

# Code generation using XSLT

## Build a flexible framework with open source tools

Skill Level: Introductory

[Leigh Dodds \(leigh@xmlhack.com\)](mailto:leigh@xmlhack.com)

Developer and editor  
Ingenta, Ltd.

28 Apr 2003

The automated generation of code, when applied correctly, can be a powerful engineering technique. This tutorial provides a basic introduction to code generation concepts, and in particular introduces XSLT as a code generation tool. The tutorial also discusses the limitations of XSLT when generating code, and demonstrates how to compensate for some of these issues using a flexible framework built with open source tools, such as Ant and Jalopy. This simple but powerful framework can be applied to the generation of many different types of code.

## Section 1. Tutorial introduction

### Tutorial overview

The automated generation of code, when applied correctly, can be a powerful engineering technique. This tutorial provides a basic introduction to code generation concepts, and in particular introduces XSLT as a code generation tool.

The tutorial also discusses the limitations of XSLT when generating code, and demonstrates how to compensate for some of these issues using a flexible framework built with open source tools, such as Ant and Jalopy. This simple but powerful framework can be applied to the generation of many different types of code.

The examples in the tutorial illustrate how to apply this framework to the generation of both Java code and XSLT stylesheets. The latter section introduces a powerful technique called **meta-stylesheets** that is sure to prove useful to any XSLT author.

## Who should take this tutorial?

This tutorial is designed for developers already familiar with XML, XSLT, and the Java language, and who are interested in combining these technologies to help automate development tasks and explore the potential benefits of code generation. The tutorial therefore assumes that you already understand how to write and apply XSLT stylesheets. The [Resources](#) include pointers to an XSLT tutorial for those needing a refresher on the basic concepts.

The tutorial examples also refer to Ant, an open source build tool. However, as the tutorial covers both the installation of the Ant and some background on its basic concepts, no detailed prior knowledge is assumed. If you're interested in learning more about Ant, see the [Resources](#) section.

Overall, the tutorial introduces the reader to the uses of XSLT as a code generation tool, providing useful tips, techniques, and advice.

While the majority of the examples discuss the generation of Java code, you can easily adapt the basic techniques to encompass other programming languages.

## Tools

While this tutorial does not discuss how to install an XSLT processor, you are not required to have a processor installed to follow the examples. Instructions on how to install Ant, the open source Java build tool, are available in the [Installing the tools](#) section. Ant includes Xalan as an embedded XSLT processor, and the later sections (see [A code generation framework](#)) show you how to apply XSLT transformations using Ant's built-in XSLT support. But, as no processor-specific extensions have been used, any XSLT processor will work should you care to manually apply the transformations.

The examples were tested against Ant 1.5.1. You can download the binary distributions of the latest version of Ant from the project Web site at: <http://ant.apache.org/bindownload.cgi>. See [Installing Ant](#) for information on how to install Ant and the Java source code formatter, Jalopy, available at: [http://sourceforge.net/project/showfiles.php?group\\_id=45216](http://sourceforge.net/project/showfiles.php?group_id=45216).

The code samples for this tutorial are available at [Downloads](#). To install the samples, simply extract the files, preserving the directory structure, into a newly created directory. The examples for each section are contained in the following

sub-directories:

- `bean` -- the Java code generation examples referenced from the [Generating JavaBeans](#) .
  - `meta` -- the XSLT stylesheet generation examples introduced in [Generating XSLT using XSLT](#) .
- 

## Section 2. Installing the tools

### Installing overview

This section covers the installation of the basic tools used in the rest of this tutorial.

**Ant** is a widely respected replacement for Make that gives sophisticated control over the process by which Java applications are built. **Make** is a Unix command-line utility used to manage the often complex series of tasks required to build and install an application. The process is controlled by a **Makefile**, which describes the command-line operations required to perform the actual build and install process.

In addition to providing functionality for Java application development, Ant also provides a number of tasks for working with XML documents, including applying XSLT transformations. This makes it a good framework for controlling code generation and subsequent compilation.

Ant has a plug-in framework that allows its functionality to be extended. One useful plug-in is the Jalopy code formatter, which is an open source tool capable of formatting any valid Java source code according to a set of user-defined rules. This allows source to be automatically made to conform to local coding conventions or styles. Along with Ant, Jalopy is used in several examples later in the tutorial.

### Installing Ant

The installation of Ant is very straightforward. First, download the latest version of Ant from the project Web site (see [Tools](#) for the link). Unzip the distribution into a newly-created local directory. Next, create an environment variable, `ANT_HOME`, that points to this directory (for example, `ANT_HOME=/usr/local/jakarta-ant-1.5.1`).

Finally, update your `PATH` environment variable so it includes `ANT_HOME` (for

example, `PATH=$PATH:$ANT_HOME`). This will ensure that the Ant shell scripts are available from the command line. Also check that you have a `JAVA_HOME` environment variable configured to point to the directory containing the Java Development Kit so that Ant can properly determine its location.

You can now invoke Ant directly from the command-line using the `ant` shell script. By default Ant will look for a file called `build.xml` in the current directory. You can alter this with the `buildfile` command-line parameter as follows:

```
ant -buildfile my-build-file.xml
```

You can prompt Ant to display a list of the targets defined in a build file using the following command-line:

```
ant -projecthelp
```

This is a useful way to produce a readable list of the functionality defined within a given build file.

## Installing Jalopy

See [Tools](#) for a link to the latest version of the Jalopy Ant task. Simply download the zip file and unpack it into a local directory.

Jalopy includes a graphical application for managing code styling preferences. This is available in the `bin` directory under the Jalopy installation directory. Add this directory to the `PATH` environment variable and run the `preferences` shell script from the command-line to explore the range of code formatting options that Jalopy provides.

Detailed documentation is available in the online manual, which is also included in the download. (See [Resources](#) for a link.)

Now that you've correctly configured all of the tools required to work through the rest of this tutorial, the next section introduces the basics of code generation.

---

## Section 3. Introduction to code generation

### A code generation primer

This section introduces the basic principles of code generation and reviews the two most common ways in which code generation is typically applied. Also covered are the benefits and limitations of using XML and XSLT as code generation tools.

Code generation is not a new idea; tools for automatically generating code have been around for many years. Yet, due to the increasing sophistication and complexity of modern application frameworks (such as J2EE or .NET), it is a technique that is once more gaining popularity. While not a substitute for a detailed understanding of the fundamentals of any framework, it is a tool that can (among the other benefits discussed in [Little languages and the benefits of code generation](#)) certainly help increase productivity by automating many repetitive tasks.

An example of the redundant complexity common to application frameworks can be taken from the J2EE platform: Creating a single Enterprise JavaBean (EJB) requires the creation of two separate interfaces -- an implementation of those interfaces and two deployment descriptors that configure that bean for a particular application server. There is a great deal of information shared between the various Java source files and configuration files, so using a code generator is a natural approach to managing that redundancy.

This approach is the most common way that code generation is used: The code generator creates the basic boilerplate code for a project or application component that a software engineer then adds to. Obviously, this is generally a one-off process, as the original code cannot be regenerated without losing the manual modifications.

The second approach to using code generation is more iterative. The generated code is typically limited to a specific functional area in an application, such as a database persistence layer. A developer can then treat this component as a **black box**, allowing the generated code to be incorporated into an application without change. This approach works best within a well-factored application framework with well-defined interfaces between its various components.

You can even use a code generator to create a complete application, assembling it from existing application components. In this form the technique is very similar to scripting an application, but without the need to actually embed a separate scripting language in the application. The code generator executes the script to create the code that glues the application together.

## Little languages and the benefits of code generation

The iterative **black box** approach to code generation has the potential to deliver the greatest benefits. Commonly, a declarative description of the desired functionality is created using a **little language** that serves as the input to the code generator. A little language is a high-level declarative language targeted at a particular application or business domain. Like code generators, little languages are not new ideas and

are used for more purposes than just driving the generation of code. Commonly encountered examples of little languages include Makefiles, Ant build files, Apache Web server config files, etc. But while both Makefiles and Ant build files are examples of little languages, they have very different syntaxes. However, they do share an application domain: the description of build processes.

A code generator processes scripts written in a little language to generate the actual program code. The little language is therefore used to create a functional description of the application or component's behavior. This achieves a good separation between the underlying implementation (the output of the code generator) and the functional description of the application (the input to the code generator). This separation has the following benefits:

- You can tune the code generator separately from any given functional description, thereby improving the quality of the generated code for all applications that use the code generation tool.
- You can transparently alter the target programming language, allowing code to be generated in a variety of target languages.
- A declarative description of a process, embodied as a little language, is more understandable as it is no longer obscured by implementation details.
- A non-engineer (for example, a business analyst) can create and manage the functional description, freeing the engineer to concentrate on the details of the framework itself.

## XML and XSLT for little languages

XML is a natural choice for creating little languages to serve as input for code generators. Without using XML, you would have to write a complete grammar and parser for the custom language, and this is a fairly specialized task that not all programmers are comfortable with. The wide range of off-the-shelf parsers and tools available for working with XML make it a natural alternative to environments like Lex and Yacc.

With XML as the underlying syntax for a little language, XSLT is a natural choice for a processing tool to convert that language into source code. In particular, its ability to generate plain-text output makes it suitable for generating a wide variety of different source code formats, including simple shell scripts, SQL statements, and even Java or C++ code. Other useful features of XSLT include:

- **Stylesheets**, which are highly portable between processors and platforms.

- **Built-in modularity (using `xsl:import` and `xsl:include`)**, which allows for the creation of reusable libraries of transformations.
- **XSLT and XPath**, which incorporate an extension mechanism that allows new elements and functions to be added to the language. You can make these extensions, while typically processor-specific, portable. See [Resources](#) for links to efforts to standardize XSLT extensions.

There are still some limitations to using XSLT for generating non-XML output, which are discussed next.

## Limitations of XSLT

There are a number of limitations to XSLT that should be accounted for when considering it as a code generation tool.

First, XSLT is not a generic transformation language. It has restricted capabilities in several key areas. Its plain-text output option is limited, making it difficult to control issues like whitespace and indenting. The built-in string manipulation functions are restricted, as is its ability to work with typed values (for example, dates and numbers). You can overcome these limitations by using extension functions. The EXSLT project (see [Resources](#)) offers a wide range of standardized functions. The XSLT 2.0 and XPath 2.0 specifications that are currently undergoing standardization will also offer more control in these areas.

Second, XSLT has limited access to the operating system, giving no control over the naming of its output, which is additionally limited to a single output file. This constrains the options available for generation and requires the creation of multiple interdependent output files -- for example, Java interfaces and their implementations, and C++ header and source files. Again, the use of extension elements can address this issue, and XSLT 2.0 will standardize this practice. Another option is to embed the use of XSLT within a code generation framework that would allow a single input file to be processed multiple times by different stylesheets. The framework introduced in Section 5 (see [A code generation framework](#)) can easily be extended to perform this type of additional processing

While not a limitation of XSLT per se, mixing program code with XSLT can make both difficult to manage: The code is less amenable to change with a typical IDE, and blocks of program source can obscure the XSLT markup. This problem can be mitigated through careful factoring in the underlying application API to minimize the amount of code required to invoke some functionality, coupled with modularization of the stylesheets.

## Section recap

This section introduced the basic code generation concepts, and in particular the two different approaches (one-off and iterative) to using code generators to help construct an application. It also highlighted the immediate benefits of using XML as a means to design and implement the little languages that serve as the input to code generators, and the advantages and disadvantages of using XSLT as the means of generating code from these functional descriptions.

The next section uses the generation of simple JavaBeans as a means of illustrating this use of XSLT. This is followed by a section that demonstrates how to build a framework for applying the code generating transformation using Ant.

---

## Section 4. Generating JavaBeans

### Generating JavaBeans overview

This section discusses how to generate Java code using XSLT. Specifically, it introduces two stylesheets that are capable of creating Java classes that conform to the JavaBeans naming conventions.

These naming conventions deliberately make class definitions very predictable, so that programming tools can easily manipulate them. For every property there is both a `set` method to store that property and a `get` method to retrieve the current value. This regular structure also makes JavaBeans very amenable to automated generation.

The input to the code-generating stylesheets will be a simple **bean markup** vocabulary, which is introduced next.

Note that you can find the code examples for this section in the `bean` sub-directory of the example distribution that you can download from [Tools](#).

### Bean markup

The DTD below describes the Bean Markup vocabulary, a simple declarative XML language for modelling JavaBeans. The complete DTD is available in the `beans/xml` directory in the examples.

```
<!ELEMENT bean (dependency*,
                description?,
                property+)>
```

```
<!ATTLIST bean
    name CDATA #REQUIRED
    serializable CDATA #IMPLIED>
<!ELEMENT dependency (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT property EMPTY>
<!ATTLIST property
    name CDATA #REQUIRED
    type CDATA #REQUIRED>
```

Each bean must have a name and, by setting the optional `serializable` attribute, you can mark it as being serializable. A bean can also have a description to document its role in the application.

A bean must have one or more named properties described by the `property` elements, with required `name` and `type` attributes. The value of the `type` attribute can be the name of any valid Java type. Classes from any package can be referenced as long as the name of that package is listed in a `dependency` element of the bean. A bean may be dependent on any number of additional packages.

An example of a simple bean (see `bean/xml/Person.xml` in the examples) is given below:

```
<bean name="Person" serializable="true">
  <dependency>java.util</dependency>
  <description>A Person</description>
  <property name="email" type="String"/>
  <property name="firstName" type="String"/>
  <property name="lastName" type="String"/>
  <property name="birthday" type="Date"/>
</bean>
```

From this declarative description of a JavaBean, it is possible to generate the actual code in line with the JavaBean naming conventions. The markup is deliberately simple in order to illustrate the code generation process; a more realistic vocabulary might allow for describing the relationships between different beans, describing additional behavior, or defining multiple beans in each markup file.

## The stylesheets: `java-util.xsl`

Converting the Bean Markup language into code relies on mapping the different markup elements to their corresponding structure(s) in the Java source. For example, the `bean` element maps to a class definition, while each `property` element maps to three outputs: a member variable, a `get` method, and a `set` method.

You can find the two stylesheets used to generate Java code from the Bean Markup in the `bean/xslt` directory in the examples (see [Tools](#)). The first of these is `java-util.xsl`. This stylesheet defines several utility templates -- XSLT named

templates that accept several parameters -- that aid in the code generation:

- `capitalizeFirstLetter` -- returns its single parameter with the first letter capitalized (if necessary). This is useful for generating method names according to the Java *camel case* convention.
- `getMethodPrefix` -- accepts a `type` parameter and returns the correct prefix for the corresponding `get` method. In the JavaBean convention, you should name the accessor methods `getPropertyName` unless they return a boolean, in which case the convention is to name them `isPropertyName`.
- `makeImport` -- creates a Java import statement, optionally introducing a wildcard.
- `makeMemberVariable`, `makeSetMethod`, `makeGetMethod` -- all of these accept `name` and `type` parameters and generate the Java code that their name indicates.

A useful tip to remember while writing code-generating transformations is to factor common functionality into a separate stylesheet. With this kind of utility stylesheet, the code can be reused in other code generation transformations.

The actual transformation is driven by a second stylesheet, which is discussed next.

## The stylesheets: bean2java.xsl

The code generation transform is controlled by the `bean2java.xsl` stylesheet that references the utility stylesheet using the `xsl:include` element. The main template in this stylesheet is shown below:

```
<xsl:template match="bean">
  <xsl:apply-templates select="dependency"/>

  <xsl:if test="@serializable = 'true'">
    <xsl:call-template name="makeImport">
      <xsl:with-param name="package"
        select="'java.io.Serializable'"/>
      <xsl:with-param name="wildcard"
        select="false()"/>
    </xsl:call-template>
  </xsl:if>

  /**
   * <xsl:apply-templates select="description"/>
   */
  public class <xsl:value-of select="normalize-space(@name)"/>
  <xsl:if test="@serializable = 'true'">
    implements Serializable
  </xsl:if>
  {
    <xsl:apply-templates select="property"/>
  }
```

```

}
</xsl:template>

```

The essential aspect to this example is that the template uses the XSLT **pull** style of processing: Rather than simply using `xsl:apply-templates` without any parameters, the stylesheet explicitly directs the processor toward particular elements in the source tree. This gives greater control over how the transformation happens. This is vital when the structural relationships within the input and output are different. For example, in the Bean Markup, the `description` and `dependency` elements are children of the `bean` element. In the Java source code, these structures map to `import` statements and Javadoc comments that must occur before the class definition.

## The stylesheets: matching property elements

The second major section in the `bean2java.xsl` stylesheet is the template that matches `property` elements. This template controls the generation of the member variables and the required `get` and `set` methods for each named property. A fragment of this template, along with one of the utility templates it invokes, is shown below.

```

<!-- from bean2java.xsl -->
<xsl:template match="property">
  <!-- .. make member variable ... -->

  <!-- .. make set method ... -->

  <xsl:call-template name="makeGetMethod">
    <xsl:with-param name="name">
      select="normalize-space(@name)" />
    <xsl:with-param name="type">
      select="normalize-space(@type)" />
  </xsl:call-template>
</xsl:template>

<!-- from java-util.xsl -->
<xsl:template name="makeGetMethod">
  <xsl:param name="name"/>
  <xsl:param name="type"/>
  <xsl:variable name="capitalizedName">
    <xsl:call-template name="capitalizeFirstLetter">
      <xsl:with-param name="property-name"
        select="$name" />
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="methodPrefix">
    <xsl:call-template name="getMethodPrefix">
      <xsl:with-param name="type"
        select="$type" />
    </xsl:call-template>
  </xsl:variable>

  public <xsl:value-of select="$type" />&sp;
  <xsl:value-of select="$methodPrefix" />
  <xsl:value-of select="$capitalizedName" />()
  {

```

```

    return _<xsl:value-of select="$name"/>;
  }
</xsl:template>

```

As demonstrated, the `makeGetMethod` template is invoked with the `name` and `type` attributes of the property as parameters. The XPath `normalize-space` function is used to ensure that redundant whitespace is stripped from either end of the attribute values. The template uses two variables to help construct the method signature. See [The stylesheets: java-util.xsl](#) for notes on the behavior of the others.

The other notable aspect of this template is the use of an entity `&sp;` as a spacer between the return type and the generated method name. Using entities to manage whitespace is discussed next.

## A note about whitespace

When generating plain-text output using XSLT, it's important to understand how the XSLT processor will normalize whitespace in your stylesheet, as this can significantly alter the results of the transformation. When generating source code, this can equate to introducing bugs into the source that will cause compilation errors.

These normalization rules are quite straightforward: Any text node in the stylesheet that consists only of whitespace characters will be stripped from the stylesheet *unless* it occurs as the value of an `xsl:text` element. For example, in the following code an XSLT processor will remove the space in between the two elements.

```
<xsl:value-of select="$a"/> <xsl:value-of select="$b"/>
```

To preserve this whitespace -- to ensure that there is a space between the type of a Java variable and the variable name -- you should alter the above as follows:

```
<xsl:value-of select="$a"/><xsl:text> </xsl:text><xsl:value-of select="$b"/>
```

Introducing these extra `xsl:text` elements adds further clutter to a stylesheet and can obscure the details of the transformation. One way to reduce the clutter is to use an entity, which is declared at the top of the stylesheet, as follows:

```
<!DOCTYPE stylesheet [ <!ENTITY sp "<xsl:text> </xsl:text>">
]>
<xsl:stylesheet/>
```

The entity will be substituted for its replacement text once the stylesheet is parsed, so it can be used in place of the `xsl:text` elements. Note that it is not enough to

simply declare the replacement text to be a single space. The stripping of whitespace happens *after* the entity has been replaced.

The `java-util.xsl` stylesheet declares several entities, including examples for carriage returns and tabs that can further clarify stylesheet structure. The entity names are also easier to remember than the equivalent Unicode character references (for example, `&#9 ;` for a tab).

You can also control the presence of whitespace in the input document, as opposed to the stylesheet, by using the `xsl:strip-space` and `xsl:preserve-space` elements. For more information on these, refer to a dedicated XSLT tutorial (see [Resources](#) for some useful pointers).

## Summary so far

At this point in the tutorial, you have reviewed the basic structure of two stylesheets that you can use to generate a Java class from the Bean Markup vocabulary. One stylesheet consists entirely of reusable utility templates and could form the basis of a library of useful code-generating templates. This stylesheet also helps to localize the actual program code, minimizing the mixture of the Java and XSLT syntaxes.

The second stylesheet performs the actual transformation using these utility templates. This stylesheet uses the **pull** style of XSLT processing to give better control over the structure of the output. Generally speaking, it is unlikely that the structure of generated code will have much correspondence to the structure of the XML vocabulary controlling the code generation.

This section has also briefly reviewed some of the whitespace handling rules in XSLT, highlighting one possible source of problems as well as presenting a solution that further promotes readability in the XSLT source.

While you can use the stylesheets introduced in this section directly (from the command-line) to manually generate the required code, the next section demonstrates how you can incorporate the transformation into an Ant build process that will allow the code generation to fit together nicely with the compilation of the end results.

---

## Section 5. A code generation framework

### Framework overview

This section walks through the basics of using Ant as a means to tie together required code generation steps with the rest of the build of a Java project. To complete this section, ensure that you have followed the instructions in [Installing the tools](#).

An Ant build process is configured using a **build file** that defines a number of **targets**, which are discrete steps in the process. Each target is made up of one or more **tasks** -- instructions that tell Ant to perform specific operations such as compiling a file or creating a directory. Ant includes a wide range of built-in tasks, and there are numerous optional tasks that have been added using the plug-in system. One Ant target can be declared to be dependent on another, thereby allowing the construction of a pipeline of operations.

This section demonstrates how to define build targets that use Ant's built-in support for XSLT to generate Java code. Also introduced is the Jalopy source formatting tool as a means to tidy up the generated code prior to compilation.

The information in this section is not intended to be a complete Ant tutorial; see [Resources](#) for pointers to more information about Ant. However, there should be enough information to guide the new Ant user through the basic steps of configuring a build file.

## The basic build file

[Figure 1](#) shows the directory structure for the bean generation example. The complete build file (build.xml) is available in the bean directory (see [Tools](#) for how to download the examples).

**Figure 1. Project Directory Structure**



The build file defines four targets:

- build -- compiles all Java code (`*.java`) from the java directory into the bin directory.
- clean -- deletes all class files from the bin directory.
- formatSource -- applies a source code formatter to all Java code in the java directory.

- `generateCode` -- applies the stylesheets introduced in the last section (see [Generating JavaBeans](#)) to the XML documents in the `xml` directory in order to generate Java code.

The details of the latter two targets are discussed in the next sections (see [Generating the source](#) and [Formatting the output](#)).

The build file also defines a number of variables (known as properties) that correspond to the actual directory names:

```
<property name="java.dir" value="java"/>
```

This is a common Ant convention that avoids the need to hardcode complete paths into the build file.

To instruct Ant to execute any of these targets, simply add the target name to the command-line when invoking Ant: `ant generateCode`.

If no target is specified then Ant will invoke the `build` target, which is the project default.

## Generating the source

The Ant `xslt` task has been written to allow a stylesheet to be applied to one or more files. Using this task allows you to invoke your code generation stylesheets automatically from within a build process controlled by Ant. The XML fragment below shows the `generateCode` target from the build file.

```
<target name="generateCode" description="Convert bean markup into Java source">
  <xslt basedir="${xml.dir}"
        destdir="${java.dir}"
        extension=".java"
        style="${xslt.dir}/bean2java.xsl">
    <include name="**/*.bxml"/>
    <outputproperty name="method" value="text"/>
  </xslt>
</target>
```

The `target` element merely defines an Ant target with a unique name and an optional description. The `xslt` element instructs Ant to carry out an XSLT transformation on a specific set of files, placing the results in a specified directory.

The files to transform are collectively defined by the `basedir` attribute, which defines a base directory in the file system, and the `include` element, which identifies a filename pattern using Ant's built-in wildcards. The example above instructs Ant to begin with the directory identified by the `xml.dir` property and find

all files in that directory or any of its sub-directories that have a .xml extension. The tree-walk is implied by the double-asterisk, which matches any directory path.

Alternatively, it is possible to identify the list of files to transform by *excluding* (rather than including) files from the list:

```
<exclude name="**/*.xml"/>
```

This example would exclude any file with a .xml extension, but all others would be included by default.

The matched files are transformed in turn using the stylesheet identified by the `style` attribute of the `xslt` element. In this example the stylesheet is `bean2java.xsl`, introduced in [The stylesheets: bean2java.xsl](#).

The transformed files will be placed in a separate destination directory indicated by the `destDir` attribute. Directory structures will be preserved, so if the original file was in a sub-directory in the base directory, then the transformed result would be stored in a similarly named directory below the destination directory.

The filename of the files will remain the same, but they will be given a new suffix as defined by the `extension` attribute. In this case, a .java extension indicates that the stylesheet is generating Java source code. The original filenames must therefore match the value of the bean's `name` attribute (see [Bean markup](#)) to ensure that the filename correctly matches that of the automatically-generated class name.

This target can now be executed to generate the Java source for all of the Bean Markup example files.

## Formatting the output

Now that the `generateCode` target has been executed, several Java source files will now be present in the `java` directory (see [The basic build file](#) for a reminder of the directory structure).

A quick look at any one of these files will show that they are very poorly formatted: Not only is the indentation askew, but the class layout isn't very friendly because the member variables and methods are freely interspersed. This is to be expected because the stylesheets didn't make any attempt to properly format the Java source. To do this using XSLT is possible, but would be tricky and would further obscure the details of the transformation.

To solve this problem, you can integrate the Jalopy source formatter into the build process. For instructions on installing Jalopy see [Installing Jalopy](#). The first step to

including Jalopy in the build process is to inform Ant of the existence of an optional task and the location of the required libraries using the `taskdef` element shown below:

```
<project name=".." default="..">
<taskdef name="jalopy"  classname="de.hunsicker.jalopy.plugin.ant.AntPlugin">
  <classpath>
    <fileset dir="JALOPY_INSTALL_DIR/lib">
      <include name="*.jar" />
    </fileset>
  </classpath>
</taskdef>

<!-- ... Ant targets defined here ...-->

</project>
```

Replace `JALOPY_INSTALL_DIR` with the location into which Jalopy was installed. This will ensure that Ant loads the required libraries during the build. The `classname` attribute indicates the name of the Jalopy Ant Task plug-in, mapping it to the element name `jalopy`.

With the new task defined, it is now possible to use it within any Ant target in the build file. Here is the `formatSource` target that uses it:

```
<target name="formatSource"  description="Format the Source, Luke!">
  <jalopy>
    <fileset dir="{java.dir}">
      <include name="**/*.java" />
    </fileset>
  </jalopy>
</target>
```

While the Jalopy task does accept several parameters, none are used here, as the defaults work well enough for the majority of cases. Refer to the Jalopy Ant plug-in documentation for more information about these parameters.

The only new piece of markup here is the `fileset` element. This is a standard Ant element that indicates a set of files that will be processed by some task. In this example, these are the files that Jalopy will be reformat. The `dir` attribute indicates the base directory containing the input files, and the `include` element operates as before to define a wildcard for matching file names.

Apply Jalopy to all of the Java source, allowing the enforcement of a coding style to all source code, not just that which was automatically generated. By default this coding formatting style is that defined by Sun, but it can be altered using the Jalopy preferences tool. Again refer to the Jalopy documentation for more information.

Invoking the `formatSource` target will reformat the source. Rechecking any of these files will demonstrate that they are now much more readable. This is one

example of where XSLT falls short of providing the fine-grain control we require when generating code, but in this case, tools like Ant and Jalopy can easily plug the gap.

## Tying it together

Having defined separate targets that allow the source code to be regenerated and reformatted at will, the remaining step is to tie these into the final compilation step. This can be efficiently achieved using Ant's dependency system as shown in the definition of the build target below:

```
<target name="build"
  depends="generateCode, formatSource"
  description="Compile the application">
  <javac srcdir="${java.dir}"
    destdir="${bin.dir}">
  </javac>
</target>
```

The build target contains a single task, `javac`, which simply invokes the Java compiler on all files in the `java` directory, instructing it to place the compiled classes in the project's `bin` directory.

The `depends` attribute on the `target` element indicates that before this target can be carried out, both the `generateCode` and `formatSource` targets must be invoked in sequence.

The end result is that simply invoking the build target will ensure that the source code will be regenerated as necessary, and all code will be correctly formatted before it is finally compiled. Thus, the code generation has become a seamless aspect of the overall build process.

## Section recap

You can easily adapt the techniques introduced in this section to give as much fine-grain control over the code generation process as is required. For example, you can apply multiple stylesheets to either the same file or separate groups of files. This would allow different code generation processes to be carried out within a single build. A single source file might even be transformed multiple times to produce different outputs.

The Bean Markup vocabulary is amenable to this kind of multiple transformation. As with Java code, it can be transformed into appropriate SQL statements to create database tables capable of storing the bean data, as well as JDBC code capable of querying those tables to instantiate or serialize bean objects. You can also generate

test harnesses to ensure that the generated code performs as expected.

The next section moves on from generating Java code to exploring how you can use XSLT stylesheets to generate other XSLT stylesheets, again using an Ant-based framework.

---

## Section 6. Generating XSLT using XSLT

### Generating XSLT using XSLT overview

This section explores the use of XSLT to generate other XSLT stylesheets. Known as **meta-stylesheets**, this powerful technique allows for the automated creation of sophisticated stylesheets. There are two specific circumstances where the technique is best exploited.

The first of these occurs when the transform generates an HTML document. Generally the stylesheet must contain the entire structure of the output document, interspersed with XSLT markup. This makes changes to the presentation difficult to achieve. Factoring out the HTML markup into a separate document and using a meta-stylesheet to regenerate the original is a simple way to remove this limitation.

A second use for this technique is when there are a related family of stylesheets, basically variations of one another, but where those variations cannot be accommodated by using stylesheet parameters alone. Rather than managing multiple files directly, introducing another level of transformation allows the common structures to be collated into a single file, making it easier to change. The variations can be described by a little language that is then transformed to regenerate the originals.

The examples in this section follow the latter approach by introducing the Element Filter vocabulary, a little language for describing filters that can be applied to XML elements during a transform. A meta-stylesheet is then described that will generate a stylesheet that's capable of applying those filtering instructions.

### The Element Filter language

The DTD below describes the Element Filter language. The operations that the language encompasses include renaming elements, removing them and optionally their children, and adding additional wrapper elements.

```

<!ELEMENT filters (element)+>
<!ELEMENT element (remove|rename|wrap)>
<!ATTLIST element
  name  NMTOKEN  #REQUIRED
  ns    CDATA    #IMPLIED>
<!ELEMENT remove EMPTY>
<!ATTLIST remove
  keep-children (true|false) #IMPLIED>
<!ELEMENT rename  EMPTY>
<!ATTLIST rename
  to    NMTOKEN  #REQUIRED
  ns    CDATA    #IMPLIED>
<!ELEMENT wrap    EMPTY>
<!ATTLIST wrap
  with  NMTOKEN  #REQUIRED
  ns    CDATA    #IMPLIED>

```

The most important tags are the `element` tags. These identify the elements to which the filters should be applied. The optional `ns` attribute indicates the namespace of the element, and is combined with the required `name` attribute to create a fully qualified namespace name.

The `remove`, `rename`, and `wrap` elements each describe a different filter operation. In this limited example, only one kind of filter can be applied to each named element.

The `remove` element indicates that the named element is to be removed from the output along with all of its child elements. However, if the `keep-children` attribute has a value of `true`, then only the named element is removed; its children will be copied into the output. For example, the following indicates that the `title` element should be removed, but all of its contents retained:

```

<element name="title">
  <remove keep-children="true"/>
</element>

```

The `rename` element specifies a new name for the matched element. The name is specified by the `to` attribute. If the new element has a different namespace, then the `ns` attribute can additionally specify it. The following example indicates that an `author` element should be replaced with an alternative `creator` element from a specified namespace:

```

<element name="author">
  <rename to="creator"
    ns="http://www.example.com/ns/document"/>
</element>

```

The last filter indicates that the matched element should become a child of another new element. The `wrap` element declares the name of the new element using its `with` attribute. The following example would wrap a `cite` element with a `blockquote` element:

```
<element name="cite">
  <wrap with="blockquote"/>
</element>
```

You can achieve all of this functionality through XSLT-based manipulations, as the next panels will illustrate.

## Implementing the filters with XSLT -- element matching

Before looking at the meta-style sheet, it is useful to review how you can map the Element Filter language onto XSLT. This will illustrate what markup the meta-style sheet needs to generate in order to create a style sheet that's capable of applying the rules described in an Element Filter document.

The basic style sheet will be an identity transformation -- a style sheet that simply copies its input to its output. This is achieved using the template shown below, which matches all elements and attributes and uses the `xsl:copy` element to copy them into the result:

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

You can find a complete example of an identity style sheet in the `meta/xslt` directory in the examples. See [Tools](#) for instructions on how to download the examples.

While not useful in isolation, the identity transform is a useful building block for constructing style sheets that manipulate only a small portion of a document, leaving the rest unaltered. This is exactly what the Element Filter language requires. Any element named in an `element` tag must be explicitly matched and processed according to one of the filter rules. Everything else remains unchanged.

Matching an element is straightforward. You can simply extend the identity style sheet to introduce a new XSLT template matching -- for instance, an `author` element:

```
<xsl:template match="author">
  <!-- filter implementation here -->
</xsl:template>
```

The Element Filter language also allows an element to be matched by both its name and its namespace identifier. Because the language doesn't include a means to specify a namespace prefix, correctly matching the element requires a template of

the following format:

```
<xsl:template
  match="*[local-name()='author' and namespace-uri() =
                                     'http://www.example.com/ns/document']">
  <!-- filter implementation here -->
</xsl:template>
```

The `match` attribute in this example matches all elements, but limits this to those elements whose local name is `author`, and whose namespace identifier is `http://www.example.com/ns/document`. These values are checked using the XPath `local-name` and `namespace-uri` functions.

These initial examples illustrate the basic XSLT templates that your meta-stylesheet must generate to match the elements that should be specially processed. The first format will be used when only an element name is declared; the second format when both a name and a namespace identifier are provided.

## Implementing the filters with XSLT -- the functionality

Now that element matching has been covered, the next logical step is to illustrate how you can implement some of the Element Filter instructions using XSLT.

The simplest to implement is the `remove` functionality. If an element is to be removed from the result, then creating an empty template that matches the element will achieve this. Alternatively, if the child nodes are to be retained, then the template can instruct the processor to continue applying templates. The original element will be removed, but the identity template will copy its children into the output, unless of course they match other templates in the stylesheet. The code below illustrates these examples:

```
<!-- removes author and all children -->
<xsl:template name="author"/>

<!-- removes title, but retains children -->
<xsl:template name="title">
  <xsl:apply-templates/>
</xsl:template>
```

To rename an element, a new element must be dynamically created. The original elements, attributes, and child nodes are then copied over so that they are associated with the new element. The following template illustrates how you can achieve this using the `xsl:element` tag, which allows for the dynamic creation of an element:

```
<!-- rename author to creator -->
```

```
<xsl:template name="author">
  <xsl:element name="creator">
    <xsl:copy-of select="@*" />
  <xsl:apply-templates/>
</xsl:element>
</xsl:template>
```

You can similarly handle wrapping one element in another, as the next example demonstrates. The slight difference is that the original element must be copied over intact -- not only its attributes and children. The following template also illustrates how to dynamically create an element in a new namespace using the `namespace` attribute on the `xsl:element` tag.

```
<!-- wrap cite in blockquote -->
<xsl:template name="cite">
  <xsl:element name="blockquote"
    namespace="http://www.example.com/ns/document">
    <xsl:copy-of select="." />
  </xsl:element>
</xsl:template>
```

These templates collectively implement all of the functionality described in the Element Filter language. While it is possible to hand craft a stylesheet that will carry out the desired transformations for any given set of elements, introducing a meta-stylesheet into the process makes it possible to generate the appropriate templates automatically. Before reviewing the functionality of this stylesheet, I must first introduce the concept of **namespace aliasing**.

## Namespace aliasing

When an XSLT engine processes a stylesheet, it looks for elements bound to the XSLT namespace `http://www.w3.org/1999/XSL/Transform` and interprets them as transformation instructions. As with all namespace-aware software, the actual prefix used is irrelevant. It is simply convention that `xsl` is used as the XSLT namespace prefix.

This presents a problem for meta-stylesheets, which include not only XSLT elements that guide the transformation but also XSLT elements that are to form part of the output. If the XSLT processor interprets all of these as instructions, then the transformation will fail to produce the expected output and will, in all likelihood, generate errors.

The XSLT specification includes a mechanism that's designed to work around this limitation. Known as **namespace aliasing**, it allows a prefix to be bound to one namespace within the stylesheet, but be automatically bound to another alternative namespace when the results are produced. The following XSLT fragment indicates how to invoke this aliasing from within a stylesheet:

```
<xsl:namespace-alias stylesheet-prefix="axsl" result-prefix="xsl"/>
```

The `xsl:namespace-alias` element has two required attributes that collectively indicate that the namespace identifier bound to one prefix (specified by `stylesheet-prefix`) is an alias for the namespace identifier bound to another prefix (specified by `result-prefix`). In other words, in the example above, when the results are generated the `axsl` prefix will be bound to the same namespace as the `xsl` prefix. Both prefixes must already be bound to a namespace, although the actual namespace identifier for the `axsl` prefix is irrelevant.

Using this functionality, it is possible to write a stylesheet that contains XSLT elements bound to an aliased namespace, thereby avoiding the XSLT processor interpreting them as transformation instructions. However, the real XSLT namespace will be substituted for the alias in the results. The following fragment illustrates this. The convention is to use `axsl` (Aliased XSLT) as the prefix, and `http://www.w3.org/1999/XSL/TransformAlias` as the namespace identifier. But neither is mandatory.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:axsl="http://www.w3.org/1999/XSL/TransformAlias">

  <xsl:namespace-alias
    stylesheet-prefix="axsl"
    result-prefix="xsl"/>

  <!-- templates... -->
</xsl:stylesheet>
```

This forms the basic structure of a meta-stylesheet. Any additional functionality is simply a matter of producing the desired elements just as you would with any other XSLT transformation. It is the namespace aliasing that magically turns the results into a processable stylesheet.

The `meta/xslt` directory of the examples (`metastylesheet-tmpl.xml`) includes a simple reusable template for building additional meta-stylesheets that are based around an identity transformation (see [Tools](#) for how to download the examples).

## The meta-stylesheet

Once you know how to create a basic meta-stylesheet and how the Element Filter language maps onto XSLT functionality, all that remains is to create the complete example. For this example, assume that elements with the `axsl` prefix will be namespace aliased to become XSLT elements.

The first step is to create a template that matches the `element` tags in an Element

Filter document using the element matching templates introduced in [Implementing the filters with XSLT -- element matching](#). The following template, taken from the meta/xslt/makeFilter.xsl stylesheet in the examples, does just that:

```
<xsl:template match="element">
  <xsl:variable name="matchPattern">
    <xsl:choose>
      <xsl:when test="@ns">
        *[local-name()='<xsl:value-of select="@name"/>'
          and namespace-uri()='<xsl:value-of select="@ns"/>']
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="@name"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <axsl:template match="{normalize-space($matchPattern)}">
    <xsl:apply-templates/>
  </axsl:template>
</xsl:template>
```

There are several things to notice here. First, a variable, `matchPattern`, is declared and initialized to either the *local name plus namespace identifier* form of match pattern or simply the name of the element. The selection of the different forms is achieved based on whether the `element` tag has an `ns` attribute.

Having identified the correct match pattern, an XSLT template is created using the namespace aliased prefix. The value of the `match` attribute becomes the value of the `matchPattern` variable with leading and trailing spaces removed. Notice that the XPath function cannot be used directly but must use the XSLT attribute value template syntax, which wraps expressions in curly braces. This is because the XSLT processor does not recognize the result elements (those with the `axsl` prefix) as being XSLT elements because they are currently aliased to another namespace.

Creating the contents of the (so far aliased) XSLT template is straightforward. The rest of the meta-stylesheet contains templates that match the `remove`, `rename`, and `wrap` elements to generate the transformations described in [Implementing the filters with XSLT -- the functionality](#). Consult the meta/xslt/makeFilter.xsl transform for full details. However, as an illustration, here is how the `wrap` element is processed:

```
<xsl:template match="wrap">
  <axsl:element name="{@with}">
    <xsl:if test="@ns">
      <xsl:attribute name="namespace"><xsl:value-of select="@ns"/></xsl:attribute>
    </xsl:if>
    <axsl:copy-of select="."/>
  </axsl:element>
</xsl:template>
```

First, a namespace alias, `xsl:element`, is created that will generate an element

whose name is defined in the `with` attribute of the `wrap` element. Second, if the `wrap` element has an `ns` attribute, then an attribute is dynamically added to the aliased `xsl:element` so that the new element is created with the correct namespace. Lastly, an aliased `xsl:copy-of` element is used to implement the copying of the newly-wrapped element.

The indirection involved in using aliased XSLT elements can be confusing at first, but comparing the inputs and outputs of the meta-style sheet can help isolate exactly what is happening when. To generate some output, all that remains is to integrate the transformation steps -- both the meta-style sheet transform and the generated Element Filter transform -- with your Ant build process.

## Applying the filter with Ant

A combination of Ant's `xslt` task and its dependency tracking can provide an effective means of handling the kind of multi-stage transformations required by meta-style sheets. The first step is simply to create a target, `makeFilter`, that processes an Element Filter document using the meta-style sheet:

```
<target name="makeFilter"
  description="Make the document filter">
  <xslt in="{xml.dir}/filter.xml"
    out="{xslt.dir}/filter.xsl"
    style="{xslt.dir}/makeFilter.xsl">
  </xslt>
```

Refer to [Generating the source](#) for more information about the `xslt` task. In the example shown, the Element Filter document, `filter.xml`, is transformed using the `makeFilter.xsl` style sheet to generate a new XSLT transform, `filter.xsl`. See [Tools](#) to find out how to download the complete examples that contain these files.

Having generated the new filter transformation, all that remains is to actually use this transformation to filter some real documents. To achieve this, create another task that uses the `xslt` target:

```
<target name="build"
  depends="makeFilter"
  description="Filter the documents">
  <xslt basedir="{in.dir}"
    destdir="{out.dir}"
    extension=".xml"
    style="{xslt.dir}/filter.xsl">
    <include name="**/*.xml"/>
  </xslt>
</target>
```

This target transforms all XML documents in the `in` directory using `filter.xsl`, placing the results in the `out` directory (see [Figure 2](#) below). Notice that the target is

declared to be dependent on the `makeFilter` target. This will ensure that if either the Element Filter document or the meta-style sheet is changed, then `filter.xsl` will be regenerated prior to it being used to process the XML documents.

**Figure 2. Transform results go in out directory**



Using Ant's flexible wildcards, it is quite simple to adapt this example so that it can create and apply multiple filters to the input. As the JavaBean generation example demonstrated, the Ant framework provides a great deal of flexibility when handling multi-stage processes.

---

## Section 7. Summary

### Tutorial summary

This tutorial has demonstrated how you can use XSLT, controlled by the Ant build tool, to automatically generate both Java code and XSLT transformations.

The tutorial began by introducing the basic concepts behind code generation, reviewing its benefits and the advantages of using XML and XSLT to implement little languages that can be used to drive code generation processes. Also reviewed were some of the limitations of XSLT you might encounter when applying it to code generation, with a proposal that embedding the transformation within a build framework (such as Ant) can eliminate some of these limitations.

The remainder of the tutorial covered, in detail, two separate examples. The first of these explored the use of XSLT to generate Java code using a Bean Markup language. This example introduced the use of Ant, showing how you can use its sophisticated task management to control the multi-step processes required to generate, format, and ultimately compile Java code. The Jalopy source code formatter was also introduced as a means of applying consistent formatting to Java source code, removing the need to attempt this using XSLT.

The last section of the tutorial introduced the concept of meta-style sheets -- XSLT

transforms that generate other XSLT transforms. The XSLT technique of namespace aliasing was discussed as an essential part of implementing these transformations. The examples showed how you could map an Element Filter language describing operations to be carried out on XML elements to an XSLT implementation -- first by hand, and then using the meta-stylesheet technique. Again, Ant was used as a means to tie together the multi-step process by making the main filter transformation dependent on the meta-stylesheet transformation.

You can easily extend the techniques and framework introduced in this tutorial for use in generating other kinds of source code. You can also adapt them to the processing of more complex little languages that would allow the generation of more sophisticated Java code, including complete applications as well as more powerful XSLT transformations.

To learn more about how code generation is being used in real projects, consult the [Resources](#) for pointers to more information. For example, Schematron is a little language for validating XML documents implemented using XSLT and meta-stylesheets. The Cocoon project also includes a code generator that's used to convert XML documents, known as XML Server Pages, into Java code suitable for plugging into the Cocoon framework.

## Downloads

Description	Name	Size	Download method
Code samples for this tutorial	x-codexslt_code.zip	10KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Find out more about the Ant build tool at [the project Web site](#).
- Explore the source formatting options in Jalopy through [the online documentation](#).
- See how [Cocoon](#) (March 2002) uses XSLT-driven code generation to automatically create [XML Server Pages](#) (April 2002) in these developerWorks tutorials.
- [Schematron](#) is a little language that's implemented using XSLT and meta-stylesheets. It can be used to validate XML documents.
- Michael Fitzgerald's article "[XML Pipelining with Ant](#)" provides some additional background on processing XML using Ant (January 2003).
- Robert DuCharme's [Transforming XML column](#) contains regular practical advice on using XSLT.
- Review the [JavaBeans documentation](#) on the Java technology Web site.
- Get started with the J2EE platform with the [Enterprise JavaBean fundamentals](#) tutorial (*developerWorks*, April 2003).
- Read [XSL Formatting Objects \(XSL-FO\) basics](#) and [XSL-FO advanced techniques](#), two tutorials by Doug Tidwell (*developerWorks*, February 2003) for more information on how to use XSLT to convert XML documents into formatting objects and then the Apache XML Project's FOP (Formatting Object to PDF) tool to convert those objects into PDF files.
- Find more information on the technologies covered in this tutorial at the developerWorks [XML](#) and [Java technology](#) zones.
- Find out how you can become an [IBM Certified Developer in XML and related technologies](#).

## Get products and technologies

- The [EXSLT](#) community initiative is a great place to download XSLT extensions. Uche Ogbuji's *developerWorks* article "[EXSLT by example](#)" uses practical examples to introduce and demonstrate some useful EXSLT functions (February 2003).
- [IBM WebSphere Studio](#) provides a suite of tools that automate XML development, both in Java and in other languages. It is closely integrated with the [WebSphere Application Server](#), but can also be used with other J2EE servers.

## About the author

### Leigh Dodds



Leigh Dodds is currently employed as an Engineering Manager at [Ingenta](#). He has been developing applications on the Java platform since 1997, and has spent the last four years working with XML and related technologies. Leigh is also a contributing editor to [xmlhack](#), and between February 2000 and June 2002, wrote the weekly "XML-Deviant" column for [XML.com](#). He holds a Bachelors degree in biological Science, and a Masters in computing. As being the father of a lively 18 month old (Ethan) is a full-time job in itself, Leigh currently spends his copious amount of free time investigating how to improve the speed of manufacturing of Round Tuits, which he believes will revolutionize the parenting business.