C H A P T E R 5

Creational Patterns

Creational patterns provide guidance on how to create objects when their creation requires making decisions. These decisions will typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to. The value of creational patterns is to tell us how to structure and encapsulate these decisions.

Often, there is more than one creational pattern that you can apply to a situation. Sometimes you can combine multiple patterns advantageously. In other cases, you must choose between competing patterns. For these reasons, it is important to be acquainted with all of the patterns described in this chapter.

If you have time to learn only one pattern in this chapter, the most commonly used one is Factory Method. The Factory Method pattern is a way for an object to initiate the creation of another object without having to know the class of the object created.

The Abstract Factory pattern is a way for objects to initiate the creation of a variety of different kinds of objects without knowing the classes of the objects created, but ensuring that the classes are a correct combination.

The Builder pattern is a way to determine the class of an object created by its contents or context.

The Prototype pattern allows an object to create customized objects without knowing their exact class or the details of how to create them.

The Singleton pattern is a way for multiple objects to share a common object without having to know whether it already exists.

The Object Pool pattern is a way to reuse objects rather than create new objects.

Factory Method

This pattern was previously described in [GoF95].

SYNOPSIS

You need to create an object to represent external data or process an external event. The type of object depends on the contents of the external data or type of event. You want neither the data source, the event source, nor the object's clients to be aware of the actual type of object created. You encapsulate the decision of what class of object to create in its own class.

CONTEXT

Consider the problem of writing a framework for desktop applications. Such applications are typically organized around documents or files. Their operation usually begins with a command to create or edit a word processing document, spreadsheet, time line, or other type of document the application is intended to work with.

A framework to support this type of application will include high-level support for common operations such as creating, opening, or saving documents. Such support will generally include a consistent set of methods to call when the user issues a command. For the purpose of this discussion, we will call the class providing the methods the `Application` class.

Because the logic to implement most of these commands varies with the type of document, the `Application` class usually delegates most commands to some sort of document object. The logic in document objects for implementing these commands varies with the type of document. However, some operations, such as displaying the title of a document, will be common to all document objects. This suggests an organization that includes:

- An application-independent document interface
- An abstract class that provides application-independent logic for concrete document classes
- Concrete application-specific classes that implement the interface for specific types of documents

Figure 5.1 shows this organization.

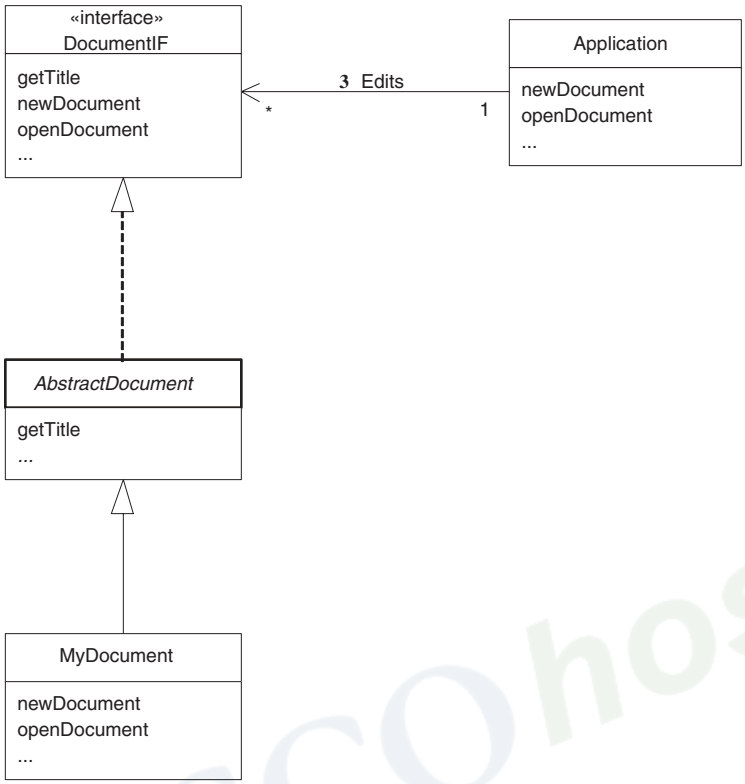

FIGURE 5.1 Application framework.

Neither Figure 5.1 nor the preceding discussion shows how an `Application` object creates instances of application-specific document classes without itself being application specific.

One way to accomplish this is for the programmer using the framework to provide a class that encapsulates logic for selecting and instantiating application-specific classes. For the `Application` class to be able to call the programmer-provided class without having any dependencies on it, the framework should provide an interface that the programmer-provided class must implement. Such an interface would declare a method that the programmer-provided class would implement to select and instantiate a class. The `Application` class works through the framework-provided interface and not with the programmer-provided class. Figure 5.2 shows this organization.

Using the organization shown in Figure 5.2, an `Application` object calls the `createDocument` method of an object that implements the `DocumentFactoryIF` interface. It passes a string to the `createDocument` method that allows it to infer which subclass of the `Document` class to

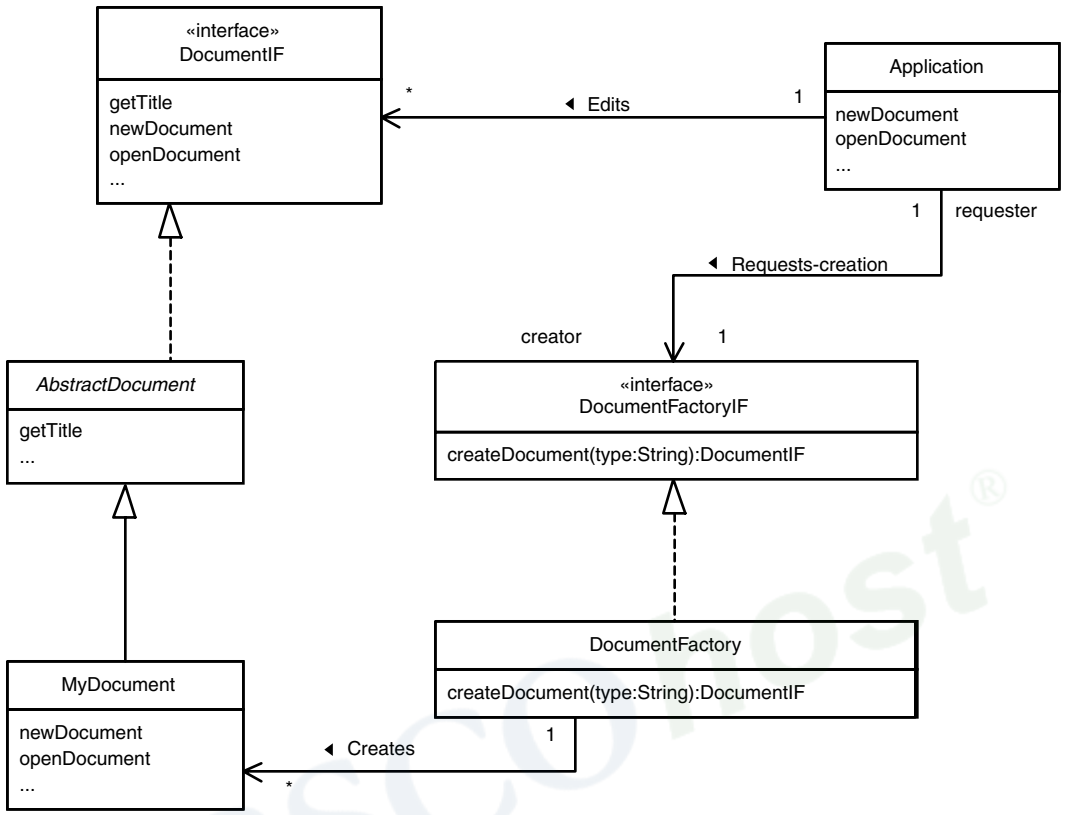

FIGURE 5.2 Application framework with document factory.

instantiate. The `Application` class does not need to know the actual class of the object whose method it calls or which subclass of the `Document` class it instantiated.

FORCES

- ☺ A class must be able to initiate the creation of objects without having any dependencies on the class of the created object.
- ☺ The set of classes a class may be expected to instantiate may be dynamic as new classes become available.

SOLUTION

Provide application-independent objects with an application-specific object to which they delegate the creation of other application-specific

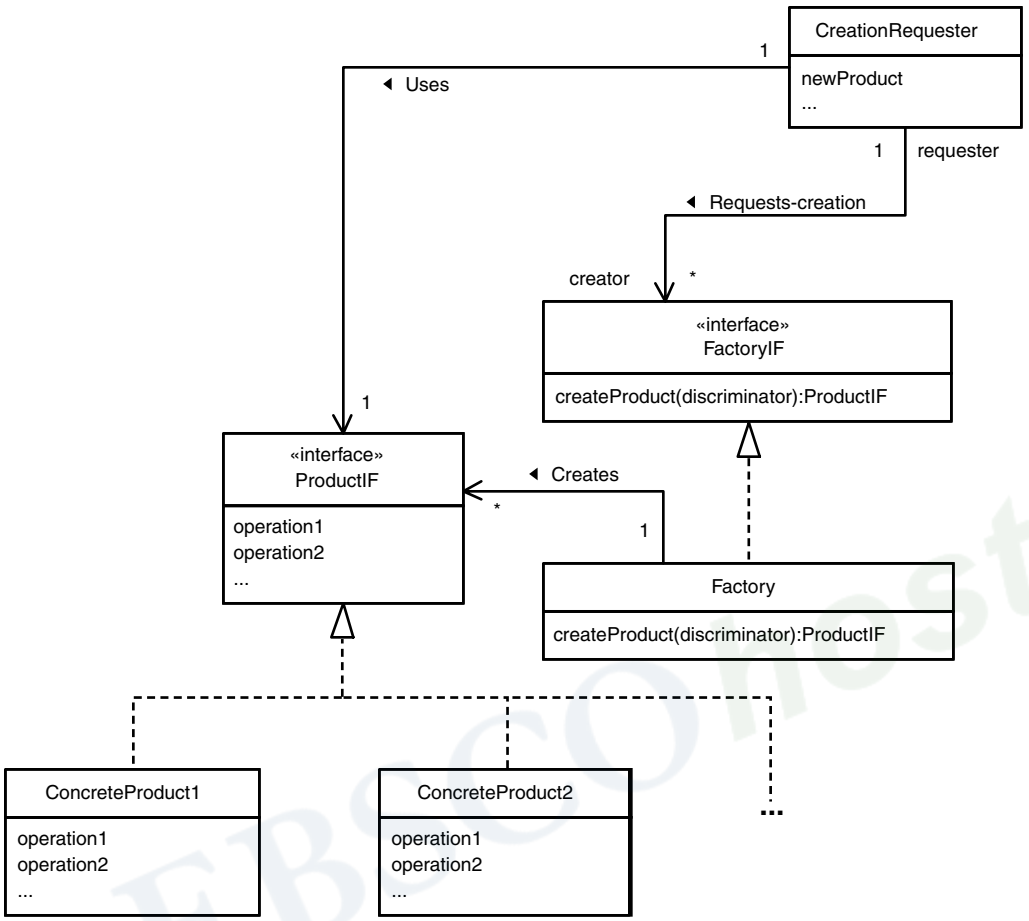

FIGURE 5.3 Factory Method pattern.

objects. Require the application-independent objects that initiate the creation of application-specific objects to assume that the objects implement a common interface.

Figure 5.3 shows the interfaces and classes that typically make up the Factory Method pattern.

The class diagram in Figure 5.3 shows the roles classes and interfaces play in the Factory Method pattern:

ProductIF. The objects created using this pattern must implement an interface in this role.

ConcreteProduct1, ConcreteProduct2, and so on. Classes in this role are instantiated by a `Factory` object. Classes in this role must implement the `ProductIF` interface.

CreationRequester. A class in this role is an application-independent class that needs to create application-specific classes. It does so indirectly through an instance of a class that implements the `FactoryIF` interface.

FactoryIF. This is an application-independent interface. Objects that create `ProductIF` objects on behalf of `CreationRequester` objects must implement this interface. Interfaces of this sort declare a method that can be called by a `CreationRequester` object to create concrete product objects. The arguments this method takes are discussed under the Implementation section for this pattern.

Interfaces filling this role will typically have a name that includes the word *Factory,* such as `DocumentFactoryIF` or `ImageFactoryIF`.

Factory. This is an application-specific class that implements the appropriate `FactoryIF` interface and has a method to create `ConcreteProduct` objects. Classes filling this role will typically have a name, such as `DocumentFactory` or `ImageFactory`, that contains the word *Factory*.

IMPLEMENTATION

In many implementations of the Factory Method pattern the `ConcreteProduct` classes do not directly implement the `ProductIF` interface. Instead, they extend an abstract class that implements the interface. See the Interface and Abstract Class pattern in Chapter 4 for a discussion of the reasons for this.

CLASS DETERMINATION BY CONFIGURATION

There are two main variations on the Factory Method pattern. There is the general case, where the class of the object to create is determined when object creation is initiated. There is also the less common case, where the class of objects that will be created is always the same and is determined before an object's creation is initiated.

A program may use a factory object that always creates an instance of the same class if the class is determined by some configuration information. For example, suppose that a company sells a point-of-sale system that is responsible for communicating with a remote computer to process credit card payments. Its classes expect to send messages to the remote computer and receive responses by using objects that implement a particular interface. The exact format of the messages to send will depend on the company that processes the credit card transactions. For each credit card

processing company, there is a corresponding class that implements the required interface and knows how to send the messages that company expects. There is also a corresponding factory class. This organization is shown in Figure 5.4.

Here is an explanation of the design shown in Figure 5.4. When the point-of-sale system starts, it reads its configuration information. The configuration information tells the point of sale that it is to pass credit card transactions either to BancOne or to Wells Fargo for processing. Based on this information, it creates either a `BancOneCCFactory` or a `WellsCCFactory` object. It accesses the object through the `CreditCardProcessorFactoryIF` interface. When it needs an object to process a credit card transaction, it calls the

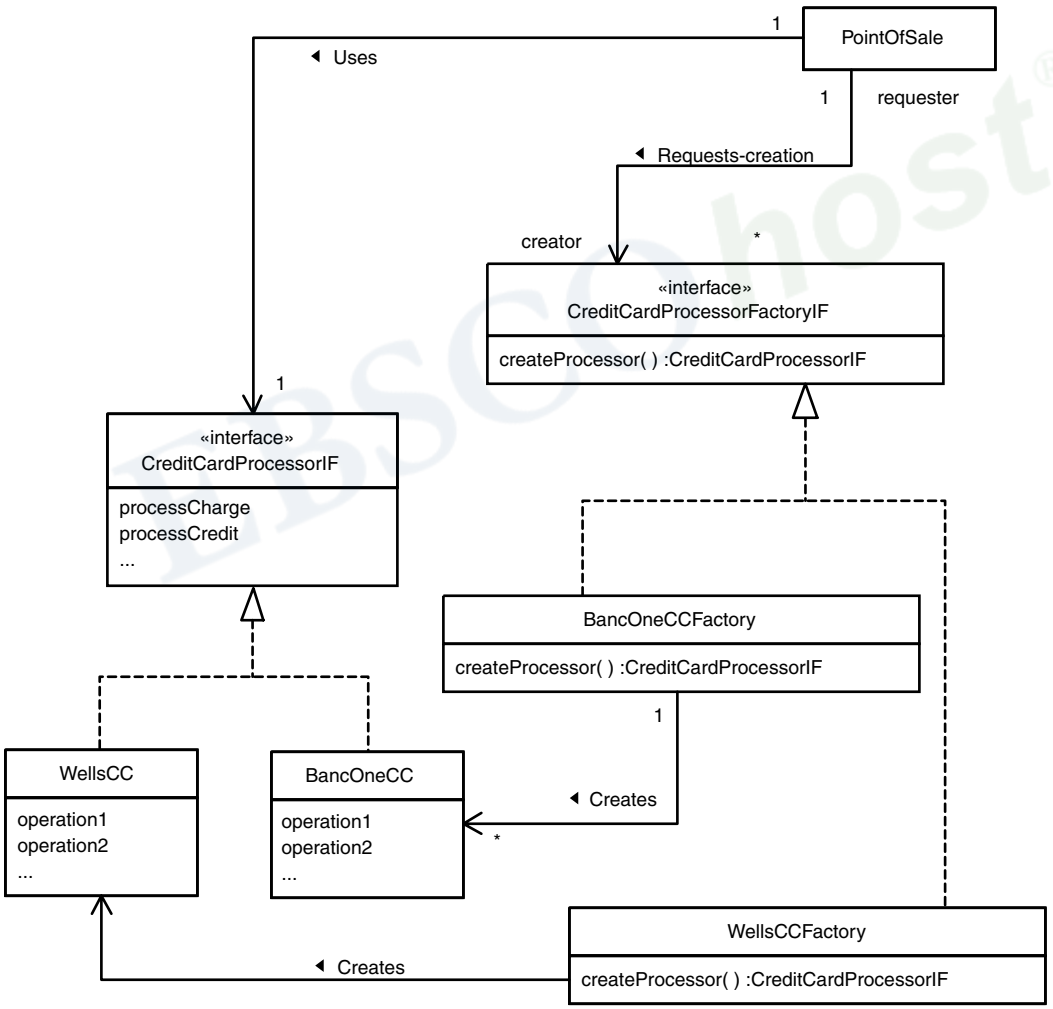

FIGURE 5.4 Credit card processing.

`createProcessor` method of its `CreditCardProcessorFactoryIF`, which creates an object that uses the configured credit card processing company.

Note that the factory class's Create method does not need any arguments because it always returns the same type of object.

DATA-DRIVEN CLASS DETERMINATION

Very often, the class of an object created by a factory object is determined by the data that it is intended to encapsulate. The class is determined by the factory object's `createProduct` method. The determination is usually based on information passed into the method as a parameter. Such `createProduct` methods often look something like this:

```
Image createImage (String ext) {
    if (ext.equals("gif")
      return new GIFImage();
    if (ext.equals("jpeg"))
      return new JPEGImage();
    ...
} // createImage(String)
```

This sequence of `if` statements works well for `createProduct` methods that have a fixed set of product classes to instantiate. To write a `createProduct` method that handles a variable or a large number of product classes, you can use the Hashed Adapter Objects pattern (described in *Patterns in Java, Volume 2*). Alternatively, you can use the various objects that indicate which class to instantiate as keys in a hash table with `java.lang.reflect.Constructor` objects for values. Using this technique, you look up an argument value in the hash table and then use the constructor object found in the hash table to instantiate the desired object.

Another point that the preceding example illustrates is that factory methods are a reasonable place to find `switch` statements or chains of `if` statements. In many situations, the presence of `switch` statements or chains of `if` statements in code indicates that a method should have been implemented as a polymorphic method. Factory methods cannot be implemented using polymorphism, because polymorphism works only after an object has been created.

For many implementations of the Factory Method pattern, the valid arguments to the factory object's `createProduct` method are a set of predetermined values. It is often convenient for the factory class to define symbolic names for each of those predetermined values. Classes that ask the factory class to create objects can use the constants that define the symbolic names to specify the type of object to be created.

Sometimes, there is more than one layer of data-driven class determination. For example, there may be a top-level factory object that is respon-

sible for creating the factory object that will create the actual product object. For this reason, the data-driven form of the Factory Method pattern is sometimes called *layered initialization*.[1]

CONSEQUENCES

The primary consequences of using the Factory Method pattern are as follows:

- ☺ The creation requester class is independent of the class of concrete product objects actually created.
- ☺ The set of product classes that can be instantiated may be changed dynamically.
- ☹ The indirection between the initiation of object creation and the determination of which class to instantiate can make it more difficult for maintenance programmers to understand.

JAVA API USAGE

The Java API uses the Factory Method pattern in a few different places to allow the integration of the applet environment with its host program. For example, each `URL` object has associated with it a `URLConnection` object. You can use `URLConnection` objects to read the raw bytes of a URL. `URLConnection` objects also have a method called `getContent` that returns the content of the URL packaged in an appropriate sort of object. For example, if the URL contains a `gif` file, then the `URLConnection` object's `getContent` method returns an `Image` object.

The way it works is that `URLConnection` objects play the role of creation requester in the Factory Method pattern. They delegate the work of the `getContent` method to a `ContentHandler` object. `ContentHandler` is an abstract class that serves as a product class that knows about handling a specific type of content. The way that a `URLConnection` object gets a `ContentHandler` object is through a `ContentHandlerFactory` object. The `ContentHandlerFactory` class is an abstract class that participates in the Factory Method pattern as a factory interface. The `URLConnection` class also has a method called `setContentHandlerFactory`. Programs that host applets call that method to provide a factory object used for all `URLConnection` objects.

[1] In the first edition of this volume, layered initialization was documented as a separate pattern. Reader feedback convinced the author that this point of view was confusing rather than helpful.

CODE EXAMPLE

For our example, suppose you are developing an application to process records in a journal file created by a point-of-sale system.[2] Every line that appears on every register tape has a corresponding record in the journal file. Your application will read all of the records in a journal file and generate summaries of the transactions in the journal file.

Your application will be required to work with point-of-sale systems from multiple manufacturers. Since the journal file produced by each manufacturer's point-of-sale system has a different format, it is especially important to keep the classes responsible for generating the summaries independent of the journal file format.

You notice that all of the journal file formats consist of a sequence of records. The types of records in the journal files are similar. You decide to make the summary-generating classes independent of the journal file formats by representing the journal files internally as a sequence of objects that correspond to the records in the journal file. With this in mind, you design a set of classes that correspond to different types of records that occur in the journal files.

The focus of this example is how your application will create the objects that represent the records in the journal files. This example uses both forms of the Factory Method pattern.

- As the application reads records from a journal file, it uses a factory object to create the objects that correspond to each record in the journal file. Each time the factory object is asked to create an object, it selects the class to instantiate based on information in the record. This is an example of a factory that performs runtime class determination.
- The class of the factory object that creates objects to represent records depends on the type of journal file that the application will be reading. Because the class of this factory object depends on configuration information, it is created by another factory object when the application reads its configuration information.

Figure 5.5 shows the class organization on which this code example is based. Here are descriptions of the classes and interfaces shown in Figure 5.5:

JournalRecordIF. Objects that encapsulate the contents of journal records are an instance of a class that implements this interface.

[2] A point-of-sale system is a high-tech cash register.

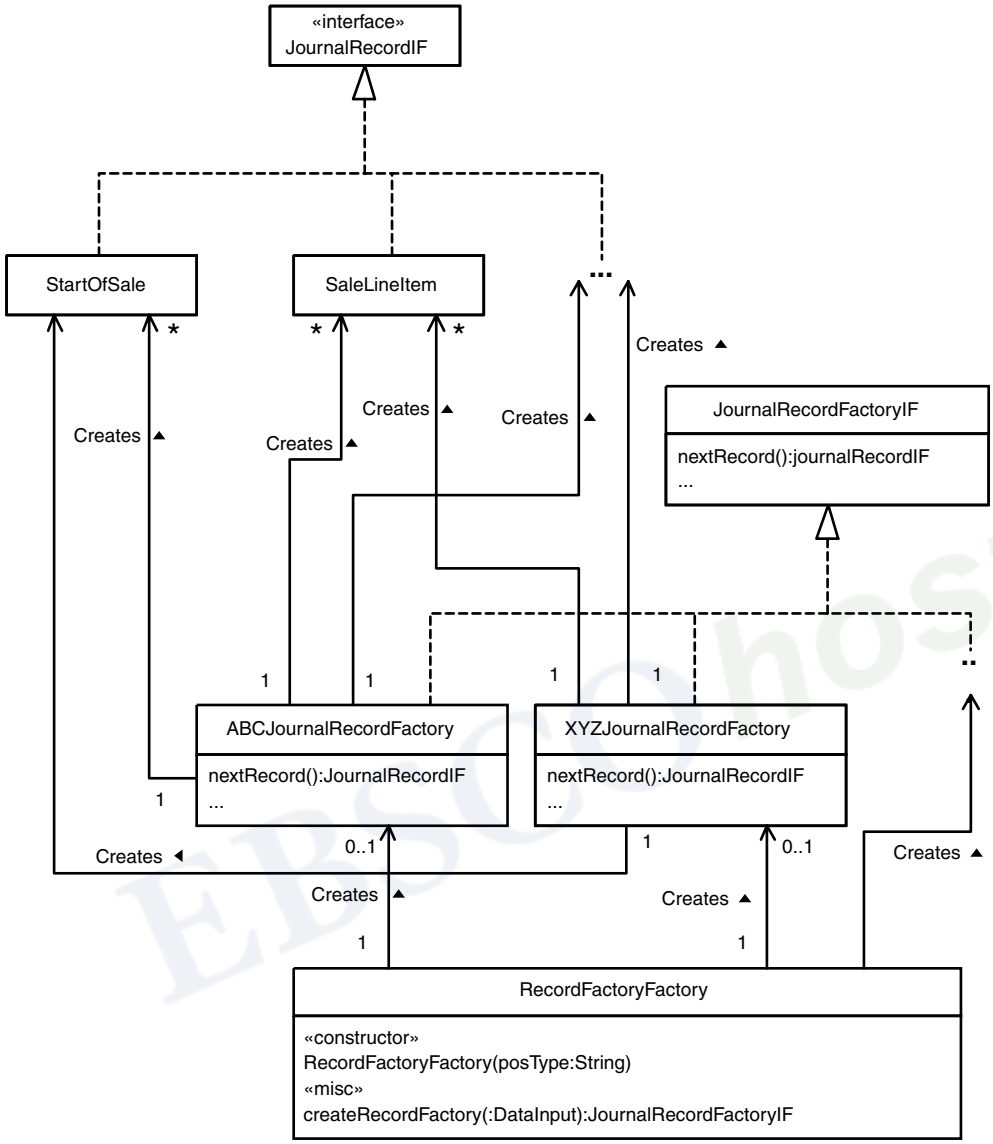

FIGURE 5.5 Journal-file-related factory classes.

Only two such classes are shown in Figure 5.5. However, in the complete application there would be many more.

StartOfSale. Instances of this class represent a journal record that indicates the beginning of a sale transaction.

SaleLineItem. Instances of this class represent a journal record that contains the details of a sale item on a register receipt.

JournalRecordFactoryIF. This interface is implemented by classes responsible for taking the information in journal file records and encapsulating it in objects that implement the `JournalRecordIF` interface. Instances of classes that implement this interface create objects that implement the `JournalRecordIF` interface, such as `StartOfSale` objects and `SaleLineItem` objects.

ABCJournalRecordFactory. Instances of this class are responsible for understanding the format of journal files produced by ABC point-of-sale systems. They read the next journal record from an object that implements `java.io.DataInput` and encapsulate its contents in an instance of a class that implements the `JournalRecordIF` interface.

XYZJournalRecordFactory. Instances of this class are responsible for understanding the format of journal files produced by XYZ point-of-sale systems. They read the next journal record from an object that implements `java.io.DataInput` and encapsulate its contents in an instance of a class that implements the `JournalRecordIF` interface.

RecordFactoryFactory. The name of a type of point-of-sale system is passed to this class's constructor. The constructed object is then used to create instances of the appropriate class that implements the `JournalRecordIF` interface for the specified type of point-of-sale system.

You may notice some structural similarity between this example that uses both forms of the Factory Method pattern and the next pattern in this book, the Abstract Factory pattern. One of the distinguishing characteristics of the Abstract Factory pattern is that the client objects are concerned with the interface implemented by the objects to be created rather than with their content.

Now let's look at the code that implements this design. We will begin with a listing of the `RecordFactoryFactory` class.

```
public class RecordFactoryFactory {
    private Constructor factoryConstructor;
```

When an instance of this class is created, its constructor is passed a string indicating the type of point-of-sale system for which the object will be a factory. It gets a `java.lang.reference.Constructor` object that it will use to construct objects and make the `Constructor` object the value of the `factoryConstructor` instance variable.

```
    // POS Types
    public static final String ABC = "abc";
    //...
```

```
    /**
     * Constructor
     *
     * @param posType
     *        The type of point-of-sale system that this object
     *        will be creating JournalRecordFactoryIF object
     *        for.
     * @throws POSException
     *         If there is a problem initializing this object.
     */
    public RecordFactoryFactory(String posType)
                                            throws POSException {
        Class[] params = { DataInput.class };
        Class factoryClass;
        if (ABC.equals(posType)) {
            factoryClass = ABCJournalRecordFactory.class;
        ...
        } else {
            String msg = "Unknown POS type: " + posType;
            throw new java.lang.IllegalArgumentException(msg);
        }
        try {
            factoryConstructor
              = factoryClass.getConstructor(params);
        } catch (Exception e) {
            String msg = "Error while constructing factory";
            throw new POSException(msg, e);
        } // try
    } // constructor(String)
```

Here is the method that creates `JournalRecordFactoryIF` objects.

```
    public JournalRecordFactoryIF createFactory(DataInput in)
                                          throws POSException {
        Object[] args = {in} ;
        Object factory;
        try {
            factory = factoryConstructor.newInstance(args);
        } catch (Exception e) {
            String msg = "Error creating factory";
            throw new POSException(msg, e);
        } //
        return (JournalRecordFactoryIF)factory;
    } // createFactory
} // class RecordFactoryFactory
```

Here is the `JournalRecordFactoryIF` interface.

```
/**
 * This interface is implemented by classes that are
 * responsible for creating objects that encapsulate the
 * contents of a point-of-sale journal file record.
 */
```

```
public interface JournalRecordFactoryIF {
    /**
     * Return an object that encapsulates the next record in a
     * journal file.
     *
     * @throws EOFException
     *         If there are no more records in the journal
     *         file.
     * @throws IOException
     *         If there is any problem reading the next
     *         record.
     */
    public JournalRecordIF nextRecord() throws EOFException,
                                               IOException;

    ...
} // interface JournalRecordFactoryIF
```

Here is the class `ABCJournalRecordFactory` that implements the `JournalRecordFactoryIF` interface for a type of point-of-sale system named ABC.

```
public class ABCJournalRecordFactory
                    implements JournalRecordFactoryIF {
    // Record Types
    ...
    private static final String SALE_LINE_ITEM = "17";
    private static final String START_OF_SALE = "4";
    ...

    private DataInput in;
    private final SimpleDateFormat dateFormat
      = new SimpleDateFormat("yyyyMMddHHmmss");

    // Counter for sequence number;
    private int sequenceNumber = 0;

    ABCJournalRecordFactory(DataInput input) {
        in = input;
    } // constructor(DataInput)

    /**
     * Return an object that encapsulates the next record in a
     * journal file.
     *
     * @throws EOFException
     *         If there are no more records in the journal
     *         file.
     * @throws IOException
     *         If there is a problem reading the file.
     */
    public JournalRecordIF nextRecord() throws EOFException,
                                               IOException {
```

```
            String record = in.readLine();
            StringTokenizer tokenizer;
            tokenizer = new StringTokenizer(record, ",");
            sequenceNumber++;

            try {
                String recordType = tokenizer.nextToken();
                ...
                if (recordType.equals(START_OF_SALE)) {
                    return constructStartOfSale(tokenizer);
                } else if (recordType.equals(SALE_LINE_ITEM)) {
                    return constructSaleLineItem(tokenizer);
                ...
                } else {
                    String msg = "Unknown record type";
                    throw new IOException(msg);
                } // if
            } catch (NoSuchElementException e) {
                // We will treat this exception as an I/O problem
                // since the record does not have all of the
                // expected fields.
                String msg = "record is missing some fields";
                IOException ioe = new IOException(msg);
                ioe.initCause(e);
                throw ioe;
            } // try
        } // nextRecord()

        private
        StartOfSale constructStartOfSale(StringTokenizer tok)
                                throws NoSuchElementException {
            String transactionID = tok.nextToken();
            tok.nextToken();        // Skip mode indicator.
            String timestampString = tok.nextToken();
            Date timestamp = parseTimestamp(timestampString);
            String terminalID = tok.nextToken();

            return new StartOfSale(terminalID,
                                sequenceNumber,
                                timestamp,
                                transactionID);
        } // constructStartOfSale(StringTokenizer)
        ...
    } // class ABCJournalRecordFactory
```

RELATED PATTERNS

Hashed Adapter Objects. The Hashed Adapter Objects pattern (described in *Patterns in Java, Volume 2*) can be used in the implementation of the Factory Method pattern. It is useful if the

set of classes that a factory object instantiates may change during the running of a program.

Abstract Factory. The Factory Method pattern is useful for constructing individual objects for a specific purpose without the construction requester knowing the specific classes being instantiated. If you need to create a matched set of such objects, then the Abstract Factory pattern is a more appropriate pattern to use.

Template Method. When the Factory Method pattern is implemented to determine the type of what will be created using configuration information, the implementation often uses the Template Method pattern.

Prototype. The Prototype pattern provides an alternate way for an object to work with other objects without knowing the details of their construction.

Strategy. If you are considering using the Factory Method pattern to vary behavior, the Strategy pattern may be a better alternative.

Abstract Factory

Abstract Factory is also known as Kit or Toolkit.

This pattern was previously described in [GoF95].

SYNOPSIS

Given a set of related interfaces, provide a way to create objects that implement those interfaces from a matched set of concrete classes. The Abstract Factory pattern can be very useful for allowing a program to work with a variety of complex external entities such as different windowing systems with similar functionality.

CONTEXT

Suppose you have the task of building a user-interface framework that works on top of multiple windowing systems, such as Windows, Motif, or MacOS. It must work on each platform with the platform's native look and feel. You organize it by creating an abstract class for each type of widget (text field, pushbutton, list box, etc.) and then writing a concrete subclass of each of those classes for each supported platform. To make this robust, you need to ensure that all the widget objects created are for the desired platform. This is where the abstract factory comes into play.

An abstract factory class defines methods to create an instance of each abstract class that represents a user-interface widget. Concrete factories are concrete subclasses of an abstract factory that implements its methods to create instances of concrete widget classes for the same platform.

In a more general context, an abstract factory class and its concrete subclasses organize sets of concrete classes that work with different but related products. For a broader perspective, consider another situation.

Suppose you are writing a program that performs remote diagnostics on computers made by a manufacturer called Stellar Microsystems. Over time, Stellar has produced computer models having substantially different architectures. Their oldest computers used CPU chips manufactured by Enginola that had a traditional complex instruction set. Since then, they have released multiple generations of computers based on their own reduced instruction set computer (RISC) architectures called ember, super-ember, and ultra-ember. The core components used in these models perform similar functions but involve different sets of components.

In order for your program to know which tests to run and how to interpret the results, it will need to instantiate objects that correspond to each core component in the computer being diagnosed. The class of each object will correspond to the type of component being tested. This means you will have a set of classes for each computer architecture. There will be a class in each set corresponding to the same type of computer component.

Figure 5.6 is a class diagram that shows the organization of the classes that encapsulate diagnostics for different kinds of components. It shows just two kinds of components. The organization for any additional kind of component would be the same. There is an interface for each type of component. For each computer architecture that is supported, there is a class that implements each interface.

The organization shown in Figure 5.6 leads us to the problem that this pattern solves. You need a way to organize the creation of diagnostic objects. You want the classes that use the diagnostic classes to be independent of the specific classes being used. You could use the Factory Method pattern to create diagnostic objects, but there is a problem that the Factory Method pattern does not solve.

You want to ensure that all the objects that you use to diagnose a computer are for that computer's architecture. If you simply use the Factory

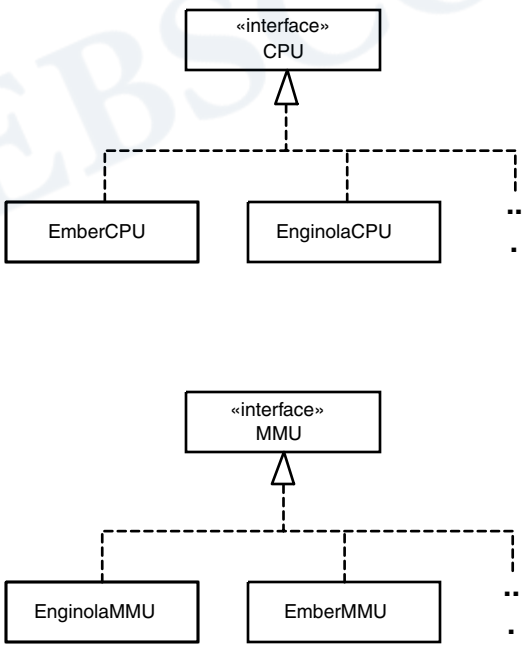

FIGURE 5.6 Diagnostic classes.

Method pattern, the classes that use the diagnostic classes will have the burden of telling each factory which computer architecture the diagnostic objects are for. You want to find a cohesive way of ensuring that all of the diagnostic objects used for a computer are for the correct architecture, without adding any dependencies to the classes that use the objects.

FORCES

- ☺ A system that works with multiple products should function in a way that is independent of the specific product it is working with.
- ☺ It should be possible for a system to be configured to work with one or multiple members of a family of products.
- ☺ Instances of classes intended to interface with a product should be used together and only with the product. This constraint must be enforced.
- ☺ The rest of a system should work with a product without being aware of the specific classes used to interface with the product.
- ☺ A system should be extensible so that it can work with additional products by adding additional sets of classes and changing at most only a few lines of code.
- ☹ The interface that a class implements is not sufficient to distinguish it from other classes that may be instantiated to create an object.

SOLUTION

Figure 5.7 is a class diagram that shows the roles classes play in the Abstract Factory pattern.

Here are descriptions of the roles that the classes and interfaces shown in Figure 5.7 play in the Abstract Factory pattern.

WidgetAIF, WidgetBIF, and so on. Interfaces in this role correspond to a service or feature of a product. Classes that implement one of these interfaces work with the service or feature to which the interface corresponds.

Because of space constraints, only two of these interfaces are shown. Most applications of the Abstract Factory pattern have more than two of these interfaces. There will be as many of these interfaces as there are distinct features or services of the products that are being used.

Product1WidgetA, Product2WidgetA, and so on. Classes in this role correspond to a specific feature of a specific product. You can generically refer to classes in this role as concrete widgets.

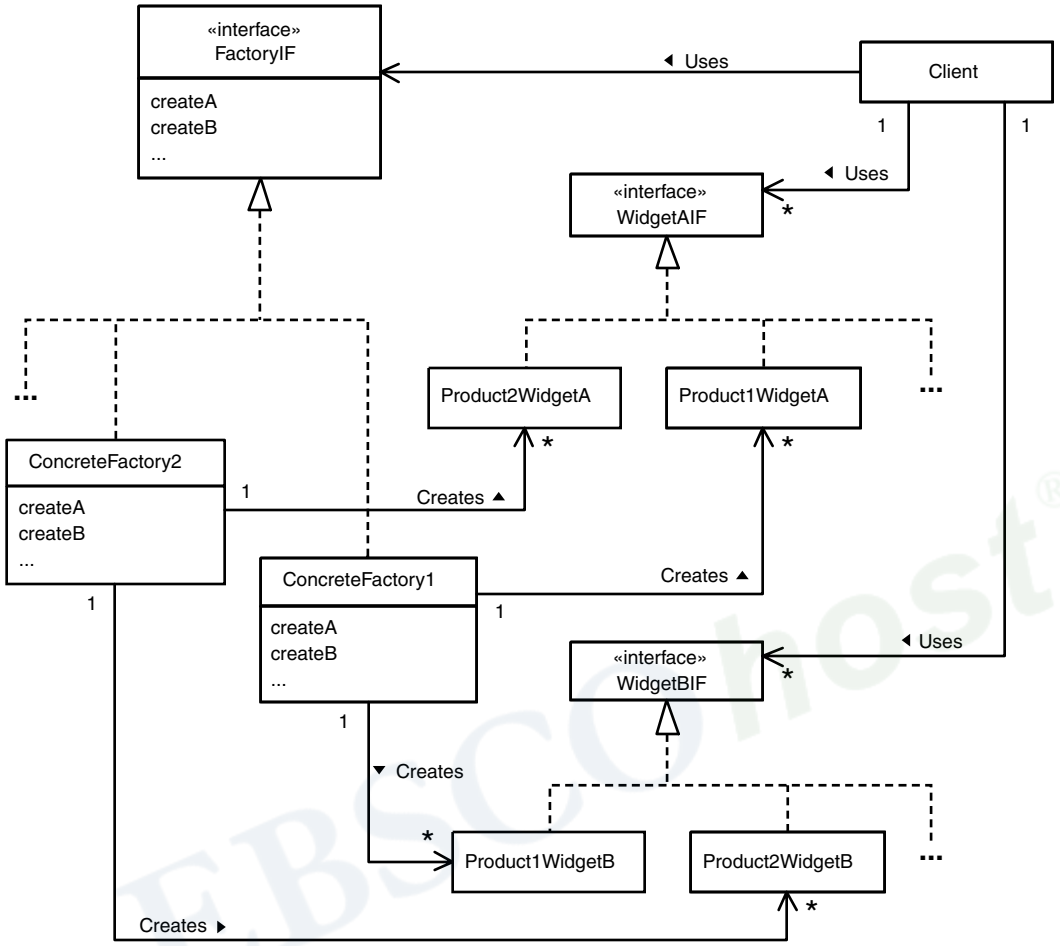

FIGURE 5.7 Abstract Factory.

Client. Classes in this role use concrete widget classes to request or receive services from the product that the client is working with. Client classes interact with concrete widget classes through a `WidgetIF` interface only. They are independent of the actual concrete widget classes they are working with.

FactoryIF. Interfaces in this role declare methods for creating instances of concrete widget classes. Each method returns an instance of a class that implements a different `WidgetIF` interface. Classes that are responsible for creating concrete widget objects implement an interface in this role.

ConcreteFactory1, ConcreteFactory2, and so on. Classes in this role implement a `FactoryIF` interface to create instances of con-

crete widget classes. Each class in this role corresponds to a different product that a `Client` class may work with. Each method of a `ConcreteFactory` class creates a different kind of concrete widget. However, all of the concrete widgets that its methods create are for working with the product that the `ConcreteFactory` class corresponds to.

Client classes that call these methods should not have any direct knowledge of these concrete factory classes, but instead should access instances of these classes through the `FactoryIF` interface.

IMPLEMENTATION

The main implementation issue for the Abstract Factory pattern is the mechanism that the client classes use to create and access `FactoryIF` objects. The simplest situation is when client objects need to work with only one product during the run of a program. In that case, some class will typically have a static variable set to the `ConcreteFactory` class that is used during the program run. The variable may be public or its value accessed through a public static method.

If the abstract factory object will use information provided by the requesting client to select among multiple concrete factory objects, you can use the Factory Method pattern.

CONSEQUENCES

- ☺ Client classes are independent of the concrete widget classes that they use.
- ☺ Adding (as opposed to writing) classes to work with additional products is simple. The class of a concrete factory object usually needs to be referenced in only one place. It is also easy to change the concrete factory used to work with a particular product.
- ☺ By forcing client classes to go through a `FactoryIF` interface to create concrete widget objects, the Abstract Factory pattern ensures that client objects use a consistent set of objects to work with a product.
- ☹ The main drawback of the Abstract Factory pattern is that it can be a lot of work to write a new set of classes to interface with a product. It can also take a lot of work to extend the set of features that the existing set of classes is able to exercise in the products that they work with.

Adding support for a new product involves writing a complete set of concrete widget classes to support that product. You must write a concrete widget class for each `WidgetIF` interface. If there are many `WidgetIF` interfaces, then it will be a lot of work to support an additional product.

Adding access to an additional feature of the products interfaced to can also take a lot of work if there are many supported products. It involves writing a new `WidgetIF` interface corresponding to the new feature and a new concrete widget class corresponding to each product.

☹ Client objects may have a need to organize widget classes into a hierarchy that serves the needs of client objects. The basic Abstract Factory pattern does not lend itself to this because it requires concrete widget classes to be organized into a class hierarchy that is independent of client objects. The difficulty can be overcome by mixing the Bridge pattern with the Abstract Factory pattern:

- Create a hierarchy of product-independent widget classes that suits the needs of the client classes. Have each product-independent widget class delegate product-specific logic to a product-specific class that implements a `WidgetIF` interface.
- Java's `java.awt` package contains a number of classes that are implemented using this variation. Classes like `Button` and `TextField` contain logic that is independent of the windowing system being used. These classes provide a native look and feel by delegating windowing system operations to concrete widget classes that implement interfaces defined in the `java.awt.peer` package.

JAVA API USAGE

The Abstract Factory pattern is used in the Java API to implement the `java.awt.Toolkit` class. The `java.awt.Toolkit` class is an abstract factory class used to create objects that work with the native windowing system. The concrete factory class it uses is determined by initialization code, and the singleton concrete factory object is returned by its `getDefaultToolkit` method.

CODE EXAMPLE

For this pattern's code example, we will return to the problem discussed under the Context section. Figure 5.8 shows an expanded class diagram that incorporates the Abstract Factory pattern.

An instance of the `Client` class manages the remote diagnostic process. When it determines the architecture of the machine it has to diagnose, it passes the architecture type to the `createToolkit` method of a `ToolkitFactory` object. The method returns an instance of a class such as `EmberToolkit` or `EnginolaToolkit` that implements the `Architecture-ToolkitIF` interface for the specified computer architecture. The `Client`

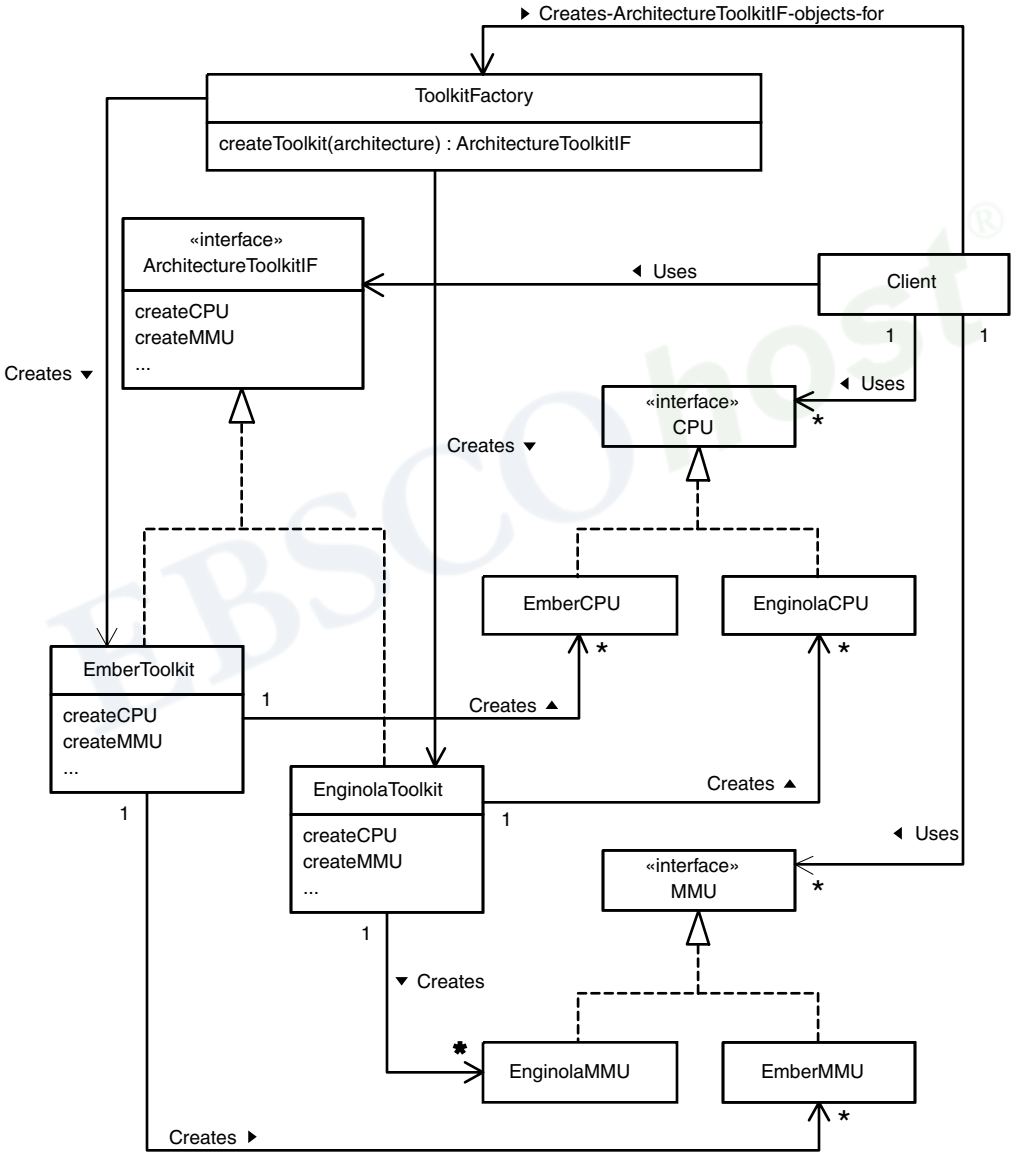

FIGURE 5.8 Diagnostic classes with abstract factory.

object can then use the ArchitectureToolkitIF object to create objects that model central processing units (CPUs), memory management units (MMUs), and other components of the required architecture.

Here is some of the Java code that implements the design for remote computer diagnostics shown in Figure 5.8. The abstract widget classes have the obvious structure.

This code is for a concrete factory class that creates instances of classes to test ember architecture computers:

```
class EmberToolkit implements ArchitectureToolkitIF {
    public CPU createCPU() {
        return new EmberCPU();
    } // createCPU()

    public MMU createMMU() {
        return new EmberMMU();
    } // createMMU()
    ...
} // class EmberFactory
```

The following is the code for the factory interface:

```
public interface ArchitectureToolkitIF {
    public abstract CPU createCPU() ;
    public abstract MMU createMMU() ;
...
} // AbstractFactory
```

This implementation of the Abstract Factory pattern uses the Factory Method pattern to create ArchitectureToolkitIF objects. Here is a listing of the factory class responsible for creating ArchitectureToolkitIF objects.

```
public class ToolkitFactory {
    /**
     * The single instance of this class.
     */
    private static ToolkitFactory myInstance
      = new ToolkitFactory();

    // Symbolic names to identify computer architectures
    public final static int ENGINOLA = 900;
    public final static int EMBER    = 901;
    // ...

    public static ToolkitFactory getInstance() {
        return myInstance;
    } // getInstance()

    /**
     * Return a newly created object that implements the
```

```
     * ArchitectureToolkitIF interface for the given computer
     * architecture.
     */
    public
    ArchitectureToolkitIF createToolkit(int architecture) {
        switch (architecture) {
          case ENGINOLA:
              return new EnginolaToolkit();

          case EMBER:
              return new EmberToolkit();
        } // switch
        String errMsg = Integer.toString(architecture);
        throw new IllegalArgumentException(errMsg);
    } // createToolkit(int)
} // class ToolkitFactory
```

Client classes typically create concrete widget objects using code that looks something like this:

```
public class Client {
    public void doIt () {
        ToolkitFactory myFactory;
        myFactory = ToolkitFactory.getInstance();
        ArchitectureToolkitIF af;
        af = myFactory.createToolkit(ToolkitFactory.EMBER);
        CPU cpu = af.createCPU();
        ...
    } // doIt
} // class Client
```

RELATED PATTERNS

Factory Method. In the preceding example, the abstract factory class uses the Factory Method pattern to decide which concrete factory object to give to a client class.

Singleton. Concrete Factory classes are usually implemented as Singleton classes.

Builder

This pattern was previously described in [GoF95].

SYNOPSIS

The Builder pattern allows a client object to construct a complex object by specifying only its type and content. The client is shielded from the details of the object's construction.

CONTEXT

Consider the problem of writing an email gateway program. The program receives e-mail messages that are in MIME format.[3] It forwards them in a different format for different kinds of e-mail systems. This situation is a good fit for the Builder pattern. It is straightforward to organize this program with an object that parses MIME messages. Each message to parse is paired with a builder object that the parser uses to build a message in the required format. As the parser recognizes each header field and message body part, it calls the corresponding method of the builder object it is working with.

Figure 5.9 shows this structure. In Figure 5.9, the `MessageManager` class is responsible for collecting MIME formatted e-mail messages and initiating their transmission. The e-mail messages it directly manages are instances of the `MIMEMsg` class.

Instances of the `MIMEMsg` class encapsulate MIME formatted e-mail messages. When a `MessageManager` object wants to transmit a message in a format other than MIME, it must rebuild a message in the desired format. The content of the new message must be the same as the MIME message or as close as the format allows.

The `MIMEParser` class is a subclass of the `MessageParser` class that can parse MIME formatted e-mail messages and pass their contents to a builder object.

[3] MIME is an acronym for Multipurpose Internet Mail Extensions. It is the standard that most email messages on the Internet conform to. You can find a description of MIME at http://mgrand.home.mindspring.com/mime.html.

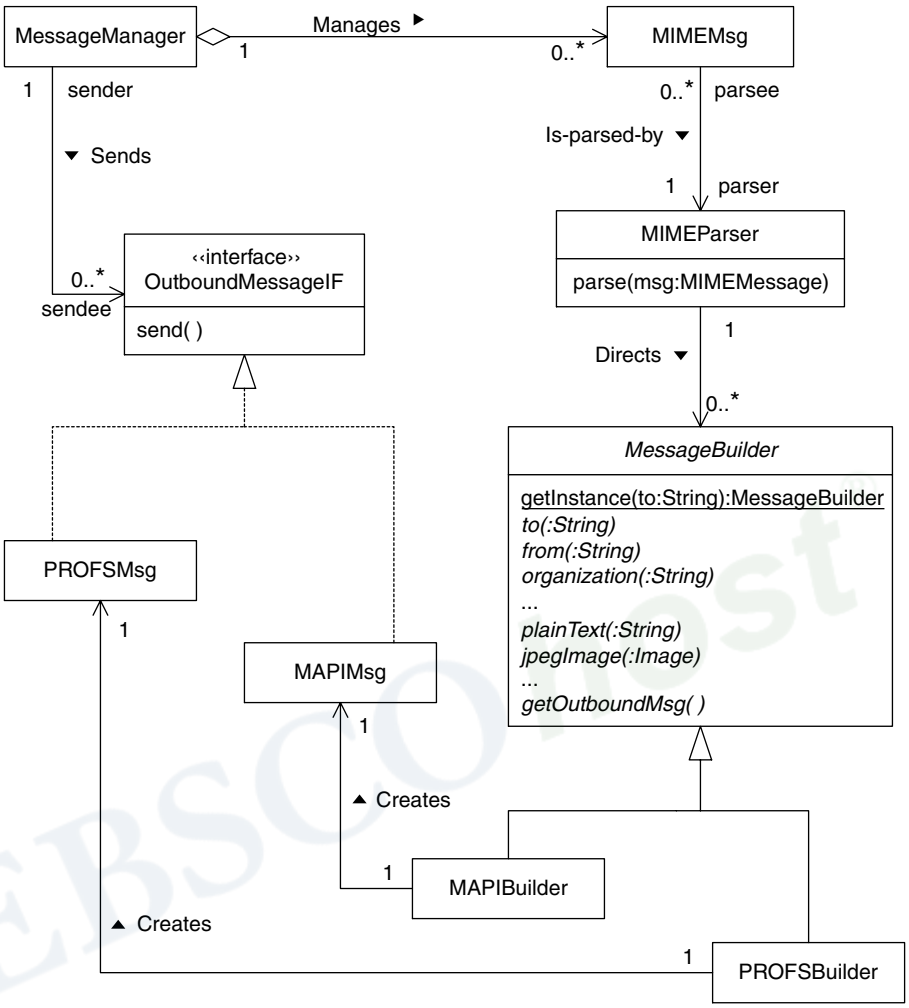

FIGURE 5.9 Builder example.

`MessageBuilder` is an abstract builder class. It defines methods that correspond to the various header fields and body types that MIME supports. It declares abstract methods that correspond to required header fields and the most common body types. It declares these methods abstract because all concrete subclasses of `MessageBuilder` should define these methods. However, some of the optional header fields such as organization and fancier body types such as `Image/Jpeg` may not be supported in all message formats, so the `MessageBuilder` class provides do-nothing implementations of these methods.

The `MessageBuilder` class also defines a class method called `getInstance`. A `MIMEParser` object passes the `getInstance` method the destination address of the message it is parsing. From the message's destination address, the `getInstance` method determines the message format needed for the new message. It returns an instance of the subclass of `MessageBuilder` appropriate for the format of the new message to the `MIMEParser` object.

The `MAPIBuilder` and `PROFSBuilder` classes are concrete builder classes for building Messaging Application Programming Interface (MAPI) and PROFS (a registered trademark of the IBM corporation) messages, respectively.

The builder classes create product objects that implement the `OutboundMsgIF` interface. This interface defines a method called `send` that is intended to send the e-mail message wherever it is supposed to go.

Figure 5.10 is a collaboration diagram that shows how these classes work together.

Here is what's happening in Figure 5.10:

1. A `MessageManager` object receives an e-mail message.

1.1 The `MessageManager` object calls the `MIMEParser` class's parse method. It will return an `OutboundMessageIF` object that encapsulates the new message in the needed format.

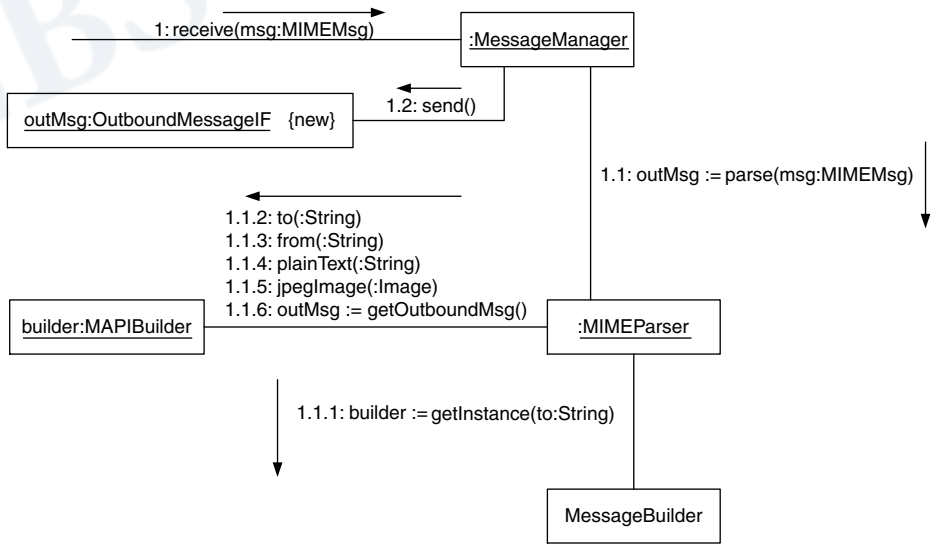

FIGURE 5.10 Builder example collaboration.

1.1.1 The `MIMEParser` object calls the `MessageBuilder` class's `getInstance` method, passing it the destination email address. By analyzing the address, the method selects a concrete subclass of the `MessageBuilder` class and creates an instance of it.

1.1.2 The `MIMEParser` object passes the destination email address to the `MessageBuilder` object's `to` method.

1.1.3 The `MIMEParser` object passes the originating email address to the `MessageBuilder` object's `from` method.

1.1.4 The `MIMEParser` object passes the email message's simple content to the `MessageBuilder` object's `plainText` method.

1.1.5 The `MIMEParser` object passes the email message's attached jpeg image to the `MessageBuilder` object's `jpegImage` method.

1.1.6 The `MIMEParser` object calls the `MessageBuilder` object's `getOutboundMsg` method to complete and fetch the new message.

1.2 The `MessageManager` object calls the `OutboundMsg` object's send method. This sends the message off and completes the message processing.

FORCES

- ☺ A program must to be able to produce multiple external representations of the same data.
- ☺ The classes responsible for providing content should be independent of any external data representation and the classes that build them. If content-providing classes have no dependencies on external data representations, then modifications to external data representation classes will not require any maintenance to content-providing classes.
- ☺ The classes responsible for building external data representations are independent of the classes that provide the content. Their instances can work with any content-providing object without knowing anything about the content-providing object.

SOLUTION

Figure 5.11 is a class diagram showing the participants in the Builder pattern.

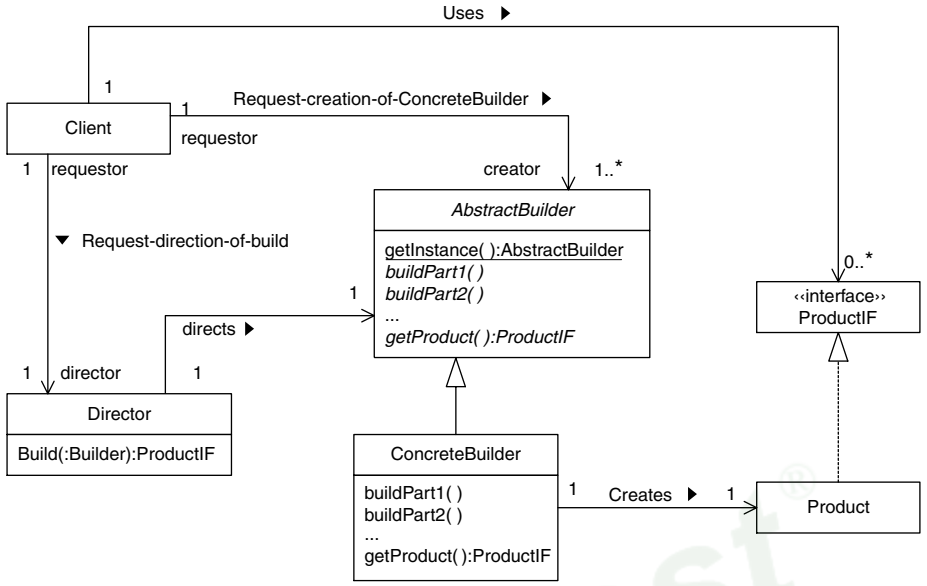

FIGURE 5.11 Builder pattern.

Here are the roles that these classes and interface play in the Builder pattern:

Product. A class in this role defines a type of data representation. All `Product` classes should implement the `ProductIF` interface so that other classes can refer to `Product` objects through the interface without having to know their class.

ProductIF. The Builder pattern is used to build a variety of different kinds of `Product` objects for use by `Client` objects. To avoid the need for `Client` objects to know the actual class of `Product` objects built for them, all `Product` classes implement the `ProductIF` interface. `Client` objects refer to `Product` objects built for them through the `ProductIF` interface, so they don't need to know the actual class of the objects built for them.

Client. An instance of a client class initiates the actions of the Builder pattern. It calls the `AbstractBuilder` class's `getInstance` method. It passes information to `getInstance` telling it what sort of product it wants to have built. The `getInstance` method determines the subclass of `AbstractBuilder` to create and returns it to the `Client` object. The `Client` object then passes the object it got

from `getInstance` to a `Director` object's `build` method, which builds the desired object.

ConcreteBuilder. A class in this role is a concrete subclass of the `AbstractBuilder` class that is used to build a specific kind of data representation of a `Director` object.

AbstractBuilder. A class in this role is the abstract superclass of `ConcreteBuilder` classes. An `AbstractBuilder` class defines a static method, typically called `getInstance`, which takes an argument that specifies a data representation. The `getInstance` method returns an instance of a concrete builder class that produces the specified data representation.

An `AbstractBuilder` class also defines methods, shown in the class diagram as `buildPart1`, `buildPart2`, ... A `Director` object calls these methods to tell the object returned by the `getInstance` method what content to put in the created object.

Finally, the builder class defines a method, typically called `getProduct`, which returns the product object created by a concrete builder object.

Director. A `Director` object calls the methods of a concrete builder object to provide the concrete builder with the content for the product object that it builds.

Figure 5.12 is a collaboration diagram showing how these classes work together.

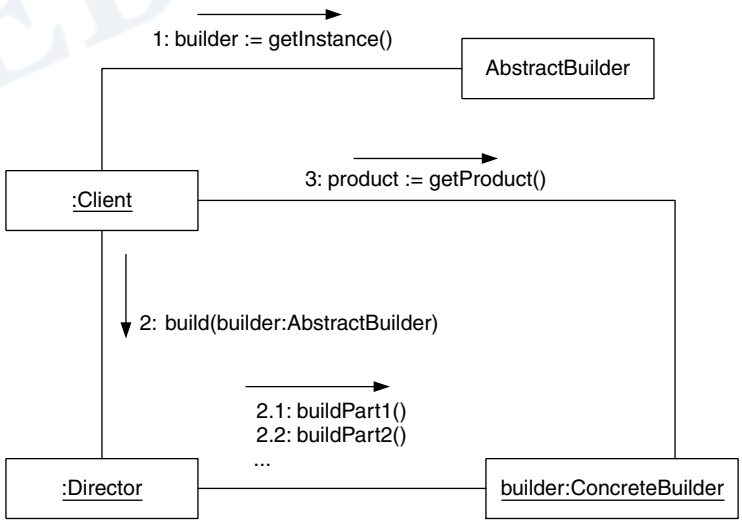

FIGURE 5.12 Builder collaboration.

IMPLEMENTATION

The essential design and implementation issue for the Builder pattern is the set of methods that the builder class defines to provide content to concrete builder objects. These methods can be a major concern because there may be a large number of them. The methods should be general enough to allow all reasonable data representations to be constructed. On the other hand, an excessively general set of methods can be more difficult to implement and to use. The consideration of generality versus difficulty of implementation raises these issues in the implementation phase:

- Each content-providing method declared by the abstract builder class can be abstract or provided with a default do-nothing implementation. Abstract method declarations force concrete builder classes to provide an implementation for that method. Forcing concrete builder classes to provide implementations for methods is good in those cases where the method provides essential information about content. It prevents implementers of those classes from forgetting to implement those methods.

  However, methods that provide optional content or supplementary information about the structure of the content may be unnecessary or even inappropriate for some data representations. Providing a default do-nothing implementation for such methods saves effort in the implementation of concrete builder classes that do not need such methods.
- Organizing concrete builder classes so that calls to content-providing methods simply add data to the product object is often good enough. In some cases, there will be no simple way to tell the builder where in the finished product a particular piece of the product will go. In those situations, it may be simplest to have the content-providing method return an object to the director that encapsulates such a piece of the product. The director object can then pass the object to another content-providing method in a way that implies the position of the piece of the product within the whole product.

CONSEQUENCES

- ☺ Content determination and the construction of a specific data representation are independent of each other. The data representation of the product can change without any impact on the objects that provide the content. Builder objects can work with different content-providing objects without requiring any changes.
- Builder provides finer control over construction than other patterns such as Factory Method by giving the director object step-by-step

control over creation of the product object. Other patterns simply create the entire object in one step.

CODE EXAMPLE

Let's look at some sample code for the classes in this example that collaborate in the Builder pattern. Instances of the `MIMEParser` class fill the role of director objects. Here is the source for the `MIMEParser` class:

```
class MIMEParser {
    private MIMEMessage msg;       // The message being parsed
    private MessageBuilder builder; // The builder object
    ...
    /**
     * Parse a MIME message, calling the builder methods that
     * correspond to message header fields and body parts.
     */
    OutboundMessageIF parse() {
        builder =MessageBuilder.getInstance(getDestination());
        MessagePart hdr = nextHeader();
        while (hdr != null) {
            if (hdr.getName().equals("to"))
              builder.to((String)hdr.getValue());
            else if (hdr.getName().equals("from"))
              builder.from((String)hdr.getValue());
            ...
            hdr = nextHeader();
        } // while hdr
        MessagePart bdy = nextBodyPart();
        while (bdy != null) {
            String name = bdy.getName();
            if (name.equalsIgnoreCase ("text/plain"))
              builder.plainText((String)bdy.getValue());
            ...
            else if (name.equalsIgnoreCase("image/jpeg"))
              builder.jpegImage((Image)bdy.getValue());
            ...
            bdy = nextHeader();
        } // while bdy
        return builder.getOutboundMsg();
    } // parse(Message)
...
    private class MessagePart {
        private String name;
        private Object value;

        MessagePart(String name, Object value) {
            this.name = name;
            this.value = value;
        } // Constructor(String, String)
```

```
        String getName() { return name; }

        Object getValue() { return value; }
    } // class MessagePart
} // class MIMEParser
```

The chains of `if` statements that occur in the `parse` method of the preceding class would be rather long if the method were fully fleshed out. MIME supports over 25 different kinds of header fields alone. A less awkward way to organize a chain of tests of object equality that result in a method call is to use the Hashed Adapter Objects pattern described in *Patterns in Java, Volume 2*.

Here is code for the `MessageBuilder` class, which fills the role of abstract builder class:

```
abstract class MessageBuilder {
    /**
     * Return an instance of the subclass appropriate for the
     * email message format implied by the given destination
     * address.
     * @param dest The address to send the email to
     */
    static MessageBuilder getInstance(String dest) {
        MessageBuilder builder = null;
        ...
        return builder;
    } // getInstance(String)

    /**
     * Pass the "to" header field value to this method.
     */
    abstract void to(String value);

    /**
     * Pass the "from" header field value to this method.
     */
    abstract void from(String value);

    /**
     * Pass the "organization" header field value to this
     * method.
     */
    void organization(String value) { }

    /**
     * Pass the content of a plaintext body part to this
      method.
      */
    abstract void plainText(String content);
...

    /**
```

```
     * complete and return the outbound email message.
     */
    abstract OutboundMessageIF getOutboundMsg() ;
} // class MessageBuilder
```

Finally, here is the code for the `OutboundMsgIF` interface:

```
public interface OutboundMsgIF {
    public void send() ;
} // interface OutboundMsgIF
```

RELATED PATTERNS

Interface. The Builder pattern uses the Interface pattern to hide the class of a `ProductIF` object.

Composite. The object built using the Builder pattern is typically a Composite.

Factory Method. The Builder pattern uses the Factory Method pattern to decide which concrete builder class to instantiate.

Template Method. The Abstract Builder class is often implemented using the Template Method pattern.

Null Object. The Null Object pattern may be used by the Builder pattern to provide do-nothing implementations of methods.

Visitor. The Visitor pattern allows the client object to be more closely coupled to the construction of the new complex object than the Builder pattern allows. Instead of describing the content of the objects to be built through a series of method calls, the information is presented in bulk as a complex data structure.

Prototype

This pattern was previously described in [GoF95].

SYNOPSIS

The Prototype pattern allows an object to create customized objects without knowing their exact class or the details of how to create them. It works by giving prototypical objects to an object that initiates the creation of objects. The creation-initiating object then creates objects by asking the prototypical objects to make copies of themselves.

CONTEXT

Suppose that you are writing a Computer-Assisted Design (CAD) program that allows its users to draw diagrams from a palette of symbols. The program will have a core set of built-in symbols. However, people with different and specialized interests will use the program. The core set of symbols will not be adequate for people with a specialized interest. These people will want additional symbols that are specific to their interests. Most users of this program will have a specialized interest. It must be possible to provide additional sets of symbols that users can add to the program to suit their needs.

This presents the problem of how to provide these palettes of additional symbols. You can easily organize things so that all symbols, both core and additional, are descended from a common ancestor class. This will give the rest of your diagram-drawing program a consistent way of manipulating symbol objects. It does leave open the question of how the program will create these objects. Creating objects such as these is often more complicated than simply instantiating a class. It may also involve setting values for data attributes or combining objects to form a composite object.

A solution is to provide the drawing program with previously created objects to use as prototypes for creating similar objects. The most important requirement for objects to be used as prototypes is that they have a method, typically called `clone`, that returns a new object that is a copy of the original object. Figure 5.13 shows how this would be organized.

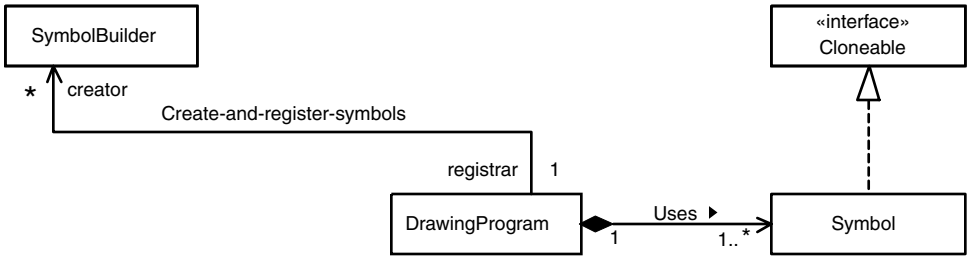

FIGURE 5.13 Symbol prototype.

The drawing program maintains a collection of prototypical `Symbol` objects. It uses the `Symbol` objects by cloning them. `SymbolBuilder` objects create `Symbol` objects and register them with the drawing program.

All Java classes inherit a method from the `Object` class called `clone`. An object's `clone` method returns a copy of the object. It does this only if the object's class gives permission. A class gives permission for its instances to be cloned if, and only if, it implements the `Cloneable` interface. Because the `Symbol` class implements the `Cloneable` interface, the drawing program is able to clone the `Symbol` objects that it manages and incorporate those objects into drawings.

FORCES

- ☺ A system must be able to create objects without knowing their exact class, how they are created, or what data they represent.
- ☺ Classes to be instantiated are not known by the system until runtime, when they are acquired on the fly by techniques such as dynamic linkage.
- ☺ The following approaches to allowing the creation of a large variety of objects are undesirable:
  - The classes that initiate the creation of objects directly create the objects. This makes them aware of and dependent on a large number of other classes.
  - The classes that initiate the creation of objects create the objects indirectly through a factory method class. A factory method that is able to create a large variety of objects may be very large and difficult to maintain.
  - The classes that initiate the creation of objects create the objects indirectly through an abstract factory class. In order for an

abstract factory to be able to create a large variety of objects, it must have a large variety of concrete factory classes in a hierarchy that parallels the classes to be instantiated.

☺ The different objects that a system must create may be instances of the same class that contain different state information or data content.

SOLUTION

Enable a class to create objects that implement a known interface by giving it a prototypical instance of each kind of object it will create. It is then able to create new objects by cloning a prototypical instance.

Figure 5.14 shows the organization of the Prototype pattern. Here are descriptions of the roles these classes and interfaces play in the Prototype pattern:

Client. The client class represents the rest of the program for the purposes of the Prototype pattern. The client class needs to create objects that it knows little about. Client classes will have a method that can be called to add a prototypical object to a client object's collection. In Figure 5.14, this method is indicated with the name `registerPrototype`. However, a name that reflects the sort of object being prototyped, such as `registerSymbol`, is more appropriate in an actual implementation.

Prototype. Classes in this role implement the `PrototypeIF` interface and are instantiated for the purpose of being cloned by the client. Classes in this role are commonly abstract classes with a number of concrete subclasses.

PrototypeIF. All prototype objects must implement the interface that is in this role. The client class interacts with prototype objects through this interface. Interfaces in this role should extend the `Cloneable` interface so that all objects that implement the interface can be cloned.

PrototypeBuilder. This corresponds to any class instantiated to supply prototypical objects to the client object. Such classes should have a name that denotes the type of prototypical object that they build, such as `SymbolBuilder`.

A `PrototypeBuilder` object creates `Prototype` objects. It passes each newly created `Prototype` object to a `Client` object's `registerPrototype` method.

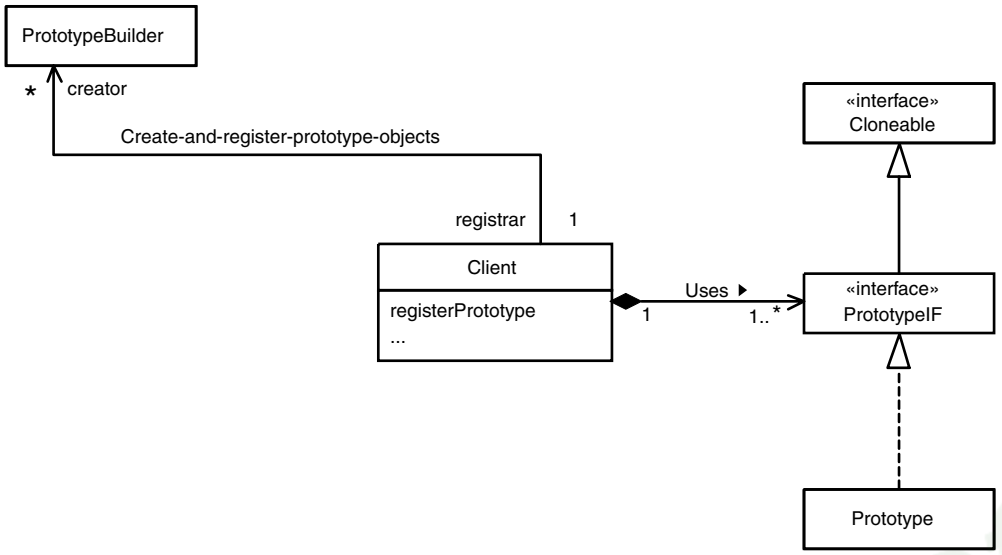

FIGURE 5.14 Prototype pattern.

IMPLEMENTATION

An essential implementation issue is how the `PrototypeBuilder` objects add objects to a client object's palette of prototypical objects. The simplest strategy is for the client class to provide a method for this purpose, which `PrototypeBuilder` objects can call. A possible drawback is that the `PrototypeBuilder` objects will need to know the class of the client object. If that is a problem, the `PrototypeBuilder` objects can be shielded from knowing the exact class of the client objects by providing an interface or abstract class for the client class to implement or inherit.

How to implement the clone operation for the prototypical objects is another important implementation issue. There are two basic strategies for implementing the clone operation:

- *Shallow copying* means that the variables of the cloned object contain the same values as the variables of the original object and that all object references are to the same objects. In other words, shallow copying copies only the object being cloned, not the objects that it refers to. Both the original and a shallow copy refer to the same objects.
- *Deep copying* means that the variables of the cloned object contain the same values as the variables of the original object, except that variables that refer to objects refer to copies of the objects referred to

by the original object. In other words, deep copying copies the object being cloned and the objects that it refers to. A deep copy refers to copies of the objects that the original refers to.

Implementing deep copying can be tricky. You will need to decide whether you want to make deep or shallow copies of the indirectly copied objects. You will also need to be careful about handling any circular references.

Shallow copying is easier to implement because all classes inherit a `clone` method for the `Object` class that does just that. However, unless an object's class implements the `Cloneable` interface, the `clone` method will refuse to work. If all of the prototypical objects your program uses will be cloning themselves by shallow copying, you can save some time by declaring the `PrototypeIF` interface to extend the `Cloneable` interface. This way, all classes that implement the `PrototypeIF` interface also implement the `Cloneable` interface.

Some objects, such as threads and sockets, cannot be simply copied or shared. Whichever copying strategy is used, if it involves references to such objects, then equivalent objects will need to be constructed for the use of the copied objects.

Unless the `Client` object's palette of prototypical objects consists of a fixed number of objects having fixed purposes, it is inconvenient to use individual variables to refer to each prototypical object. It is easier to use a collection object that can contain a dynamically growing or shrinking palette of prototypical objects. A collection object that plays this role in the Prototype pattern is called a *prototype manager*. Prototype managers can be fancier than just a simple collection. They may allow objects to be retrieved by their attribute values or other keys.

If your program will have multiple client objects, then you have another issue to consider. Will the client objects have their own palette of prototypical objects or will all of them share the same palette? The answer will depend on the needs of your application.

CONSEQUENCES

- ☺ A program can dynamically add and remove prototypical objects at runtime. This is a distinct advantage offered by none of the other creational patterns in this book.
- ☺ A `PrototypeBuilder` object can simply supply a fixed set of prototypical objects.
- ☺ A `PrototypeBuilder` object may provide the additional flexibility of allowing new prototypical objects to be created by object composition and changes to the values of object attributes.

- ☺ The client object may also be able to create new kinds of prototypical objects. In the drawing program example we looked at previously, the client object could very reasonably allow the user to identify a sub-drawing and then turn the sub-drawing into a new symbol.
- ☺ The client class is independent of the exact class of the prototypical objects that it uses. Also, the client class does not need to know the details of how to build the prototypical objects.
- ☺ The `PrototypeBuilder` objects encapsulate the details of constructing prototypical objects.
- ☺ By insisting that prototypical objects implement an interface such as `PrototypeIF`, the Prototype pattern ensures that the prototypical objects provide a consistent set of methods for the client object to use.
- ● There is no need to organize prototypical objects into any sort of class hierarchy.
- ☹ A drawback of the Prototype pattern is the additional time spent writing `PrototypeBuilder` classes.
- ☹ Programs that use the Prototype pattern rely on dynamic linkage or similar mechanisms. Installation of programs that rely on dynamic linkage or similar mechanisms can be more complicated. They may require information about their environment that may otherwise not be needed.

JAVA API USAGE

The Prototype pattern is the very essence of JavaBeans. JavaBeans are instances of classes that conform to certain naming conventions. The naming conventions allow a bean creation program to know how to customize them. After a bean object has been customized for use in an application, the object is saved to a file to be loaded by the application while it is running. This is a time-delayed way of cloning objects.

CODE EXAMPLE

Suppose that you are writing an interactive role-playing game. That is, a game that allows the user to interact with simulated characters. One of the expectations for this game is that people who play it will grow tired of interacting with the same characters and want to interact with new characters. For this reason, you are also developing an add-on to the game that consists of a few pre-generated characters and a program to generate additional characters.

The characters in the game are instances of a relatively small number of classes such as Hero, Fool, Villain, and Monster. What makes instances of the same class different from each other is the different attribute values that are set for them, such as the images that are used to represent them, height, weight, intelligence, and dexterity.

Figure 5.15 shows some of the classes involved in the game.

Here is the code for the `CharacterIF` interface, the interface that serves the role of `PrototypeIF`.

```
public interface CharacterIF extends Cloneable {
    public String getName() ;
    public void setName(String name) ;
    public Image getImage() ;
    public void setImage(Image image) ;
```

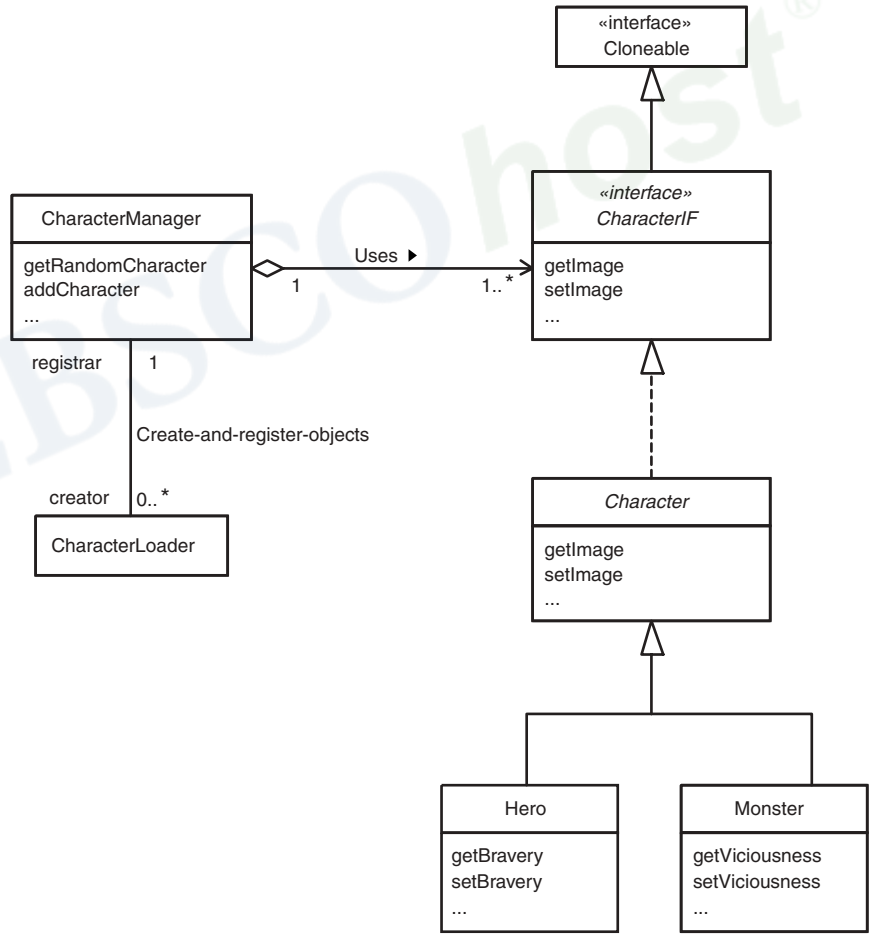

FIGURE 5.15 Prototype example.

```
    public int getStrength() ;
    public void setStrength(int strength) ;
    ...
} // class CharacterIF
```

Here is the code for the `Character` class, the abstract class that serves the role of `Prototype`:

```
public abstract class Character implements CharacterIF {
...
    /**
     * Override clone to make it public.
     */
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            // should never happen because this class
            // implements Cloneable.
            throw new InternalError();
        } // try
    } // clone()

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public Image getImage() { return image; }

    public void setImage(Image image) { this.image = image; }
...
} // class Character
```

Most of this is just simple accessor methods. The one less-than-obvious method is the `clone` method. All objects inherit a `clone` method from the `Object` class. Because the `clone` method is not public, the `Character` class must override it with a public declaration, just to make it accessible to other classes.

Here is source code for the `Hero` class, one of the classes that serve in the Prototype role:

```
public class Hero extends Character {
    private int bravery;
...
    public int getBravery() { return bravery; }

    public void setBravery(int bravery) {
        this.bravery = bravery;
    }
} // class Hero
```

The Monster class is similar to the Hero class.

Following is the code for the `CharacterManager` class that serves in the role of client class:

```
public class CharacterManager {
    private Vector characters = new Vector();
...
    /**
     * Return a copy of random character from the collection.
     */
    Character getRandomCharacter() {
        int i = (int)(characters.size()*Math.random());
        Character c = (Character)characters.elementAt(i);
        return (Character)c.clone();
    } // getRandomCharacter()

    /**
     * Add a prototypical object to the collection.
     */
    void addCharacter(Character character) {
        characters.addElement(character);
    } // addCharacter(Character)
...
} // class CharacterManager
```

Here is the code for the `CharacterLoader` class that fills the role of `PrototypeBuilder`:

```
/**
 * This class loads character objects and adds them to
 * the CharacterManager.
 */
class CharacterLoader {
    private CharacterManager mgr;
    /**
     * Constructor
     * @param cm
     *        The CharacterManager this object will work with.
     */
    CharacterLoader(CharacterManager cm) {
        mgr = cm;
    } // Constructor(CharacterManager)

    /**
     * Load character objects from the specified file.
     * Since failure only affects the rest of the program to
     * the extent that new character objects are not loaded,
     * we need not throw any exceptions.
     */
    int loadCharacters(String fname) {
        int objectCount = 0;   // The number of objects loaded
```

```
            // If construction of InputStream fails, just return
            try {
                InputStream in;
                in = new FileInputStream(fname);
                in = new BufferedInputStream(in);
                ObjectInputStream oIn = new ObjectInputStream(in);
                while(true) {
                    Object c = oIn.readObject();
                    if (c instanceof Character) {
                        mgr.addCharacter((Character)c);
                    } // if
                } // while
            } catch (Exception e) {
            } // try
            return objectCount;
        } // loadCharacters(String)
    } // class CharacterLoader
```

RELATED PATTERNS

Composite. The Prototype pattern is often used with the Composite pattern. The composite is used to organize prototype objects.

Abstract Factory. The Abstract Factory pattern can be a good alternative to the Prototype pattern if the dynamic changes that the Prototype pattern allows to the prototypical object palette are not needed.

`PrototypeBuilder` classes may use the Abstract Factory pattern to create a set of prototypical objects.

Façade. The client class commonly acts as a façade that separates the other classes that participate in the Prototype pattern from the rest of the program.

Factory Method. The Factory Method pattern can be an alternative to the Prototype pattern when the palette of prototypical objects never contains more than one object.

Decorator. The Prototype pattern is often used with the Decorator pattern to compose prototypical objects.

Singleton

This pattern was previously described in [GoF95].

SYNOPSIS

The Singleton pattern ensures that only one instance of a class is created. All objects that use an instance of that class use the same instance.

CONTEXT

Some classes should have exactly one instance. These classes usually involve the central management of a resource. The resource may be external, as is the case with an object that manages the reuse of database connections. The resource may be internal, such as an object that keeps an error count and other statistics for a compiler.

Suppose you need to write a class that an applet can use to ensure that no more than one audio clip is played at a time. If an applet contains two pieces of code that independently play audio clips, then it is possible for both to be playing at the same time. When two audio clips play at the same time, the results depend on the platform. The results may range from confusing, with users hearing both audio clips together, to terrible, with the platform's sound-producing mechanism unable to cope with playing two different audio clips at once.

To avoid the undesirable situation of two audio clips playing at the same time, the class you write should stop the previous audio clip before starting the next audio clip. A way to design a class to implement this policy while keeping the class simple is to ensure that there is only one instance of the class shared by all objects that use that class. If all requests to play audio clips go through the same object, then it is simple for the object to stop the last audio clip it started before starting the next audio clip. Figure 5.16 shows such a class.

The constructor for the `AudioClipManager` class is private. This prevents another class from directly creating an instance of the `AudioClipManager` class. Instead, to get an instance of the `AudioClipManager` class, other classes must call its `getInstance` method. The `getInstance` method is a static method that always returns the same instance of the `AudioClipManager` class. The instance it returns is the instance referred to by its private static variable `instance`.

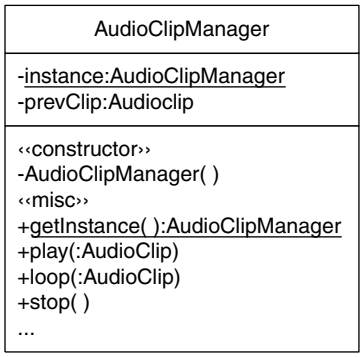

FIGURE 5.16 Audio clip manager.

The rest of the `AudioClipManager` class's methods are responsible for controlling the playing of audio clips. The `AudioClipManager` class has a private instance variable named `prevClip`, which is initially null and later refers to the last audio clip played. Before playing a new audio clip, the instance of the `AudioClipManager` class stops the audio clip referred to by the `prevClip`. That ensures that the previously requested audio clip stops before the next audio clip starts.

FORCES

- ☺ There must be at least one instance of a class. Even if the methods of your class use no instance data or only static data, you may need an instance of the class for a variety of reasons. Some of the more common reasons are that you need an instance to pass a method of another class or that you want to access the class indirectly through an interface.
- ☺ There should be no more than one instance of a class. This may be the case because you want to have only one source of some information. For example, you may want to have a single object that is responsible for generating a sequence of serial numbers.
- ☺ The one instance of a class must be accessible to all clients of that class.
- ☹ An object is cheap to create but takes up a lot of memory or continuously uses other resources during its lifetime.

SOLUTION

The Singleton pattern is relatively simple, since it involves only one class. The organization of this class is shown in Figure 5.17.

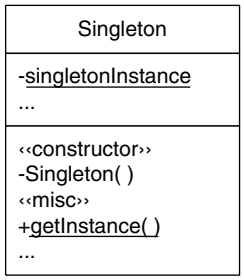

FIGURE 5.17 Singleton.

A singleton class has a static variable that refers to the one instance of the class you want to use. This instance is created when the class is loaded into memory. You should implement the class in a way that prevents other classes from creating additional instances. That means ensuring that all of the class's constructors are private.

To access the one instance of a singleton class, the class provides a static method, typically called `getInstance` or `get`*`Classname`*, which returns a reference to the one instance of the class.

IMPLEMENTATION

Though the Singleton pattern involves a relatively simple solution, there are a surprising number of subtleties in its implementation.

PRIVATE CONSTRUCTOR

To enforce the nature of a singleton class, you must code the class in a way that prevents other classes from directly creating instances of the class. A way to accomplish this is to declare the class's constructors private. Be careful to declare at least one private constructor. If a class does not declare any constructors, then the Java compiler automatically generates a default public constructor.

LAZY INSTANTIATION

A common variation on the Singleton pattern occurs in situations where the instance of a Singleton may not be needed. You do not know whether the program will need to use the singleton instance until the first time that the getInstance method is called. In situations like this, postpone creation of the instance until the first call to `getInstance`.

ALLOWING MORE THAN ONE INSTANCE

Another variation on the Singleton pattern stems from the fact that it has a class's instantiation policy encapsulated in the class itself. Because the instantiation policy is encapsulated in the class's `getInstance` method, it is possible to vary the creation policy. Some possible policies are to have `getInstance` alternately return one of two instances or to periodically create a new instance for `getInstance` to return.

This sort of situation can come up when you are using small, fixed individual objects to represent external resources and you want to balance the load between them. When you generalize this to an arbitrary or variable number of objects, you get the Object Pool pattern.

MAKING COPIES OF A SINGLETON

A singleton object is usually intended to be the only instance of its class. Even in the case in which you allow more than one instance of a singleton object, you generally want to ensure that creation of singleton objects is entirely under the control of the singleton class's `getInstance` object. This means that you don't want any other classes making copies of a singleton object.

One consequence of this is that a singleton class should not implement the `java.lang.Cloneable` interface. It should not be possible for another class to make a copy by calling a singleton object's `clone` method.

Serialization is a mechanism that Java provides for converting the contents of objects into a stream of bytes. *Deserialization* is a mechanism that Java provides for converting a stream of bytes created by serialization to objects that have the same contents as the original objects. There is a detailed description of serialization and deserialization on page 377.

Serialization can be used to copy objects, but copying is not usually the intended use of serialization. These are the two most common uses for serialization:

- Serialization is used to make objects persistent by writing them to a file as a stream of bytes. This allows persisted objects with the same contents to be recreated later on. This is the mechanism by which JavaBeans are saved and reconstituted.
- Serialization is used to support remote procedure calls using RMI. RMI is used by EJB (Enterprise JavaBeans). RMI uses serialization to pass argument values to remote procedure calls and also to pass the return value back to a remote caller.

You will generally not want to perform either of these functions with a singleton object. In general, Singleton objects should not be included in

a serialized stream of objects. You can ensure this by never directly serializing a singleton object and by not having any classes save a reference to a singleton object in its instance variables. Instead, have clients of a singleton class call the singleton class's `getInstance` methods every time they want to access a singleton object.

CONCURRENT GetInstance CALLS

If there is any chance that multiple threads may call a singleton class's `getInstance` method at the same time, then you will need to make sure that the `getInstance` method does not create multiple instances of the singleton class. Consider the following code:

```
public class Foo {
    private Foo myInstance;
    ...
    public static Foo getInstance() {
        if (myInstance==null) {
            myInstance = new Foo();
        } // if
        return myInstance;
    } // getInstance()
    ...
} // class Foo
```

If two threads call the `getInstance` method at the same time and there were no previous calls to the `getInstance` method, then both calls will see that the value of myInstance is null, and both calls will create an instance of Foo. To avoid this problem, you can declare the `getInstance` method to be synchronized. This will ensure that only one thread at a time is able to execute the `getInstance` method.

Declaring the `getInstance` method to be synchronized does add a small amount of overhead by forcing every call to the `getInstance` method to get a synchronization lock before it proceeds.

A SUBTLE BUG

There is a rather subtle bug that can occur in implementations of the Singleton pattern. It can cause a singleton class to create and initialize more than one instance of itself over time. The problem occurs in programs that refer to a singleton class only through other classes that are dynamically loaded, as described in the Dynamic Linkage pattern.

Some programs are organized so that they dynamically load a set of classes, use them for a while, and then stop using them. Applets, servlets, and MIDlets are managed this way. When a program stops using classes, the classes it is finished with are garbage-collected. This is normally a

good thing. If a program keeps no references to classes after it is finished with them and garbage collection of classes is enabled, the unused classes will eventually be garbage-collected.

This behavior can be a problem for singleton classes. If a singleton class is garbage-collected, it will be loaded again if there is another dynamic reference to it. After the class has been loaded a second time, the next request for its instance will return a new instance! This can produce unexpected results.

Suppose you have a singleton class whose purpose is to maintain performance statistics. Consider what happens if it is garbage-collected and reloaded. The first time its `getInstance` method is called after it is loaded a second time, it will return a new object. Its instance variables will have their initial values, and previously collected statistics will be lost.

If a class is loaded by a `ClassLoader` object, then it will not be garbage-collected until the `ClassLoader` object is eligible to be garbage-collected. If this is not a practical way for you to manage the lifetime of a singleton object, there is a more general way: Ensure that there is a reference, direct or indirect, from a live thread to the object that is not to be garbage-collected. The class in the following listing can be used to do just that.

```
public class ObjectPreserver implements Runnable {
    // This keeps this class and everything it references from
    // being garbage-collected
    private static ObjectPreserver lifeLine
      = new ObjectPreserver();

    // Since this class won't be garbage-collected, neither
    // will this HashSet or the object that it references.
    private static HashSet protectedSet = new HashSet();

    private ObjectPreserver() {
        new Thread(this).start();
    } // constructor()

    public synchronized void run() {
        try {
            wait();
        } catch (InterruptedException e) {
        } // try
    } // run()

    /**
     * Garbage collection of objects passed to this method
     * will be prevented until they are passed to the
     * unpreserveObject method.
     */
    public static void preserveObject(Object o) {
        protectedSet.add(o);
    } // preserveObject()
```

```
    /**
     * Objects passed to this method lose the protection
     * from garbage collection.
     */
    public static void unpreserveObject(Object o) {
        protectedSet.remove(o);
    } // unpreserveObject(Object)
} // class ObjectPreserver
```

If the class object that encapsulates a class or one of a class's instances is passed to the `preserveObject` method of the `ObjectPreserver` class shown in the preceding listing, then that class will not be garbage-collected.

CONSEQUENCES

- ☺ Exactly one instance of a singleton class exists.
- ☺ The singleton class's `getInstance` method encapsulates the creation policy for the singleton class so that classes that use a singleton class do not depend on the details of its instantiation.
- ● Other classes that want a reference to the one instance of the singleton class must get that instance by calling the class's `getInstance` static method, rather than by constructing the instance themselves.
- ☹ Subclassing a singleton class is awkward and results in imperfectly encapsulated classes. In order to subclass a singleton class, you must give it a constructor that is not private. Also, since static functions cannot be overridden, a subclass of a singleton class must leave its superclass's `GetInstance` method exposed.

JAVA API USAGE

The Java API class `java.lang.Runtime` is a singleton class. It has exactly one instance. It has no public constructors. To get a reference to its one instance, other classes must call its static method `getRuntime`.

CODE EXAMPLE

The following listing shows a Java class you can use to avoid playing two audio clips at the same time. The class is a singleton class. You can access its instance by calling its static `getInstance` method. When you play audio clips through that object, it stops the last audio clip it was playing before it starts the newly requested one. If you play all audio clips through the

AudioClipManager object, then there will never be more than one audio clip playing at the same time.

```
public class AudioClipManager implements AudioClip{
    private static AudioClipManager instance
      = new AudioClipManager();
    private AudioClip prevClip; // previously requested audio clip

    /**
     * This private constructor is defined so the compiler
     * won't generate a default public constructor.
     */
    private AudioClipManager() { }

    /**
     * Return a reference to the only instance of this class.
     */
    public static AudioClipManager getInstance() {
        return instance;
    } // getInstance()
...
    /**
     * Stop the previously requested audio clip and play the
     * given audio clip.
     * @param clip the new audio clip to play.
     */
    public void play(AudioClip clip) {
        if (prevClip != null)
          prevClip.stop();
        prevClip = clip;
        clip.play();
    } // play(AudioClip)
...
    /**
     * Stop the previously requested audio clip and play the
     * given audio clip in a loop.
     * @param clip the new audio clip to play.
     */
    public void loop(AudioClip clip) {
        if (prevClip != null)
          prevClip.stop();
        prevClip = clip;
        clip.loop();
    } // play(AudioClip)

    /**
     * Stops playing this audio clip.
     */
    public void stop() {
        if (prevClip != null)
          prevClip.stop();
    } // stop()
} // class AudioClipManager
```

RELATED PATTERNS

You can use the Singleton pattern with many other patterns. In particular, it is often used with the Abstract Factory, Builder, and Prototype patterns.

Cache Management. The Singleton pattern has some similarity to the Cache Management pattern. A Singleton is functionally similar to a Cache that contains only one object.

Object Pool. The Object Pool pattern is for managing an arbitrarily large collection of similar objects rather than just a single object.

Object Pool

SYNOPSIS

Manage the reuse of objects when a type of object is expensive to create or only a limited number of a kind of object can be created.

CONTEXT

Suppose you have been given the assignment of writing a class library to provide access to a proprietary database. Clients will send queries to the database through a network connection. The database server will receive queries through the network connection and return the results through the same connection.

For a program to query the database, it must have a connection to the database. A convenient way for programmers who will use the library to manage connections is for each part of a program that needs a connection to create its own connection. However, creating database connections that are not needed is bad for a few reasons:

- It can take a few seconds to create each database connection.
- The more connections there are to a database, the longer it takes to create new connections.
- Each database connection uses a network connection. Some platforms limit the number of network connections that they allow.

Your design for the library will have to reconcile these conflicting forces. The need to provide a convenient API for programmers pulls your design in one direction. The high expense of creating database connection objects and a possible limit to the number of concurrent database connections pulls your design in another direction. One way to reconcile these forces is to have the library manage database connections on behalf of the application that uses the library.

The library will use a strategy to manage database connections based on the premise that a program's database connections are interchangeable. So long as a database connection is in a state that allows it to convey a query to the database, it does not matter which of a program's database connections is used. Using this observation, the database access library will be designed to have a two-layer implementation of database connections.

A class called `Connection` will implement the upper layer. Programs that use the database access library will directly create and use `Connection` objects. `Connection` objects will identify a database, but will not directly encapsulate a database connection. Only while a `Connection` object is being used to send a query to a database and fetch the result will it be paired with a `ConnectionImpl` object. `ConnectionImpl` objects encapsulate an actual database connection.

The library will create and manage `ConnectionImpl` objects. It will manage `ConnectionImpl` objects by maintaining a pool of them that are not currently paired up with a `Connection` object. The library will create a `ConnectionImpl` object only when it needs to pair one with a `Connection` object and the pool of `ConnectionImpl` objects is empty. The class diagram in Figure 5.18 shows the classes that will be involved in managing the pool of `ConnectionImpl` objects.

A `Connection` object calls the `ConnectionPool` object's `AcquireImpl` method when it needs a `ConnectionImpl` object, passing it the name of the database it needs to be connected with. If any `ConnectionImpl` objects in the `ConnectionPool` object's collection are connected to the specified database, the `ConnectionPool` object returns one of those objects. If there are no such `ConnectionImpl` objects in the `ConnectionPool` object's collection, then it tries to create one and return it. If it is unable to create a `ConnectionImpl` object, it waits until an existing `ConnectionImpl` object is returned to the pool by a call to the `releaseImpl` method and then it returns that object.

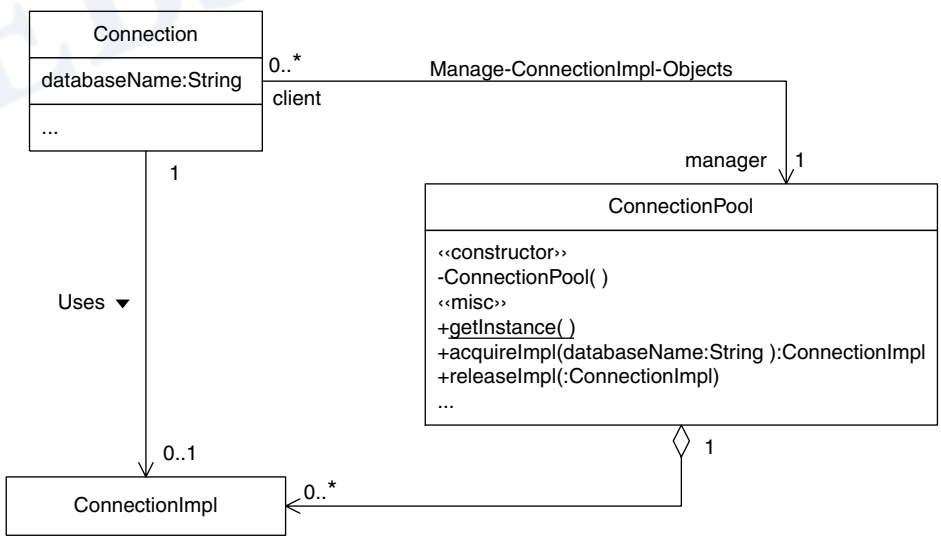

FIGURE 5.18 ConnectionImpl pool management.

The `ConnectionPool` class is a singleton. There should be only one instance of the `ConnectionPool` class. The class's constructor is private. Other classes access the one instance of the `ConnectionPool` class by calling its `getInstance` method, which is static.

There are many reasons a `ConnectionPool` object's `AcquireImpl` method may be unable to create a `ConnectionImpl` object. Among those reasons may be a limit on the number of `ConnectionImpl` objects it can create that connect to the same database. The reason for the limit is to be able to guarantee that a database will be able to support a minimum number of clients. Since there will be a maximum number of connections each database can support, limiting the number of connections to a database that each client can have allows you to guarantee support for a minimum number of client programs.

FORCES

- ☺ A program may not create more than a limited number of instances of a particular class.
- ☺ Creating instances of a particular class is sufficiently expensive that creating new instances of that class should be avoided.
- ☺ A program can avoid creating some objects by reusing objects that it has finished with rather than letting them be garbage-collected.
- ☺ The instances of a class are interchangeable. If you have multiple instances on hand, you can arbitrarily choose one to use for a purpose. It does not matter which one you choose.
- ☺ Resources can be managed centrally by a single object or in a decentralized way by multiple objects. It is easier to achieve predictable results by managing resources centrally with a single object.
- ☺ Some objects consume resources that are in short supply. Some objects may consume a lot of memory. Some objects may periodically check to see whether some condition is true, thereby consuming CPU cycles and perhaps network bandwidth. If the resources that an object consumes are in short supply, then it may be important that the object stop using the resource when the object is not being used.

SOLUTION

If instances of a class can be reused, avoid creating instances of the class by reusing them. The class diagram in Figure 5.19 shows the roles that classes play in the Object Pool pattern.

The `ConnectionPool` class is a singleton. There should be only one instance of the `ConnectionPool` class. The class's constructor is private. Other classes access the one instance of the `ConnectionPool` class by calling its `getInstance` method, which is static.

There are many reasons a `ConnectionPool` object's `AcquireImpl` method may be unable to create a `ConnectionImpl` object. Among those reasons may be a limit on the number of `ConnectionImpl` objects it can create that connect to the same database. The reason for the limit is to be able to guarantee that a database will be able to support a minimum number of clients. Since there will be a maximum number of connections each database can support, limiting the number of connections to a database that each client can have allows you to guarantee support for a minimum number of client programs.

FORCES

- ☺ A program may not create more than a limited number of instances of a particular class.
- ☺ Creating instances of a particular class is sufficiently expensive that creating new instances of that class should be avoided.
- ☺ A program can avoid creating some objects by reusing objects that it has finished with rather than letting them be garbage-collected.
- ☺ The instances of a class are interchangeable. If you have multiple instances on hand, you can arbitrarily choose one to use for a purpose. It does not matter which one you choose.
- ☺ Resources can be managed centrally by a single object or in a decentralized way by multiple objects. It is easier to achieve predictable results by managing resources centrally with a single object.
- ☺ Some objects consume resources that are in short supply. Some objects may consume a lot of memory. Some objects may periodically check to see whether some condition is true, thereby consuming CPU cycles and perhaps network bandwidth. If the resources that an object consumes are in short supply, then it may be important that the object stop using the resource when the object is not being used.

SOLUTION

If instances of a class can be reused, avoid creating instances of the class by reusing them. The class diagram in Figure 5.19 shows the roles that classes play in the Object Pool pattern.

Here are descriptions of the roles classes play in the Object Pool pattern, as shown in Figure 5.19:

Reusable. Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.

Client. Instances of classes in this role use `Reusable` objects.

ReusablePool. Instances of classes in this role manage `Reusable` objects for use by `Client` objects. It is usually desirable to keep all `Reusable` objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the `ReusablePool` class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its `getInstance` method to get the one instance of the `ReusablePool` class.

A `Client` object calls a `ReusablePool` object's `acquireReusable` method when it needs a `Reusable` object. A `ReusablePool` object maintains a collection of `Reusable` objects. It uses the collection of `Reusable` objects to contain a pool of `Reusable` objects that are not currently in use. If there are any `Reusable` objects in the pool when the `acquireReusable` method is called, it removes a `Reusable` object from the pool and returns it. If the pool is empty, then the `acquireReusable` method creates a Reusable object if it can. If the `acquireReusable` method can-

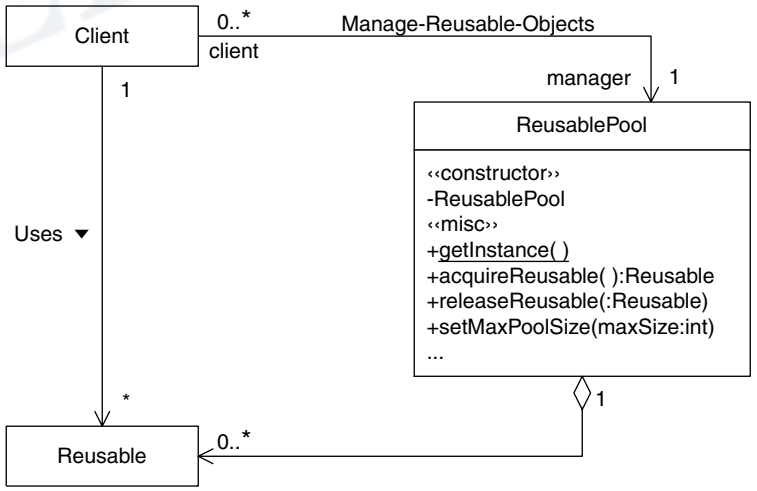

FIGURE 5.19 Object Pool pattern.

not create a new `Reusable` object, then it waits until a `Reusable` object is returned to the collection.

`Client` objects pass a `Reusable` object to a `ReusablePool` object's `releaseReusable` method when they are finished with the object. The `releaseReusable` method returns a `Reusable` object to the pool of `Reusable` objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of `Reusable` objects that may exist. In such cases, the `ReusablePool` object that creates `Reusable` objects is responsible for not creating more than a specified maximum number of `Reusable` objects. If `ReusablePool` objects are responsible for limiting the number of objects they will create, then the `ReusablePool` class will have a method for specifying the maximum number of objects to be created. That method is indicated in Figure 5.19 as `setMaxPoolSize`.

IMPLEMENTATION

Here are some of the issues to consider when implementing the Object Pool pattern.

ENSURING A MAXIMUM NUMBER OF INSTANCES

In many cases, the object that manages an object pool is supposed to limit the number of instances of a class that can be created. It is easy for an object to limit the number of objects it creates. However, to robustly enforce a limit on the total number of objects created, the object responsible for managing the object pool must be the only object able to create those objects.

You can ensure that a class is instantiated only by the class that manages the object pool. You can do that by making the managed class's constructor(s) private and implementing the pool management class as a static member class of the managed class. If you do not have control over the structure of the class whose instances are to be managed, you may be able to add that structure to the class through inheritance.

DATA STRUCTURE

When there is a limit on the number of objects that may be created or just a limit on the size of the object pool, using a simple array is usually the best way to implement the object pool. When there is no limit on the size of the object pool, using an `ArrayList` is an appropriate way to implement the object pool.

USING SOFT REFERENCES

The Object Pool pattern keeps objects that are not being used available for reuse. If the program that is using an object pool is running out of memory, then you would like the garbage collector to be able to remove objects from the pool and reclaim the memory that they occupy. You can arrange for the garbage collector to do this by using soft references.

Soft references are implemented in the Java API by the class `java.lang.ref.SoftReference`. A reference to another object is passed to the constructor of a `SoftReference` object. Immediately after a `SoftReference` object is constructed, its `get` method returns the object reference that was passed to its constructor. The interesting thing about `SoftReference` objects is that they are special to the garbage collector. If the only live reference to an object is through a `SoftReference` object, then the garbage collector will set the reference in the `SoftReference` object to null so that it can safely reclaim the storage occupied by the referenced object.

If the object pool refers to objects in the pool through soft references, then the garbage collector will reclaim the storage occupied by the objects if there are no other references to the objects and the Java virtual machine (JVM) is running low on memory. This use of the `SoftReference` class is shown in the code example.

LIMITING THE SIZE OF THE POOL

There are some cases where soft references are not a satisfactory solution to limiting the excess resources used by pool-managed objects.

- If the objects being managed use an external resource, you may not always want to wait for the garbage collector to finalize the objects before freeing up the resource.
- If your program needs to run on JVMs older than version 1.2, you cannot use soft references.

An alternative to using soft references is to limit the number of objects that may be in the object pool. If there is a limit on the number of objects in an object pool, it is generally less than the number of objects that the pool will allow to exist at once.

When a pool-managed object is no longer needed, it is released back to the `ReusablePool` object. If the pool already contains its maximum number of objects, then the released object is not added to the pool. If the object is tying up any external resources, it is told to release them. Since the object is not added to the pool, the object may be eligible for garbage collection at this time.

The techniques of limiting the size of a pool and using soft references are generally not used together. Because the garbage collector can clear a soft reference at any time, there is no way of ensuring that a pool always has an accurate count of the number of objects it contains.

MANAGING STATEFUL OBJECTS

One of the assumptions that underlie an object pool is that the objects it manages are interchangeable. When a client acquires an object from a pool, it expects the object to be in a known state. If it is possible for clients to alter the state of pool-managed objects, there has to be a way for the pool to ensure that its objects return to the expected state. These are the most common ways this is done:

- The pool explicitly resets the state of an object before a client acquires the object. This may not be possible in some cases. For example, once a database connection is closed, it may not be possible for the pool to reopen it. Besides, you really don't want clients closing the database connection.
- You can prevent clients from changing the state of pool-managed objects by using the Decorator pattern. For example, to do this with a database connection, you would create a wrapper object that implements the same interface as a real database connect object. The wrapper object would delegate all of its methods to a real database connection object, except for the method that closes the connection. That method would be implemented to do nothing.

CONSEQUENCES

- ☺ Using the Object Pool pattern avoids the creation of objects. It works best when the demand for objects does not vary greatly over time.
- ☺ Keeping the logic to manage the creation and reuse of a class's instances in a separate class from the class whose instances are being managed results in a more cohesive design. It eliminates interactions between the implementation of a creation and reuse policy and the implementation of the managed class's functionality.

CODE EXAMPLE

Implementations of the Object Pool pattern use one of two techniques to ensure that they do not consume an excessive amount of memory with objects awaiting reuse. Either they use soft references to refer to the

objects or they limit the number of objects that may be in the pool. Since these two techniques are somewhat different, there is an example of each one in this section. Both classes are generic. They can be used to keep a pool of any kind of objects.

Here is a listing of a class that implements the object pool using soft references.

```
public class SoftObjectPool implements ObjectPoolIF {
    /**
     * This collection contains the objects that are waiting
     * to be reused.
     */
    private ArrayList pool ;
```

Because there is no fixed limit on the number of objects that can be in the pool awaiting reuse, this class uses an `ArrayList` object to contain its collection of objects.

Because this class is generic, it does not know how to create the objects that it manages. Instead, it delegates the responsibility for creating objects to an object that implements an interface named `CreationIF`. The `CreationIF` interface is listed toward the end of this section.

```
    private CreationIF creator ;
```

Instances of this class are responsible for limiting the number of pool-managed objects that may exist at one time. The `instanceCount` variable contains the number of pool-managed objects that currently exist. The `maxInstances` variable contains the maximum number of pool-managed objects that may be under management by this object pool. An object pool will not create a new object unless the value of `instanceCount` is less than the value of `maxInstances`.

```
    private int instanceCount;
    private int maxInstances ;
```

Because this class is a generic object pool, it does not know in advance what kind of objects its clients want it to manage. As an inexpensive sanity check, this class requires that a class be passed as one of the arguments to its constructors. When an object is released to the pool for reuse, the pool checks that the object is an instance of the specified class. The class used for this purpose is the value of the `poolClass` variable.

```
    private Class poolClass;

    /**
     * Constructor
     *
```

```
     * @param poolClass
     *        The Class to instantiate to create Pool Objects.
     * @param creator
     *        The object that the pool will delegate the
     *        responsibility for creating objects that it will
     *        manage.
     */
    public SoftObjectPool( Class poolClass,
                           CreationIF creator ) {
        this( poolClass, creator, Integer.MAX_VALUE);
    } // constructor(Class, CreationIF, int)

    /**
     * Constructor
     *
     * @param poolClass
     *        The Class to instantiate to create Pool Objects.
     * @param creator
     *        The object that the pool will delegate the
     *        responsibility for creating objects that it will
     *        manage.
     * @param maxInstances
     *        The maximum number of instances of the poolClass
     *        class that the pool should allow to exist at one
     *        time.  If the pool is asked to produce an
     *        instance of poolClass when there are no objects
     *        in the pool awaiting reuse and there are this
     *        many pool-managed obects in use, then the pool
     *        will not produce an object until an object is
     *        returned to the pool for reuse.
     */
    public SoftObjectPool( Class poolClass,
                           CreationIF creator,
                           int maxInstances ) {
        this.creator = creator;
        this.poolClass = poolClass;
        pool = new ArrayList();
    } // constructor(Class, CreationIF, int, int)

    /**
     * Return the number of objects in the pool that are
     * awaiting reuse.  The actual number may be less than
     * this because what is returned is the number of soft
     * references in the pool.  Any or all of the soft
     * references may have been cleared by the garbage
     * collector.
     */
    public int getSize() {
        synchronized (pool) {
            return pool.size();
        } // synchronized
    } // getSize()
```

```
    /**
     * Return the number of pool-managed objects that
     * currently exist.
     */
    public int getInstanceCount() {
        return instanceCount;
    } // getInstanceCount()

    /**
     * Return the maximum number of pool-managed objects that
     * this pool will allow to exist at one time.
     */
    public int getMaxInstances() {
        return maxInstances;
    } // getMaxInstances()

    /**
     * Set the maximum number of objects that this pool will
     * allow to exist at one time.
     *
     * If the pool is asked to produce an instance of
     * poolClass when there are no objects in the pool
     * awaiting reuse and there are already this many
     * pool-managed objects in use, then the pool will not
     * produce an object until an object is returned to the
     * pool for reuse.
     *
     * @param newValue
     *        The new value for the maximum number of
     *        pool-managed objects that may exist at one time.
     *        Setting this to a value that is less than the
     *        value returned by the getInstanceCount method
     *        will not cause any objects to be deleted.
     *        It will just prevent any new pool-managed
     *        objects from being created.
     */
    public void setMaxInstances(int newValue) {
        maxInstances = newValue;
    } // setMaxInstances()

    /**
     * Return an object from the pool.  If there is no object
     * in the pool, one will be created unless the number of
     * pool-managed objects is greater than or equal to the
     * value returned by the getMaxInstances method.  If the
     * number of pool-managed objects exceeds this amount,
     * then this method returns null.
     */
    public Object getObject() {
        synchronized (pool) {
            Object thisObject = removeObject();
            if (thisObject!=null) {
                return thisObject;
```

```
            } // if thisObject
            if (getInstanceCount() < getMaxInstances()){
                return createObject();
            } else {
                return null;
            } // if
        } // synchronized (pool)
    } // getObject()

    /**
     * Return an object from the pool.  If there is no object
     * in the pool, one will be created unless the number of
     * pool-managed objects is greater than or equal to the
     * value returned by the getMaxInstances method.  If the
     * number of pool-managed objects exceeds this amount,
     * then this method will wait until an object becomes
     * available for reuse.
     *
     * @throws InterruptedException
     *         If the calling thread is interrupted.
     */
    public Object waitForObject() throws InterruptedException {
        synchronized (pool) {
            Object thisObject = removeObject();
            if (thisObject!=null) {
                return thisObject;
            } // if thisObject
            if (getInstanceCount() < getMaxInstances()){
                return createObject();
            } else {
                do {
                    // Wait until notified that an object has
                    // been put back in the pool.
                    pool.wait();
                    thisObject = removeObject();
                } while (thisObject==null);
                return thisObject;
            } // if
        } // synchronized (pool)
    } // waitForObject()

    /**
     * Remove an object from the pool array and return it.
     */
    private Object removeObject() {
        while (pool.size()>0) {
            SoftReference thisRef
              = (SoftReference)pool.remove(pool.size()-1);
            Object thisObject = thisRef.get();
            if (thisObject!=null) {
                return thisObject;
            } // if thisObject
            instanceCount—;
```

```
            } // while
            return null;
        } // removeObject()

        /**
         * Create an object to be managed by this pool.
         */
        private Object createObject() {
            Object newObject = creator.create();
            instanceCount++;
            return newObject;
        } // createObject()

        /**
         * Release an object to the Pool for reuse.
         *
         * @param obj
         *        The object that is available for reuse.
         */
        public void release( Object obj ) {
            // no nulls
            if ( obj == null ) {
                throw new NullPointerException();
            } // if null
            if ( !poolClass.isInstance(obj)) {
                String actualClassName = obj.getClass().getName();
                throw new ArrayStoreException(actualClassName);
            } // if isInstance
            synchronized (pool) {
                pool.add(obj);
                // Notify a waiting thread that we have put an
                // object in the pool.
                pool.notify();
            } // synchronized
        } // release()
    } // class SoftObjectPool
```

Here is a listing of a class that limits the number of objects that may be awaiting reuse instead of using soft references. The portions of it that are the same as the previous class are left out of this listing.

```
public class ObjectPool implements ObjectPoolIF {
    private int size ;
```

Because there is a definite limit on the number of objects this class will allow to be awaiting reuse, this class can use a simple array to contain them. It uses the instance variable named size to count the number of objects that are actually awaiting reuse.

```
    /**
     * This array contains the objects that are waiting to be
     * reused.  It is managed as a stack.
```

```
     */
    private Object[] pool ;
...
    /**
     * Internal operations are synchronized on this object.
     */
    private Object lockObject = new Object();
```

See the discussion of the Internal Lock Object pattern for a more detailed explanation of lock objects.

```
...
    /**
     * Constructor
     *
     * @param poolClass
     *        The Class to instantiate to create Pool Objects.
     * @param creator
     *        The object that the pool will delegate the
     *        responsibility for creating objects that it will
     *        manage.
     * @param capacity
     *        The number of currently unused objects that this
     *        object pool may contain at once.
     * @param maxInstances
     *        The maximum number of instances of the poolClass
     *        class that the pool should allow to exist at one
     *        time.
     */
    public ObjectPool( Class poolClass,
                       CreationIF creator,
                       int capacity,
                       int maxInstances ) {
        size = 0;
        this.creator = creator;
        this.maxInstances = maxInstances;
        pool = (Object[])Array.newInstance(poolClass,
                                           capacity);
    } // constructor(Class, CreationIF, int, int)

    /**
     * Return the number of objects in the pool that are
     * awaiting reuse.
     */
    public int getSize() {
        return size;
    } // getSize()

    /**
     * Return the maximum number of objects that may be in the
     * pool awaiting reuse.
     */
    public int getCapacity() {
```

```
        return pool.length;
    } // getCapacity()

    /**
     * Set the maximum number of objects that may be in the
     * pool awaiting reuse.
     *
     * @param newValue
     *        The new value for the maximum number of objects
     *        that may be in the pool awaiting reuse.  This
     *        must be greater than zero.
     */
    public void setCapacity(int newValue) {
        if (newValue<=0) {
            String msg = "Capacity must be greater than zero:"
                       + newValue;
            throw new IllegalArgumentException(msg);
        } // if
        synchronized (lockObject) {
            Object[] newPool = new Object[newValue];
            System.arraycopy(pool, 0, newPool, 0, newValue);
            pool = newPool;
        } // synchronized
    } // setCapacity(int)

...

    /**
     * Return an object from the pool.  If there is no object in
     * the pool, one will be created unless the number of
     * pool-managed objects is greater than or equal to the
     * value returned by the <code>getMaxInstances</code>
     * method.  If the number of pool-managed objects exceeds
     * this amount, then this method returns null.
     */
    public Object getObject() {
        synchronized (lockObject) {
            if (size>0) {
                return removeObject();
            } else if (getInstanceCount() < getMaxInstances()){
                return createObject();
            } else {
                return null;
            } // if
        } // synchronized (lockObject)
    } // getObject()

    /**
     * Return an object from the pool.  If there is no object in
     * the pool, one will be created unless the number of
     * pool-managed objects is greater than or equal to the
     * value returned by the <code>getMaxInstances</code>
     * method.  If the number of pool-managed objects exceeds
```

```
     * this amount, then this method will wait until an object
     * becomes available for reuse.
     *
     * @throws InterruptedException
     *         If the calling thread is interrupted.
     */
    public Object waitForObject() throws InterruptedException {
        synchronized (lockObject) {
            if (size>0) {
                return removeObject();
            } else if (getInstanceCount() < getMaxInstances()){
                return createObject();
            } else {
                do {
                    // Wait until notified that an object has
                    // been put back in the pool.
                    wait();
                } while(size<=0);
                return removeObject();
            } // if
        } // synchronized (lockObject)
    } // waitForObject()

    /**
     * Remove an object from the pool array and return it.
     */
    private Object removeObject() {
        size--;
        return pool[size];
    } // removeObject()

...

    /**
     * Release an object to the Pool for reuse.
     *
     * @param obj
     *        The object that is available for reuse.
     * @throws ArrayStoreException
     *         If the given object is not an instance of the
     *         class passed to this pool object's constructor.
     * @throws NullPointerException
     *         If the given object is null.
     */
    public void release( Object obj ) {
        // no nulls
        if ( obj == null ) {
            throw new NullPointerException();
        } // if null
        synchronized (lockObject) {
            if (getSize() < getCapacity()) {
                pool[size] = obj;
                size++;
```

```
                // Notify a waiting thread that we have put an
                // object in the pool.
                lockObject.notify();
            } // if
        } // synchronized
    } // release()
} // class ObjectPool
```

Here is a listing of the `CreationIF` interface.

```
public interface CreationIF {
    /**
     * Return a newly created object.
     */
    public Object create() ;
} // interface creationIF
```

RELATED PATTERNS

Cache Management. The Cache Management pattern manages the reuse of specific instances of a class. The Pool pattern manages and creates instances of a class that can be used interchangeably.

Factory Method. The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation.

Singleton. Objects that manage object pools are usually singletons.

Thread Pool. The Thread Pool pattern (discussed in *Patterns in Java, Volume 3*) is a specialized form of the Object Pool pattern.

Lock Object. The Lock Object pattern may be used in the implementation of the Object Pool pattern.