
2	0	0	0.047 sec

Figure 17 - Build Forge Bill of Materials

There are a large number of interfaces delivered with Build Forge and new ones are continually being made available by IBM and its community of users⁵.

Build Forge scheduling

Build Forge allows projects to be scheduled and executed based on a number of criteria. The interface to the scheduler is graphical and uses similar concepts to that of the popular UNIX “cron” scheduler. An example of a schedule to build the *Hotel de Java* project every 20 minutes is illustrated in Figure 18.

⁵ For some examples visit www.buildmeister.com/modules/PDdownloads

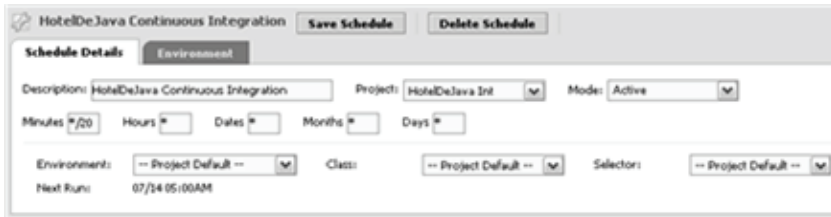


Figure 18 - Build Forge schedule

This is a continuous integration schedule which also requires the out-of-the-box version control (in this case ClearCase) adaptor interface to be enabled. This adaptor checks for changes in the repository, so although I have scheduled the project to build every 20 minutes, it will only do so if changes are found. You can create as many schedules as you want; other scenarios would include implementing daily or nightly builds at specific times.

Summary

In this chapter I have described how you can integrate Ant with both CruiseControl and IBM Rational Build Forge to schedule, execute and report on a complete end-to-end build process ensure traceability and auditability in your build process. By combining build scripting tools such as Ant with higher level build control tools you can adopt a more regular build process that in many ways can take care of itself. The productivity gain from automating these aspects can be significant.

As well as using build control tools to execute the mechanics of your build process such as compilation, you can also use them to automate aspects of your release and deployment process. This is particularly true of Build Forge where its security capabilities allow controlled deployment and execution. Using some of the capabilities of these tools together with the packaging and deployment capabilities of Ant that I discussed in chapter 3, you can implement a very

sophisticated, scaleable and productive end-to-end build and release process.

APPENDIX A

Extending Ant

*Sometimes when you innovate, you make mistakes.
It is best to admit them quickly, and get on with
improving your other innovations.*
Steve Jobs

One of the more useful features of Apache Ant is that it can be easily extended by writing new Java classes. This appendix looks at why and how you might want to extend Ant and gives an example of how to achieve it.

Why extend Ant?

One of main advantages of Apache Ant is its extensibility; since it is written in Java you can extend it by simply writing new Java classes. The obvious question here is why would you want to extend it? Does it not do everything that you need to already? Well, if you are using a set of well established tools: IDEs, compilers, application servers, databases and so on, then yes it probably will. However, what if you want to integrate with a tool that there is no current Ant integration

for? Similarly, what if you want to extend an existing integration to a tool, maybe you have your own in-house tools or you simply just want to add some new features to Ant?

In this section, I will walk through how to extend Ant. For the purposes of our implementation, I will be creating a simple integration to the ClearCase `cleartool` command. However, the basic process is the same no matter what implementation you will be carrying out.

An example task

Apache Ant already comes with a set of pre-integrated ClearCase tasks which you could use quite happily for integrating Ant and ClearCase together. 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 a `cleartool` subcommand is added or updated by IBM, then this integration becomes out of date and needs to be reworked. An alternative approach is to create a new Ant task that is a simple refactored integration to the `cleartool` command, and that meets the following requirements:

- The task shall support all current and future `cleartool` commands and arguments.
- The task shall be able to send its output to or read its input from a specified file.
- It shall be possible to specify the location of the `cleartool` command executable.
- It shall be possible to specify whether failure of an individual `cleartool` command fails the build or not.

For example, if you wanted to execute a "cleartool describe" command on a ClearCase file from Ant and send the output to a file you could write an Ant build.xml file as shown below.

```
<?xml version="1.0"?>
<project name="cc-ant" default="main">
  <target name="main">
    <ccexec failonerror="true" output="ccexec.out">
      cleartoolpath="C:\Program
Files\Rational\ClearCase\bin">
        <arg value="describe" />
        <arg value="-long" />
        <arg value="build.properties" />
      </ccexec>
    </target>
  </project>
```

As you can see from this example, this new Ant task is called `<ccexec>`; it 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 is used to indicate that Ant should terminate the build if this particular command fails. There is also an additional attribute `cleartoolPath` which you can use to specify the directory location where the ClearCase `cleartool` command can be found.

Implementing a new task

Given that this is what you might want to achieve how would you go about it? Well fortunately, you can reuse a significant amount of existing Ant functionality to achieve this. There is already an Ant task called `Exec` which works in a similar manner and allows you to execute any command line program. In this case, you simply need to

extend the Java class responsible for this task, specify the new attributes and hardcode the executable that is to be run as `cleartool`.

Create a new Java class

The first step in creating a new Ant task is deciding what the name of your task should be and what existing Ant class you want to extend. Most of the time you will extend `org.apache.tools.ant.Task`, however there are a number of other useful classes that you can extend; in our case I am extending the `Exec` task which is represented by the class `org.apache.tools.ant.ExecTask`. You can extend this class by implementing the following skeleton code:

```
package org.apache.tools.ant.taskdefs.optional;

import org.apache.tools.ant.taskdefs.ExecTask;

public class ClearToolExec extends ExecTask {
    public ClearToolExec() {
        ...
    }
    ...
}
```

I will call this class `ClearToolExec.java` and place it in the directory:

```
src\main\org\apache\tools\ant\taskdefs\optional
```

Write attribute setter methods

For each attribute that the task is to support, you need to write a setter method¹. Attributes can be thought of as parameters to the task in question. In the `build.xml` file above, I used attributes called `failonerror` and `cleartoolPath`. The `failonerror` attribute is already implemented by the `ExecTask` class so it will be inherited, however I will need to implement the `cleartoolPath` attribute myself. At the same time I will also implement an attribute called `cleartoolCmd` which specifies the name of the ClearCase `cleartool` executable program. It will probably always be set to `cleartool` but you never know!

Any setter method that is implemented must be a "public void" method which take a single argument. The name of the method must begin with `set`, followed by the attribute name. It is recommended that the first character of the name be in uppercase, and the rest in lowercase, for example `setCleartoolpath`. The `set` methods take a single parameter representing the value of the attribute to be set. In our case the Java type for both of the arguments will be `String`. I also need to create some private variables to hold the values of the attributes. To implement the setter (and getter) methods you would therefore write the following:

```
/* the name of the cleartool command */
private String ct_cmd;

/* the path to where it resides */
private String ct_path;

public String getCleartoolcmd() {
    return ct_cmd;
}
```

¹ It is good practice to write getter methods as well.

```
}

public void setCleartoolcmd(String ct_cmd) {
    this.ct_cmd = ct_cmd;
}

public String getCleartoolpath() {
    return ct_path;
}

public void setCleartoolpath(String ct_path) {
    this.ct_path = ct_path;
}
```

Add support for nested elements and character data

If the task that you are implementing is allowed to contain Ant types or other tasks then you need to implement some additional functionality. If the task will contain nested tasks, for example `<parallel>` then your class must implement the interface `org.apache.tools.ant.TaskContainer`. If your task will contain nested types, for example `<fileset>` then you will need to implement an `add` or `addConfigured` method². For example, let us assume that you wanted your `cleartool` task to support the `<fileset>` type so that you could specify a set of files as arguments, as in the following:

```
<ccexec>
```

² Ant already has a task called `<apply>` which supports nested `<fileset>` elements being invoked against an executable. You could extend this task in a similar way

```

    <arg value="checkout" />
    <arg value="-nc" />
    <fileset dir="${dir.src}">
        <include name="**/*.java"/>
        <exclude name="**/*Test*" />
    </fileset>
</ccexec>

```

In this case you would have to implement a new add method that must be a “public void” method and takes a FileSet as a parameter. The name of this method must begin with add followed by the element name, in this case addFileset. Example code to achieve this would be as follows:

```

import org.apache.tools.ant.types.FileSet

private Vector filesets = new Vector();

...

public void addFileset(FileSet set) {
    this.filesets.addElement(set);
}

public void execute() throws BuildException {
    if (filesets.size() > 0) {
        for (int i = 0; i < filesets.size(); i++) {
            ...
            add entry to argument list
            ...
        }
    }
}

```

In this case the `execute` method would iterate over the `FileSet`, adding each entry as an argument to the `cleartool` command. I am not going to include the functionality required to complete this example, as doing so would require the addition of a large amount of code. However, for a complete example refer to the source code for the `<ccstage>` task in `ClearAntLib`.

Supporting character data

If your task is to support character data between its start and end tags, for example as in the `<echo>` task:

```
<echo>here is some text</echo>
```

Then you will also need to write a `"public void addText(String)"` method. For more information on how to include nested elements refer to the Ant manual³.

Write an execute method

Next you would need to write an `execute` method; this should be a `"public void"` method, that takes no arguments and that throws a `BuildException`. You throw a `BuildException` in your own code if something goes wrong, for example the required attributes have not been initialized. In our case I do not need to throw this exception, instead I will leave it to the superclass. The `execute` method is where I will be implementing the core of the task. In our example I would write the following:

```
public ClearToolExec() {  
    setCleartoolcmd("cleartool");  
}
```

³ <http://ant.apache.org/manual>


```
        setCleartoolpath("");
    }

    public void execute() throws BuildException {
        if (getCleartoolpath().equals("")) {
            setResolveExecutable(true);
            setExecutable(getCleartoolcmd());
        } else {
            setExecutable(getCleartoolpath() +
                System.getProperty("file.separator") +
                getCleartoolcmd());
        }
        super.execute();
    }
}
```

Note that I have included the constructor for `ClearToolExec` here as well to help demonstrate the logic. Basically, this task looks to see if a `cleartoolPath` attribute has been supplied, if so it will try to find the executable on that path, if not it will let the `ExecTask` method try and resolve it (where it will look in the current directory, the `PATH` environment variable and so on).

Register the task

If the task that is being created is a completely new task then you will need to register it with Ant. This is done by including an entry in the file:

```
src/main/org/apache/tools/ant/taskdefs/default.properties
```

In this file you need to add an entry that maps the name of the Ant task as it will be used in the `build.xml` file to its Java implementation class. For example, I would map our class to the name `ccexec` as follows:

```
ccexec=org.apache.tools.ant.taskdefs.optional.ClearToolExec
```

Re-compile the source code

Once this is complete, to compile the new task, you would then build and install the Ant source distribution in the standard manner, usually by re-running the `build.bat` or `build.sh` file in the top level directory.

Final source code

The completed source code for our simple `<ccexec>` task is given below:

```
package org.apache.tools.ant.taskdefs.optional;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.taskdefs.ExecTask;

public class ClearToolExec extends ExecTask {
    /* the name of the cleartool command */
    private String ct_cmd;
    /* the path to where it resides */
    private String ct_path;

    public ClearToolExec() {
        setCleartoolcmd("cleartool");
        setCleartoolpath("");
    }

    public String getCleartoolcmd() {
        return ct_cmd;
    }
}
```

```
public void setCleartoolcmd(String ct_cmd) {
    this.ct_cmd = ct_cmd;
}

public String getCleartoolpath() {
    return ct_path;
}

public void setCleartoolpath(String ct_path) {
    this.ct_path = ct_path;
}

public void execute() throws BuildException {
    if (getCleartoolpath().equals("")) {
        setResolveExecutable(true);
        setExecutable(getCleartoolcmd());
    } else {
        setExecutable(getCleartoolpath() +
            System.getProperty("file.separator") +
            getCleartoolcmd());
    }
    super.execute();
}

}
```

As you can see this is a relatively straightforward example, with a limited amount of coding required. However if your task needs to be more complicated, then you obviously have the full capabilities of Java at your disposal.

Summary

Although at first it might seem complex to create new Ant tasks or types, with a little bit of Java knowledge it is relatively

straightforward. In this chapter I have looked at how to create a simple integration to a command line tool – which is often a common requirement. You can see how a large number of other examples have been implemented by spending some time studying the Ant source code.

APPENDIX B

Building the Reference Application

The *Hotel de Java* reference application is available for download from SourceForge. To download the source code you will need access to the Subversion `svn` command. If you do not have Subversion installed then visit <http://subversion.tigris.org> for information on how to do this.

First, create a new workspace directory to hold the application (for example `C:\workspaces`), then navigate into this directory and checkout the source using the following commands:

```
cd C:\workspaces
svn co https://hoteldejava.svn.sourceforge.net/ ↵
svnroot/hoteldejava/trunk hoteldejava
```

This will checkout the latest version of the application from the trunk into the directory `hoteldejava`. A number of directories will be created inside your workspace as follows:

- `HotelDJ` – directory containing common scripts and master build script.
- `HotelDJEAR` – directory containing Enterprise Archive definitions and resources.
- `HotelDJEJB` – directory containing EJB source code.

- `HotelDJEJBClient` – directory containing EJB client access source code files.
- `HotelDJWeb` – directory containing web front end application.

Note that this particular directory structure has been created for ease of use within the Eclipse IDE. If you used an alternate IDE or the command line environment only then you could migrate the contents of the `HotelDJ` to the top level of the workspace.

To build the application from the command line, make sure that the Ant executable is on your path and then execute the following commands:

```
cd HotelDJ
ant build
```

This will navigate into all of the source directories; compile the code and package up the application. Note that you will need an active Internet connection since the build uses the Ant tasks for Apache Maven to download all the required libraries and dependencies.

When the build has completed there will be a file called `HotelDeJava.ear` in the `HotelDJEAR` `dist` directory. This Enterprise archive can be installed directly in Apache Geronimo using the Geronimo Administration Console¹. There is also a `deploy` target that can be used to install the application from the command line.

For more information on the reference application, its requirements, and how to install the application visit the *Hotel de Java* project's quick start page².

¹ <http://cwiki.apache.org/GMOxDOC11/geronimo-administration-console.html>

² <http://hoteldejava.sourceforge.net/quick-start.html>

Note that the application also contains a set of Apache Maven build scripts, which are primarily used to generate the project web site but can also be used to build the complete application. If you want to see how Ant and Maven compare then you can study these two sets of files.

Index

A

Ant

- build auditing listener, 117
- build avoidance, 52, 53
- conditional logic, 67
- debugging, 51
- defensive programming, 54
- macros, 62
- portability, 50
- pre-defined tasks, 64

Anthill, 26

Apache Derby, 21

Apache Geronimo, 21, 83, 86, 87, 95, 158

Apache Ivy, 50

Apache Maven

- Ant tasks, 71
- defining a Classpath, 77
- dependency management, 75
- executing from CruiseControl, 129
- installing Ant tasks, 71
- publishing libraries, 78
- repositories, 72
- versus Ant, 24
- versus Apache Ivy, 50

Apache Tomcat, 23, 83, 94

B

- BEA, 24
- BEA WebLogic, 83
- Bill of Materials, 140
 - Build Forge example, 141
- BOM. *See* Bill of Materials
- Build Engineers, 19
- Build Forge*, 26, 96
 - architecture, 133
 - installation, 134
 - versus CruiseControl, 139
 - workspace, 135
- build.properties, 58
- buildinfo.properties, 60

C

- CheckStyle, 140
- checksums, 92
- Classpath
 - Ant support, 24
 - defined, 36
- ClearAntLib, 109
- ClearCase, 25, 97
 - and Ant, 109
 - baselines, 113
 - view, 98
- configuration record, 118
- continuous integration, 121, 128, 142
- CruiseControl*, 25, 96, 125
 - configuration file, 124
 - dashboard, 124, 131
 - installation, 123
 - versus Build Forge, 133
 - workspace, 126
- CVS, 25, 97, 99, 129

D

- DB2, 133, 134

- default.properties, 42, 58
- dependency
 - in detail, 43
- depends
 - overview, 36
- deployment descriptors, 86
- Deployment Engineers, 18
- derived object, 118

E

- EAR. *See* Enterprise Archive
- Eclipse, 21, 25, 27, 35, 52, 134, 158
- Eclipse Web Tools Platform, 21
- ElectricCommander*, 26
- Enterprise archive
 - creating in Ant, 90
 - defined, 84

F

- FileSets
 - as resource collections, 47
- FTP, 28, 73, 94, 122

G

- Groovy, 69

H

- Hotel de Java, 19, 20
 - building in Build Forge, 136
 - download, 157
- HTTP, 67, 83

I

- ibiblio, 72
- IBM, 24
- IBM WebSphere, 83, 86, 87, 96
- IDE
 - Ant integration, 25

- debugging Ant, 52
- Upgrading Ant version, 27

IDEA, 25

J

James Duncan Davidson, 23

JAR. *See* Java archive

Java archive

- creating in Ant, 90
- defined, 83

Java Development Kit, 25, 100

Java EE. *See* Java Enterprise Edition

Java Enterprise Edition

- packaging, 81

JavaScript, 69

JDK. *See* Java Development Kit

JUnit, 28, 32, 37, 130, 138, 139, 140

L

library.properties, 77

M

make, 23

manifest, 84

- creating in Ant, 89

Microsoft SQL Server, 133

Model-View-Controller, 57

MVC. *See* Model-View-Controller

MySQL, 133

N

NetBeans, 25

O

Oracle, 133

P

- Perforce, 97
- POM. *See* Project Object Model
- Project Object Model, 74
 - creating, 75
 - example, 79
- properties
 - build int, 40
 - from the environment, 41
 - in detail, 39
 - issues with, 55
- property files, 41
- PVCS, 97
- Python, 69

R

- regular expressions
 - using, 88
- Release Engineers, 18
- resource collections
 - referencing, 47
- Ruby, 28, 69, 70

S

- signing archives, 93
- StatSvn, 99, 106
- Subversion, 98, 100
 - and Ant, 99
 - baselines, 104
 - working copy, 98
- Sun, 24
- SvnAnt, 99
- Sybase, 133

T

- targets
 - conditional execution, 44
 - default, 43

- in detail, 42
 - overview, 30
- tasks
 - in detail, 42
 - overview, 30
- The Buildmeister
 - book, 27
 - website, 15, 16
- ThoughtWorks, 121
- transitive dependencies, 49, 78
- types
 - filters, 45
 - in detail, 45
 - mappers, 45
 - path-like structures, 45
 - redirectors, 45
 - resource collections, 45

U

Uniform Resource Name, 72
URN. *See* Unified Resource Name

V

Visual SourceSafe, 97
Visual Studio.NET, 134

W

WAR. *See* Web archive
Web archive

- creating in Ant, 90

Web archives, 85

X

XDoclet, 21
XML, 30, 55, 85, 124, 139

- as Ant build script, 34
- namespace, 110

XSLT, 35, 115

