6
Code Generation

Developing enterprise software requires a mixture of two mindsets: the creative and the mundane. The creative mindset calls for software engineers to apply their expertise to the task of building reliable, scaleable solutions. For the mundane, however, software engineers must resign themselves to the drudgery of the many repetitive tasks that are an all too common part of enterprise software development.

Code generation methods offer a means of delivering enterprise solutions extremely rapidly by reducing the mundane, repetitive tasks developers face. Consequently, code generators have tremendous potential for rapid development projects, and if used correctly, they can both shorten development timeframes and improve software quality and accuracy. Moreover, code generation can also make the software development process a far more enjoyable experience by freeing developers of routine tasks, enabling them to focus their attention on building better solutions.

This chapter discusses the different approaches to code generation and looks at best practices for applying code generation techniques to the development of enterprise software for the J2EE platform. Specifically, we cover a new programming paradigm, attribute-oriented programming, and explore how this and other code generation techniques can deliver higher developer productivity and more accurate software, and can help make the project team significantly more agile.

To achieve this aim, this chapter covers the following areas:

- The different types of code generators and the benefits they provide
- Apache's Velocity template engine for building custom code wizards
- Attribute-oriented programming for simplifying the development of J2EE components using XDoclet
- The issues involved in making code generation techniques part of the development process

Collectively, we'll see how the many facets of code generation can facilitate an adaptive approach to software development.

What Is Code Generation?

Code generators are software constructs that write software. It is code-writing code.

Code generation is a technique already common on development projects. The code wizards offered by most integrated development environments (IDE) are a form of template-driven code generation with which most of us are familiar. Other examples include the forward-engineering capabilities of modeling tools and the use of the Java decompiler (JAD) to reverse-engineer Java source from bytecode.

The use of code generation techniques is wide and varied, but code generators can be loosely categorized as either *passive* or *active* [Hunt, 1999].

Passive code generators.

These code generators are run once to provide the developer with a flying start. They produce code that the developer then modifies. Code wizards are an example of passive generators.

Active code generators.

Code generators that fall into the active category produce code that is continually regenerated throughout the project. The code produced is not modified by the developer and is overwritten each time the generator is run.

The distinction between active and passive code generation is a subtle one and essentially comes down to how the generated code is maintained over the course of the project. If the code generator is made part of the build process and generates fresh source each time a build is instigated, then we are using active generation. If the output from the same generator is taken by the software engineer, modified, and placed under source control, then we have passive code generation.

Both techniques can save time and effort, although the two techniques are applied quite differently. These next sections examine the uses of each type of code generation.

Passive Code Generators

The advantages of passive code generators are well known to developers thanks to the popularity of code wizards used by virtually all IDEs. Such are the benefits of passive code generators that few developers would consider using an IDE that did not number code wizards among its features.

Passive code generators are typically template driven. The template is a parameterized blueprint of the code to be produced. The generator is supplied with the necessary parameters, and

the code generator undertakes a basic parameter-substitution process on the template in order to generate the required output.

Passive code generators are simple and therefore easily written. Open source tools are available that make the task of developing passive code generators all but trivial. One such offering is *Velocity* from the Apache Software Foundation.

Velocity is a powerful Java-based template engine. Although its use is primarily aimed at the generation of dynamic content at runtime, it is also an effective tool for writing your own code generators. Full details of the Velocity template engine can be obtained from http://jakarta.apache.org/velocity.

Velocity can be used to write all manner of specialized code generators for your project, including supplementing the code wizards already provided by your favorite IDE.

Here is an example of a rudimentary code wizard written using Velocity. This simple generator accepts a series of parameters, such as class name and author, and outputs a Java file as the result.

Code Generation with Apache Velocity

Velocity is a template-driven engine, so the first task is to define a template for our class wizard. Listing 6–1 shows the basic template on which our newly generated classes are based.

Listing 6–1 Velocity Template

wizard_template.vm

```
/*
* Created by: $author
 *
 */

public class $class extends $base {

  #if ($add_ctor == true)
  public $class() {
  }
  #end

  #if ($add_main == true)
  public static void main(String[] args) {
  }
  #end

} // $class
```

The template is defined using the Velocity Template Language (VTL), a special markup language specifically designed for working with templates. The example illustrates the use of the two parts of VTL, *references* and *directives*.

VTL references are the parameterized part of the template and are substituted for actual data when the Velocity engine is run. A VTL reference is any string commencing with the `$` character. The example includes five references: `$class`, `$base`, `$author`, `$add_ctor`, and `$add_main`.

VTL directives are used for controlling the generation of output. The VTL notation specifies that directives start with the `#` character. From the template shown in Listing 6–1, VTL directives have been used to determine if either a constructor or a `main` method is required for the generated class. This is accomplished using the `#if` directive combined with the references `$add_ctor` and `$add_main`.

Table 6–1 lists the available VTL directives.

Table 6–1 VTL Directives

Directive	Description	Example

`#foreach`	Iterates through a list of objects.	`#foreach ($item in $list)`
		`  $item.Name`
		`#end`

`#set`	Assigns a value to a reference.	`#set ($myRef = $yourRef)`

`#if`
`#else`
`#elseif`
	For constructing conditional statements.	`#if ($expenditure > $income)`
		`  Save`
		`#end`

`#parse`	Renders another Velocity template.	`#parse ("mail.vm")`
`#include`	Renders the named files without parsing.	`#include ("footer.txt")`

`#macro`	Defines a new VTL directive based on existing directives known as a Velocimacro (VM).	`#macro (vmmacro $arg1 $argn)`
		`  #if ($arg1 > $argn )`
		`  Show $arg1`
		`  #end`
		`#end`

`#stop`	Stops execution of the template engine.	`#if ($condition) #stop #end`

`##`
`#+`
`+#`
	Comments.	`## Single`
		`#*`
		`Multi line`
		`*#`

With our template defined, the next task is to map actual data to the references in the template. Listing 6–2 shows the implementation for the code generator.

Listing 6–2 The ClassWizard Code Generator

```
// ClassWizard.java

import java.io.FileWriter;

import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.VelocityEngine;

public class ClassWizard {

  public static void main(String[] args) {

    // Set the name for the new class
    //
    String className = "MyNewClass";

    try {
      // Rev up the velocity engine
      //
      VelocityEngine engine = new VelocityEngine();
      engine.init();

      // Get the template from the engine
      //
      Template template =
        engine.getTemplate("wizard_template.vm");

      // Create context and add data
      //
      VelocityContext context = new VelocityContext();
      context.put("class", className);
      context.put("author", "John Smith");
      context.put("base", "MyBaseClass");
      context.put("add_ctor", new Boolean(false));
      context.put("add_main", new Boolean(true));

      // Generate output from the template
      //
      FileWriter writer = new FileWriter(className + ".java");

      template.merge(context, writer);

      writer.flush();
    } catch (Exception ex) {
      System.err.println(ex);
    }
  }

} // ClassWizard
```

The code generator initializes an instance of the Velocity engine and loads in the template. Data is mapped to each VTL reference defined in the template using a `VelocityContext`. Once all references have been bound to data, the `VelocityContext` is merged with the `Template` instance to produce the output file.

The output generated after compiling and running the code is shown in Listing 6–3.

Listing 6–3 Generated File

MyNewClass.java

```
/*
 * Created by: John Smith
 *
 */

public class MyNewClass extends MyBaseClass {

  public static void main(String[] args) {
  }

} // MyNewClass
```

So as not to obscure the example, the parameters supplied to the Velocity context were hard-wired in the code. This is less than ideal for a generic code wizard. I will leave it as an exercise for you to supply the code generator with data for binding with the template at runtime. Parameters could be supplied as command-line arguments, from a file, or from a swing-based user interface.

Benefits of Passive Code Generation

Passive code generators are a productive tool for getting the developer off to a good start. They can provide an extensive head start, and they range from generating small classes, as shown in the example, to generating entire object hierarchies.

When generating from a template that embodies software engineering best practices, the code they generate is consistently of a high quality. Where the code generator is developed inhouse, it is expected that the resulting code will always comply with company standards, assuming of course that this holds true for the template.

As a final note, not all passive code generators are code wizards. Another possible use of a passive generator is for migrating systems between platforms. One company I encountered used a code generator extensively for converting Oracle Forms applications to Java. The company initially began as part of a larger IT department, when the development team was first charged with the task of porting a large Oracle Forms application to Java. The application to be ported was extensive, and the conversion effort was expected to tie up a large team over a long timeframe. Due to the high costs of a manual approach, the company instead decided to invest its energy into the development of a code generator to automate the initial conversion effort.

The generator they produced was unable to convert every line of code. However, it was able to convert enough of the application that only a small development team was needed to fix up those areas too complex to convert automatically.

The project was completed successfully, and the development team went on to offer its services to other organizations undertaking similar conversion exercises. Thanks to the code generator developed from the previous project, the company was able to leverage this for considerable competitive advantage when competing for Oracle Forms conversion work. From a sales perspective, it could tick all the right boxes: do the job quicker, to a high standard, and at a lower cost.

Active Code Generators

Active code generators have the power to deliver extremely high productivity gains. Unlike passive generators, the active code generator does not rely upon the services of a software engineer to modify the code generated. Instead, the active code generator is able to inject code directly into the application as and when needed.

Active code generators typically comprise three different parts: metadata, a pattern template, and the code generator itself:

Metadata.

Metadata is data that describes other data, thereby giving data shape and form. Metadata is available from many sources, including databases, XML schemas, and UML models.

Pattern template.

The pattern template is used by the code generator to render the data described by the metadata in a new form.

Code generator.

The code generator binds the metadata with the pattern template to produce the desired output. The code generator can also embody code generation rules that further define the form of the generated code.

Figure 6–1 illustrates how metadata, templates, and code generator combine to produce source code.

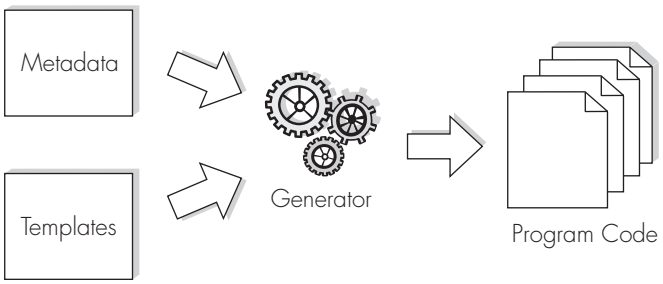

Figure 6–1 Elements of active code generation.

Benefits of Active Code Generation

The benefits of code generation are best conveyed by example. One such example comes from a project I was involved with in which a developer on the team was able to apply an active code generator very successfully.

A requirement of the system was to transmit information across a system boundary via a messaging service to the client's enterprise information system (EIS). The payload of the message was an XML document, and the project plan called for the development of a suitable XML schema to define the contents of the message for validation purposes.

The customer stipulated that the structure of the XML document must be driven by the schema of the database being used as a staging area for holding requests ready for sending. The physical design of the database was not under the control of the project but was instead designed and maintained by the customer's own enterprise data group.

One of the first tasks on the project plan called for the specification of the XML schema for the messages, using the design of the customer-supplied database as a model. On the surface, this appeared to be one of those mundane tasks common to enterprise development projects. However, instead of taking the traditional approach and handcrafting a first cut of the XML schema, the developer charged with the task opted to build a code generator.

The code generator was written in Java and used a JDBC connection to query the metadata held in the database. The metadata from the database provided all the information necessary from which to generate a schema.

The code generation approach was successful for the following reasons:

Increased speed.

The time taken to write the code generator and produce the first version of the XML schema was less than the time estimated to produce the XML schema by hand.

Improved accuracy.

The code generator removed the opportunity for human error, which is a serious flaw of the handcrafting approach. Once the code generator had been adequately tested, a repeatable, accurate result was possible each time the generator was run.

Better agility.

The underlying database was prone to change. Changes in the database design were easily accommodated by simply rerunning the code generator. By incorporating the code generator into the project's daily build process, it was possible to guarantee that database and XML schema remained in sync.

This ability to rerun the code generator to produce a new XML schema at virtually no cost made the project highly agile. The customer's database was covered contractually by change-control procedures. Thus, any change by the customer to the database that resulted in an impact on the project team was a chargeable piece of work. The database did indeed change throughout the life of the project, but the code generator made it possible to turn these changes around with minimal cost to the customer and a high degree of accuracy. For this reason, the project team established a reputation with the customer as a group capable of working very productively and delivering changes quickly.

Active Code Generators and Boilerplate Code

The ability of active code generators to turn around change very quickly and at low cost makes them a powerful tool for rapid development. The more of an application's code that can be generated, the less code has to be written by the development team. In this way, the use of active code generators increases the productivity of the language, thereby making the team agile. We code only what is needed, and the code generator and the compiler do the rest

Active code generators are especially well suited to J2EE projects, as they offer a solution to the large amounts of *boilerplate* code that must be written when developing a J2EE application. Boilerplate code is code that has to be implemented for software to comply with the demands of a particular framework. This type of code is extraneous from a business perspective because it implements no business functionality and exists solely for the purposes of the hosting platform.

For the J2EE platform, one of the worst offenders in this regard is the Enterprise JavaBean (EJB). An enterprise bean is a heavyweight component that requires a considerable amount of framework-dependent code to be written in order to be able to interact with the EJB container.

Some of the additional program artifacts associated with EJB technology include the following:

Configuration information.

J2EE applications require extensive configuration in order to operate. For an enterprise bean, it is necessary to produce both the standard configuration information for the EJB deployment descriptor and vendor-specific deployment information for the target J2EE application server.

Required interfaces.

Each enterprise bean deployed into the container must provide home and remote interfaces. The list grows if local interfaces are also to be supported on the enterprise bean.

Design pattern artifacts.

Though not strictly classed as boilerplate code, the architecture of a system employing EJB technology relies upon the application of best-practice J2EE design patterns. The use of patterns such as *Transfer Object* and *Business Delegate* results in standard classes being developed to support the enterprise bean in its environment. Although these additional classes are not mandated by the J2EE platform, good architectural design necessitates their development.

These additional program artifacts make for a code-intensive development process that is not conducive to the needs of rapid development. Thankfully, active code generation addresses this issue by generating much of the boilerplate code on our behalf.

A programming paradigm has been conceived that seeks to employ active code generation techniques to overcome the problem of developing heavyweight components for code-intensive platforms like J2EE. This paradigm is known as *attribute-oriented programming* and is discussed next.

Attribute-Oriented Programming

Active code generators can work effectively given well-defined metadata and a suitable pattern against which to generate the required code. Attribute-oriented programming leverages the benefits of active code generation by augmenting the code with special metadata tags known as *attributes*.

What Are Attributes?

Attribute-oriented programming languages enable special declarative tags, or attributes, to be embedded within the body of the code. During compilation, these attributes are extracted by the compiler and passed to an attribute provider, responsible for generating code based on the properties of the attribute and the nature of the code within which it is embedded.

The purpose of attributes is to enable the base language to be extended through the provision of custom-attribute providers. The attribute providers are essentially active code generators that inspect the code surrounding the attribute and produce further code.

Attributes Versus Preprocessor Directives

For those familiar with languages such as C and C++, attribute-oriented programming might sound very similar to *preprocessor directives*, a language construct not available in Java. Preprocessor directives are special instructions within the code that are handled by a preprocessor prior to compilation. Typically, these directives take the form of macros that are expanded within the body of the code before the source is compiled.

Preprocessor macros are a simple, but powerful, form of code generation and are used extensively within C and C++. Such is the power of the preprocessor that early C++ compilers were essentially preprocessors that expanded macros to generate straight C programming language code that was then compiled.

Though macros are powerful, if used incorrectly, they can result in some very ugly code that is extremely difficult to read and maintain. Macros can considerably alter the appearance of the language, making it very difficult for anything other than the C++ preprocessor to parse. In the past, complex macros have stopped the reverse-engineering features of some high-end modeling tools dead in their tracks. This might be why James Gosling decided to stay well clear of preprocessor directives with Java, presumably preferring instead to keep the language clean and uncluttered.

Attributes differ from preprocessor directives in that the generated code is not expanded within the body of the main code, but rather is written to a separate source file that is later compiled. This approach has the advantage that the original code is not changed behind the scenes, as is the case with macros. Instead, the newly generated file can be both viewed and debugged.

Attributes also serve to annotate the code, whereas macros are embedded into the control flow of the program. This difference makes attributed code much easier to read and understand than code that is plagued with the overzealous use of macros.

J2SE 5.0 Annotations and Attributes

Microsoft has augmented its C++ compiler to support attributes embedded within the C++ language to simplify the development of COM objects for the Microsoft platform. The .NET platform also supports the use of attributes, although these are used more to direct the behavior of a component at runtime rather than at compile time.

The release of J2SE 5.0 brings *annotations* to Java, a new language construct synonymous with the concept of attributes for declaring metadata within the body of a program. The new annotations construct that now forms part of J2SE 5.0 was defined under the Java Community Process as JSR-175, *A Metadata Facility for the Java Programming Language*.

J2SE 5.0 annotations realize the attribute concept, enabling existing program elements, such as classes, methods, and fields, to be annotated with additional metadata information. This information embedded within the code can then be used by development tools to generate additional program artifacts.

The compiler is also able to store some annotations as class-file attributes, making them accessible at runtime via a new reflection API. The existence of runtime attributes presents application frameworks with the opportunity to inject bytecode at runtime instead of generating code as part of a precompilation step.

Work is already underway on a number of different technologies that leverage the power of Java's new metadata facility. As an example of how annotations are used within the body of a program, here is a code snippet that uses annotations defined by JSR-181, *Web Services Metadata for the Java Platform*.

```
@WebService
public class BusinessService
{
  @WebMethod
  public string myBusinessMethod() {
    .
    .
    .
  }
}
```

The annotations in the code are identified by the `@` symbol, a notation commonly found inside comment blocks for use with Java's Doclet API. One of the objectives of JSR-181 is to allow the developer to convert a plain Java class into a Web Service simply by adding predefined annotation types.

From the example code snippet, the annotation types of `@WebService` and `@WebMethod` instruct a JSR-181-aware precompilation tool to generate the Web Services Definition Language (WSDL), schema, and other program artifacts necessary to expose the class as a Web Service.

The Java community has awaited the implementation of JSR-175 as part of J2SE 5.0 with some considerable excitement. During the wait, the open source community stepped in to fill the void by bringing the benefits of attributes to the Java platform using another approach.

Attributes and J2SE 5.0 Annotations

> The availability of annotations in J2SE 5.0 puts the Java community in a state of transition. During the time it takes for developers to move to J2SE 5.0, we can expect to see tools like XDoclet that rely upon metadata switching from attributes as Javadoc comments to true J2SE 5.0 annotations.
>
> The concepts behind attribute-oriented programming remain unchanged. Instead, annotations provide a formal language construct to support declarative programming paradigms.

Introducing XDoclet

XDoclet is an open source product that makes the power of attribute-oriented programming available to Java developers. As XDoclet precedes J2SE 5.0 annotations, it does so by enabling custom attributes to be specified within Java code as innocuous, compiler-friendly, Javadoc tags.

XDoclet began life as *EJBDoclet* and was the brainchild of leading open source contributor Rickard Öberg. The original EJBDoclet was built as a Javadoc plug-in that used the Doclet API

to create custom Javadoc tags. With this ingenious approach, Öberg used his newly defined tags to generate Java files rather than the HTML pages commonly associated with Javadoc output.

As the name implies, EJBDoclet was primarily concerned with code generation for EJB support. However, since its inception, EJBDoclet has extended its portfolio to include software components other than enterprise beans. In line with this wider scope, the product name was changed to XDoclet, and it was launched as an active open source project. For the purposes of this discussion, we focus on the enterprise-bean-generation capabilities of XDoclet.

The custom Javadoc tags of XDoclet make it possible to generate much of the boilerplate code necessary for enterprise bean development.

The best way to appreciate the time XDoclet can save on a project is to look at an example. The next sections do just this, putting XDoclet through its paces by seeing just how much boilerplate code can be generated from the skeleton of a simple session bean.

The first step is to install the XDoclet software.

Installing XDoclet

XDoclet is maintained as a Source Forge project and can be downloaded from `http://xdoclet.sourceforge.net`. Full installation and setup instructions, along with information on the open source license, are provided as part of the download.

XDoclet is run using the Jakarta Ant build utility from Apache. Ant has proven extremely popular among the Java community and has overtaken build tools such as *make* as the de facto standard for building Java applications. A copy of Ant can be freely downloaded from the Apache site, see `http://ant.apache.org`.

If you are unfamiliar with Ant, you may wish to read the documentation that comes with the installation before continuing with this chapter. However, the Ant syntax is relatively straightforward, and the examples covered are explained in detail. We start by covering the Ant build file necessary to run XDoclet over our source files.

Setting Up the Ant Build File

The XDoclet installation comes with a number of Ant tasks that are used for invoking the doclet engine from Ant build files. The two main Ant tasks XDoclet provides are `<ejbdoclet>` for EJB support and `<webdoclet>` for Web applications. Since we are generating the code for an enterprise bean, the example covers the use of the `<ejbdoclet>` Ant task.

Ant is covered in Chapter 12.

Listing 6–4 shows the relevant extracts from the Ant build file.

Listing 6–4 Ant Build File

```
<!-- Setup xdoclet classpath -->
<path id="project.class.path">
    <fileset dir="${libs.dir}/xdoclet-1.2">
        <include name="*.jar"/>
    </fileset>
    <pathelement location="${j2ee.lib}"/>
</path>

<!-- Define the ejbdoclet task -->
<taskdef name="ejbdoclet"
            classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath refid="project.class.path"/>
</taskdef>

<!-- Invoke xdoclet compilation -->
<target name="generate">
    <ejbdoclet destdir="gen-src"
               ejbspec="2.0">

        <!-- Source files to be processed by xdoclet -->
        <fileset dir="src">
            <include name="**/*Bean.java"/>
        </fileset>

        <!-- Code generation options -->
        <remoteinterface pattern="{0}Remote"/>
        <homeinterface/>
        <localinterface/>
        <localhomeinterface/>
        <utilobject cacheHomes="true"
                    includeGUID="true"
                    kind="physical"/>

        <!-- Generated deployment descriptors location -->
        <deploymentdescriptor destdir="${ejb.dd.dir}"/>

        <!-- Container specific -->
        <weblogic destdir="${ejb.dd.dir}"
                    xmlencoding="UTF-8"
                    validatexml="true"/>
    </ejbdoclet>
</target>
```

The comments in the build file highlight the various steps necessary for setting up and running the doclet engine. Next, we go over each step in more detail.

Setup the XDoclet Classpath

As with any Java program, the classpath must be configured to point to all the libraries the running application requires. The `<path>` tag builds up a classpath that includes all of the XDoclet libraries and the EJB library:

```
<path id="project.class.path">
    <fileset dir="${libs.dir}/xdoclet-1.2">
        <include name="*.jar"/>
    </fileset>
    <pathelement location="${j2ee.lib}"/>
</path>
```

The `<fileset>` tag provides a convenient shorthand for adding all of the libraries in the XDoclet directory to the classpath without having to specify each individual library. The locations of the various libraries are defined with Ant properties, which are set up elsewhere in the build file. The use of properties is considered good practice for Ant build files because it allows the build file to be tailored for different environments.

Define the XDoclet EJB Task

Ant needs to know what to do when it encounters the `<ejbdoclet>` tag. The `<taskdef>` task tells Ant where it can locate the class that will handle the `<ejbdoclet>` tag on behalf of Ant:

```
<taskdef name="ejbdoclet"
            classname="xdoclet.modules.ejb.EjbDocletTask">
    <classpath refid="project.class.path"/>
</taskdef>
```

Attributes for the task specify the name of the new tag and the implementing XDoclet class. The classpath established earlier gives directions to the libraries required by the `<ejbdoclet>` task.

Invoke XDoclet from Ant

The `<ejbdoclet>` task initiates XDoclet, providing all the configuration details the doclet engine requires in order to generate code from the Java source files.

We use this task to tell XDoclet the location of the destination directory for all generated code and the EJB version to be supported. Taken from the example, we have the following:

```
<ejbdoclet destdir="gen-src"
           ejbspec="2.0">
```

The `destdir` attribute specifies the location for all source generated by XDoclet, while the EJB version is specified with the `ejbspec` attribute.

The next step is to supply XDoclet with the location of all source code to be parsed. This is achieved with the `<fileset>` task, as shown below:

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

The `<fileset>` task submits a list of all files for processing to XDoclet. For the example, all source resides in the `src` directory, and we've further informed XDoclet we are only interested in files with a filename that matches the `*Bean.java` pattern.

warning

> Do not use the same directory for handwritten and generated code. Keep the two separate to avoid the risk of inadvertently modifying generated program artifacts.

The ability to specify different directories for source and generated output is critical to good housekeeping, as it keeps the source that is produced by XDoclet distinct from code managed by hand. This separation lessens the likelihood of a developer inadvertently changing generated code, which would result in any changes being lost once the build process was rerun. Moreover, the separation of the two code types also enables *clean* operations to be implemented more easily, as all code under the `gen-src` directory can safely be deleted.

The next elements determine just what the `<ejbdoclet>` task is to generate from the source it parses. In this case, we produce remote, home, and local interfaces as well as a deployment descriptor and helper class for the enterprise bean. This is achieved with elements nested inside the `<ejbdoclet>` task as follows:

```
<remoteinterface pattern="{0}Remote"/>
<homeinterface/>
<localinterface/>
<localhomeinterface/>
<utilobject cacheHomes="true"
            includeGUID="true"
            kind="physical"/>
<deploymentdescriptor destdir="${ejb.dd.dir}"/>
```

Additional instructions can be passed to the doclet engine by specifying attributes for the nested elements. For example, with the <remoteinterface> tag, we specify that the word *Remote* be appended to the end of the remote interface name.

For the deployment descriptor, we request that it be generated to a different directory than that of the other generated code. Typically, deployment descriptors are generated directly into the target build directory to simplify the packaging of the enterprise bean later in the build process.

The final element in the example is vendor-specific and generates a deployment descriptor proprietary to BEA WebLogic Server:

```
<weblogic destdir="${ejb.dd.dir}"
          xmlencoding="UTF-8"
          validatexml="true"/>
```

XDoclet is not shackled to a single J2EE vendor and provides support for generating deployment descriptors for many of the different application servers currently available. The use of these elements makes it easy to target multiple vendor application servers by including elements for each server within the `<ejbdoclet>` task. Refer to the XDoclet documentation for a complete list of all the application servers supported.

With the build file created, we are nearly all set to start generating code for our session bean. All that is needed is a Java file complete with attributes.

Creating a Session Bean

To begin building our enterprise bean, we must first write the implementation for the session bean and add the various XDoclet attributes as Javadoc tags. Listing 6–5 shows the source for a basic Customer session bean, complete with XDoclet attributes.

Listing 6–5 Session Bean with XDoclet Attributes

```
// CustomerServiceBean.java

package customer;

import java.rmi.RemoteException;

import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * @ejb.bean
 *  name="CustomerService"
 *  jndi-name="CustomerServiceBean"
 *  type="Stateless"
 *
 * @ejb.resource-ref
 *  res-ref-name="jdbc/CustomerDataSource"
 *  res-type="javax.sql.Datasource"
 *  res-auth="Container"
 *
 * @weblogic.resource-description
 *  res-ref-name="jdbc/CustomerDataSource"
 *  jndi-name="CustomerDS"
```

```
 *
**/
public abstract class CustomerServiceBean
  implements SessionBean {

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void createCustomer(CustomerVO customer) {
  }

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void updateCustomer(CustomerVO customer) {
  }

  /**
   * @ejb.interface-method tview-type="both"
   */
  public CustomerVO getCustomer(int customerID) {
    return new CustomerVO();
  }

} // CustomerServiceBean
```

The code in Listing 6–5 is a very minimal implementation of a session bean. However, from this one source file, all the code necessary to assemble a complete enterprise bean can be generated. From this example, the following file types are produced:

- `ejb-jar.xml` deployment descriptor
- `weblogic-ejb-jar.xml` proprietary deployment descriptor
- Home, remote, and local interfaces for the enterprise bean
- Helper class for accessing the enterprise bean

Producing all of these files by hand is a slow, tedious, and error-prone process. XDoclet does all of the grunt work for us with the help of a few carefully placed attributes in the bean class.

Declaring XDoclet Attributes

The XDoclet attributes masquerade as Javadoc tags. They take the form of a namespace followed by a tag name located within the namespace scope. Properties of the tag are passed in as named arguments. Here is the format of an XDoclet tag:

```
@namespace.tag name="value" name2="value"
```

Within the code of the bean, XDoclet tags are embedded at the class and method levels. Each tag augments the information already defined in the Java. The doclet engine does not simply parse the file, looking for tags and ignoring the code. Instead, where XDoclet encounters a tag, it uses the Java Doclet API to retrieve information on the code annotated by the tag. All of this information, both tag and source, is used in the code generation process.

Class-Level Tags

The class-level `@ejb.bean` tag gives the bean a name, the JNDI lookup name, an optional display name, and a type. For the example, a type of `stateless` has been defined for a stateless session bean. Here is how this information is expressed with XDoclet tags:

```
@ejb.bean name="CustomerService"
  jndi-name="CustomerServiceBean"
  type="Stateless"
```

The example also has two further tags at the class level: `@ejb.resource-ref` and `@weblogic.resource-description`. Each of these tags defines a resource; in the example, a data source has been defined. The output from each tag is output directly to the deployment descriptors. The tag with the namespace `@weblogic` is a proprietary tag and results in the details of the data source being written to the `weblogic-ejb.xml` deployment descriptor. Equally, other proprietary tags can be used here to support alternative J2EE vendors.

XDoclet supports a comprehensive range of tags and allows a complete deployment descriptor to be specified entirely within the code as attributes. For the full list of tags, refer to the XDoclet documentation.

Method-Level Tags

Method-level tags give fine-grained control for each method. The example illustrates the use of the `@ejb.interface-method` to tell the doclet engine whether the method is to be part of the bean's remote or local interface. The options available are `remote`, `local`, or `both`. Wanting to generate as much code as possible from the example, I have specified `both`, which as the name implies sees the method added to each interface type.

With the build file created and source file suitably adorned with attributes, all that remains is to execute the build file and examine the output.

Inspecting the Output

Issuing `ant generate` from the command line unleashes XDoclet upon the source, and the result is all the files necessary to assemble a complete session bean. Table 6–2 lists the files created by XDoclet.

Table 6–2 XDoclet Generated Files for the `CustomerService` Session Bean

Name	Description
`ejb-jar.xml`	EJB deployment descriptor
`weblogic-ejb-jar.xml`	Deployment descriptor for BEA WebLogic Server
`CustomerServiceHome.java`	Home interface
`CustomerServiceLocal.java`	Local interface
`CustomerServiceLocalHome.java`	Home for local interface
`CustomerServiceRemote.java`	Remote interface
`CustomerServiceUtil.java`	Helper class for accessing the session bean

The list in Table 6–2 illustrates the amount of baggage associated with a single enterprise bean. Each file is related to and must be kept in sync with the originating bean class. A change in the bean class must be reflected across all of the different interfaces and deployment descriptors. Performing this task manually is monotonous, time consuming, and subject to error.

The tag-like attributes of the XDoclet engine enable all of these files to be generated from a single source. A change to the bean class can be replicated out to all files associated with the enterprise bean by running the XDoclet engine.

XDoclet is an active code generator, and consequently, the output of the doclet engine can be considered disposable. The generation of all source files by XDoclet should be made part of the build process. In this way, the bean class, deployment descriptors, and EJB interfaces are kept in sync.

Discrepancies in these files can often be hard to detect. The relationship between a remote interface and the implementing bean class is not enforced by the compiler. This relationship is defined by configuration parameters in the enterprise bean's deployment descriptor. Errors in configuration-based relationships are difficult to detect, as they require runtime tests to determine any fault. The use of a code generator in this instance removes the possibility of such time-consuming errors occurring.

Let's examine what has been generated from our solitary bean class. Listing 6–6 shows an extract from the `ejb-jar.xml` for the session bean. As can be seen, all the pertinent information from the bean class has been extracted by the doclet engine and defined in the EJB deployment descriptor.

Listing 6–6 Generated ejb-jar.xml

```
<!-- Session Beans -->
<session >
  <description><![CDATA[]]></description>

  <ejb-name>CustomerService</ejb-name>

  <home>customer.CustomerServiceHome</home>
  <remote>customer.CustomerServiceRemote</remote>
  <local-home>customer.CustomerServiceLocalHome</local-home>
  <local>customer.CustomerServiceLocal</local>
  <ejb-class>customer.CustomerServiceBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>

  <resource-ref >
    <res-ref-name>jdbc/CustomerDataSource</res-ref-name>
    <res-type>javax.sql.Datasource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

</session>
```

The bean class defines the vendor-specific tag, `@weblogic.resource-description` at the class level. Output from this tag is proprietary to WebLogic Server and is written to the WebLogic deployment descriptor, `weblogic-ejb-jar.xml`. Listing 6–7 contains an extract from the deployment descriptor. The information generated by the `@weblogic.resource-description` tag is represented in the deployment descriptor by the element `<resource-description/>`.

Listing 6–7 Generated weblogic-ejb-jar.xml

```
<weblogic-enterprise-bean>
  <ejb-name>CustomerService</ejb-name>
  <stateless-session-descriptor>
  </stateless-session-descriptor>
  <reference-descriptor>
    <resource-description>
      <res-ref-name>jdbc/CustomerDataSource</res-ref-name>
      <jndi-name>CustomerDS</jndi-name>
    </resource-description>
  </reference-descriptor>
  <jndi-name>CustomerServiceBean</jndi-name>
  <local-jndi-name>CustomerServiceLocal</local-jndi-name>
</weblogic-enterprise-bean>
```

A useful tool for viewing the different Java classes and interfaces generated by XDoclet, and indeed any code generator, is a modeling tool with reverse-engineering capabilities.

Reverse engineering code into model form is covered in Chapter 5.

Figure 6–2 illustrates a class diagram produced using Borland's Together ControlCenter and displays all the classes generated for the enterprise bean that are contained by the customer package. The diagram emphasizes the absence of any relationships between the originating bean class and the generated classes. These relationships are defined within the EJB deployment descriptors and so are not covered by the UML notation.

You may find reverse engineering very helpful when working with generated code. After running the code generator, the modeling tool can display what new classes have been injected into the model by the generator and how they impact the existing design model.

Our example has reached the point where it can be compiled and packaged as an enterprise bean. In getting to this point, a sizeable amount of code was generated for what is a very simple session bean with a minimum set of XDoclet tags. Now that we know how to generate the code, the next question is how we manage the code generated.

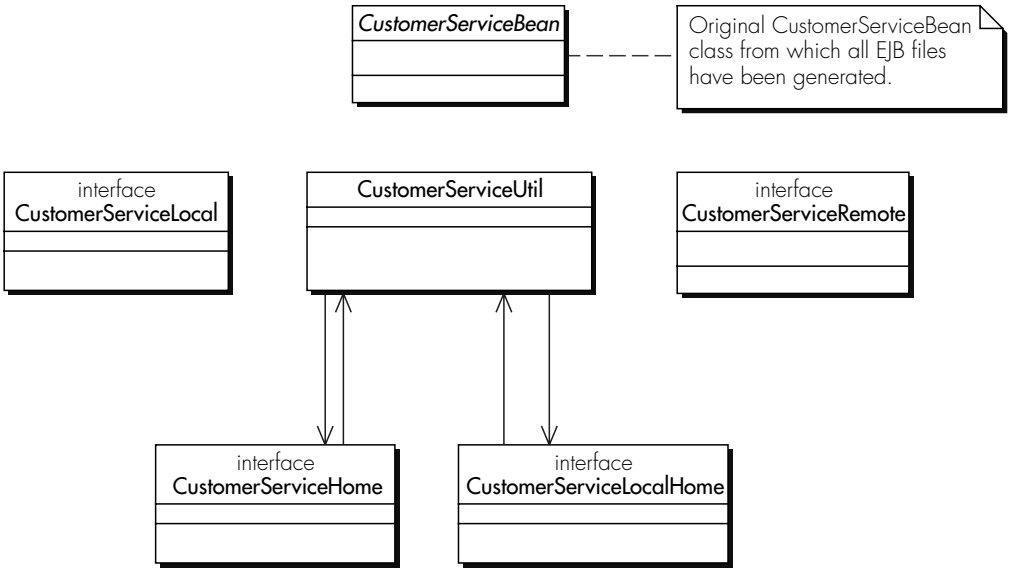

Figure 6–2 Class diagram for the `CustomerServiceBean`-generated class.

Working with Actively Generated Code

So far, we have covered the reasons why code generators are a valuable tool for the rapid developer and how code generation techniques can be applied to the development of J2EE solutions. In this section, we examine a set of guidelines for working with actively generated code.

Guidelines for Code Generation

Although active code generators such as XDoclet increase productivity on a project, the code generated must be carefully managed as part of the project build process. Failure to achieve this aim can result in confusion among developers as to which code can be modified and which code is the responsibility of the generator. Confusion of this nature negates the rapid development advantages code generators offer.

The guidelines look to avoid this confusion, enabling the full benefits of code generation techniques to be achieved.

Key Points for Code Generation

- Generate code as part of the build process.
- Keep generated code separate from other source.
- Do not place actively generated code under source control.
- Be wary when using refactoring tools.
- Avoid mixing active and passive code generation.

Generate Code as Part of the Build Process

Actively generated code is a disposable commodity. It can be quickly reproduced at near zero cost. Consequently, making the code generator part of the build process offers an effective method for ensuring all built code is an accurate reflection of the metadata upon which it is based.

Executing the code generator only when it is believed the underlying metadata has changed presents the risk of the code becoming out of sync with the source metadata. This risk can be easily mitigated by ensuring all code is generated afresh as part of a regular build process.

Keep Generated Code Separate from Other Source

To safeguard against the problem of a developer accidentally modifying generated code, it is good practice to have the build process generate the code in a directory separate from that of other source. In this way, the generated code is treated as if it were a compiled binary.

Alternate source directories are useful in this scenario. They enable the generated source to be cleanly separated from handwritten code and allow an IDE to correctly recognize the code as part of the application.

Do Not Place Actively Generated Code Under Source Control

Placing code that is actively generated under source control invites developers to mistakenly modify code that will be overwritten the next time the code generator is run. This practice also makes the build process complex, because it must integrate with the source control system.

To avoid these problems, generated code should not be placed under source control. Instead, place the metadata used by the code generator under source control if this is possible. Likewise, if the code generator has been produced inhouse, it should also reside under source control. With this approach, the build process gains an additional step:

1. Compile the code generator.
2. Run the code generator against the metadata.
3. Build the application from all sources.

It is possible to forego building the inhouse code generator, instead relying on previously built binaries. However, this approach risks failing to pick up on rule or pattern changes within the generator, which might otherwise impact the code generated. While compiling the generator may slow down the build process, it has the advantage that all generated source is guaranteed current.

Be Wary When Using Refactoring Tools

Refactoring tools have become a popular feature in many IDEs. They allow the software engineer to make sweeping changes to the code structure across the entire code base. They are a powerful development aid and are used extensively on projects adopting an agile development process.

Refactoring tools, however, show little respect for the distinction between generated code and handwritten code. The impact of a refactoring exercise can easily result in generated code being modified inadvertently by the tool. This is a problem, because the amended code will be overwritten when the code generator is next run.

Where an inhouse code generator is used, the responsibility falls to the developer to ensure the code produced by the generator reflects the changes applied by the refactoring tool.

Attribute-oriented programming tags, as used by XDoclet, are also vulnerable to this practice. As XDoclet tags are embedded within comments, they are usually overlooked by refactoring tools. Consequently, the generated code becomes out of step with the rest of the code base once the doclet engine is run.

Fortunately, the Java compiler can help to detect such problems, as can a rigorous testing schedule. Nevertheless, caution must be exercised if refactoring tools are to be used in conjunction with active code generators.

Avoid Mixing Active and Passive Code Generation

The guidelines outlined so far have focused on ways to avoid actively generated code from being inadvertently modified by developers. Unfortunately, the need to modify actively generated code is sometimes inescapable.

When this situation arises, several options are available to protect the modified source from being overwritten:

Modify the code generator.

In preference to updating the code, if an inhouse generator is used, modify the code generated to reflect the required code changes.

Extend the generated code.

The object-oriented paradigm offers many ways to extend the functionality of software components, including inheritance and composition. In some cases, the *Decorator* pattern may also prove effective. Consider these software engineering approaches before modifying any generated code.

Use merge tools.

A suitable merge tool, as is commonly used with merge-based source control systems such as CVS, enables modified code to be integrated with generated code. However, unless a *trivial merge* is the result of the union of the two code versions, a manual merge with the tool is required. Such an automated build system would be extremely cumbersome to implement.

Don't do it.

Finally, modifying actively generated code is problematic at best, regardless of what methods are used to manage the result. Think hard before taking this step, and proceed only if all other options have been exhausted.

The last point cannot be overemphasized. Having to guard against accidental code loss in the modified code adversely affects the productivity gains offered by active code generators. Keep it simple, and leave the generated code alone.

Summary

Code generation is one of the cornerstone techniques of rapid development. Through the application of both passive and active code generators, we saw how it is possible to speed up the development process and simplify the task of creating complex J2EE components, such as enterprise beans, with attribute-oriented programming.

Now J2SE annotations make program-level metadata available at both compilation and runtime. This new language feature is expected to herald a new wave of development tools that simplify the development of Java applications. Already, the draft EJB 3.0 specification is looking to use annotations to enable developers to define enterprise beans as lightweight classes. We can therefore expect annotations to feature prominently in future versions of J2EE servers.

Code generation delegates the boring, humdrum tasks to the code generator, leaving us free to work creatively and imaginatively on the problem of building better solutions for the customer. After all, it is the application of the creative mindset that makes software engineering such an engaging profession.

Chapter 7 continues with the theme of code generation and focuses on one of the richest sources of metadata available, the database.

Additional Information

Another form of programming based on the principles of code generation is *generative programming*. This term was first coined by Krzysztof Czarnecki and Ulrich Eisenecker in their book, *Generative Programming: Methods, Tools, and Applications* [Czarnecki, 2000]. Generative programming involves using active code generators to solve families of software engineering problems rather than using custom solutions to solve individual problems.

The *Code Generation Network* site, found at `http://www.codegeneration.net`, provides links to various code generation resources, including a wide range of code generators for Java developers.