Writing Better Ant Scripts: Techniques, Patterns and Antipatterns

Make your builds more manageable, maintainable, and understandable

Douglas Bullard
Nike, Inc.

2/19/2008

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Topics covered in this presentation

- Patterns
- Antipatterns
- Techniques
- Putting it all together: Designing master/project build scripts

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

"I suffered for this, now it's your turn"

George Harrison, *"I, Me, Mine"*

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Common problems with Ant build scripts:

- Many scripts are complicated, hard to understand
  - Old scripts are never upgraded
  - Workarounds for limitations in older versions of Ant made obsolete by new Ant tasks
- Little or no reuse within or across projects
  - Every script is different, every script is new
  - Differences between scripts can be confusing to developers
- Difficult to debug
- Impossible to tell what versions of libraries used
- Difficult to upgrade to new versions of Ant

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Goals of writing better Ant scripts:

- Standardize build scripts
- Maximize reuse of code within the project
- Maximize reuse of code across projects
- Improve readability
- Improve productivity
- Limit number of visible targets to minimize confusion
- Allow easier upgrading to new versions of Ant

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Patterns

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Reuse code with <macrodef>

- One of the most powerful ways of reusing Ant code is the proper use of <macrodef>
- Macrodefs allow you to define a "private method" with "parameters", called attributes
  - Repeated invocations can use different values for the attributes without conflict

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Reuse code with <macrodef>

- Macrodefs are better than <ant> and <antcall>
  - Most uses of <antcall> can be replaced by macrodefs
  - Macrodefs aren't targets
    - Putting code into macrodefs limits visibility
    - Once you define a property, it's defined forever. This limits the ability to use the same target more than once with different property settings
      - <antcall> and <ant> tasks cab get around this, but care must be taken
      - <antcall> and <ant> tasks are *slow!*
      - <antcall> runs all targets again!
- Macrodefs allow easier code flow than trying to specify "depends"

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Example of code reuse with <macrodef>

This is a simple example of repeating blocks of code with minor differences in structure.

```
<target name="compile">
   <javac srcdir="${source.java.dir}"
      classpathref="classpath.main.compile"
      destdir="${compile.dir}"
      debug="${compile.debug}"
      debugLevel="${compile.debugLevel}"
      deprecation="${compile.deprecation}"
      includeAntRuntime="false"
      optimize="${compile.optimize}">
   </javac>
   <javac srcdir="${unit.test.source.dir}"
      classpathref="classpath.test.compile"
      destdir="${compile.dir}"
      debug="${compile.debug}"
      debugLevel="${compile.debugLevel}"
      deprecation="${compile.deprecation}"
      includeAntRuntime="true"
      optimize="${compile.optimize}">
   </javac>
   <javac srcdir="${int.test.source.dir}"
      classpathref="classpath.test.compile"
      destdir="${compile.dir}"
      debug="${compile.debug}"
      debugLevel="${compile.debugLevel}"
      deprecation="${compile.deprecation}"
      includeAntRuntime="true"
      optimize="${compile.optimize}">
   </javac>
</target>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Example of code reuse with <macrodef>

Notice that the same method is used several times with different arguments. This makes the main body of code easier to read, and avoids calling a target with <ant> or <antcall>, getting around "properties are forever" issue.

Note that the "includeant" attribute has as default of false - you don't have to include it as an argument.

```
<target name="compile">
  <compilecode srcdir= "${source.java.dir}" classpath="classpath.main.compile"/>
  <compilecode srcdir= "${unit.test.source.dir}" includeant="true" classpath="classpath.test.compi
  <compilecode srcdir= "${int.test.source.dir}" includeant="true" classpath="classpath.test.compil
</target>

<macrodef name="compilecode">
   <attribute name="srcdir"/>
   <attribute name="includeant" default="false"/>
   <attribute name="classpath"/>
   <sequential>
      <javac srcdir="@{srcdir}"
         classpathref="@{classpath}"
         destdir="${compile.dir}"
         debug="${compile.debug}"
         debugLevel="${compile.debugLevel}"
         deprecation="${compile.deprecation}"
         includeAntRuntime="@{includeant}"
         optimize="${compile.optimize}">
      </javac>
   </sequential>
</macrodef>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Elements let you insert whole chunks of XML

Macrodef usage:

```
<doTests fork=”no”>
  <whatToTest>
    <batchtest fork="yes"
       haltonerror="false"
       haltonfailure="false"
       todir="{junit.report.dir}">
      <fileset dir="@{filesetDir}">
        <include name="@{includeName}"/>
      </fileset>
    </batchtest>
  </whatToTest>
</doTests>
```

Macrodef definition:

```
<macrodef name="doTests">
    <attribute name=”fork” default=”no”/>
    <element name="whatToTest" optional="no"/>
    <sequential>
        <junit
                printsummary="on"
                haltonfailure="false"
                fork="@{fork}"
                showoutput="true"
                failureproperty="test.failed"
                errorproperty="test.failed">
            <sysproperty key="app.root.dir" value="${app.root.dir}"/>
            <sysproperty key="fromant" value="yep"/>
            <classpath refid="runtest.classpath"/>
            <formatter type="xml"/>
            <formatter type="brief" usefile="false"/>
            <jvmarg value="-Demma.coverage.out.file=${coverage.dir}/metadata/coverage.emma"/>
            <jvmarg value="-Demma.coverage.out.merge=true"/>
            <whatToTest/>
        </junit>
    </sequential>
</macrodef>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Elements let you insert whole chunks of XML

Macrodef usage:

```
<doTests fork="no">
  <whatToTest>
    <test fork="yes"
          haltonerror="false"
          haltonfailure="false"
          name="@{className}"
          todir="${junit.report.dir}"/>
  </whatToTest>
</doTests
```

Macrodef definition:

```
<macrodef name="doTests">
    <attribute name="fork" default="no"/>
    <element name="whatToTest" optional="no"/>
    <sequential>
        <junit
                printsummary="on"
                haltonfailure="false"
                fork="@{fork}"
                showoutput="true"
                failureproperty="test.failed"
                errorproperty="test.failed">
            <sysproperty key="app.root.dir" value="${app.root.dir}"/>
            <sysproperty key="fromant" value="yep"/>
            <classpath refid="runtest.classpath"/>
            <formatter type="xml"/>
            <formatter type="brief" usefile="false"/>
            <jvmarg value="-Demma.coverage.out.file=${coverage.dir}/metadata/coverage.emma"/>
            <jvmarg value="-Demma.coverage.out.merge=true"/>
            <whatToTest/>
        </junit>
    </sequential>
</macrodef>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Chaining and discovery with <subant>

- Allows addition of new project module build files without changing master build script
- Two variants of <subant>
  - Execute the same build file but use different base directories for each invocation - use “genericantfile” attribute
  - Execute a specified list of build scripts, executing same target in each build script (takes a fileset or filelist - note that order can’t be specified in fileset, so use filelist if order matters)

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Chaining and discovery with <subant>

- Same build file but use different base directories

Invoking the "buildmodules" target calls the "deploy" target in the "masterbuld.xml" script using a different basedir each time. It will use as the basedir any directory that begins with "April_08" that is a subdirectory of "projects"

```
<project name="Master" default="buildModules">

    <target name="buildModules">
       <subant target="deploy" genericantfile="./masterbuild.xml">
          <dirset dir="../projects" includes="April_08*"/>
       </subant>
    </target>
.
.
</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Chaining and discovery with <subant>

- Same target, multiple build files from a fileset:

Invoking the "buildmodules" target calls the "deploy" target in any build script found in any subdirectory of "projects/April_08"

```
<project name="Master" default="buildModules">

    <target name="buildModules">
        <subant target="deploy">
            <fileset dir="../projects/April_08" includes="**/build.xml"/>
        </subant>
    </target>
.
.
</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Ant script inheritance with "Master" build scripts

- Importing and overriding of master scripts can be done, mimicking object inheritance and overriding of behavior
- <import> can be used to import another Ant script into the current script
- Common code can be placed into the master build script
  - Project build scripts only contain unique code for that project
  - When a script is imported into another script, the importing script can override targets from the imported script

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Ant script inheritance with "Master" build scripts

- Abstract Targets
  - Targets can be referenced in the "master" script which aren't defined there
  - Must be defined in the importing script, or else Ant will fail when run
- No-op Targets
  - Empty targets defined in the "master" script which do nothing
  - May be overridden in the importing script for more functionality

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Example "build-master.xml"

Note that this does not define targets clean or ivy, they must be defined by the importing file.

The target deploy is a no-op target – no work will be done unless they is overridden.

init-properties defines two properties

```
<project name="master" default="deploy" >
    <target name="init" depends="clean, init-properties"/>

    <target name="init-properties">
       <property name="source.dir"   value="./src"/>
       <property name="build.dir" value="./build"/>
    </target>

    <target name="compile" depends="init,ivy">
        <javac srcdir="${source.dir}" destdir="${build.dir}/classes">
           <classpath refid="build.classpath"/>
        </javac>
    </target>

    <target name="jar" depends="compile">
        <jar destfile="${build.dir}/lib/${jar.name}"
             basedir="${build.dir}/classes"/>
    </target>

    <target name="deploy"/>
</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Example of importing file

The master file is imported using the <import> task.

This file only needs to define the abstract targets clean, and ivy, specified in the master build file, plus any custom targets.

The init-properties and deploy targets in the master file are overriden in this example

```
<project name="Some Project" default="deploy">
    <import file="build-master.xml"/>

    <target name="init-properties">
       <property file="build.properties"/>
    </target>

    <target name="clean">
       <delete includeemptydirs="true" failonerror="true" quiet="true">
          <fileset dir="${target.dir}/classes"/>
          <fileset dir="${target.dir}/dependencies"/>
          <fileset dir="${target.dir}/dist"/>
          <fileset dir="${target.dir}/stage"/>
       </delete>
    </target>

    <target name="ivy" unless="no-ivy">
       <ivy-resolve file="${ivy.dep.file}" transitive="true"/>
       <ivy-retrieve sync="true"/>
    </target>

    <target name="deploy" depends="jar">
          ...
    </target>
</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Using <fail>

- "Build Failed" isn't very informative
- Missing expected properties don't fail build
- Provide more useful information by using <fail>

```
<fail unless "thisdoesnotexist" message="Missing property thisdoesnotexist"/>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- Build scripts don't always tell what version you're using
- Jars don't always have versioned names or manifests with the version in them
- This leads to library dependency hell when setting up projects
- Do you really have all the jars (or the right versions) needed for Hibernate? Or Spring?
- Version conflicts can cause unpredictable behavior

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- Solution: Which is more informative? Something like this...

```
<fileset dir="${global.lib.dir}">
      <include name="commons-beanutils.jar"/>
      <include name="commons-collections.jar"/>
      <include name="commons-digester.jar"/>
      <include name="commons-logging.jar"/>
      <include name="commons-validator.jar"/>
      <include name="commons-resources.jar"/>
      <include name="jakarta-oro.jar"/>
      <include name="struts.jar"/>
      <include name="struts-el.jar"/>
      <include name="commons-lang.jar"/>
      <include name="jstl.jar"/>
      <include name="standard.jar"/>
      <include name="commons-pool.jar"/>
      <include name="displaytag.jar"/>
</fileset>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- Solution: Which is more informative? Or this...

```
<dependencies>
    <dependency org="org.apache"          name="log4j"     rev="1.2.8"    conf="dist-ear"/>
    <dependency org="org.hibernate"       name="hibernate" rev="3.2.0.ga" conf="dist-ear,source,javadoc"/>
    <dependency org="org.apache"          name="struts"    rev="1.3.8"    conf="dist-ear,source"/>
    <dependency org="org.apache"          name="struts-el" rev="1.3.8"    conf="dist-ear,source"/>
    <dependency org="org.springframework" name="spring"    rev="2.0"      conf="dist-ear"/>
</dependencies>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- Solution: Use a dependency manager
  - Ivy + Ant ~= Maven dependency management
  - Ivy is an Apache project
  - Jars are downloaded, cached in local repository, and your specified project library location
  - Ivy can store libraries with generic names, no versions – don't need to change scripts or IDE projects when upgrading
  - Ivy can use the ibiblio and Maven2 repositories, the Ivy repository, or your own (corporate shared libraries, anyone?)
    - This gives the architect control over what library versions are available for use in projects
    - Multiple versions of libraries can be used in different projects without confusion
  - Easy distribution of libraries allows for easy packaging

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- Ivy supports transitive dependencies
  - Ivy not only brings in your project dependencies, but any dependencies they might have as well, and the dependencies of the dependencies of the dependencies, etc
  - When you create your own shared libraries, you write an XML dependency file for the libraries, declaring its own dependencies, then whenever you use this libraries you simply declare a dependency on it.
  - Ivy produces browser-viewable dependency reports when run, and has an .XSL template for viewing Ivy config files in browser

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Managing library dependencies

- More information on Ivy at http://ant.apache.org/ivy/index.html

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Pattern: Proper location of external Ant libraries

- Tasks not native to Ant need their jars imported into Ant
  - Most people place these in the Ant/lib dir
  - Most people are wrong
- Solution: Put these into an external directory, and explicitly declaring the classpath for the task in the taskdef
- This will make upgrading to the next version of Ant much easier
  - Example:

```
<taskdef name="commit" classname="net.nike.build.ant.task.svn.SvnCommitTask">
    <classpath>
        <fileset dir="${build.lib.dir}">
           <include name="nikesvn.jar"/>
           <include name="javasvn.jar"/>
           <include name="commons-collections.jar"/>
        </fileset>
    </classpath>
</taskdef>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipatterns

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipattern: Too many targets

- Do all those targets really need to be visible?
  - Solution: reduce "public" visibility
    - Use macrodefs, where it's possible to make things "private"
    - Prefacing target names with a hyphen makes them impossible to execute from the command line (i.e., "-compile-jaxb")
    - Fill out description attribute for all "public" targets
    - Include a "info" or "usage" target, with a complete list of the public targets and their documentation
      - Make "info" your default target
      - Workaround to prevent duplicate description code:

```
<target name="info" description="Shows all usable commands">
    <exec executable="cmd">
            <arg value="/c"/>
            <arg value="build"/>
            <arg value="-p"/>
    </exec>
</target>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipattern: "Spaghetti code"

- Overuse of <ant> or <antcall> makes build scripts difficult to understand
- Developers will accept stuff like this in their Ant scripts they would never accept in Java code
  - Build scripts are almost never code reviewed
- Property settings can make it difficult to determine what's really going to happen
  - The order in which targets are called may set properties differently, resulting in the same <ant> invocation doing different things
- Solution: code reviews, use macrodefs, simplify build scripts

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipattern: Duplicate code

- Many targets have very similar code
  - Solution: <macrodef> allows reuse of code with differences
- Many projects have identical code
  - Solution: use "master" build scripts for all projects, override and extend with project-specific build scripts as needed

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipattern: “Mystery” code

- Undocumented Ant scripts are as bad as undocumented Java code
- Solution: document it!
  - Build scripts need to be included in code reviews
  - All targets should have documentation
  - “Public” targets should have descriptions

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Antipattern: "Winnebago" script

- Examination of Ant scripts often reveal very large scripts are doing many different things
- Example - a single script built/deployed a J2EE application, built/deployed batch loaders, built and jarred an applet, and created/configured a WebLogic domain
- Solution: Break really large scripts up into smaller scripts
  - Some of these tasks were separated into a separate script - each was smaller and more understandable than the original
  - By doing this, the WebLogic domain creation script can now be reused
- Solution: use "master" build scripts for all projects, override and extend with project-specific build scripts as needed

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Techniques

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Treat build scripts like first-class components

- Design your build scripts
- KISS
- Code review your build scripts
- Keep build scripts up to date with new Ant features when they simplify your code

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Write your own Ant tasks

- Ant tasks are easy to write
- Custom tasks can do things Ant can't
- Custom tasks can make your build scripts more understandable
- Complex behavior is neatly tucked into a single task
  - <list-files> is an example of a custom Nike task
- A custom task should provide good documentation

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Use the Ant-Contrib Tasks

- Sourceforge project to create useful Ant tasks
- <for> and <foreach> iterate over a list, or list of paths, and calls a target for each token
  - Optional ability to run executions in parallel
  - Number of max threads can be limited
  - <for> has an optional "keepgoing" attribute. If set to true, all iterations will execute, even if one fails

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Use the Ant-Contrib Tasks

- <trycatch> gives control of possible failures
- <throw> lets you rethrow a caught exception
- <if> allows if/then/else/elseif format of flow
- <switch> allows execution based on the switched value
- AntPerformanceListener gives task durations in printout
  - ant -listener net.sf.antcontrib.perf.AntPerformanceListener target
- <stopwatch> allows timing of blocks of code

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Use the Ant-Contrib Tasks

- Ant-Contrib site: http://ant-contrib.sourceforge.net/tasks/tasks/index.html

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Technique: Build file visualization tools

- YWorks Ant Explorer
  http://www.yworks.com/en/products_antexplorer_about.htm
  - Good for viewing single scripts
  - Interactive
  - Shows property trees
  - Plugin for Eclipse, IDEA (but no IDEA 7), standalone
  - Doesn't work with multiple scripts, macrodefs, antcalls, taskdefs
- AntScriptVisualizer
  http://www.nurflugel.com/webstart/AntScriptVisualizer/
  - Good for viewing single or multiple scripts
    - Shows taskdefs, macrodefs, ant and antcalls
    - PDF, PNG, or SVG output

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Putting it all together:
Designing master/project build scripts

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Design goals of master/app build scripts

- Put common code into a single location
- Unify the way applications are built across projects
- Simplify the application build scripts
  - Minimize the number of visible targets
  - New applications should only have to write a simple build script
    - Import the master build script
    - Create a build.properties file with the expected properties required by the master build file
    - Goal of 10 lines or less for a vanilla project
- Bridge the differences between WebLogic and ATG J2EE projects

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Abstract target in a master script

master-build.xml

```
<?xml version="1.0"?>
<project name="master" >

  <target name="compile”  depends=”init">
    <echo >Master Compile</echo>
  </target>

</project>
```

build.xml

```
<?xml version="1.0"?>
<project name="cr" basedir=".." >
<import file="master-build.xml"/>

    <target name="deploy" depends="compile"/>

    <target name=”init”>
       .
       .
       .
    </target>
</project>
```

Output

```
ant deploy
Buildfile: build.xml

compile:
     [echo] Master Compile

deploy:

BUILD SUCCESSFUL
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Simple inheritance of a target from a master script

master-build.xml

```
<?xml version="1.0"?>
<project name="master" >

  <target name="compile">
    <echo >Master Compile</echo>
  </target>

</project>
```

build.xml

```
<?xml version="1.0"?>
<project name="cr" basedir=".." >
<import file="master-build.xml"/>

    <target name="deploy" depends="compile"/>

</project>
```

Output

```
ant deploy
Buildfile: build.xml

compile:
     [echo] Master Compile

deploy:

BUILD SUCCESSFUL
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Overriding a target from a master script

master-build.xml

```
<?xml version="1.0"?>
<project name="master" >

  <target name="compile">
    <echo >Master Compile</echo>
  </target>

</project>
```

build.xml

```
<?xml version="1.0"?>
<project name="cr" basedir="..">
<import file="master-build.xml"/>

    <target name="deploy" depends="compile"/>

    <target name="compile">
        <echo>Child compile</echo>
    </target>

</project>
```

Output

```
ant deploy
Buildfile: build.xml

compile:
     [echo] Child compile

deploy:

BUILD SUCCESSFUL
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Extending a target from a master script

master-build.xml

```
<?xml version="1.0"?>
<project name="master" >
  <target name="compile" depends="init">
     <super-compile/>
  </target>

  <macrodef name="super-compile">
     <sequential>
        <echo>Master super.compile</echo>
     </sequential>
  </macrodef>

</project>
```

build.xml

```
<?xml version="1.0"?>
<project name="cr" basedir="..">
<import file="master-build.xml"/>

    <target name="deploy" depends="compile"/>

    <target name="compile" depends="init">
        <echo>before master compile</echo>
        <super-compile/>
        <echo>after master compile</echo>
    </target>

</project>
```

Output

```
ant deploy
Buildfile: build.xml

compile:
     [echo] before master compile
     [echo] Master super.compile
     [echo] after master compile

run:

BUILD SUCCESSFUL
```

Note: the child target has to honor it's parent's dependencies (init) for behavior to be as expected

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Validating required properties in the app script

build.xml

```
<?xml version="1.0"?>
<project name="cr" basedir="..">
    <import file="master-build.xml"/>

    <property name=”project.name” value=”ClaimsAndReturns”/>
    <property name=”release.number” value=”5.0”/>

    <target name="deploy" depends="compile"/>

</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Validating required properties in the app script

master-build.xml

```
<?xml version="1.0"?>
<project name="master">

  <macrodef name="validate-property">
    <attribute name="propertyName"/>
      <sequential>
        <fail message="@{propertyName} is a required property"
              unless="@{propertyName}"/>
        <echo>Validated existence of property @{propertyName}</echo>
      </sequential>
  </macrodef>

  <macrodef name="validate-properties">
    <sequential>
      <validate-property propertyName="project.name"/>
      <validate-property propertyName="release.number"/>
      <validate-property propertyName="required.property"/>
    </sequential>
  </macrodef>

  <target name="init">
     <validate-properties/>
  </target>

  <target name="compile" depends="init">
     <super-compile/>
  </target>

</project>
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Validating required properties in the app script

Output

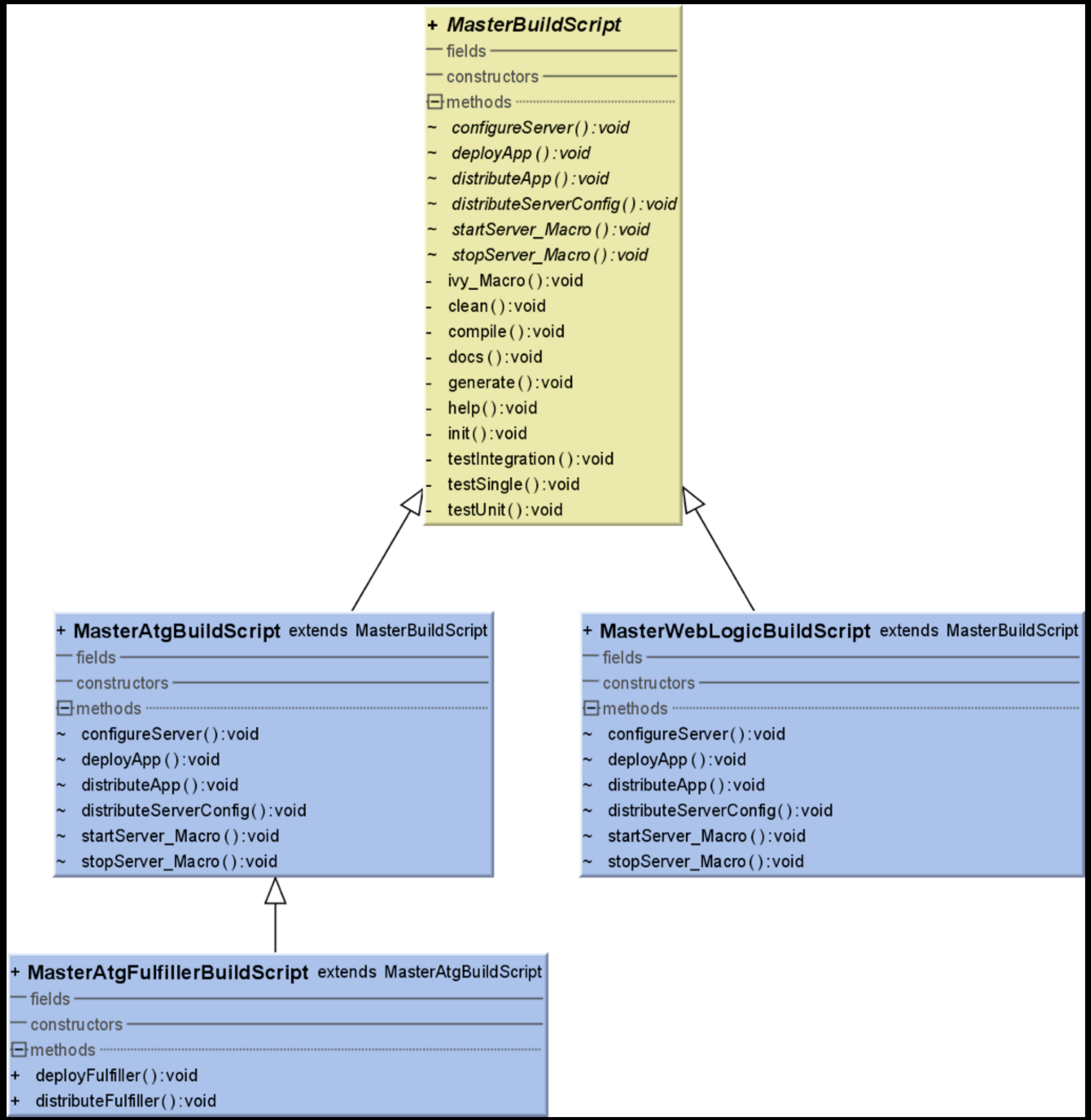
```
ant deploy
Buildfile: build.xml

init:
     [echo] Validated existence of property project.name
     [echo] Validated existence of property release.number

BUILD FAILED
build/master-build.xml:66: The following error occurred while executing
this line:
build/master-build.xml:60: The following error occurred while executing
this line:
build/master-build.xml:47: required.property is a required property
```

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Designing the build script hierarchy


+ MasterBuildScript
fields
constructors
methods
~ configureServer():void
~ deployApp():void
~ distributeApp():void
~ distributeServerConfig():void
~ startServer_Macro():void
~ stopServer_Macro():void
- ivy_Macro():void
- clean():void
- compile():void
- docs():void
- generate():void
- help():void
- init():void
- testIntegration():void
- testSingle():void
- testUnit():void
+ MasterAtgBuildScript extends MasterBuildScript
fields
constructors
methods
~ configureServer():void
~ deployApp():void
~ distributeApp():void
~ distributeServerConfig():void
~ startServer_Macro():void
~ stopServer_Macro():void
+ MasterWebLogicBuildScript extends MasterBuildScript
fields
constructors
methods
~ configureServer():void
~ deployApp():void
~ distributeApp():void
~ distributeServerConfig():void
~ startServer_Macro():void
~ stopServer_Macro():void
+ MasterAtgFulfillerBuildScript extends MasterAtgBuildScript
fields
constructors
methods
+ deployFulfiller():void
+ distributeFulfiller():void

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Issues encountered

- Property file resolution
  - Desirable for each build script to have it's own properties file
  - Ant can't do property resolution across <property> calls
  - Solution
    - Each script (master-build.xml, weblogic-master-build.xml, and the project build.xml) have their own properties file, named appropriately
    - The script we use to run Ant (sets JDK, etc). concatenates all properties files into one build.properties, which is read by all

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Issues encountered

- Project directory structure
  - Currently, many projects have different directory strutures and naming conventions
    - Example: src vs. source
- Solutions: either
  - Enforce a common directory structure and naming convention
  - Allow users to map unconventional structures via build.properties
- Standard directory structures and naming conventions were chosen

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Issues encountered

- Master build files location
  - Subversion tag was chosen
  - Projects could use with an externals
  - Allows versioned control of scripts
  - Tags can be made read-only (and should!)

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Summary: Advantages of Master/App Build Scripts

- Increased productivity
- All application build scripts look the same
- New build scripts are trivial to create
- Potential for errors and bugs is greatly reduced
- More centralized control over build scripts and configuration

Writing Better Ant Scripts: Techniques, Patterns, and Antipatterns

Examples of master and product build files