Craig Walls
Norman Richards

# XDoclet IN ACTION

MANNING

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830
Fax: (203) 661-9018
email: orders@manning.com



Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Tiffany Taylor
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-932394-05-2

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 07 06 05 04 03

# *brief contents*

# 2 Getting started with XDoclet

***This chapter covers***

- Generating a todo list
- Using XDoclet with Ant
- Core XDoclet concepts
- Code generation patterns

*The beginning of knowledge is the discovery of something we do not understand.*

—Frank Herbert (1920–1986)

XDoclet is a code generation tool that promises amazing benefits for some of the most common Java development tasks. XDoclet can help you produce applications faster and with less effort. You'll eliminate redundancies and write less code. You'll escape "deployment descriptor hell" and improve the manageability of your applications. That's a pretty tall order for a few javadoc-inspired attributes to fill. But as you'll see, XDoclet does an astonishing amount of work.

One of the difficulties in talking about XDoclet is that XDoclet is both a framework and a diverse set of code generation applications. Although the details of each application are different (EJB code generation is different than Struts code generation, both of which are different from JMX code generation), the core concepts and usages have a lot in common.

In this chapter, we'll look at the basic XDoclet framework concepts that permeate all XDoclet applications. But before we do that, let's jump right in and see XDoclet in action.

## 2.1 XDoclet in action

Every programmer knows that code is never finished. There's always another feature to add, a refactoring to apply, or a bug to fix. It's common for programmers to comment on these issues in the code to remind themselves (or other programmers) of an action that needs to be taken.

How do you keep track of those tasks to complete? Ideally, you compose a tidy todo list. XDoclet provides an extremely useful (and often overlooked) todo generator that can handle this task. This is a perfect, noncommittal opportunity to get started with XDoclet and introduce XDoclet into a project.

### 2.1.1 A common issue

Suppose you are developing a class for an application involving spoons:

```
public class Matrix {
    // TODO - need to handle the case where there is no spoon
    public void reload() {
        // ...
        Spoon spoon = getSpoon();
        // ...
    }
}
```

Ideally, the next time you visit this code, you will be able to handle the null spoon problem. But what if you don't revisit this code any time soon? How will you remember that you still have some work you want to do in this class? You could search through all of your source files looking for the text *TODO*. Unless your IDE has built-in todo list support, that is far from an ideal solution. XDoclet has a better alternative: If you mark issues on classes and method using the `@todo` tag, XDoclet can generate a todo report for your project.

### 2.1.2 Adding an XDoclet tag

The following code shows the trivial modification needed to convert your ad hoc todo item into something a bit more formal:

```
public class Matrix {
    /**  @todo need to handle the case where there is no spoon */
    public void reload() {
          // …
    }
}
```

This simple javadoc-inspired tag is all the metadata XDoclet needs. XDoclet will use the information in the tag, including its relationship to the class and the method in question, to produce a todo report.

### 2.1.3 Integrating with Ant

To generate your todo list, you need to make sure XDoclet is properly installed. If you don't have XDoclet installed yet, see the instructions in appendix A. Make sure you have an `init` target that is modeled after the one in the appendix. At the bare minimum, your `init` target should contain the following task definition for the `<documentdoclet>` task:

```
<taskdef name="documentdoclet"
          classname="xdoclet.modules.doc.DocumentDocletTask"
          classpathref="xdoclet.lib.path" />
```

The `<documentdoclet>` task is one of the core XDoclet applications. You'll see quite a few others as you go on.

Now you can add a todo target to the Ant build file that invokes the todo list task:

```
<target name="todo"
        depends="init">
    <documentdoclet destdir="todo">
        <fileset dir="${dir.src}">
            <include name="**/*.java" />
```

**Start <documentdoclet> task context; output goes to todo directory**

```
        </fileset>

        <info />    <-- <info> subtask does all the work
    </documentdoclet>
</target>
```

The code uses the `<info>` subtask to look for `todo` attributes in all your source files. HTML summaries are placed in the `todo` directory.

---

**NOTE** XDoclet makes extensive use of Ant. Although you don't need to be an Ant master to use XDoclet, you need at least a basic working knowledge of Ant. We'll point out any tricky usages of Ant as we go, but if you need a refresher, we recommend *Java Development with Ant*.[1]

---

### 2.1.4 Generating a professional-looking todo list

The todo summary pages that XDoclet creates are very professional looking. Using the summary pages, you can see at a glance which packages and classes have outstanding todo items (along with count information). Todo items can be associated with methods, classes, and fields, and the report clearly distinguishes between them. Class-level todo items are marked with the word *class*, and method-level todo items are marked with an *M* and the signature of the related method. Constructor and field-related todo items are similarly marked.

Figure 2.1 shows a todo list generated by XDoclet. This todo list is for the 1.2b2 release of XDoclet (generating a todo list for the single class we just looked at wouldn't make a terribly compelling example) and is distributed with XDoclet. XDoclet's todo list is distributed in the `doc/todo` directory. Not only is todo information useful for the XDoclet development team, but it can also play a role as part of the documentation of the system.

This may not seem like a huge task at first, but consider for a moment that the only effort expended here was to add machine-readable `@todo` tags to the source files instead of free-form comments that can only be recognized by a human reader. The output produced is easier to read and more useful to programmers than that produced by many commercial issue-tracking systems.

---

[1] Erik Hatcher and Steve Loughran, *Java Development with Ant* (Greenwich, CT: Manning, 2002).


**Figure 2.1 The todo list for the XDoclet source tree**

## 2.2 Tasks and subtasks

XDoclet does much more than produce variations on the javadoc inline documentation theme. XDoclet is best known as a tool for generating Enterprise JavaBean (EJB) interfaces and deployment descriptors. However, XDoclet is actually a full-featured, attribute-oriented code generation framework. J2EE code generation is the showcase application of XDoclet, but as you will see throughout this book, XDoclet's usefulness extends far beyond J2EE and project documentation.

### 2.2.1 XDoclet tasks

Although we'll talk about using XDoclet to generate code, it's more correct to say that you use a specific XDoclet task, such as `<ejbdoclet>`, to generate code. Each XDoclet task focuses on a single domain and provides a spectrum of code generation options within that domain.

**DEFINITION** *Tasks* are the high-level code generation applications available in XDoclet.

Table 2.1 shows the seven core XDoclet tasks and the space of code generation tasks they address. The core tasks ship with XDoclet and are available for use out of the box.

**Table 2.1 The seven core XDoclet tasks and their scopes**

| Task | Scope |
|---|---|
| `<ejbdoclet>` | EJBs—enterprise beans, utility classes, deployment descriptors |
| `<webdoclet>` | Web development—servlets, filters, taglibs, web frameworks |
| `<hibernatedoclet>` | Hibernate persistence—configuration, Mbeans |
| `<jdodoclet>` | JDO—metadata, vendor configuration |
| `<jmxdoclet>` | JMX—MBean interfaces, mlets, configuration files |
| `<doclet>` | Custom templates for ad hoc code generation |
| `<documentdoclet>` | Project documentation like the todo list |

`<ejbdoclet>` is by far the most developed XDoclet task and the most widely used. Many projects using XDoclet only make use of EJB code generation; however, EJBs and web development typically go hand in hand, and `<webdoclet>` is the next most popular XDoclet task. Of course, it is possible (and highly desirable) for a single development project to use more than one XDoclet task, but each XDoclet task exists separately from the others and doesn't interact directly with them.

### 2.2.2 XDoclet subtasks

One aspect of XDoclet sets it apart from single-purpose code generation tools: XDoclet tasks are collections of fine-grained subtasks that perform very narrow types of code generation within the domain of the task.

**DEFINITION** *Subtasks* are the single-purpose code generation procedures provided by a task.

Tasks provide a context and grouping to manage related subtasks (see figure 2.2). The subtasks are responsible for performing the code generation. It's not uncommon for a task to invoke multiple subtasks to perform various aspects of a larger code generation task. For example, when working with EJBs, you might want to generate a home interface for each bean, a remote interface for each

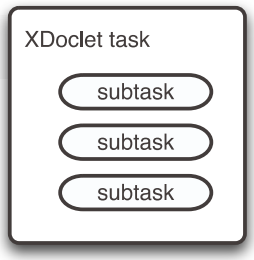


**Figure 2.2**
**Tasks are groups of related code generation subtasks that work together.**

bean, and the `ejb-jar.xml` deployment descriptor. These are three separate code generation subtasks in the context of the `<ejbdoclet>` task.

Subtasks can be invoked in any combination and in any order to provide exactly the level of code generation needed on any project. Subtasks within a single XDoclet task often share functionality and use the same XDoclet tags in the source files. This means that once you get started with a task, it's usually easy to leverage your work by using related subtasks.

### *Subtask interaction*

Let's get an idea of how subtasks relate by looking at the subtasks for `<ejbdoclet>`. Suppose you're creating a CMP (container managed persistence) entity bean. You'll want to use several `<ejbdoclet>` subtasks:

- `<deploymentdescriptor>`—To generate the `ejb-jar.xml` deployment descriptor
- `<localhomeinterface>`—To generate the local home interface
- `<localinterface>`—To generate the local interface

In the process of doing so, you'll mark some basic information about your entity bean's CMP fields. When you deploy your bean, you may want to provide basic mapping information that specifies table and column names in a relational database using vendor-specific deployment descriptors. XDoclet lets you leverage your existing CMP XDoclet attributes and add relational mapping attributes. With just a few additions, you can now invoke, for example, both the `<jboss>` subtask and the `<weblogic>` subtask to generate deployment descriptors for those application servers. XDoclet provides varying levels of support for almost a dozen application servers, multiplying your initial effort greatly.

But that's only the tip of the iceberg. You could also use the `<entitycmp>` subtask to generate a subclass of your bean that provides a concrete implementation of the entity bean interface methods you aren't interested in implementing yourself.

If you've also used the `<valueobject>` subtask to generate a value object for your bean, then the `<entitycmp>` subtask will generate methods to construct your value object for you.

Feeling dizzy yet? Unfortunately, the `<cupofcoffee>` subtask hasn't made it into XDoclet (yet), so we'll take a rest.

The point here isn't to explain all the `<ejbdoclet>` subtasks (and we've barely scratched the surface) or overwhelm you with the possibilities but to show how a task's many subtasks can work together. Once you've overcome the initial effort to get started with an XDoclet task, it pays handsomely to explore the related subtasks—the development costs are considerably cheaper and the rewards significantly higher than approaching a subtask in isolation.

## 2.3 Invoking tasks from Ant

XDoclet is married to Ant. XDoclet tasks are exposed as Ant build tasks, and there is no other supported way to invoke XDoclet. Ant is the de facto Java build tool, so this isn't much of a limitation. In fact, quite the opposite is true—the close relationship with Ant makes XDoclet a perfect fit in almost any Ant build process.

Figure 2.3 shows the XDoclet task from figure 2.2 in the context of Ant. XDoclet tasks are not merely exposed in Ant, they are actually Ant tasks. For this reason, XDoclet is only accessible from within Ant.

### 2.3.1 Declaring tasks

XDoclet isn't distributed with Ant, so you have to download and install XDoclet to be able to use it (see appendix A for installation instructions). Each XDoclet

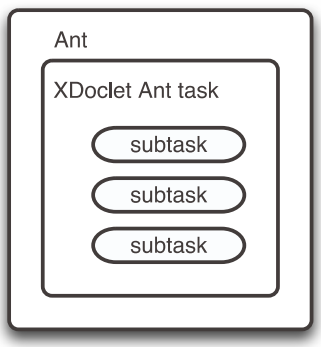


**Figure 2.3 XDoclet tasks are Ant tasks and are only accessible from within Ant.**

task you use must be declared using Ant's `<taskdef>` task. If you follow the instructions in appendix A, you should have the following task definition:

```
<taskdef name="ejbdoclet"
         classname="xdoclet.modules.ejb.EjbDocletTask"
         classpathref="xdoclet.lib.path"/>
```

If you aren't familiar with Ant, this code tells Ant to load the `<ejbdoclet>` task definition. You could name the task anything you like, but it's better to stick with the standard naming conventions to avoid confusion. The `classname` and `classpathref` attributes tell Ant where to locate the XDoclet classes that implement this task. If you want to use other XDoclet tasks, you must include `<taskdef>` declarations for those tasks as well.

It's common to wrap all the XDoclet task definitions in a single Ant target than can be added as a dependency to any targets that need to use them. You probably already have an `init` target in your build file, which is the perfect place to add your XDoclet task definitions (and if you don't, you can create one). Here's an `init` target that declares both the `<ejbdoclet>` and `<webdoclet>` tasks in addition to the `<documentdoclet>` task you used earlier:

```
<target name="init">
    <taskdef name="documentdoclet"
             classname="xdoclet.modules.doc.DocumentDocletTask"
             classpathref="xdoclet.lib.path" />
    <taskdef name="ejbdoclet"
             classname="xdoclet.modules.ejb.EjbDocletTask"
             classpathref="xdoclet.lib.path"/>
    <taskdef name="webdoclet"
             classname="xdoclet.modules.web.WebDocletTask"
             classpathref="xdoclet.lib.path"/>
</target>
```

Now that the tasks are defined, XDoclet is ready to go.

### 2.3.2 Using tasks

You can use declared tasks inside any target. Within the context of a task, you can invoke subtasks. Let's jump straight into a target that invokes the `<ejbdoclet>` task. Don't worry about the specifics of the syntax; at this point you should just note the basic concepts:

```
<target name="generateEjb" depends="init">

    <ejbdoclet destdir="${gen.src.dir}">

        <fileset dir="${src.dir}">
```

**<ejbdoclet> task; everything takes place in this context**

```
            <include name="**/*Bean.java"/>
        </fileset>

        <deploymentdescriptor                          Subtask that overrides destdir
             destdir="${ejb.deployment.dir}"/>         attribute on <ejbdoclet>


        <homeinterface />                              Subtasks that inherit
        <remoteinterface />                            context defined by
        <localinterface />                             <ejbdoclet>
        <localhomeinterface />
    </ejbdoclet>
</target>
```

Think of the task as providing a configuration context in which the subtasks operate (remember that the subtasks do the code generation work). When you invoke a subtask, you inherit the context of the task, but you are free to override those values if it makes sense. You override the `destdir` property in this example because two different types of generation are going on: the `<deploymentdescriptor>` subtask, which is a task that creates a deployment descriptor, and the various interface generation subtasks that generate Java source code. The deployment descriptor needs to be placed in a location convenient for inclusion in your EJB JAR file, but the generated code should be placed where you can invoke the Java compiler. Even though the subtasks are closely related, the nature of the tasks are different enough that you want different defaults.

The `<fileset>` attribute also applies to all the subtasks. Because it's a complex Ant type (a set of files) as opposed to a simple text or numeric value, you have to declare it as a sub-element of the task (don't confuse these configuration elements with subtasks). Of course, if you wanted to give a different input file set to any of your subtasks, you could place another `<fileset>` sub-element inside those subtasks.

Many more configuration options are available. We'll introduce the most useful ones as we cover each task and subtask in later chapters. The appendixes provide a more complete reference to the various configuration options available in each task and subtask.

## 2.4 Tagging your code with attributes

Reusable code generation systems need input to produce interesting output. A parser generator needs a description of the language to parse in order to generate the parser. A business object code generator needs a domain model to know

which business objects to generate. XDoclet takes as its input Java source files and generates related classes or deployment/configuration files.

However, the information needed to generate code isn't always available in the original source files. Consider a servlet-based application where you want to generate a `web.xml` file. The servlet source file only contains the class name and the appropriate servlet interface methods. It doesn't contain any information about how to map the servlet to a URI pattern or what initialization parameters the servlet needs. Of course, this makes sense—if the class told you those details, you would hardly need to place that information in the `web.xml` file in the first place.

XDoclet doesn't know this information either. Fortunately, the solution is simple. If the information isn't available in the source files, you can put it there in the form of XDoclet attributes. XDoclet parses the source files, extracts the attributes, and passes them along to the templates to provide the critical data needed to generate code.

### 2.4.1 The anatomy of an attribute

XDoclet attributes are nothing more than javadoc extensions. They look and work just like javadoc attributes and can be placed inside javadoc documentation comments. Documentation comments are simple; they begin with `/**` and end with `*/`. Here's a quick example:

```
/**
 * This is a javadoc comment. Comments can span multiple
 * lines and each line can start with an asterisk ('*'), which
 * will be ignored in parsing.
 */
```

Any text between the comment markers is a javadoc comment and will be visible to XDoclet. Comment blocks are always associated with an entity in the Java source file and are placed immediately before the element they are associated with; comment blocks not immediately preceding a commentable entity are ignored. Classes (and interfaces) can have comment blocks, as can methods and fields:

```
/**
 *  a class comment
 */
public class SomeClass
{
    /** a field comment */
    private int id;
```

```
    /**
     * constructors can have comments
     */
    public SomeClass() {
        // ...
    }

    /**
     *  and so can methods
     */
    public int getId() {
        return id;
    }
}
```

Comment blocks have two sections: the description section and the tag section. The tag section begins when the first javadoc `@tag` is seen. Javadoc tags have two components: the tag name and the tag description. The tag description is optional and can span multiple lines:

```
/**
 *  This is the description section.
 *  @tag1 the tag section begins here
 *  @tag2
 *  @tag3 the previous tag had no tag description. This
 *  one has a multi-line description
 */
```

XDoclet adds a layer of expressiveness to javadoc tags by adding parameterized tags. With XDoclet, you can add *name*="*value*" parameters in the description portion of the javadoc tag. This minor change drastically increases the expressiveness of javadoc tags and makes it much easier to expose rich metadata. The following is an entity bean method with XDoclet attributes:

```
/**
   * @ejb.interface-method
   * @ejb.relation
   *     name="blog-entries"
   *     role-name="blog-has-entries"
   * @ejb.value-object
   *     compose="com.xdocletbook.blog.value.EntryValue"
   *     compose-name="Entry"
   *     members="com.xdocletbook.blog.interfaces.EntryLocal"
   *     members-name="Entries"
   *     relation="external"
   *     type="java.util.Set"
   */
  public abstract Set getEntries();
```

Parameterized tags allow for logical groupings of related attributes. You can express meta-information about the class that is rich enough to generate code from, yet readable enough that a programmer who is familiar with the tags can quickly understand how the class is used. (If it's not clear what is being expressed here, don't worry; you'll see what these specific tags mean when we look at EJBs in chapter 4.)

Note that the tag names here start with `ejb`. XDoclet follows the convention of tag names of the form *`namespace.tagname`*. This ensures that XDoclet tags are not confused with javadoc tags, and that tags related to one task do not conflict with tags from another task.

---

**NOTE** For a more technical review of javadoc comments, see chapter 18 of the Java Language Specification (http://java.sun.com/docs/books/jls/).

## 2.5 Code generation patterns

XDoclet is a template-based code generation engine. At a high level, output files are generated by evaluating templates in varying contexts. The expressiveness of the templates and the nature of the contexts in which they are evaluated determine exactly what XDoclet can and can't generate. If you are evaluating the XDoclet platform, it's very important to understand these concepts. Otherwise you may miss out on some of the power of XDoclet or, worse, be caught by surprise by its limitations.

XDoclet runs in the context of the Ant build tool. XDoclet provides Ant tasks and subtasks that you use to interact with the XDoclet engine. Tasks act as containers for the subtasks, which are responsible for performing the code generation. Subtasks perform their work by invoking templates. A template provides a cookie-cutter image of the code you will generate. The template can draw on the input source files, including the metadata attributes in those source files, to provide data to drive the template. Additionally, templates may provide merge points that allow the user to plug in template fragments (merge files) to further customize the generated code.

Figure 2.4 shows the flow of information in an XDoclet task. We'll look more closely at each of these components in the following sections.

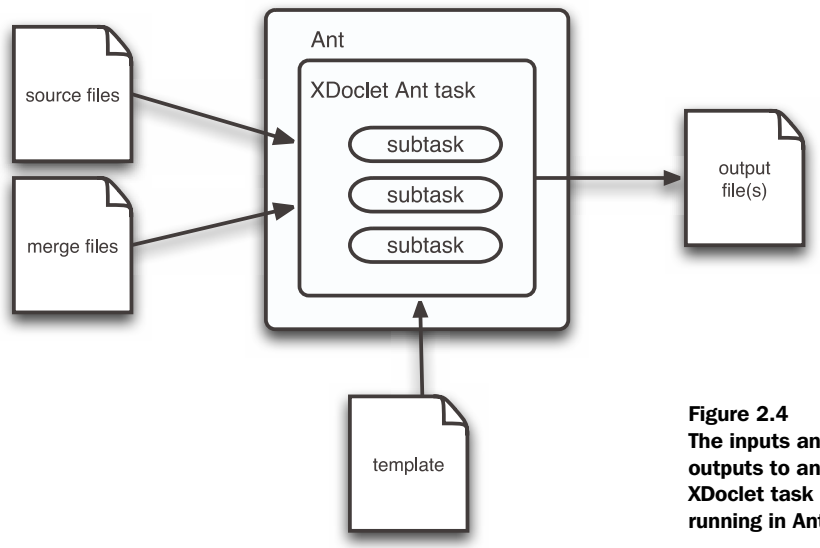


**Figure 2.4 The inputs and outputs to an XDoclet task running in Ant.**

### 2.5.1 Template basics

XDoclet generates code using code templates. A *template* is a prototypical version of the file you want to generate. The template is marked up using XML tags that instruct the template engine how to adjust the generated code based on the input classes and their metadata.

---

**DEFINITION** *Templates* are generic cookie-cutter images of code or descriptors to be generated. When the template is evaluated, the specific details are filled in.

---

Templates are always applied in some context. A template may be applied in the context of one class (*transform generation*), or it may be applied in a global context (*aggregate generation*). Understanding the difference is critical to understanding the types of tasks to which XDoclet can be applied.

When you use XDoclet to generate deployment descriptors, you are using aggregate generation. A deployment descriptor isn't related to any one class; instead, it aggregates information about multiple classes into one output file. Figure 2.5 shows the how aggregate generation looks. In this model, the template is evaluated once and generates one output file, regardless of how many input files you have.

In transform generation, the template is evaluated once for each source file, with that input class as the context. The template generates one output file for each input file (see figure 2.6).

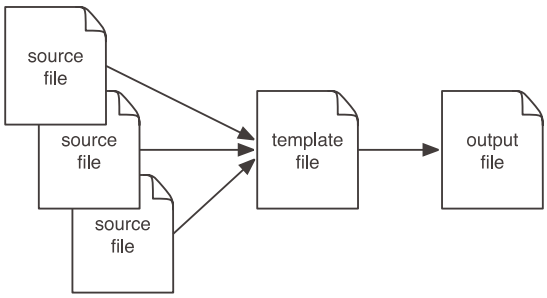


**Figure 2.5 Aggregate generation**

Generating local and remote interfaces for an EJB is a good example of transform generation. It's obvious that the interface has a one-to-one correspondence with the bean class. You transform the bean into its interface using information about the class (its methods, fields, interfaces, and so on) and its XDoclet attributes. There's no other information you need.

It might seem backward to generate the interface from the implementation. If you were writing the code by hand, you would likely write the interface first and develop your implementation class based on that interface. XDoclet doesn't work well in this model because it can't generate the business logic to fill in the methods declared in the interface. If the business logic could be described using XDoclet attributes or could be derived from naming conventions (JavaBean accessors such as `setName` and `getName`, for example), then it would be possible to generate code from the interface. However, in general, it isn't realistic to work in this direction. It's much simpler to provide the implementation and describe how the interface methods relate to it.[2]

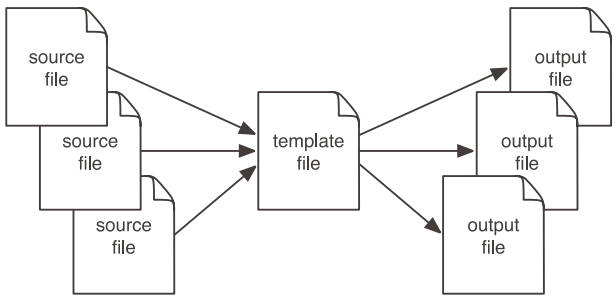


**Figure 2.6 Transform generation**

[2] A notable exception is mock objects, where you generate a mock object implementation class from an interface. We'll look at mock objects in chapter 10.

The key difference between aggregate and transform generation is the context information. Even in a code generation task that involves generating only a single Java file as output, aggregate generation is not normally used because the context can provide vital information such as the package in which you should generate the class and the name of the class. Without context information, these details must be configured separately.

### 2.5.2 Template tags

We've gone into quite a bit of depth about templates without showing you what one looks like. Template files are like JSP files. They contain text and XML tags, which are evaluated to produce text that is copied directly to the output file. Only XML tags that are in a namespace beginning with `XDt` are XDoclet template tags; other XML tags are not treated with any significance by XDoclet. The following fragment is typical of XDoclet templates:

```
public class
    <XDtClass:classOf><XDtEjbFacade:remoteFacadeClass
      /></XDtClass:classOf>
    extends Observable
{
    static <XDtClass:classOf><XDtEjbFacade:remoteFacadeClass
         /></XDtClass:classOf> _instance = null;
    public static
        <XDtClass:classOf><XDtEjbFacade:remoteFacadeClass
          /></XDtClass:classOf> getInstance() {
      if (_instance == null) {
          _instance = new
                <XDtClass:classOf><XDtEjbFacade:remoteFacadeClass
                /></XDtClass:classOf>();
        }
      return _instance;
  }
}
```

Looking at the template, you can see that it generates a basic class definition with a static field `instance` and a static method `getInstance` that controls access to the instance. Thinking about Java syntax, you can easily deduce that the XDoclet template tags are intended to produce the name of the class you are generating, although it probably isn't obvious how the tags work.

Even if you'll never write a template, it's important to understand how templates are evaluated. Sooner or later, you'll invoke an XDoclet task that fails or that doesn't produce the output you expect it to, and the fastest course of action may be to look at the template to figure out what's wrong.

Let's look at the static field declaration:

```
static <XDtClass:classOf><XDtEjbFacade:remoteFacadeClass
              /></XDtClass:classOf> _instance = null;
```

To XDoclet, this template fragment looks much simpler:

```
static <tag /> _instance = null;
```

XDoclet evaluates `tag` and places the output of the tag, if any, back into the document. Some tags perform a computation and place the output back into the stream. These tags are called *content tags*, because they produce content.

The other type of tag is the *body tag*. Body tags have text between their begin and end tags. What makes a body tag particularly interesting is that this text can itself be a template fragment, which the enclosing tag can evaluate. In the case of the `XDtClass:classOf` tag we just looked at, its body is a template that consists of a single tag:

```
<XDtEjbFacade:remoteFacadeClass />
```

The `classOf` tag evaluates this template, which happens to produce a fully qualified class name as its output, and chops off the package name, leaving only the class name. Body tags don't necessarily have to evaluate their bodies—they can choose whether to do so based on external conditions they check (they might check whether you're generating an interface instead of a class, for example). These are *conditional tags*. Other body tags provide iterator type functionality, evaluating their body multiple times. An example might be a tag that evaluates its body once for each method in a class.

XDoclet tags provide high-level code generation functions, but they can be somewhat inflexible and less expressive than you want. Rather than trying to provide a general-purpose template engine, XDoclet instead has focused on creating an extensible template engine. It's easy to write your own tags using the full expressiveness and capabilities of the Java platform. You'll see how to do this in chapter 11.

## 2.6 Customizing through merging

Code generation systems are often rejected because they are perceived as producing rigid, inflexible code. Most code generation systems don't let you edit the generated code; so, if the system isn't flexible enough, your best bet for extensibility is to use inheritance to extend the generated classes or to apply common design patterns like Proxy and Adaptor to achieve the desired behavior.

These are all heavy alternatives to generating the code you want. Given the WYGIWYG (what you generate is what you get) nature of code generation and the high cost of working with code that doesn't function the way you want, it's highly desirable for a code generation system to be customizable enough that you can generate the best code possible up front.

XDoclet offers customization through *merge points*—points in a template file where the template designer allows you to insert code at runtime. Sometimes a merge point covers an entire region of the generation template, allowing you to not only add to the template, but also fundamentally change what is generated. Merge points are a cooperative effort between the creator of the code generation task and the user. If the creator of the task doesn't give you adequate merge points, you might not be able to customize the code the way you like. However, the core XDoclet tasks have very well-defined merge points.

---

**DEFINITION** *Merge points* are predefined extension points in templates that allow you to customize the generated code at runtime.

---

Let's look at a few quick examples from the XDoclet source tree. At the end of the template for generating primary keys for entity beans, the code defines the following merge point:

```
<XDtMerge:merge file="entitypk-custom.xdt"></XDtMerge:merge>
```

If you create a merge file named `entitypk-custom.xdt` in your `merge` directory, then the contents of that template will be included at the merge point. Your customizations have all the power of top-level templates and can perform any computation a template can (including defining custom tags and defining their own merge points).

In this case, the merge point uses the same file for each class context the template is run in. It's also possible for the merge file to use a unique merge file for every class context. This is useful when you want to customize only a small number of classes or when you don't want to write a template and would instead prefer to write the customization for each target. Regardless of the motivation, per-class merge points are easy to recognize: They have the standard XDoclet per-class marker, `{0}`, in their name. Here is a larger example for generating security role references in `ejb-jar.xml`:

```
<XDtMerge:merge file="ejb-sec-rolerefs-{0}.xml">
 <XDtClass:forAllClassTags tagName="ejb:security-role-ref">
```

```
        <security-role-ref>
            <role-name><XDtClass:classTagValue
                         tagName="ejb:security-roleref"
                         paramName="role-name"/></role-name>
            <role-link><XDtClass:classTagValue
                        tagName="ejb:security-roleref"
                        paramName="role-link"/></role-link>
        </security-role-ref>
    </XDtClass:forAllClassTags>
</XDtMerge:merge>
```

This part of the template is executed inside a tag that iterates over the beans in the project. For each bean, XDoclet looks for a merge file specific to that bean by substituting the symbolic name of the current class context for the `{0}` in the merge file name. For example, for the `BlogFacadeBean` you will develop later in this book, XDoclet would try to load the merge file `ejb-sec-rolerefs-BlogFacade.xml`.

If the merge file is not present, then the template code inside the merge tag is evaluated. This means that merge points not only can provide customizable additions to a generated file, but also can define replacements for portions of a template. Not all XDoclet tasks provide replacement merge points, preferring instead to provide additive merge points only. It is up to the task creator to decide what types of merge points are appropriate for the task they are providing.

The only missing detail is how XDoclet locates your merge files. Each XDoclet task or subtask provides a `mergeDir` attribute that allows you to specify the directory where you place your merge files.

---

**NOTE** Throughout the book, we will identify important merge points. However, for more complete documentation on merge points, see appendix B. If you want to learn how to write XDoclet templates for merge points, see chapter 11.

## 2.7 The big picture

Although they are never explicitly stated, two principles run throughout the design of XDoclet:

- Every piece of information in the system should have a single authoritative source, and every use of the information should derive from that one source.
- Information about a class should be kept with that class.

Information is easier to deal with when it isn't replicated and when the distance between it and related information is minimized. It's hard to argue with this as an abstract concept, but the application to your code might be less clear.

These principles can be seen clearly in Enterprise JavaBeans. In fact, it was the messy world of EJBs that inspired the writing of EjbDoclet (the predecessor of XDoclet). To create a single EJB, multiple classes must be written. Of course, there is the bean itself, but you also need to be concerned with the home interface and the local and/or remote interfaces to the bean. In the case of an entity bean, you may have value objects to pass data to the outside world and data access objects to retrieve data from the database. If your entity bean has a `name` field, how many times is this piece of information repeated? The bean will have `get` and `set` accessor methods on it. These methods will likely be on the local/remote interfaces, or on a value object if that pattern is employed. Adding a new field could require you to touch a half dozen (or more) classes.

Using XDoclet, you can capture the information about the attribute in a single location, the bean implementation class. To expose the accessor methods in the local or remote interface, you can insert metadata in the source, along with all the other information needed to derive those classes. But you can do more than that. You can mark the attribute as persistent and have XDoclet generate the appropriate lines in the deployment descriptor. You can even specify the column name you want to map to in your database, to be placed in the application server–specific deployment descriptor. In other words, you can capture all the information about a class in the class itself and derive the related information from it.

Eliminating redundancy is important, as is keeping related information together. Think about the problem with software documentation. It can be incredibly difficult to keep the documentation in sync with the code. It's tempting to change the code and then decide to update the documentation "tomorrow." But one type of documentation has proven relatively easy to keep in sync: javadoc.

With javadoc, API documentation is kept in the same file as the code. When the documentation is sitting in the same file as the source, it's hard to not take the extra bit of time to update the documentation. This is the principal of *locality*: The closer information is, the easier it is to update. And the easier it is to update, the more likely the developer is to update it.

This principal isn't limited to documentation. Java developers face scattered systems all the time. Classes use resource bundles defined in separate files. Servlets and EJBs have deployment information in far away deployment descriptors. Keeping these information sources up to date is just plain difficult. Although

XDoclet won't magically make your entire application stay in sync, it does provide tools to eliminate a lot of the effort that would otherwise be wasted on managing application information across multiple files.

It's important to understand these two ideas because they are the problems that XDoclet solves best. If your development suffers from redundancy and excessively spread-out information, you will find that XDoclet does particularly well at solving your problems.

## 2.8 Summary

Each XDoclet code generation task is different. The tasks are configured differently, use different tags, and produce different code. However, despite these differences, they all rely on the same core concepts. Once you understand the mechanics of XDoclet code generation, jumping from generating EJB data objects to generating servlet deployment information is relatively straightforward. If you find yourself confused about how some part of XDoclet works, take a step back from the specific task you're working in and look at the core XDoclet concepts involved. Chances are, the problem will be much clearer when you remove the details of the domain you're working in.

JAVA

# XDoclet IN ACTION

Craig Walls and Norman Richards

Are you tired of repeatedly writing essentially the same Java code? XDoclet will take the burden of common development tasks off your shoulders by writing code for you. XDoclet is a metadata driven, code generation engine for Java. It generates deployment descriptors, interfaces, framework classes and other utility classes your project requires from simple JavaDoc-style comments.

*XDoclet in Action* is a complete resource for XDoclet code generation. With many short code examples and a full-scale J2EE example, the book shows you how to use XDoclet with EJBs, Servlets, JMX, and other technologies. You'll also learn how to customize XDoclet beyond its out-of-the-box capabilities to generate code specific to your application. This book shows you how to write less code, how to keep your application components in sync, and how to keep your deployment, interface, utility and other information all in one place.

## What's Inside

- Introduction to XDoclet
- Best practices and techniques
- How to customize XDoclet
- How to use XDoclet with
  - EJB
  - Servlets
  - Struts and WebWork
  - JDO
  - Hibernate
  - JMX
  - SOAP
  - MockObjects
  - JBoss and WebLogic
  - Eclipse and IDEA

**Craig Walls**, an XDoclet project committer, has been a software developer since 1994 and a Java fanatic since 1996. He lives in Dallas, Texas. **Norman Richards** has developed software for a decade and has been working with code generation techniques for much of that time. He is an avid XDoclet user and evangelist. Norman lives in Austin, Texas.

"This is the first serious XDoclet book, and the authors got it right!"

—Michael Yuan
JavaWorld Author

"XDoclet is the missing link for complex code generation ... This book will quickly teach you how to build and deploy J2EE projects with just a click."

—Daniel Brookshier, author of *JXTA: Java P2P Programming*

"I learned a lot reading it ... I was immediately filled with new ideas on how to use XDoclet. It's perfect."

—Rickard Öberg
inventor of EJBdoclet

"The patterns described make a lot of sense to me ... I've done things the hard way in the past."

—Nathan Egge
Argon Engineering

"... a fantastic job ... written clearly and effectively, with humor too.

—Erik Hatcher, co-author of *Java Development With Ant*

www.manning.com/walls

AUTHOR ONLINE Authors respond to reader questions

Ebook edition available

MANNING