

# ANT REFACTORING

Magnus Mickelsson

Today, those who work in agile teams have learned to constantly improve their code using refactoring techniques. However, build scripts still tend to degenerate as more functionality is added. Magnus Mickelsson discusses a way to counter this, and gets a library of reusable parts as a bonus.

I have been responsible for creating scripts for automating software build processes lots of times, and I have to tell you – it's mostly kind of boring. On the flip side, we're able to reduce problems due to human errors, and can build in a minute instead of laboring manually for an hour or more. On the flop side, it feels like you've done it a thousand times before, with only minor changes from the norm.

If this is the case, I think you have not looked enough at the features that appeared with Ant 1.6: *macrodef* and *import*. A *macrodef*, short for a macro definition, can be seen as defining a reusable piece of Ant functionality – if you think of it in terms of Java code, this would be a reusable utility method.

An *import* is a way to include external Ant functionality into another script.

If you construct a set of *macrodefs* for the most commonly used tasks, you end up with a generic library to choose from when constructing new build scripts. Your build script construction then becomes a matter of putting already prepared functionality together in a way that satisfies the current demands, instead of reinventing the wheel. Build script construction and project setup can be reduced from or days hours to minutes.

How do we achieve this, when we have a huge script that we dare not touch due to its size and complexity? We can do it via a form of *Ant script refactoring*.



First, let us have a look at what macrodefs and imports are, to get an idea of the mechanisms to use. I am assuming here that you are familiar with the basics of writing Ant scripts - if you're not, have a look in the references section for a link to the Ant user manual.

## MACRODEF

A macrodef for compiling Java code using the javac Ant task could look like this:

```
<macrodef name="m_compile">
  <attribute name="build.dir" default="{build.dir}" />
  <attribute name="classpath.id" default="default.classpath" />
  <attribute name="src.dir" default="{src.java.dir}" />
  <sequential>
    <mkdir dir="@{build.dir}" />
    <javac destdir="@{build.dir}" debug="{debug}"
      optimize="{optimize}">
      <classpath refid="@{classpath.id}" />
      <src path="@{src.dir}" />
    </javac>
  </sequential>
</macrodef>
```

The `{property}` properties in this context are standard Ant properties that the macro gets from the Ant script that includes the macro. The `@{attribute}` functionality may not be quite so familiar, but it's not as confusing as you might think. It's simply the attributes defined in the actual macro above, and in terms of Java code you could say it corresponds to method parameters.

This means that you can both use macrodef attributes, which get their values from the Ant script calling the macro (or defaults), and general Ant properties, which are defined once in the Ant script on a more global, static level.

The `<sequential>` tag marks the border between attribute definitions and the actual macro functionality; within this tag you place the Ant script functionality you want the macro to have. In this example, we create a directory and then run the javac task to compile Java code into the created directory.

You can of course add lots of other Ant functionality in between sequential-tags, but that may reduce the chances of providing a macro that's actually reusable and somewhat generic. Try to analyze what would make good reusable fragments when doing this; you won't regret it later on.

## IMPORT

An Ant import means that you import Ant functionality from an external source into your build script, letting you divide and conquer when the script's size and complexity are getting out of hand.

An import can look like this:

```
<import file="${buildfiles.dir}/macro/compile.xml" />
```

This statement imports the XML from the compile.xml file into the place where the import statement is. It's a neat way to break out technical details, which can sometimes obscure what the script actually does in terms of the build process.

## USING A MACRODEF FROM AN ANT SCRIPT

When you have imported a macrodef into your build script using an import statement like above, the macro can be used in standard Ant targets like this:

```
<target name="compile">
  <m_compile/>
</target>
```

That's all, assuming that the macrodef has some valid default attributes. If you need to pass in some attributes to the macrodef which are different from the defaults, it can look like this:

```
<target name="compile">
  <m_compile src.dir="src/test" build.dir="build/testclasses/"
    classpath.id="test.classpath.id"/>
</target>
```

Note that you can use standard Ant properties as input, so instead of "src/test" you could use "\${src.test.dir}".

Now let's take a look at how a complete, working build script could be implemented using macrodef and imports.

```
<project name="hibernatespring" basedir="." default="continuous">
  <property file="build.properties"/>
  <property file="c:/j2ee/cb/buildfiles/build.properties"/>
  <import file="c:/j2ee/cb/buildfiles/default-build.xml"/>
  <!-- Composite targets building on macro base functionality -->
  <target name="continuous" depends="clean, info, init, compile, test-
compile, test, jdepend, checkstyle" description="Target for use from CC -
build and generate reports"/>
  <target name="dev" depends="info, init, compile" description="Used for
local development build"/>
  <target name="dev-clean" depends="clean, dev" description="Dev but
cleaning out old code first"/>
  <target name="dev-test" depends="dev, test-compile, test"
description="Compile and run local unit tests"/>
</project>
```

What I've done here is to create a `default-build.xml` file which contains the standard functionality of a typical project using the Hibernate and Spring frameworks. I have a local `build.properties` file that defines needed properties for this particular project, and I have a central `build.properties` file that defines the directory structure of a typical project, and other generic properties that rarely change.

My `default-build.xml` contains loads of macrodef imports, definitions of typical classpaths and other needed properties, and Ant target definition like this one:

```
<target name="checkstyle">
  <m_checkstyle />
</target>
```

These targets can then be used by a project-specific build script; to be combined as you see fit to live up to the requirements of a certain build, as you could see in the build script of the `hibernatespring` project above.

So, if you think that these scripts may be more maintainable, how do you get there?

A suggestion of steps for refactoring Ant scripts could be as follows:

1. Draw a visual representation of the major build script flows, for instance on a whiteboard.
2. Run the most important flows of the build script to see what they do in practice, and ensure that they work. Take notes.
3. Make a list of typical functions in the build script, like “compile”, “create a ZIP package” etc.
4. Prioritize the list of functions based on which function is used the most amount of times within the build script. For instance, a compile function could be used both for standard code as well as for test code. Also, put a little “dot” on so called “low hanging fruit” - functions that are not so complex.
5. Take the simplest and most used Ant function, and copy the Ant script functionality into a new file, called for instance “compile.xml” if you’re working on compile functionality.
6. Make that script fragment into a macrodef. Define attributes for all used Ant properties, and let the attributes be those Ant properties.
7. Import the new macrodef into your build script using the import statement, comment out the old script functionality and replace it with a macrodef call.
8. Try to run the main flow of the script and ensure that it still works.
9. See if there are other places where that functionality is used that would mean adding attributes to the macro. If so, add those attributes and replace the script in those other places, by using the same macrodef.
10. By now, your script code should have been reduced somewhat. Remove the function you have now turned into a macrodef from your list of possible build refactorings, and repeat the process for the next one.

## CITERUS NYHETSBRV FÖR DIG SOM VILL LYCKAS MED MJUKVARUUTVECKLING

Just as with “normal” refactoring it takes time and patience to understand and know how to apply the hints I’ve given in practice, but I hope you now have some idea of the possibilities with Ant refactoring and learning to use Ant 1.6 features.

The main idea, as always, is to make it easier to understand and maintain build scripts – and to make your organization more efficient by not having to spend more time on automation than what is needed. You do utilize automation, don’t you? If not, your competitors must love competing with you.

Happy Ant farming! ■ ■ ■

### LEARN MORE ABOUT ANT

- Ant user manual for the latest version - <http://ant.apache.org/manual/index.html>
- Ant import task - <http://ant.apache.org/manual/CoreTasks/import.html>
- Ant macrodef task - <http://ant.apache.org/manual/CoreTasks/macrodef.html>



**Are you in need help with your Ant farm? Get in touch with farmer Magnus at [magnus dot mickelsson at citerus dot se](mailto:magnus dot mickelsson at citerus dot se).**