TheElementsOfAntStyle

The original version of this document was created by Steve Loughran and Erik Hatcher for Java Development with Ant. It was used by permission (verified with the author ErikHatcher) AndrewCOliver

- http://www.manning.com/antbook
- http://www.amazon.com/exec/obidos/ASIN/1930110588

The Elements of Ant Style

General principles

Let Ant be Ant.

Don't try to make Ant into Make. (submitted by Rich Steele, eTrack Solutions, Inc.) Ant is not a scripting language. It is a mostly declarative description of steps. The declarative nature of Ant can be a source of confusion for new users, especially if a scripting language is expected.

Design for componentization.

A small project becomes a large project over time; splitting up a single build into child projects with their own builds will eventually happen. You can make this process easier by designing the build file properly from the beginning, being sure to:

- Use <property location> to assign locations to properties, rather than values. Not only does this stand out, it ensures that the properties are bound to an absolute location, even when they are passed to a different project.
- Always define output directories using Ant properties. This lets master build files define a single output tree for all child projects.

Design for maintenance.

Will your build file be readable when you get back to it six months after the project is finished? Will it execute on a clean machine? Follow these points:

- Document the build process. XML may be a file format that is both human readable and machine readable, but it is not the most easily read format for either party. A text file covering the build and deploy process will be appreciated by your successors. Of critical importance is the list of which programs and libraries are needed for the build; without it, running the build will be a trial-and-error process.
- Use comments liberally.
- Avoid dependencies on programs and JAR files outside the source tree: keep everything you can under source code control for later re-creation of development environments. That includes Ant itself, especially if you have changed it in any way.
- Keep deployment usernames and passwords out of build files. Passwords should change over time, and security is always an issue. Keep them in property files out of source code control. If a password is required for a build process, the user can be alerted to it being missing by using

```
<fail message="Provide user.password" unless="user.password"/>
```

- Never neglect false positive test case failures. Even if you know that testWorstCase always fails, four months later someone else might have the maintenance task and waste ages trying to find out why the build is reporting errors. At best, fix them; otherwise exclude them from the test suite.

Environment conventions The items in this section assume that you are launching Ant through the wrapper scripts, such as the provided ant.bat, ant.sh, antrun.pl, or antrun.py.

Run without a classpath.

There is no need to manually set your system CLASSPATH environment variable. When running through the Ant wrapper scripts, the libraries in ANT_HOME/lib are automatically placed into the system CLASSPATH before invoking Ant. Having a CLASSPATH variable simply increases the risk of Jar clash.

Place commonly used Ant library dependencies in Ant's lib directory.

In some cases it is required that libraries be in the system classpath. JUnit's library is one of them, when using the <junit> task.

Use ANT_OPTS to control Ant?s virtual machine settings.

Some tasks may require more memory, which you can set in the ANT_OPTS environment variable, using the appropriate mechanism for your platform:

```
set ANT_OPTS=-Xmx500M
set ANT_OPTS=-Xmx500M ; export ANT_OPTS
```

Use ANT_ARGS to set fixed command-line switches.

You may always want to use the -emacs and the NoBannerLogger:

```
set ANT_ARGS=-emacs -logger org.apache.tools.ant.NoBannerLogger
export ANT_ARGS=-emacs -logger org.apache.tools.ant.NoBannerLogger
```

Other settings that may be useful in ANT_ARGS are:

```
-Dbuild.compiler=jikes
-listener org.apache.tools.ant.tools.listener.Log4jListener
-propertyfile my.properties.
```

Formatting conventions

Readability and maintainability is the prevailing rationale for most of these items. Note: some IDE's and XML editors have an annoying habit of reformatting build.xml automatically - use these with caution if you care about the aesthetics of your build file. If your build file will be manually edited and readability is desired, craft it your way and if a tool attempt to spoil it, complain to the vendor and avoid using it.

Provide the <?xml?> directive.

Include the encoding if there are characters outside the ASCII range:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Use consistent indentation.

Keep <project> at the very left edge, along with the <?xml ?> tag. Two or four spaces is typical, no hard tabs. Keep closing elements aligned with opening elements, as in <target> here:

```
<?xml version="1.0"?>
<project>
..<target name="init">
....<mkdir dir="${build.dir}"/>
..</target>
</project>
```

One-line elements are acceptable.

This typically only applies to elements that combine their start and finish tags into one.

```
<echo message="hello"/>
```

One line also works for short begin and end pairs.

```
<echo>build.dir = ${build.dir}</echo>
```

Break up long lines.

Follow similar conventions as in your Java coding. Lines typically should not be longer than 80 characters, although other considerations may lower this limit. Break lines when they become longer than the limit, or readability would be increased by breaking them. These guidelines assist in breaking long lines.

Place the first attribute of an XML element on the same line as the start element tag, and place subsequent attributes on new lines indented to the same level as the first attribute.

```
<javac destdir="${build.classes.dir}"
       debug="${build.debug}"
       includeAntRuntime="yes"
       srcdir="${src.dir}"
/>
```

If an attribute value still pushes past the established line length limit, consider splitting the value into multiple properties and concatenating their values. Close self-contained elements on a new line, as shown here, with the /> characters aligned vertically with the opening < - this helps visually notice the entire block as a unit.

Whitespace is your friend.

Include blank lines between logical groupings. Examples include between: logical sets of property definitions, targets, and groupings of tasks within a target.

Define tasks, datatypes, and properties before targets.

Some tasks are allowed outside targets: <taskdef>, <typedef>, and <property>. All datatype declarations are also allowed outside of targets. When possible, place task, datatype, and property definitions prior to the first target as child elements of <project>.

```
<?xml version="1.0" ?>
<project name="library" default="main">

  <property name="tasks.jar" location="dist/tasks.jar"/>
  <taskdef resource="taskdef.properties" classpath="${tasks.jar}"/>

  <path id="the.path" includes="${tasks.jar}"/>

  <target name="usetasks">
    <sometask refid="the.path"/>
  </target>

</project>
```

Some exceptions apply, such as compiling and using a custom task in the same build - this requires the <taskdef> to be inside a target dependent upon compilation.

Ant 1.6 lets you declare anything outside of targets. Use this only for defining things (such as with the <condition> task, rather than putting code with side effects there. When you invoke ant with the -projecthelp option, all the code outside of a target may get executed.

Order attributes logically and consistently.

Define targets with name first so that it is easy to spot visually.

```
<target name="deploy" depends="package" if="deploy.server">
  <!-- ... -->
</target>
```

For commonly used tasks, such as <javac>, establish a preferred ordering of attributes and be consistent across projects:

```
<javac srcdir="${src.dir}"
       destdir="${build.classes.dir}"
       classpathref="compile.classpath"
       debug="${build.debug}"
       includeAntRuntime="yes"
/>
```

Use XML entity references to include common fragments.

```
<?xml version="1.0"?>
<!DOCTYPE project [
  <!ENTITY properties SYSTEM "file:../properties.xml">
]>
<project name="Sub-project" default="main">

  &properties;

</project>
```

Ant 1.6 adds an <import> statement that lets you import XML fragments into the build file, supporting ant properties in the declaration of the file to import. Use this in preference to to the entity reference trick, when you can. Be aware that the rules as to *where* the import is inserted into the file are different between the two mechanisms -consult the <import> documentation for up to date details.

Naming conventions

General

Use a common naming scheme among targets, datatypes, and properties.

Combine meaningful names, such as install, docs, and webapp, with meaningful types like webapp.name, webapp.classpath, and docs.patternset.

Targets

Use consistent target names.

Standard target names keep a build file understandable over time and by new developers. In a project with separate subprojects, each with their own build file, it is important to keep the names consistent between projects, not just for the benefit of programmers but for any master build mechanism. Well-known Ant targets provide a "walk up and use" experience.

The following targets are common to many builds. Always avoid changing the behavior of a well-known target name. You do not need to implement all of these in a single project.

```
all              Build and test everything; create a distribution, optionally install.
clean            Delete all generated files and directories.
deploy           Deploy the code, usually to a remote server.
dist             Produce the distributables.
distclean        Clean up the distribution files only.
docs             Generate all documentation.
init             Initialize the build: create directories, call <tstamp> and other common actions.
install          Perform a local installation.
javadocs         Generate the Javadoc pages.
printerdocs      Generate printable documents.
test             Run the unit tests.
uninstall        Remove a local installation.
```

Never override a well-known target name with a different behavior, as then the build file will behave unexpectedly to new users. For example, the docs task should not install the system as a side effect, as that is not what is expected.

Separate words in target names with a hyphen.

For example, use install-lite instead of installLite, install_lite or install lite.

Use consistent default target name across projects.

A standard default project target name allows for easy invocation from the command-line.

```
<project name="elements" default="default">

  <target name="clean">
    <!-- ... -->
  </target>

  <target name="default" depends="..."/>

</project>
```

The targets main and default make good standard default target names, although this is an area of personal preference. Whatever you choose, be consistent across all projects. To invoke a default clean build, run ant clean default .

Properties

Use standard suffixes for properties directory and library properties.

Use .dir for directory paths and .jar for JAR file references.

Prefix compilation properties with "build.".

The reason is somewhat historical, as build.compiler is Ant's magic property to control which compiler is used. In keeping with this special property, build.debug should be used to control the debug flag of <javac>. Likewise, other <javac> parameters that you wish to control dynamically should use the build. prefix.

State the role of the property first, then the type of property.

For example, a property representing whether tests need to be run or not is represented as tests.uptodate.

Separate words with a dot (.) character in property names.

Lowercase property names.

Environment variables are an exception.

Load environment variables with ?env.? prefix.

```
<property env="env"/>
```

The case of the properties loaded will be dependent upon the environment variables, but they are typically uppercase: env.COMPUTERNAME, for example, is the computer name on a Windows platform.

Abstract name and location of third-party libraries.

Using indirection keeps a build file decoupled from the location and version of third-party libraries.

build.xml

```
<property name="lib.dir" location="../somewhere/libs"/>
<property file="${lib.dir}/lib.properties"/>

...
<classpath id="compile.classpath"
           path="${oro.jar}:${xalan.jar}"
/>
```

lib.properties

```
xalan.jar=${lib.dir}/java/jakarta-xalan2/xalan.jar
oro.jar=${lib.dir}/java/jakarta-oro/jakarta-oro-2.0.6.jar
```

(submitted by Stephane Bailliez)

There are several ways to accomplish this indirection to varying degrees of flexibility and complexity. The important point is that a build file not be hard-coded with library version information.

Datatypes

Use property name formatting conventions for datatype IDs.

In other words, lowercase datatype id?s and separate words with the dot (.) character.

Standardize datatype suffixes.

Paths which represent classpaths end with .classpath, like compile.classpath. End patternset id?s with .patternset (or simply .pattern, but be consistent). Filesets end with .files.

Directory structure

Standardize directory structures.

A consistent organization of all projects' directory structures makes it easier for build file reuse, either by cut-and-paste editing, or within library build files. It also makes it easier for experienced Ant users to work with your project. A good directory structure is an important aspect of managing any software development project. In relation to Ant, a well thought out directory structure can accomplish something simply and elegantly, rather than struggling with tangled logic and unnecessarily complex fileset definitions.

The following table lists directory names commonly found in Ant projects. The build and dist directories should contain nothing in them that Ant cannot build, so clean can clean up just by deleting them.

```
    build           Temporarily used as a staging area for classes and more.
    dist            Distribution directory.
    docs            Documentation files stored in their presentation format.
    etc             Sample files.
    lib             Project dependencies, typically third party .jar files.
    src             Root directory of Java source code, package directory structure below.
    src/xdocs       Documentation in XML format, to be transformed into presentation format during the build
process.
    src/META-INF    Metadata for the JAR file.
    web             Root directory of web content (.html, .jpg, .JSP).
    web/WEB-INF     Web deployment information, such as web.xml.
```

The actual naming and placement of directories somewhat controversial, as many different projects have their own historical preference. All good layouts tend to have these features:

- Source files are cleanly split from generated files; Java class files are never generated into the same directory as the source. This makes it much easier to clean a project of all generated content, and reduces the risk of any accidental destruction of source files.
- Java files are laid out in package hierarchy, with subdirectories such as com and org, which contain vendor and project names beneath them. This is critical for Ant's Java file dependency checking, but also helps you to manage very large projects.
- Library files used are kept with the project. This avoids implicit dependencies on files from elsewhere that have been stuck onto the classpath somehow.
- Distribution files are separate from intermediate files. This lets you clean up the intermediate files while keeping the redistributable output.

Documentation conventions

Define a project <description>.

This is visible in the build file, but is also displayed when the ?projecthelp switch is used.

```
    <project name="elements" default="default">
      <description>The Elements of Ant Style project</description>

      <target name="default"
              description="Public target"
      />

    </project>
```

Running ant -projecthelp shows the description

```
    Buildfile: build.xml
    The Elements of Ant Style project
    Main targets:

    default  Public target

    Default target: default
```

Use comments liberally.

Be succinct and do not repeat what is already obvious from the build file elements. There is no need to say <!-- compile the code --> followed by <javac>.

Here is an example of a block comment to separate sections visibly.

```
    <!-- ================================================== -->
    <!-- Datatype declarations                              -->
    <!-- ================================================== -->
    <path id="compile.classpath">
      <pathelement location="${lucene.jar}"/>
      <pathelement location="${jtidy.jar}"/>
    </path>
```

One general comment (no pun intended) regarding XML comments. The above "comment" is actually 3 comments from XML's perspective. The following convention ensures that if you do something like use XSLT to generate reports from your build.xml files, then the XSLT won't have to do the extra work of taking the individual comments and combining them into one. Also, tools that display XML will show a single comment, not three distinct comments. It's not so bad in this example, but when the comment is 20 lines long, 20 individual 1-line comments can become hard to follow.

```
<!-- ===================================================
     Datatype declarations
     =================================================== -->
<path id="compile.classpath">
  <pathelement location="${lucene.jar}"/>
  <pathelement location="${jtidy.jar}"/>
</path>
```

Define a usage target.

If your build file is used mostly by new users, having the default target display usage information will help them get started.

```
<project name="MyProject" default="usage" basedir="..">
  <target name="usage">
    <echo message="  Execute 'ant -projecthelp' for build file help."/>
    <echo message="  Execute 'ant -help' for Ant help."/>
  </target>
  ...
</project>
```

(submitted by Bobby Woolf)

You may, alternatively, want your default target to perform the primary build of the project but still have a usage target that can be invoked if needed.

Alias target names to provide intuitive entry points.

For example, you may want to have a usage target but users would also type ant help and expect assistance. An alias is simply an empty target which depends on a non-empty target.

```
<target name="help" depends="usage"/>
```

Include an "all" target that builds it all.

This may not be your default target though. This target should at least create all artifacts including documentation and distributable, but probably would not be responsible for installation.

Give primary targets a description.

Targets with a description appear as "Main targets" in the -projecthelp output. The description should be succinct and provide information beyond what the actual target name implies.

```
<target name="gen-ejb"
        description="Generate EJB code from @tags"
        depends="init">
  <!-- ... -->
</target>
```

Higher-level build process documentation or diagrams can be generated from a build file using XSL transformations or other techniques (submitted by Greg 'Cosmo' Haun). This is a rationale for keeping descriptions short. Use XML comments for more detailed information if necessary.

Add an XSLT stylesheet to make build.xml browsable.

Right after the <?xml ... ?>, you can add a stylesheet reference, e.g.

```
<?xml-stylesheet type="text/xsl" href="./ant2html.xsl"?>
```

Ant will ignore it, but if you open build.xml in a browser, it will show a nicely formatted version of the script.

A sample stylesheet can be found at http://www-106.ibm.com/developerworks/xml/library/x-antxsl/examples/example2/ant2html.xsl

(tip by Jim Creasman, taken from an IBM developerworks article, entered by Niels Ull Harremoës)

Programming conventions

General

Copy resources from source path to classpath.

Metadata resources are often kept alongside source code, or in parallel directory trees to allow for customer customization or test data, for example. These files are typically property files or XML files, such as resource bundles used for localization. Tests may assume these files are available on the classpath. Copying these non-source files to the directory where source files are compiled to allows them to be picked up automatically during packaging and testing.

```
<copy todir="${test.classes.dir}">
  <fileset dir="${test.src.dir}" includes="**/*.properties"/>
</copy>
```

Targets

Clean up after yourself.

Make sure that all artifacts generated during a build are removed with a clean target. You may also want a cleandist target such that the clean target only removes the temporary build files but leaves the final distributable, and cleandist removes everything including the generated distributable.

Properties

Use properties to name anything that could need overriding by the user or a parent project.

Every output directory deserves a unique property name, suffixed by .dir. Other items that make good candidates for properties are: configuration files, template files used for code generation, distributable file names, application names, user names, and passwords.

Use location for directory and file references.

This results in the full absolute path being resolved and used for the property value, rather than just the string (likely relative path) value.

```
<property name="build.dir" location="build "/>
```

Handle creation and deletion of property referenced directories individually.

Do not assume that hierarchically named properties refer to hierarchically structured directories.

```
<project name="DirectoryExample" default="init">
  <property name="build.dir" location="build"/>
  <property name="build.classes.dir" location="${build.dir}/classes"/>

  <target name="init">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>

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

The <mkdir> task will make multiple levels of directories if they do not exist, so if only <mkdir dir="${build.classes.dir}"/> was used in the init target, both the build and build/classes directories would be created. However, because properties are designed to be overridden you cannot assume that build. classes.dir is physically under build.dir. A master build file may wish to redirect specific output.

Understand and utilize property rules.

Properties are immutable. This fact alone can be the source of frustration for those seeking to implement variables, or the source of great flexibility when used properly. A property sticks with its first value set, and all future attempts to change it are ignored. Load properties in the desired order of precedence. The needs of the project, development team, and organization dictate the specific order needed. Loading user-specific properties is a convention all projects should follow. The following code is an example of typical property ordering:

```
<!-- Load environment variables -->
<property environment="env"/>

<property name="user.properties.file"
          location="${user.home}/.build.properties"
/>

<!-- Load the project specific settings -->
<property file="build.properties"/>

<!-- Load user specific settings -->
<property file="${user.properties.file}"/>
```

This fragment loads all environment variables as Ant properties first. The user-specific properties file is mapped to an Ant property so that its location can itself be overridden, but it is not loaded until after the project-specific properties are loaded. This allows a project to have more say in its settings than a users preference, if it is needed.

I find it more useful to 'override' the project properties with user specific properties. To do this, the order of the property file loads need to be switched so that the user properties are loaded first (Bret Hansen)

Base hierarchically named directory properties from parent directory property.

Don't do this:

```
<property name="build.dir" location="build"/>
<property name="build.classes" location="build/classes"/>
```

Do this instead:

```
<property name="build.dir" location="build"/>
<property name="build.classes" location="${build.dir}/classes"/>
```

The difference becomes apparent when a user or master build wants to override build.dir. In the first example, only build.dir is relocated to the overridden directory, but build.classes will remain under the base directory. In the second example, overriding build.dir has the desired effect of moving all child directories too.

Achieve conditional logic with <property>.

New users to Ant often overlook this technique when searching for ways to achieve conditional logic.

Defaulting the build.number property to zero if it is not set in a properties file takes advantage of property immutability.

```
<property file="build.properties"/>
<property name="build.number" value="0"/>
```

If build.number is loaded from build.properties, the second <property> task is essentially ignored.

Setting properties differently depending on the server only requires a single build file line.

```
<property file="${server.name}.properties"/>
```

The server.name property could be set at the command-line, set from the environment (with some indirection from <property name="server.name" value="${env.COMPUTERNAME}"/>), or from a previously loaded properties file.

Use prefix to uniquely identify similar property names.

Because of property immutability and name clash possibilities, the <property> task allows prefixing all loaded properties.

To load two server configuration files with unique prefixes, use prefix.

```
<property file="server1.properties" prefix="server1"/>
<property file="server2.properties" prefix="server2"/>
```

Both files may have a server.name property, but they will be accessible as server1.server.name and server2.server.name Ant properties.

Datatypes

Define reusable paths.

This eases build file maintenance. Adding a new dependency to the compile classpath, in this example, automatically includes it in the test classpath.

```
<path id="compile.classpath">
  <pathelement location="${lucene.jar}"/>
  <pathelement location="${jtidy.jar}"/>
</path>

<path id="test.classpath">
  <path refid="compile.classpath"/>
  <pathelement location="${junit.jar}"/>
  <pathelement location="${build.classes.dir}"/>
  <pathelement location="${test.classes.dir}"/>
</path>
```

Classpath

This section deals with classpath issues within your build file.

Use explicit classpaths wherever possible.

Although some libraries must be in ANT_HOME/lib, keep the ones that are not needed there in a separate location, and refer to them with Ant properties to allow for overriding. Avoid, if possible, using filesets to pull *.jar into a path declaration, as this makes the build file break if conflicting JAR files are added later.

```
<path id="task.classpath">
  <pathelement location="${build.dir}"/>
  <pathelement location="${antbook-common.jar}"/>
  <pathelement location="${lucene.jar}"/>
</path>

<java classname="org.example.antbook.tasks.Searcher"
      fork="true"
      classpathref="task.classpath">
  <arg file="${index.dir}"/>
  <arg value="${query}"/>
</java>
```

Turn off includeAntRuntime on <javac>.

By default, includeAntRuntime is true. There is no need for it to be in the classpath unless you are building custom Ant tasks. Even then, it is not necessary as you can include ${ant.home}/lib/ant.jar in the classpath manually.

Use the refid <property> variant for string representation of a path.

This can be useful for debugging purposes, or especially handy when invoking Java programs with <apply>.

This example invokes JavaNCSS on a set of source files, producing an XML output file.

```
<path id="ncss.classpath">
  <fileset dir="${ncss.lib.dir}" includes="**/*.jar" />
</path>

<target name="ncss" depends="compile">
  <property name="cp" refid="ncss.classpath" />
  <apply executable="java"
         parallel="true"
         dir="${src.dir}"
         output="${build.dir}/${ant.project.name}-ncss.xml">
    <arg line="-cp %classpath%;${cp}" />
    <arg value="javancss.Main" />
    <arg line="-package -xml" />
    <fileset dir="${src.dir}" includes="**/*.java" />
  </apply>
</target>
```

(submitted by Paul Holser)

Testing

Write tests first!

While not directly related to Ant, we find it important enough to say whenever we can.
Corollary: if you can't do them first, do them second.

Standardize test case names.

Name test cases *Test.java. In the <junit> task use <fileset dir="${test.dir}" includes="**/*Test.class"/>. The only files named *Test.java are test cases that can be run with JUnit. Name helper classes something else. Another useful convention is naming base, typically abstract, test cases *TestCase.

Provide visual and XML test results.

Use <formatter type="brief" usefile="false"/> and <formatter type="xml"/>. The brief formatter, without using a file, allows for immediate visual inspection of a build indicating what caused the failure. The XML formatter allows for reporting using <junitreport>.

Incorporate the single test case trick

The nested <test> and <batchtest> elements of <junit> allow for if/unless conditions. This facilitates a single test case to be run when desired, or all tests by default.

```
<junit printsummary="no"
       errorProperty="test.failed"
       failureProperty="test.failed"
       fork="${junit.fork}">
  <classpath refid="test.classpath"/>

  <test name="${testcase}" if="testcase"/>
  <batchtest todir="${test.data.dir}" unless="testcase">
    <fileset dir="${test.classes.dir}"
             includes="**/*Test.class"
    />
  </batchtest>
</junit>
```

During development, a single test case can be isolated and run from the command-line:

```
ant -Dtestcase=org.example.antbook.TroublesomeTest
```

Fail builds when tests fail.

By default, the <junit> task does not halt a build when failures occur. If no reporting is desired, enable haltonfailure and haltonerror. However, reporting of test cases is often desired. To accomplish reporting of test failures and failing the build together, borrow this example:

```
<junit printsummary="no"
       errorProperty="test.failed"
       failureProperty="test.failed"
       fork="${junit.fork}">
  <!-- ... -->
</junit>

<junitreport todir="${test.data.dir}">
  <fileset dir="${test.data.dir}">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames" todir="${test.reports.dir}"/>
</junitreport>

<fail if="test.failed">
  Unit tests failed.  Check log or reports for details
</fail>
```

When tests fail, the property test.failed is set, yet processing continues. The conditional <fail> stops the build after reports are generated.

Code test cases that need data to adapt.

Place test data files alongside test cases, copy the files to the test classpath during the build, and access them using Class.getResource. Read test configuration information from system properties, and set them from Ant. Both of these techniques are illustrated in this example.

```
<copy todir="${test.classes.dir}">
  <fileset dir="${test.src.dir}" excludes="**/*.java"/>
</copy>
<junit printsummary="no"
       errorProperty="test.failed"
       failureProperty="test.failed"
       fork="${junit.fork}">
  <classpath refid="test.classpath"/>
  <sysproperty key="docs.dir" value="${test.classes.dir}"/>
  <!-- ... -->
</junit>
```

System.getProperty is used to retrieve the value of docs.dir. Our tests can be controlled easily and values adjusted through Ant properties.

Cross-platform issues

Launch even native scripts in a cross-platform manner.

Disabling the vmlauncher setting of <exec> executes through Ant?s launcher scripts in ANT_HOME/bin.

```
<exec executable="script" vmlauncher="false" />
```

This will launch "script" on UNIX and "script.bat" on Windows OS's. This assumes the UNIX script version does not have a suffix and the Windows version has a supported suffix as defined in the PATHEXT environment variable. This works because setting the vmlauncher attribute to false causes the command to be executed through cmd.exe on Windows NT/2000/XP, antRun.bat on Windows 9x and antRun on UNIX. Otherwise, with JVM's 1.3 and above, the command is executed directly, bypassing any shell or command interpreter. (Submitted by Bill Burton)

Debugging

Log important information, at the appropriate level.

The <echo> task has an optional level attribute. Use it to provide information at verbose and debug levels.

```
<echo level="verbose">Seen with -verbose</echo>
<echo level="debug">Seen with ?debug</echo>
```

Adding diagnostic output at the debug level can help troubleshoot errant property values, yet the output will not be seen during normal builds. Run ant -debug to see such output.

Add a debug target.

Adding a debug target with no dependencies with an <echoproperties> can shed light on possible misconfiguration.

```
<target name="debug">
  <echoproperties/>
</target>
```

Because properties can be defined inside targets, simply running ant debug will not necessarily display them all. Running two targets from a single-command line execution will, as properties retain their values across target invocations. For example, if the init target sets properties, running ant init debug will display the properties set by init.

Increase Ant's verbosity level.

By default, messages displayed to the console during Ant builds are only a fraction of the messages generated by the Ant engine and tasks. To see all logged output, use the -debug switch:

```
ant -debug
```

The -debug switch generates an enormous amount of output. A good compromise is the -verbose switch, which outputs more than the default informational level of output, but less than the debugging output.