



## Chapter 7

# Generative Programming

Generative programming (GP) is code reuse via the automation of code development. Instead of writing Java code directly, you describe a problem in a specification language tailored to the problem space, and then you employ a tool to generate the necessary Java source code or bytecode. Java is widely hailed as a language suitable for object-oriented development, but it is equally suited for GP. In fact, object-oriented programming and GP are complementary, and many of the most exciting technologies in the Java world today combine the two. This chapter has four purposes:

1. Present the motivations for using GP.
2. Develop a taxonomy of the binding times and modes that are possible in Java and the tools that each employs.
3. Demonstrate how GP is already in wide use in the Java world, especially J2EE.
4. Present examples of GP that will jump-start your thinking on how to use GP in your own projects.

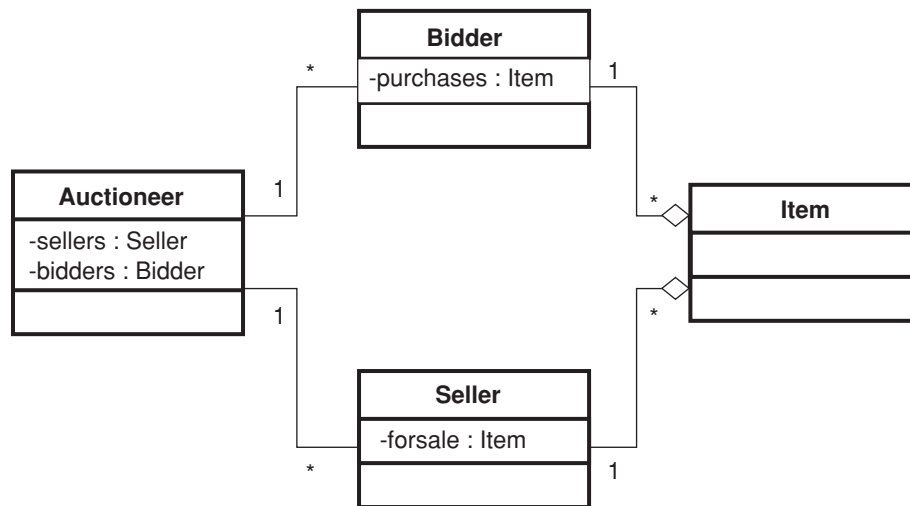
### 7.1 Why Generate Code?

The reason to generate code is simple: to efficiently capture and reuse knowledge of a problem domain. [Cle01] provides several useful terms to describe *domain analysis*, the design process that often leads to a GP implementation. Domain analysis identifies the *commonalities* and *variabilities* of a family of related software systems. Commonalities are standard features that are coded into the system and shared by all permutations of the system. Variabilities are



features that can differ in various products, or in different invocations of the same product. At some point in the lifecycle of a system, you must make a choice, or *specification*, for each variability. The point in time that a choice is made is the *binding time*.

For example, consider the simple online bidding system depicted in Figure 7–1. The commonalities shown in the figure are the relationships between auctioneers, bidders, sellers, and items to be sold. You can imagine some of the variabilities: legal bid increments, number of bidding rounds, and the types of information available about each item. Choices can be binary (Does the item have a picture?), numeric (How many bidding rounds will there be?) or something much more complex. The binding time for each choice depends on the implementation, as you will see.



**Figure 7–1 A simple online bidding system**

### 7.1.1 Object-Oriented Approaches to Modeling Variabilities

In a traditional object-oriented design, variabilities are modeled with a combination of inheritance and parameterization. The code in Listing 7–1 shows fragments of two different approaches to implementing an `Auctioneer` interface, which is responsible for enforcing the number of bidding rounds and the legal bid increments. The `ThreeRoundFiveDollar` implementation uses inheritance

to model each choice. With this approach, each combination of the number of rounds and the minimum bid increment would result in a distinct concrete implementation.

The second example, `AuctioneerImpl`, models all possible `Auctioneer`s with a single implementation class. The specification of rounds and minimum bid increment are made explicitly at runtime by passing in parameters.

#### Listing 7-1 Modeling Auctioneer with Inheritance and Parameterization

```
//using inheritance to model every variability
public class ThreeRoundFiveDollar implements Auctioneer {
    public void runAuction(Item i) {
        for (int n=0; n<3; n++) {
            runBidRound(i, 5);
        }
    }
    //etc.
}

//using parameters to model every variability
public class AuctioneerImpl extends Auctioneer {
    public AuctioneerImpl(int rounds, int minIncrement) {
        this.rounds = rounds;
        this.minIncrement = minIncrement;
    }
    public void runAuction(Item i) {
        for (int n=0; n<rounds; n++) {
            runBidRound(i, bid);
        }
    }
    //etc.
}
```

In this example, the `AuctioneerImpl` is obviously the better design; because the choices are across a range of values, the first approach might require an unlimited number of subclasses. Since the choices do not imply different logic or storage requirements, the `AuctioneerImpl` class can trivially encode the choices as parameter values.

One can just as easily concoct a scenario that favors inheritance over parameterization. Imagine that the `Items` being sold can have text information, a

picture, or a movie. This situation favors inheritance, as Listing 7–2 demonstrates. Each choice in Listing 7–2 is binary—either the media is present or it is not. The number of possible classes in an inheritance-based solution is therefore bounded. Because each choice implies different storage and logic, the parameterized implementation is inefficient. The `ItemImpl` must keep fields for each possible data type, and it must execute branching logic each time through its display method. Real problems and real designs tend to fall between the two extremes and employ parameterization and inheritance in tandem.

### Listing 7–2 Modeling Items with Inheritance and Parameterization

```
//using inheritance to model every variability
public class ItemWithText implements Item {
    Text t;
    public void display() {
        t.print();
    }
}

//using parameterization to model every variability
public class ItemImpl implements Item {
    Text t;
    Image i;
    Movie m;
    public void display() {
        if (t) t.print();
        else if (i) i.draw();
        else if (m) m.play();
    }
}
```

#### 7.1.2 Thinking in Terms of Bind Time

Now consider the bind times implied by each approach. With the inheritance-based solution, each different specification is instantiated as a different concrete class. Therefore, the specification is bound during development, which is often called compile-time binding.

This has important consequences. The *developer* must choose the specification since the choice is made during development. Of course, the developer

could be acting on detailed instructions from an end user, but the important point is that the end user cannot change the specification later, after the developer is gone.

For the parameterized solution, the specification is bound at runtime by passing in parameters. This implies that an end user can choose the specification at runtime if the program yields control of the parameters. In general, later binding gives more flexibility to the user, but earlier binding may offer better performance. The compiler can optimize code based on your specifications only if the specifications are available before the compiler runs.

The Java development world has four obvious bind times.

1. Compile-time binding happens when the compiler runs.
2. Design-time binding happens when a designer configures the initial state of an already compiled component. JavaBeans are designed specifically with design-time binding in mind; designers often use a visual tool to examine and modify bean properties.
3. Deployment-time binding occurs when components are installed onto the network where they will be used. Deployment-time binding is distinct because even though the developer may no longer be present, a system administrator will be.
4. Runtime binding occurs after an application starts to execute.

To the generative programmer, bind time is a very important issue that needs to be treated separately from the actual specification that is bound.

### 7.1.3 Separating Specification from Bind Time

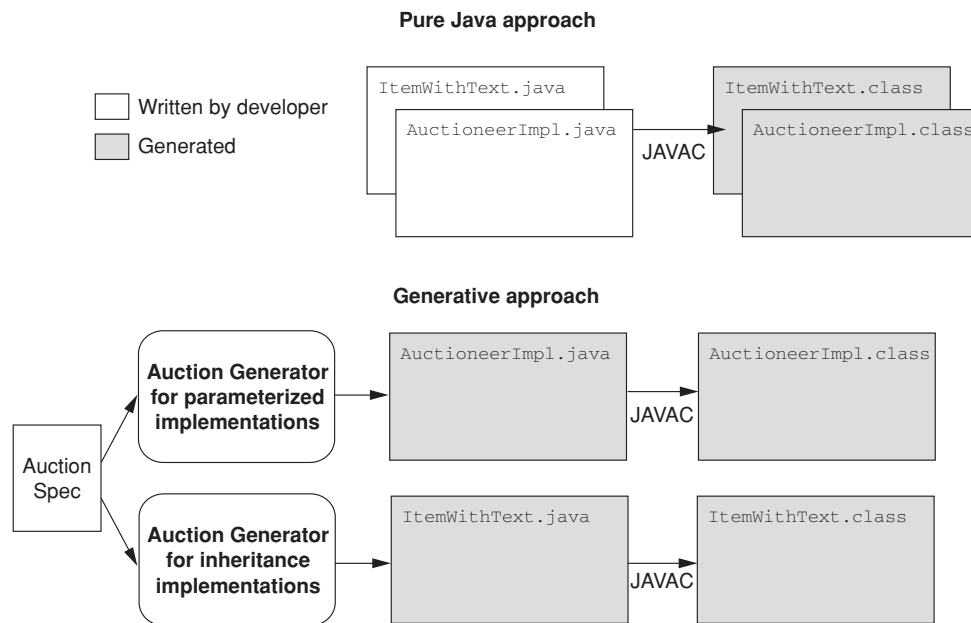
The online bidding examples discussed earlier suffer from a basic flaw. The pure Java approach hopelessly tangles the binding time with the specification chosen. A single object design cannot elegantly handle both issues simultaneously, as the examples demonstrate. Changing the specification requires only small edits, but switching from inheritance to parameterization requires a wholesale rewrite of the code.

To take a generative approach to the auction simulation, you need to separate these two concerns into distinct artifacts: a specification document that enumerates the choices for each variability, and a set of *generators* that process the

specification document and produce the application code. An example specification document for the auction simulation is shown in Listing 7–3. This specification describes an entire bidding system, and it will be processed by one or more generators, as shown in Figure 7–2. The generators manufacture the program that will actually run the simulation. The specification document encodes the choices, and the generators select the binding times.

### Listing 7–3 Specification Document for the Auction Simulation

```
auction.rounds=3
auction.minIncrement=5
item.1=text
item.2=movie
```



**Figure 7–2 Generating an auction from a specification**

The separation of specification data into a separate layer lets you experiment with different binding strategies. From the same specification, you could generate any number of different implementations, including the examples above. You could have separate generators that prioritized readability of the

generated code, fast performance, small code size, or any other measure that you value. If your bind-time priorities change, the critical domain knowledge of your specification is not lost. You simply reuse it with a new generator.

Object designs have other issues besides binding time that tend to *cross-cut* object hierarchies. A cross-cutting issue does not fit neatly into a single class. Synchronization, transactions, security, and auditing are notorious cross-cutters. Dealing with them tends to litter small amounts of code across many of the classes in your design. With a separate specification, you can hoist these concerns out of your code into a single place, and then you can generate the code that interleaves them.

#### 7.1.4 Choosing a Specification Language

The generative approach has a number of other desirable properties as well. While the configuration document in Listing 7–3 is in the format of a Java properties file, you can choose the format most convenient to your problem domain (just remember that your generators will have to read this format). You do not have to use the same language for the generators and for the generated code, so everything you have seen so far applies equally well to any programming language. Nor are generators limited to creating application code. You could also generate documentation, test scripts, and deployment instructions. Generating all these project artifacts from a shared configuration document makes it much easier to keep various elements of your project in sync.

#### 7.1.5 Reuse Requires More Than One Use

The primary disadvantage to a generative approach, as with other reuse approaches, is that you must build more than one system for the initial effort to pay off. If you only plan to build one application within a problem domain, then analyzing commonalities and variabilities will be valuable, but the effort to develop a suite of generators will not. Fortunately, many development projects do belong to a large family of similar tasks. The online auction in Figure 7–1 is a good example, as is an online shopping cart, and several other kinds of online transactions.

### 7.1.6 A Little Domain Analysis Is a Dangerous Thing

This has been a lightning introduction to generative programming. In the interest of space, I have been economical with the truth. Domain analysis can be far more intricate than it is when you use it to simply identify variabilities, and it often leads to implementation strategies other than GP. If you want to know more, [Cle01] provides a gentle introduction to GP using Java and XML, and [CE00] stands to be the bible of this emerging field. The purpose of the remainder of this chapter is not to rehash these books in capsule form; instead, the objective is to look at the possible relationships between GP and the component services described earlier in this book. Class loading, type information, and metadata explode the simplistic notion of binding time presented thus far, and they greatly enhance the utility of GP on the Java platform.

## 7.2 Why Generate Code with Java?

Since the principles of generative programming apply to other languages as well, why use Java? One could argue that Java is not particularly well suited to code generation. After all, C++ has built-in support for code generation with macros and templates. Scripting languages like Perl are very good with string operations and might be better suited for writing generators. Despite these valid objections, Java is particularly suited to GP for five reasons:

1. High quality type information acts as a valuable implicit specification document.
2. Flexible class loading supports any combination of binding times and binding modes.
3. Java source files are simple to read and generate.
4. Java bytecode files are simple to read and generate.
5. Generated code can provide dramatic performance improvements which can obviate the overhead of the VM.

### 7.2.1 Type Information Acts as a Free Specification Document

High quality type information and reflection can act as a specification document for GP. Best of all, you do not even have to write the document since it is implicit in all Java classes. Many services can be generated from type information alone.



As a very small example, consider your Java IDE. Most IDEs have an “implement interface” wizard that creates a new class to implement some interface. The wizard uses reflection to build a Java file with all the method signatures already in place and just waits for you to fill them in. This is a tiny but useful example of GP.

### 7.2.2 Class Loading Supports Flexible Binding Modes

The class loading architecture makes it easy to load new classes on-the-fly at runtime. This shatters the simplistic assumptions made about bind time in the previous section. In order to capture the possibilities opened by dynamic class loading, you need to augment bind time with the notion of *bind mode*. Bind time is *when* a decision is encoded, while bind mode is *how* that decision is encoded. At one extreme, static bind mode means that decisions are frozen into the code. At the other extreme are dynamic bindings, which are encoded as runtime branches, perhaps via a parameter or virtual method invocation. With dynamic class loading, you can bind specifications at runtime and still have excellent performance by generating a class that statically binds the specification.

Because class loading is so flexible, generators can create classes that efficiently encode binding decisions at runtime. Dynamic proxies are one example; they use type information to generate an implementation of a batch of interfaces specified at runtime. Another example is JavaServer Pages (JSP). JavaServer Pages have their own, presentation-oriented configuration document. You write your code as a JSP page and then drop it into a JSP container. The container acts as a generator and converts this format into a normal Java source file, which it then compiles and loads dynamically.

### 7.2.3 Java Source Is Easy to Generate

The simplicity of Java syntax encourages code generation projects. It is easy to write a program that will emit a valid Java source file. Because Java source files do not have macros or templates, it is also easy to use a Java file as input to a generator or even to write a generator that modifies a file in-place. On the minus side, since macros and templates are not supported, you are forced to build your own generation schemes to mimic these capabilities.

### 7.2.4 Java Binary Classes Are Easy to Generate

The class file format is also straightforward, so you can write a generator that emits valid Java class files, and omits the source code step entirely. This feature is crucial if your generator will execute at runtime in an environment where a compiler may not be available. The portability of the class file format also guarantees that your generated code will work on any compliant Java platform.

### 7.2.5 Code Generation Boosts Performance

Perhaps the most important motivation for GP in Java is the potential for performance gains. These performance gains come from two sources. First, your code generators are free to generate efficient code regardless of readability.<sup>1</sup> Your domain knowledge is stored in the specification file and in the generators, so these are the artifacts that need to be readable and maintainable. Second, you can get late binding semantics with early binding performance.

In general, early binding makes for better performance. For example, if you could hard-code *all* your choices during development, your code would not need conditional statements or virtual methods at all, and it would be blazingly fast. Of course, many choices must be made at runtime. Generative techniques allow you to use runtime binding with a static binding mode, which enables you to generate the code once and reuse it for future iterations. For example, Java serialization uses reflection every time you read or write a Java instance. You could write a generator that uses reflection only once during development to generate a helper class that binds statically to the field values.<sup>2</sup>

### 7.2.6 Levels of Commitment to Code Generation

It is useful to divide code generation schemes by scale. Code generation in-the-large is a complete commitment to code generation for an application. Here, the

---

1. This is true only up to a point since you may need to read the generated code when you are debugging. Ideally debugging tools can relate the generated code back to the specification, but this will not always be the case.

2. This would also require that the fields be at least package-protected instead of private, and it is an argument in favor of small packages with shared access to class fields. With some additional effort you could even optimize access to private fields by modifying classes as they were being loaded.

entire architecture assumes that generation is being used, and in fact, it may only accept specification files as inputs. Helper components are provided to service the generated code, and they may not even be documented or accessible for direct programmer consumption. This style of code generation is widely used in the J2EE architecture. For example, JavaServer Pages (JSPs) and Enterprise JavaBeans (EJBs) are worthless without a code generation step that creates the code that the client will actually invoke.

Code generation in-the-small suggests techniques that can be used within part of a project, at the class, method, or field level. Code generation in-the-small can enhance your development process in a more encapsulated way, without binding you to a particular architecture such as J2EE. This chapter will leverage J2EE for examples of generation in-the-large, and it will introduce some custom examples for generation in-the-small.

### 7.3 A Taxonomy of Bind Times and Modes

The flexibility of Java class loading means that you can bind your specifications at any time and in any mode. Table 7–1 categorizes some generative programming techniques by bind time and mode. Notice that the divisions are somewhat arbitrary. Because Java preserves full type information in its compiled class file format, most of these techniques *could* be used at any time. Deployment time and runtime have been combined in the table because dynamic class loading makes it straightforward to redeploy at runtime. I have listed each technique where it is most likely to be used today. Some of these divisions will change in the future; for example, future versions of `rmic` might generate stubs at runtime.

**Table 7–1 Generative Programming in Java by Bind Time and Mode**

	Static Bind Mode	Dynamic Bind Mode
<b>Development time</b>	IDE wizards, <code>rmic</code> , JavaBeans	Default serialization
<b>Design time</b>		JavaBeans
<b>Deployment/runtime</b>	JSP, EJB	EJB, dynamic proxies

In general, as you move up in the table, you have more services available, both human and software. At development time you have access to developers, high-end developer machines, and end users. At runtime you have only end users and whatever software they install. As you move left in the table, you *need* more services because there is more code to generate, but the resulting code can be faster if the overhead of generating the code is affordable.

Start from the top of the table and work down. IDE wizards run at development time and produce code that is compiled into the application. The RMI stub compiler (`rmic`) also runs during development and produces implementations that are specific to a particular remote object, while default serialization is more dynamic. Although field types are bound during development, they are traversed reflectively at runtime. JavaBean property types and names are chosen during development, but their values can be dynamically modified at design time. Java-Server Pages are translated into servlets, which are then statically compiled at runtime.

Table 7-1 lists EJB under both static and dynamic bind mode. The Enterprise JavaBeans specification is flexible; EJB functionality can be produced by generating static code or by passing parameters through dynamic code. Dynamic proxies are, of course, dynamic. The dynamic proxy architecture generates an in-memory class file that forwards all of its interfaces to an `InvocationHandler`, which almost always uses dynamic invocation to forward the call to another object.

Another way to characterize various styles of generative programming is by inputs and outputs. In the Java world, inputs might be any combination of Java source code, Java class files, and non-Java vocabularies suited to the problem domain. Outputs are typically Java source or class files. Table 7-2 organizes some common Java technologies by inputs and outputs. Generators that work only with Java class files use Java type information to build connectors. RMI stubs connect objects in different virtual machines, and dynamic proxies connect objects through an intermediary handler. JSP defines its own file format that includes embedded Java code, and EJB includes both Java code and XML-based deployment descriptors. SOAP is an XML specification for describing request



and response methods in a language-neutral way. SOAP generators take the XML description of types and generate language-specific mappings.

**Table 7–2 Generative Programming in Java by Inputs and Outputs**

Inputs/Outputs	Source Files	Class Binaries
Java class binaries	RMI stubs, IDE wizards	RMI stubs, dynamic proxies
Non-Java data	SOAP	
Mixed data	JSP, EJB	EJB

Note that source files are easier to generate than class files. In fact, most of the technologies that output class files “cheat” by emitting and compiling a source code file. The only example listed in Table 7–2 that goes directly to byte-code is dynamic proxies, which can run on client-side machines that do not have access to a compiler.

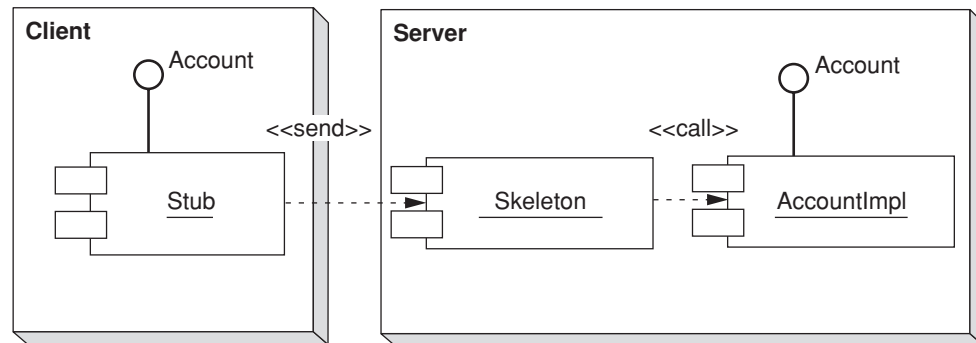
Subsequent sections of this chapter describe RMI, JSP, and EJB in more detail. Dynamic proxies are described earlier in the book in §3.4.

## 7.4 Code Generation in RMI

Java Remote Method Invocation (RMI) uses code generation to build *stubs* and *skeletons*. A stub implements a remote interface by serializing method calls into a stream and forwarding that stream to the actual implementation class, often on a different physical machine. RMI clients never hold a direct reference to an implementation class; instead they use a local stub class, which forwards the call to the implementation. A skeleton receives stream-encoded method calls from a stub, converts them back into call stacks, and invokes the corresponding methods. Figure 7–3 shows an RMI call passing through a stub and skeleton to the implementation class.

To create RMI stubs and skeletons, you run the Java RMI stub compiler `rmic`, passing in the name of a remotable class—one that implements the `Remote` marker interface. This ties stub generation to development time, or deployment time at the latest, since you cannot expect clients to have (or correctly





**Figure 7-3 RMI calls pass through stubs.**

use) `rmic`. The `rmic` tool reads the remote class's type information to discover method signatures, which it then uses to generate the stubs and skeletons.

The RMI stub compiler does not generate class files directly (although this is certainly possible in theory). Instead, it first generates a Java source code file, and then it invokes the compiler on it. Unlike the situation with dynamic proxies, generating a source file first is a reasonable approach because `rmic` is installed along with the compiler. Normally, `rmic` deletes the source file so that you never see it, but the `-keep` option allows you to see the Java code for the stubs if you like.

If you refer back to Table 7-1, you will see that `rmic` is listed as a compile-time use of code generation. However, the RMI stubs and skeletons are driven entirely by type information and require no additional semantic knowledge of the interfaces being implemented. This sounds like a situation tailor-made for reflection, and indeed it is. Skeletons can be replaced entirely by a single generic skeleton that uses the `Method` class's dynamic invocation capabilities to invoke the correct method at runtime. Similarly, dynamic proxies can be generated at runtime to take the place of specific stubs, requiring only a single generic stub that knows how to serialize arguments and communicate over the network. This would move RMI stub generation down two rows in the table, into the "runtime" category.

In fact, RMI is moving toward this more dynamic approach to stubs and skeletons. In SDK 1.2, RMI added the ability to use generic skeletons, as discussed

above. A future version of RMI will probably allow dynamic proxies to be used in place of stubs, and [Öbe00] demonstrates how to trick the SDK 1.3 implementation of RMI into using dynamic proxies.

Automating the generation of stubs at runtime is a big win for developers. Because stubs are generated at runtime, there is no need to figure out how to make stub classes available to clients—an exercise in class loading gymnastics that often stymies rookie RMI developers. The potential disadvantage of runtime stub generation via dynamic proxies is that dynamic proxies use reflection, which imposes a performance penalty on every method call. In the case of RMI, this performance issue is a red herring. Dynamic proxies are hundreds of times faster than the simplest RMI calls across machines, especially when network latency and likely file operations are taken into account.

## 7.5 Code Generation in JSP

JSP represents an entirely different use of code generation. JSP provides a web-content-oriented language that can include escapes to blocks of Java code. The idea is that web developers experienced with HTML and XML can design pages that have a substantial amount of static content, and then they can occasionally use escape sequences to introduce blocks of Java code. These blocks of Java code execute when the page is accessed and can add dynamic content to the page.

A JSP engine converts JSP syntax into a normal Java source file containing the code for a Java servlet, which it then compiles and executes. Listing 7–4 shows a simple Hello.jsp servlet that displays a greeting. Normal text in the page is sent directly to the client as HTML by default. The text bracketed by `<% %>` represents special instructions to be evaluated by the JSP engine.

Listing 7–5 shows the servlet generated by this simple JSP page; it has been edited for space and for readability on the printed page.<sup>3</sup> The generated servlet is simply normal Java code. When a page request arrives at the servlet container, it locates an instance of the appropriate servlet and invokes its `doGet`

---

3. I used Tomcat to generate this servlet. Tomcat is open source and is the reference implementation for servlets. See <http://jakarta.apache.org> for more details. If you want to see the servlets generated by your JSPs, they are the Java files with funky names in the `tomcat/work` directory.

method. The servlet `doGet` method is forwarded to the generated servlet's `_jspService` method, which then writes back to the client through the `out` variable. If you look through the generated servlet, you can find Java code corresponding to each line in the JSP. The `page import` directive becomes a simple import statement. Normal text blocks are simply written through the `out` variable. Code in an expression, delimited by `<%= expr %>`, is evaluated, and the result is written back through the `out` variable.

#### Listing 7-4 A Simple Hello.jsp

```
<%@ page import = "java.util.*" %>

<h1>Hello</h1>
Hello, you have reached this page at
<%= new Date().toString() %>. Have a nice day.
```

#### Listing 7-5 Servlet Generated from Hello.jsp

```
import javax.servlet.*;import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;
public class _0002fhello_0002ejsphello_jsp_2
extends HttpJspBase {
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
```



```

try {
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html; charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this, request,
        response, "", true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    // HTML // begin [file="E:\\gj\\jakarta-tomcat-\\
    //3.2.1\\webapps\\ROOT\\hello.jsp";from=(0,34);to=(4,0)]
    out.write("\r\n\r\n<h1>Hello</h1>\r\nHello, you have reached
        this page at \r\n");
    // end
    // begin [file="E:\\gj\\jakarta-tomcat-\\
    //3.2.1\\webapps\\ROOT\\hello.jsp";from=(4,3);to=(4,26)]
    out.print( new Date().toString() );
    // end
    // HTML // begin [file="E:\\gj\\jakarta-tomcat-\\
    //3.2.1\\webapps\\ROOT\\hello.jsp";from=(4,28);to=(6,0)]
    out.write(". Have a nice day.\r\n\r\n");
    // end
} catch (Exception ex) {
    if (out.getBufferSize() != 0) out.clearBuffer();
    pageContext.handlePageException(ex);
} finally {
    out.flush();
    _jspxFactory.releasePageContext(pageContext);
}
}
}

```

The JSP in Listing 7–4 and the servlet in Listing 7–5 are functionally equivalent. However, Listing 7–4 is much easier to read. In this example, code generation enables a syntax that is more suited to a specific problem domain than Java. For web content that is mostly text, the JSP syntax is simpler than a servlet with hundreds of calls to `out.write`.

JSP code generation is different from dynamic proxies or RMI stubs in that type information is not needed to generate the code. The transformations are based mostly on the JSP text. In fact, programming syntax errors in the JSP will

pass undetected through the servlet generation stage, only to be detected when the servlet is compiled.

The JSP conversion is also more performance sensitive than dynamic proxy invocation. Fortunately, there is no need to convert the JSP every time a request comes in. The conversion and compilation can take several seconds. However, the conversion needs to be done only once. The JSP engine only translates the page and compiles the resulting servlet once when the page is first accessed,<sup>4</sup> and then it caches the class to service future requests. If you direct your browser to a JSP that has not yet been compiled, you will see a substantial pause before the page is returned. Subsequent requests for the same page will return instantaneously.<sup>5</sup>

## 7.6 Code Generation in EJB

The most interesting use of code generation in the J2EE environment is Enterprise JavaBeans. Despite the similar names, EJBs are completely unrelated to JavaBeans.<sup>6</sup> An EJB represents data and logic that executes in a server environment. There are two primary kinds of EJBs. Session beans represent short-lived conversational state between a client and server, and entity beans represent long-lived data, often in a back end database.

For our purpose here, the interesting thing about EJBs is that they leverage generated code to add semantics beyond the semantics specifically encoded in Java classes. Most importantly, EJBs can acquire transactional semantics at deployment time. Developers write their Java code as normal, and application deployers describe the transactional requirements of the beans in an XML-based deployment descriptor.

To build an Enterprise JavaBean, you specify the following four things:

1. The *home interface* specifies how clients find or create the bean.
2. The *remote interface* defines business methods that clients call.

---

4. The JSP page could also be compiled on application startup, or it could be recompiled when the source on the disk changes, but the basic point remains the same.

5. Well, maybe not instantaneously, but any delays you experience will not be related to converting the page.

6. There are only so many coffee metaphors, so the coolest ones have to be reused.

3. The *bean* object contains implementation code.
4. The *deployment descriptor* describes services the deployer wants to make available to the bean.

Listing 7–6 shows fragments from each of these text files. The home interface and remote interface delineate separate interfaces for creating, finding, and using an object. The bean object executes a simple `transfer` operation by withdrawing from one account and adding to another.

#### Listing 7–6 Relevant Fragments of an EJB

```
//home interface
public interface TellerSessionHome extends EJBHome {
    public TellerSession create() throws CreateException,
        java.rmi.RemoteException;
}

//remote interface
public interface TellerSession extends EJBObject {
    public boolean deposit(Money m, Account a)
        throws java.rmi.RemoteException, TellerException;
    public boolean transfer(Money m, Account a1, Account a2)
        throws java.rmi.RemoteException, TellerException;
}

//bean
public class TellerSessionBean implements SessionBean {
    //several other methods omitted
    public boolean transfer(Money m, Account a1, Account a2) {
        a1.withdraw(m);
        a2.deposit(m);
    }
}

<!-- deployment descriptor -->
<ejb-jar>
    <enterprise-beans><session>
        <ejb-name>Teller</ejb-name>
        <transaction-type>Container</transaction-type>
    </session></enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
```

```

        <ejb-name>Teller</ejb-name>
        <method-interfaces>Remote</method-interfaces>
        <method-name>*</method-name>
    </method>
    <transaction-attribute>Required</transaction-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>

```

What happens if `transfer` fails halfway, for instance, `withdraw` succeeds but `deposit` fails? Ideally, the entire operation should rollback, and the client should receive an exception indicating that the operation failed. Of course, this could be accomplished by explicit transaction programming, but the code would be much more complex, as Listing 7-7 shows (additions to the original code are in bold). The `transfer` method must begin by looking up a transaction object via the Java Naming and Directory Interface (JNDI). Then, all data access in the rest of the method should *enlist* on this transaction, specifying that if the transaction fails, any changes should be rolled back.

#### Listing 7-7 EJB-Like Code, but with Manual Transaction Programming

```

public class TellerSessionBean implements SessionBean {
    //several other methods omitted
    public boolean transfer(Money m, Account a1, Account a2) {
        Context ctx = new InitialContext();
        UserTransaction tx = (UserTransaction)
            ctx.lookup("java:comp/UserTransaction");
        tx.setTransactionTimeout(30);
        tx.begin();
        try {
            a1.withdraw(m, tx);
            a2.deposit(m, tx);
        } catch (Throwable t) {
            tx.rollback();
            throw t;
        }
        tx.commit();
    }
}

```

Ironically, an encapsulated design makes it difficult to know where data access might be occurring. To be safe, you must recode all the methods on the `Money` and `Account` objects to take a transaction object as a parameter. This changes interfaces as well as implementations, as the extra parameter to `withdraw` and `deposit` demonstrates. There are other concerns as well. If a method throws an uncaught exception, then the transaction should abort quickly so that any resources associated with the transaction are released as soon as possible.

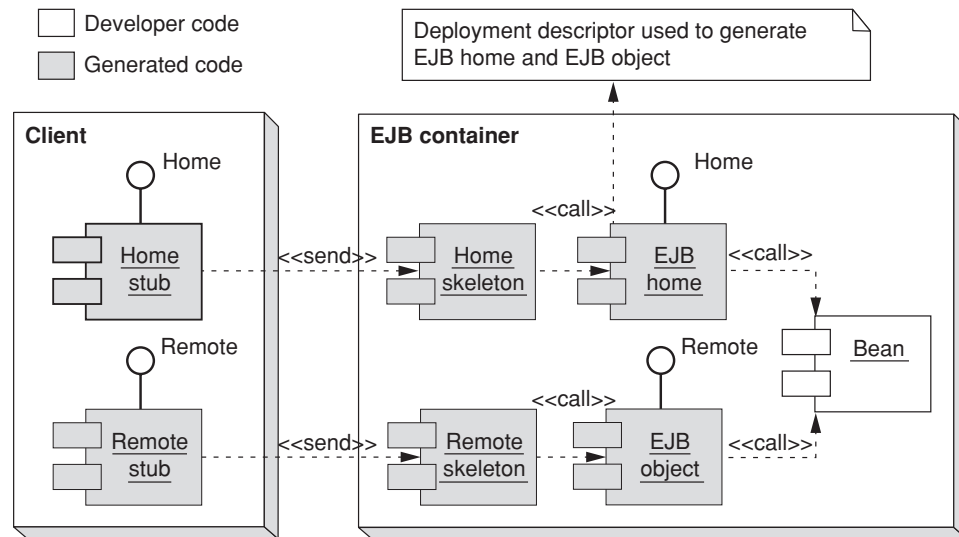
Transaction programming is a concern that cross-cuts traditional object designs. As Listing 7-7 shows, the transactional code is interleaved with the code that executes the business logic. All of the other classes, such as `Money` and `Account`, would need similar interleaving. This spreads transactional code throughout an application, making it difficult to extend and maintain.

### 7.6.1 The Deployment Descriptor

EJB attacks the cross-cutting problem by separating the transactional aspect of the system into a separate XML file called a deployment descriptor. In Listing 7-6, the `container-transaction` element specifies that the remote methods of `Teller` should always be protected by a transaction. This causes the container to generate or otherwise simulate the bolded code from Listing 7-7. Whenever a client calls a method on `Teller`'s remote interface, the container will create a transaction object. If `Teller` then calls out to other objects such as `Money` and `Account`, the container will propagate the transaction to these components as well. As a result, all of the work done on behalf of a `Teller` remote method can be bound to the same transaction, even if dozens of other objects are involved.

Thanks to the deployment descriptor, control of the transaction is situated in a single location that is easy to maintain. There are other transaction settings, not shown here, that allow components to block the flow of a transaction, or to start a different transaction even if one transaction is already in process. The deployment descriptor also supports other cross-cutting aspects, such as security roles, and more aspects may be added in the future.

In order to create transactions and to guarantee that transactions flow from one component to another as specified by the descriptor, the EJB container needs to intercept all calls into an EJB. Containers typically do this by generating additional classes, either at deployment time or possibly even on-the-fly at run-time. Figure 7-4 shows the classes typically generated by an EJB container and their relationship to classes that you author.



**Figure 7-4** Classes generated by an EJB container

EJB generates stubs and skeletons that are similar to RMI stubs and skeletons. These classes are generated from type information and handle forwarding method calls around the network. EJB containers use the information in the deployment descriptor to generate the EJB home and EJB object, which handle aspects such as transactions and security before invoking the business logic of the bean itself. Clients make RMI connections to the EJB home and EJB object, *never* to the bean itself.

Relate Figure 7-4 back to the definition of a component given at the beginning of the book: “an independent unit of production and deployment that is



combined with other components to assemble an application.” An EJB is a single component that contains the following nine classes:

1. Home interface\*
2. Remote interface\*
3. Home stub
4. Remote stub
5. Home skeleton
6. Remote skeleton
7. EJB home
8. EJB object
9. The bean itself\*

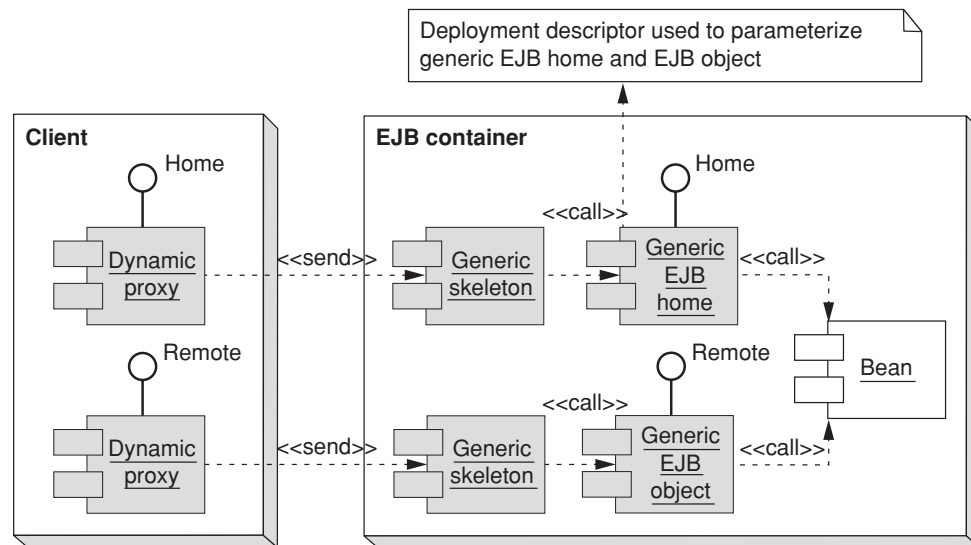
Of these nine classes, you only need to write the three labeled with asterisks—the interfaces and the bean. The other classes are generated from type information and from the data in the deployment descriptor. In other words, much of the work of authoring an EJB component is not writing Java code at all. Instead, you write some Java code, write some XML, and the container builds most of the Java code that eventually executes on both the client and the server.

### 7.6.2 Alternate Implementations

Not all EJB containers function exactly as described above. Many of the services provided by generated classes could also be provided in other ways. Figure 7–5 shows an EJB container that makes minimal use of code generation. The client-side stubs must be generated, but they could be generated at runtime using dynamic proxies. All of the generated classes on the server side have been replaced with generic classes that perform the same operations.

This is very different from the previous figure because the generic classes are not specific to any bean. Instead, they use reflection’s `Method.invoke` to call any EJB in a generic fashion. From a generative programming perspective, the difference is one of bind mode. The container architectures in Figure 7–4 and Figure 7–5 both bind the deployment descriptor during deployment (hence





**Figure 7-5 EJB without code generation**

the name), but the former uses a static binding mode, and the latter uses a dynamic binding mode. Given that both designs accomplish the same thing, why go to the trouble of generating the additional classes shown in Figure 7-4?

There are two reasons why all containers will use at least some code generation and many containers will rely heavily on it. First, code generation is necessary to preserve Java syntax on the client. Clients expect to call methods on the home and remote interfaces, which means that stubs *must* be generated. The only real question is the binding time.

Second, code generation permits optimizations that are difficult or impossible with generic code. This is the crux of the matter. If the container provides a generic service layer for components, as in Figure 7-5, then the deployment descriptor must be bound into the code by passing parameters at runtime. Passing and interpreting these parameters takes time, possibly on every method invocation. If the container generates the code instead, then it has to deal with the specifics of the component only once, at generation time.

Actually writing performance tests to compare different techniques for implementing an EJB container would be an interesting task, but it would take us very far afield. Instead, the next examples will evoke similar issues within a



scope amenable to some simple performance tests. You will see techniques to statically bind strongly typed collections and serialization, both of which are normally dynamically bound in Java. In both of these examples, code generation will replace generic code with code specifically tuned to the task at hand. This will boost performance by eliminating repetitive tasks at runtime.

If you plan to use code generation to improve the performance of an application, you should beware of the possible tradeoffs. In theory, generated code can run faster because you can hard-code values that might otherwise be parameters, or remove levels of indirection that would otherwise be implemented as virtual methods. However, these benefits must be offset against the additional effort to generate code, and the increased memory footprint if you generate several blocks of very similar code. Which of these factors will predominate is project and virtual machine dependent, but for many simple tasks code generation provides a clear performance advantage. Also, it is often easier to generate a specific solution than to code the logic needed to implement a generic solution.

## 7.7 Generating Strongly Typed Collections

As a simple example of the performance tradeoffs that drive a GP design, consider the collection classes in the `java.util` package. The various collections (`ArrayList`, `HashMap`, etc.) are all of type `Object`—in other words, the collections are entirely generic. If you want to use a collection in a type-safe fashion, then you must write additional code to enforce type safety at runtime, as shown here:

```
// Must pay runtime cost of casting to String
// also possible that cast might fail
String value = (String) stack.pop();
```

One workaround to this problem is to write your own strongly typed collections, such as hand-coded `StringStack`, `IntStack`, and so on. Such work is tedious, error-prone, and better suited to code generation. Since all the conceivable strongly typed `Stacks` look mostly the same, it is straightforward to generate the source code for them. One approach is to write a JSP page based on the source code for the generic version of the collection. Listing 7–8 shows a JSP

page that extracts parameters from an HTTP query string and uses them to generate a specific stack class. Because the source code for different stack classes is similar, most of the JSP page is simply static text.

### Listing 7-8 JSP Page That Generates Strongly Typed Stack Classes

```
<%
response.setContentType("text/plain");
String packageName = request.getParameter("package");
String type = request.getParameter("type");
String name=request.getParameter("name");
if ((packageName == null) || (type == null) || (name == null)) {
    throw new Error("must specify package, name, and type");
}
%>
package <%= packageName %>;

import java.util.*;
/**
 * This stack class was generated by StronglyTypedStack.jsp.
 *
 * @author Stuart Halloway
 */
public
class <%= name %>Stack extends Vector {
    public <%= name %>Stack() {

        public <%= type %> push(<%= type %> item) {
addElement(item);
return item;
        }
        public synchronized <%= type %> pop() {
<%= type %> obj;
int len = size();
obj = peek();
removeElementAt(len - 1);
return obj;
        }
        public synchronized <%= type %> peek() {
int len = size();
if (len == 0)
    throw new EmptyStackException();
```

```

        return (<%= type %>) elementAt(len - 1);
    }
    public boolean empty() {
        return size() == 0;
    }
    public synchronized int search(<%= type %> o) {
        int i = lastIndexOf(o);
        if (i >= 0) {
            return size() - i;
        }
        return -1;
    }
}

```

The specification that drives this generator is simply three text strings: the package name, the new class name, and the name of the class that the stack holds. To generate a `StringStack` class in the `com.develop` package, you would install the JSP page in a JSP engine, and then browse to the site with the following HTTP request:

```

http://yoursite/yourwebapp/StronglyTypedStack.jsp?
package=com.develop&type=java.lang.String&name=String

```

Then, you would simply paste the text content of your browser into a Java source code file and compile. To complete this example, you would want to automate the entire process from a build tool. You could eliminate the browser from the equation by writing your own simple HTTP client that automates connecting to the server, retrieving the source code, and saving it to file.

Strongly typed stack classes have two potential advantages over the generic `java.util.Stack` class. First, they enforce correct usage at compile time by type-checking the references involved in the `push` and `pop` operations. Second, the generated classes can be made more efficient than the generic `Stack` class. The generator shown in Listing 7–8 can provide the former, but not the latter. The generated code leverages the weakly typed `Vector` class so that it still has to execute a type cast for every `pop` operation.

Of course, nothing limits you to a single flavor of generator. Now that you have defined a specification, you could create a second generator that provides both advantages. This second generator would generate a strongly typed

`Vector` base class that uses a strongly typed array as its backing store. This would avoid type casts and provide a slight performance advantage over the standard API classes. Of course, the advantages of the generated code must be weighed against increased memory usage. The generated `Stack` and `Vector` classes take extra space in memory, possibly quite a bit of space if you generated dozens or hundreds of different varieties.

I am not going to evaluate the various arguments for and against strongly typed collections here. The weight you give to the various pros and cons will depend strongly on the specifics of your project. The important issue here is not which option you choose, but rather that you have multiple options available. Code generation makes it easy to try the various possibilities without having to hand code them all. The JSP example shown earlier generates only a single class, but you could design an entire project to include a parameterized generation step as part of the build process. If you have a compiler available, you could even generate different versions of your component at runtime based on the specifics of the current environment.

### 7.7.1 Code Generation Language versus Target Language

The environment you use to generate code need not have anything in common with the environment you are generating code *for*. In this example, using JSP as the generator language for Java code has several advantages. JSP has a well-known syntax, and implementations are freely available.<sup>7</sup> Also, JSP is more convenient than using Java to generate code, especially when the generation is driven mostly from a static template.

JSP also has several noteworthy disadvantages stemming from the fact that the language was not originally intended for code generation. As used here, JSP requires a separate server process, expects arguments to be passed as HTTP `GET` parameters, and returns a single source file. For generating source code, you might prefer to have a tool that runs as part of a build process, provides a convenient (or even customizable) syntax for the specification, and returns an entire collection of files. With additional effort you could coerce a JSP engine to

---

7. The reference implementation of JSP is open source; see [Jakarta].

do all of these things, or you might consider just writing your generator from scratch. The next section gives an example of a simple generator written in straight Java code.

## 7.8 Generating Custom Serialization Code

Serialization is the perfect example of a generic service. Simply mark your object as `Serializable`, and at runtime the `ObjectOutputStream` class and friends will use reflection to extract/construct your object's instance state. Unfortunately, default serialization's heavy use of reflection imposes a performance penalty that is noticeable in some situations. In Chapter 4, you saw several options for manually customizing serialization. Some of these options could be used to improve serialization performance, but they would require you to hand-author the serialization code.

Generative programming offers an attractive middle ground. A generator can use reflection to generate custom code that is more efficient than default serialization. If done properly, this provides the best of both worlds. The serialization code is fast because it is compiled into the object, and it is error-free because it is generated directly from type information.

Consider the `Externalizer` class shown in Listing 7-9. `Externalizer` uses reflection to analyze a preexisting class and generate appropriate source code for `readExternal`, `writeExternal`, and `serialVersionUID`. The `serialVersionUID` is calculated trivially by calling an accessor method on serialization's `ObjectStreamClass` representation of the class. The `readExternal` and `writeExternal` methods are calculated by iterating over a class's serializable fields to produce the appropriate calls to read and write methods. The base class `GeneratorBase` (not shown in the listing) provides helper methods that open a Java file and insert the generated code.

### Listing 7-9 The Externalizer

```
package com.develop.generators;

import java.io.*;
import java.lang.reflect.*;
```

```

public class Externalizer extends GeneratorBase {

    public String primitiveName(Class cls) {
        if (!cls.isPrimitive()) {
            throw new IllegalArgumentException(cls +
                " is not primitive");
        }
        String name = cls.getName();
        return name.substring(0,1).toUpperCase() +
            name.substring(1);
    }

    public void readExternal(Class cls, Field[] fields) {
        indentPrint("public void readExternal(ObjectInput oi) " +
            "throws ClassNotFoundException, IOException {"", 0);
        if (cls.getSuperclass() != Object.class)
            indentPrint("super.readExternal(oi);", 1);
        for (int n=0; n<fields.length; n++) {
            Field f = fields[n];
            if (0 != (f.getModifiers() &
                (Modifier.STATIC + Modifier.TRANSIENT)))
                continue;
            Class fldClass = f.getType();
            if (fldClass.isPrimitive()){
                indentPrint(f.getName() + " = oi.read" +
                    primitiveName(fldClass) + "();", 1);
            } else if (fldClass == String.class) {
                indentPrint(f.getName() + " = oi.readUTF();", 1);
            } else {
                indentPrint(f.getName() + " = oi.readObject();", 1);
            }
        }
        indentPrint("}", 0);
    }

    public void writeExternal(Class cls, Field[] fields) {
        indentPrint("public void writeExternal(ObjectOutput oo) "+
            "throws IOException {"", 0);
        if (cls.getSuperclass() != Object.class)
            indentPrint("super.writeExternal(oi);", 1);
        for (int n=0; n<fields.length; n++) {
            Field f = fields[n];
            if (0 != (f.getModifiers() &
                (Modifier.STATIC + Modifier.TRANSIENT)))

```

```

        continue;
        Class fldClass = f.getType();
        if (fldClass.isPrimitive()){
            indentPrint("oo.write" + primitiveName(fldClass) +
                "(" + f.getName() + ");", 1);
        } else if (fldClass == String.class) {
            indentPrint("oo.writeUTF(" + f.getName() + ");", 1);
        } else {
            indentPrint("oo.writeObject(" + f.getName() + ");", 1);
        }
    }
    indentPrint("}", 0);
}

public void serialVersionUID(Class cls) {
    ObjectOutputStream ocs = ObjectOutputStream.lookup(cls);
    indentPrint("private final static long serialVersionUID="
        + ocs.getSerialVersionUID() + "L;", 0);
}

public void generate(Class cls, PrintStream out) {
    this.out = out;
    beginGenerated();
    serialVersionUID(cls);
    Field[] fields = cls.getDeclaredFields();
    writeExternal(cls, fields);
    readExternal(cls, fields);
    endGenerated();
}
}

```

Listing 7–10 shows a simple `SerializeMe` class after it was modified by the `Externalizer`. The code generated by the `Externalizer` is shown here in bold. The modified version of `SerializeMe` will serialize and deserialize more efficiently than the original version because there is no need to use reflection at runtime to access the class's type information or instance fields. At the same time, you can rely on the correctness of the code because it is generated directly from type information. If you want to use a large number of `Externalizable` classes in your application, you should take the `Externalizer` (or something like it) and make it part of your build process.

**Listing 7-10 SerializeMe after Modification by Externalizer**

```

import java.io.*;

public class SerializeMe implements Externalizable {
    public SerializeMe() {
        i = 1;
        f = 10;
        l = 100;
        d = 1000;
        s = "serialize me ";
    }

    int i;
    float f;
    long l;
    double d;
    String s;

    /**{@@ BEGIN CODE GENERATION BY class Externalizer @@}}
    //edit at your own risk...
    private final static long serialVersionUID=
        -2726536721571465800L;
    public void writeExternal(ObjectOutput oo) throws IOException {
        oo.writeInt(i);
        oo.writeFloat(f);
        oo.writeLong(l);
        oo.writeDouble(d);
        oo.writeUTF(s);
    }
    public void readExternal(ObjectInput oi)
        throws ClassNotFoundException, IOException {
        i = oi.readInt();
        f = oi.readFloat();
        l = oi.readLong();
        d = oi.readDouble();
        s = oi.readUTF();
    }
    /**{@@ END CODE GENERATION BY class Externalizer @@}}
}

```

Another approach to generating serialization code is shown in Listing 7-11. This version of `SerializeMe` contains serialization code from a different generator.



The `UnreflectiveSerialize` generator<sup>8</sup> uses the `GetField` and `PutField` hooks to modify default serialization. With these hooks, metadata is included in the stream format just as if the class had been serialized using the default mechanism.

#### Listing 7-11 `SerializeMe` after Modification by `UnreflectiveSerialize`

```
import java.io.*;
public class SerializeMe implements Serializable {
    public SerializeMe() {
        i = 1;
        f = 10;
        l = 100;
        d = 1000;
        s = "serialize me ";
    }

    int i;
    float f;
    long l;
    double d;
    String s;

    /**{@@ BEGIN CODE GENERATION BY class UnreflectiveSerialize
    //edit at your own risk...
    private final static long serialVersionUID=
        -2726536721571465800L;
    private static final ObjectOutputStream[] serialPersistentFields =
    {
        new ObjectOutputStream("i", int.class),
        new ObjectOutputStream("f", float.class),
        new ObjectOutputStream("l", long.class),
        new ObjectOutputStream("d", double.class),
        new ObjectOutputStream("s", java.lang.String.class),
    };
    private void writeObject(ObjectOutputStream oos)
        throws IOException {
        ObjectOutputStream.PutField pf = oos.putFields();
        pf.put("i", i);
        pf.put("f", f);
```

---

8. The `UnreflectiveSerialize` generator is not shown here for brevity, but it is included with the book's source code.

```

        pf.put("l", l);
        pf.put("d", d);
        pf.put("s", s);
        pf.write(oos);
    }
    private void readObject(ObjectInputStream ois)
        throws ClassNotFoundException, IOException {
        ObjectInputStream.GetField gf = ois.readFields();
        i = gf.get("i", 0);
        f = gf.get("f", 0.0f);
        l = gf.get("l", 0L);
        d = gf.get("d", 0.0);
        s = (java.lang.String) gf.get("s", null);
    }
    //{@@ END CODE GENERATION BY UnreflectiveSerialize @@}
}

```

The `UnreflectiveSerialize` generator is helpful if you need to support multiple versions of a class over time. Imagine that you have a large data class with a few dozen fields. If only a few fields change, most of the code in `readObject` and `writeObject` will look exactly like the code generated by `UnreflectiveSerialize`. You could generate the basic code, and then hand-edit only the few lines that need to change. This demonstrates a general principle: Code generation during development is flexible because you can fix problems by hand. Therefore, a 90 percent solution is far better than nothing at all. By contrast, code generation at runtime must be exact, since no developer is present to adjust it.

## 7.9 Onward

In this chapter you have seen several widely different examples of generative programming. In all the examples, the goal is to reuse your knowledge of the problem domain. GP works in tandem with, not in opposition to, traditional OO techniques for reuse. Some generative schemes build bytecode, like dynamic proxies. Others build source code, like the serialization example. Some code generation tools bind your choices at runtime, like JSP. Others bind at deployment time or compile time, like the RMI stub compiler.

Sometimes your commitment to generative programming is pervasive and tied into special support libraries. For example, a component either is an EJB, and it requires all the associated container goo, or it is not. On the other end of the spectrum, you may use simple code generation helpers like interface wizards at the level of a single file, without any implication for your overall application architecture. Code generation inputs range from existing Java classes (dynamic proxies), to classes plus additional metadata (EJB), to custom languages (JSP).

Given the wide array of options, you need some way to impose order onto chaos and choose generative techniques that are appropriate for your own applications. This section will make some suggestions as to when code generation is useful, and how you should decide which types of tools to employ.

The most obvious example of when code generation is useful is in providing *generic service components*. A generic service component is a component that can add functionality to other components without having compile-time knowledge of the components that it will be working with. Many of the examples in this chapter fit this description. Dynamic proxies are generated at runtime to implement interfaces that were not previously known. Serialization streams the state of an object without advance knowledge of the object's fields. EJB containers add transactional semantics to objects without knowing in advance what methods the objects may have.

Consider EJB first. EJB requires code generation in order to ensure type safety. Refer back to Figure 7-4. Without the generated stubs, clients would be forced to use some sort of generic invocation mechanism. With the generated stubs, clients are able to communicate via a well-known interface instead.

EJB code generation tools can be used at any time during the component lifecycle. Depending on your container, you might generate the support classes during development, at deployment, at runtime, or some combination of all three. The reason for this flexibility is that EJBs are server-side code. The only client-side components are stubs, which can be downloaded dynamically from the server anyway. Because EJB is a server-side technology, you can reasonably expect access to a compiler and whatever other tools you may need to generate the stubs, EJB home, and EJB object. These tools are available throughout

the component lifecycle, up to and including runtime. Since a compiler is available, you do not have to worry about generating bytecode directly. You can generate source code and then run it through the compiler.

Dynamic proxies, by contrast, are much more constrained. Because they are part of the core API, proxies must work in all sorts of Java environments. A compiler will not always be available, so proxies cannot count on compiling source code. Instead, they must generate bytecode directly. This makes dynamic proxies much more difficult to generate than the EJB support classes.<sup>9</sup> Also, dynamic proxies must be generated at runtime since they are created in response to an API call at runtime. Generating code later in the development cycle is more flexible for the user but more difficult for the developer.

As another example of this principle, consider the simple serialization code generators shown in §7.8. In all likelihood, you would use these tools at development time. This is more convenient for the developer since the generated code does not even have to be 100 percent correct—you can always edit it to fix small problems. However, it is less flexible for users of the object because serialization semantics are frozen into the object during development. By contrast, JSP code generators often execute at runtime. This allows even a web administrator to make cosmetic changes to the appearance of a page, without shutting down the web server.<sup>10</sup>

Another issue to consider is the type of inputs required by a code generation scheme. Many of the examples in this chapter require only the type information that is available to any Java object. However, some of the examples add their own metadata as well. EJB functionality is controlled by external XML deployment descriptors, and JSP provides an entire separate syntax with occasional escapes to Java code. In general, code generation schemes that build Java code from non-Java data have one or more of the following properties:

1. A problem domain that is well understood and repetitive
2. Syntaxes more expressive than Java in solving the problem
3. Special support libraries that are called from generated code

---

9. Consider how many programmers write in bytecode instead of Java. Or, compare the number of dynamic proxy implementations with the number of EJB implementations.

10. Whether this is a good idea or not is a site management decision.

Both EJB and JSP have the first property. EJBs repeat the same sequence over and over: check security, acquire transaction, use data, commit/abort. All of the steps other than “use data” are well understood and repetitive to code. JSP pages tend to do the same things again and again, such as generating a standard HTML structure to fill in with user-requested content. JSPs are also a good example of the second property. If most of the work on a page is static content, then Java code degenerates into a boring sequence of write instructions. EJB exemplifies the third property. You do not need to explain *how* the support libraries should implement transactions or security checks; instead, you need only *declare* the parameters to be used. Switching from a functional to a declarative approach also changes the locus of decision-making. Changing a declarative setting does not require a programmer since no code changes.

## 7.10 Resources

The examples in this chapter only scratch the surface of generative programming. Moreover, they are biased toward services that can leverage type information. For a gentle, general introduction to generative techniques using Java and XML see [Cle01]. For a more complete treatment not limited to Java, see [CE00].

