Design Patterns

M.1 Introduction

Most of the examples provided in this book are relatively small. These examples do not require an extensive design process, because they use only a few classes and illustrate introductory programming concepts. However, some programs are more complex—they can require thousands of lines of code or even more, contain many interactions among objects and involve many user interactions. Larger systems, such as air-traffic control systems or the systems that control a major bank’s thousands of automated teller machines, could contain millions of lines of code. Effective design is crucial to the proper construction of such complex systems.

Over the past decade, the software-engineering industry has made significant progress in the field of design patterns—proven architectures for constructing flexible and maintainable object-oriented software. Using design patterns can substantially reduce the complexity of the design process. Designing an air-traffic control system will be a somewhat less formidable task if developers use design patterns. Design patterns benefit system developers by

- helping to construct reliable software using proven architectures and accumulated industry expertise.
- promoting design reuse in future systems.
- helping to identify common mistakes and pitfalls that occur when building systems.
- helping to design systems independently of the language in which they will ultimately be implemented.
- establishing a common design vocabulary among developers.
- shortening the design phase in a software-development process.

The notion of using design patterns to construct software systems originated in the field of architecture. Architects use a set of established architectural design elements, such as arches and columns, when designing buildings. Designing with arches and columns is
a proven strategy for constructing sound buildings—these elements may be viewed as architectural design patterns.

In software, design patterns are neither classes nor objects. Rather, designers use design patterns to construct sets of classes and objects. To use design patterns effectively, designers must familiarize themselves with the most popular and effective patterns used in the software-engineering industry. In this appendix, we discuss fundamental object-oriented design patterns and architectures, as well as their importance in constructing well-engineered software.

This appendix presents several design patterns in Java, but these can be implemented in any object-oriented language, such as C++ or Visual Basic. We describe several design patterns used by Sun Microsystems in the Java API. We use design patterns in many programs in this book, which we will identify throughout our discussion. These programs provide examples of the use of design patterns to construct reliable, robust object-oriented software.

History of Object-Oriented Design Patterns

During 1991–1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—collectively known as the “Gang of Four”—used their combined expertise to write the book Design Patterns: Elements of Reusable Object-Oriented Software. This book describes 23 design patterns, each providing a solution to a common software design problem in industry. The book groups design patterns into three categories—creational design patterns, structural design patterns and behavioral design patterns. Creational design patterns describe techniques to instantiate objects (or groups of objects). Structural design patterns allow designers to organize classes and objects into larger structures. Behavioral design patterns assign responsibilities to classes and objects.

The gang-of-four book showed that design patterns evolved naturally through years of industry experience. In his article Seven Habits of Successful Pattern Writers,1 John Vlissides states that “the single most important activity in pattern writing is reflection.” This statement implies that, to create patterns, developers must reflect on, and document, their successes (and mistakes). Developers use design patterns to capture and employ this collective industry experience, which ultimately helps them avoid repeating the same mistakes. New design patterns are being created all the time and are introduced rapidly to designers worldwide via the Internet.

Design patterns are a somewhat advanced topic that might not appear in most introductory course sequences. As you proceed in your Java studies, design patterns will surely increase in value. If you are a student and your instructor does not plan to include this material in your course, we encourage you to read this material on your own.

Section M.8 presents a list of Web resources pertaining to design patterns and their relevance to Java programming. As you proceed through this appendix, you may want to consult the provided URLs to learn more about a particular design pattern introduced in the text, or to read about new developments in the design patterns community.

M.2 Introducing Creational, Structural and Behavioral Design Patterns

In Section M.1, we mentioned that the “Gang of Four” described 23 design patterns using three categories—creational, structural and behavioral. In this and the remaining sections

of this appendix, we discuss design patterns in each category and their importance, and how each pattern relates to the Java material in the book. For example, several Java Swing components that we introduce in Chapter 11 and Chapter 22 use the Composite design pattern. Figure M.1 identifies the 18 Gang of Four design patterns discussed in this appendix.

Many popular patterns have been documented since the Gang-of-Four book—these include the concurrency design patterns, which are especially helpful in the design of multithreaded systems. Section M.4 discusses some of these patterns used in industry. Architectural patterns, as we discuss in Section M.5, specify how subsystems interact with each other. Figure M.2 lists the concurrency patterns and architectural patterns that we discuss in this appendix.

### M.2.1 Creational Design Patterns

**Creational design patterns** address issues related to the creation of objects, such as preventing a system from creating more than one object of a class (the Singleton creational design pattern) or deferring until execution time the decision as to what types of objects are going to be created (the purpose of the other creational design patterns discussed here).

For example, suppose we are designing a 3-D drawing program, in which the user can create several 3-D geometric objects, such as cylinders, spheres, cubes, tetrahedrons, etc. Further suppose that each shape in the drawing program is represented by an object. At compile time, the program does not know what shapes the user will choose to draw. Based

<table>
<thead>
<tr>
<th>Section</th>
<th>Creational design patterns</th>
<th>Structural design patterns</th>
<th>Behavioral design patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section M.2</td>
<td>Singleton</td>
<td>Proxy</td>
<td>Memento, State</td>
</tr>
<tr>
<td>Section M.3</td>
<td>Factory Method</td>
<td>Adapter, Bridge, Composite</td>
<td>Chain of Responsibility, Command, Observer, Strategy, Template Method</td>
</tr>
<tr>
<td>Section M.5</td>
<td>Abstract Factory</td>
<td>Decorator, Facade</td>
<td></td>
</tr>
<tr>
<td>Section M.6</td>
<td>Prototype</td>
<td></td>
<td>Iterator</td>
</tr>
</tbody>
</table>

**Fig. M.1** | 18 Gang-of-Four design patterns discussed in this appendix.

<table>
<thead>
<tr>
<th>Section</th>
<th>Concurrency design patterns</th>
<th>Architectural patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section M.4</td>
<td>Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock, Two-Phase Termination</td>
<td></td>
</tr>
<tr>
<td>Section M.5</td>
<td></td>
<td>Model-View-Controller, Layers</td>
</tr>
</tbody>
</table>

**Fig. M.2** | Concurrency design patterns and architectural patterns discussed in this appendix.
on user input, this program should be able to determine the class from which to instantiate an appropriate object for the shape the user selected. If the user creates a cylinder in the GUI, our program should “know” to instantiate an object of class Cylinder. When the user decides what geometric object to draw, the program should determine the specific subclass from which to instantiate that object.

The Gang-of-Four book describes five creational patterns (four of which we discuss in this appendix):

- Abstract Factory (Section M.5)
- Builder (not discussed)
- Factory Method (Section M.3)
- Prototype (Section M.6)
- Singleton (Section M.2)

**Singleton**

Occasionally, a system should contain exactly one object of a class—that is, once the program instantiates that object, the program should not be allowed to create additional objects of that class. For example, some systems connect to a database using only one object that manages database connections, which ensures that other objects cannot initialize unnecessary connections that would slow the system. The **Singleton design pattern** guarantees that a system instantiates a maximum of one object of a class.

Figure M.3 demonstrates Java code using the Singleton design pattern. Line 4 declares class Singleton as final, so subclasses cannot be created that could provide multiple instantiations. Lines 10–13 declare a private constructor—only class Singleton can instantiate a Singleton object using this constructor. Line 7 declares a static reference to a Singleton object and invokes the private constructor. This creates the one instance of

```java
// Singleton.java
// Demonstrates Singleton design pattern

public final class Singleton
{
    // Singleton object to be returned by getInstance
    private static final Singleton singleton = new Singleton();

    // private constructor prevents instantiation by clients
    private Singleton()
    {
        System.err.println("Singleton object created.");
    } // end Singleton constructor

    // return static Singleton object
    public static Singleton getInstance()
    {
        return singleton;
    } // end method getInstance
} // end class Singleton
```

**Fig. M.3** | Class Singleton ensures that only one object of its class is created.
class Singleton that will be provided to clients. When invoked, static method `Singleton.getInstance` (lines 16–19) simply returns a copy of this reference.

Lines 9–10 of class SingletonTest (Fig. M.4) declare two references to Singleton objects—firstSingleton and secondSingleton. Lines 13–14 call method `getInstance` and assign Singleton references to firstSingleton and secondSingleton, respectively. Line 17 tests whether these references both refer to the same Singleton object. Figure M.4 shows that firstSingleton and secondSingleton indeed are both references to the same Singleton object, because each time method `getInstance` is called, it returns a reference to the same Singleton object.

### M.2.2 Structural Design Patterns

**Structural design patterns** describe common ways to organize classes and objects in a system. The Gang of Four book describes seven structural design patterns (six of which we discuss in this appendix):

- Adapter (Section M.3)
- Bridge (Section M.3)
- Composite (Section M.3)
- Decorator (Section M.5)
- Facade (Section M.5)
- Flyweight (not discussed)
- Proxy (Section M.2)

```java
1 // SingletonTest.java
2 // Attempt to create two Singleton objects
3
4 public class SingletonTest
5 {
6     // run SingletonExample
7     public static void main(String args[]) 
8     {
9         Singleton firstSingleton;
10        Singleton secondSingleton;
11
12        // create Singleton objects
13        firstSingleton = Singleton.getInstance();
14        secondSingleton = Singleton.getInstance();
15
16        // the "two" Singletons should refer to same Singleton
17        if (firstSingleton == secondSingleton)
18            System.err.println("firstSingleton and secondSingleton refer to the same Singleton");
19     } // end main
20 } // end class SingletonTest
```

**Fig. M.4** Class `SingletonTest` create Singleton object more than once.
Proxy
An applet should always display something while images load to provide positive feedback to users, so they know the applet is working. Whether that “something” is a smaller image or a string of text informing the user that the images are loading, the Proxy design pattern can be applied to achieve this effect. Consider loading several large images (several megabytes) in a Java applet. Ideally, we would like to see these images instantaneously—however, loading large images into memory can take time to complete (especially across a network). The Proxy design pattern allows the system to use one object—called a proxy object—in place of another. In our example, the proxy object could be a gauge that shows the user what percentage of a large image has been loaded. When this image finishes loading, the proxy object is no longer needed—the applet can then display an image instead of the proxy. Class java.awt.JProgressBar can be used to create such proxy objects.

M.2.3 Behavioral Design Patterns
Behavioral design patterns provide proven strategies to model how objects collaborate with one another in a system and offer special behaviors appropriate for a wide variety of applications. Let us consider the Observer behavioral design pattern—a classic example illustrating collaborations between objects. For example, GUI components collaborate with their listeners to respond to user interactions. GUI components use this pattern to process user interface events. A listener observes state changes in a particular GUI component by registering to handle its events. When the user interacts with that GUI component, the component notifies its listeners (also known as its observers) that its state has changed (e.g., a button has been pressed).

We also consider the Memento behavioral design pattern—an example of offering special behavior for many applications. The Memento pattern enables a system to save an object’s state, so that state can be restored at a later time. For example, many applications provide an “undo” capability that allows users to revert to previous versions of their work.

The Gang-of-Four book describes 11 behavioral design patterns (eight of which we discuss in this appendix):

- Chain of Responsibility (Section M.3)
- Command (Section M.3)
- Interpreter (not discussed)
- Iterator (Section M.2)
- Mediator (not discussed)
- Memento (Section M.2)
- Observer (Section M.3)
- State (Section M.2)
- Strategy (Section M.3)
- Template Method (Section M.3)
- Visitor (not discussed)
**Memento**
Consider a painting program, which allows a user to create graphics. Occasionally the user may position a graphic improperly in the drawing area. Painting programs offer an “undo” feature that allows the user to unwind such an error. Specifically, the program restores the drawing area to its state before the user placed the graphic. More sophisticated painting programs offer a history, which stores several states in a list, allowing the user to restore the program to any state in the history. The Memento design pattern allows an object to save its state, so that—if necessary—the object can be restored to its former state.

The Memento design pattern requires three types of objects. The originator object occupies some state—the set of attribute values at a specific time in program execution. In our painting-program example, the drawing area acts as the originator, because it contains attribute information describing its state—when the program first executes, the area contains no elements. The memento object stores a copy of necessary attributes associated with the originator’s state (i.e., the memento saves the drawing area’s state). The memento is stored as the first item in the history list, which acts as the caretaker object—the object that contains references to all memento objects associated with the originator. Now, suppose that the user draws a circle in the drawing area. The area contains different information describing its state—a circle object centered at specified x-y coordinates. The drawing area then uses another memento to store this information. This memento becomes the second item in the history list. The history list displays all mementos on screen, so the user can select which state to restore. Suppose that the user wishes to remove the circle—if the user selects the first memento, the drawing area uses it to restore the blank drawing area.

**State**
In certain designs, we must convey an object’s state information or represent the various states that an object can occupy. The State design pattern uses an abstract superclass—called the State class—which contains methods that describe behaviors for states that an object (called the context object) can occupy. A State subclass, which extends the State class, represents an individual state that the context can occupy. Each State subclass contains methods that implement the State class’s abstract methods. The context contains exactly one reference to an object of the State class—this object is called the state object. When the context changes state, the state object references the State subclass object associated with that new state.

**M.2.4 Conclusion**
In this section, we listed the three types of design patterns introduced in the Gang-of-Four book, we identified 18 of these design patterns that we discuss in this appendix and we discussed specific design patterns, including Singleton, Proxy, Memento and State. In the next section, we introduce some design patterns associated with AWT and Swing GUI components. After reading this section, you should understand better how Java GUI components take advantage of design patterns.

**M.3 Design Patterns Used in Packages java.awt and javax.swing**
This section introduces those design patterns associated with Java GUI components. It should help you understand better how these components take advantage of design patterns and how developers integrate design patterns with Java GUI applications.
M.3 Design Patterns Used in Packages java.awt and javax.swing

M.3.1 Creational Design Patterns

Now, we continue our treatment of creational design patterns, which provide ways to instantiate objects in a system.

Factory Method

Suppose that we are designing a system that opens an image from a specified file. Several different image formats exist, such as GIF and JPEG. We can use method `createImage` of class `java.awt.Component` to create an `Image` object. For example, to create a JPEG and GIF image in an object of a `Component` subclass—such as a `JPanel` object—we pass the name of the image file to method `createImage`, which returns an `Image` object that stores the image data. We can create two `Image` objects, each containing data for two images having entirely different structures. For example, a JPEG image can hold up to 16.7 million colors, a GIF image up to only 256. Also, a GIF image can contain transparent pixels that are not rendered on screen, whereas a JPEG image cannot.

Class `Image` is an abstract class that represents an image we can display on screen. Using the parameter passed by the programmer, method `createImage` determines the specific `Image` subclass from which to instantiate the `Image` object. We can design systems to allow the user to specify which image to create, and method `createImage` will determine the subclass from which to instantiate the `Image`. If the parameter passed to method `createImage` references a JPEG file, method `createImage` instantiates and returns an object of an `Image` subclass suitable for JPEG images. If the parameter references a GIF file, `createImage` instantiates and returns an object of an `Image` subclass suitable for GIF images.

Method `createImage` is an example of the Factory Method design pattern. The sole purpose of this factory method is to create objects by allowing the system to determine which class to instantiate at runtime. We can design a system that allows a user to specify what type of image to create at runtime. Class `Component` might not be able to determine which `Image` subclass to instantiate until the user specifies the image to load. For more information on method `createImage`, visit

java.sun.com/j2se/5.0/docs/api/java/awt/Component.html

M.3.2 Structural Design Patterns

We now discuss three more structural design patterns. The Adapter design pattern helps objects with incompatible interfaces collaborate with one another. The Bridge design pattern helps designers enhance platform independence in their systems. The Composite design pattern provides a way for designers to organize and manipulate objects.

Adapter

The Adapter design pattern provides an object with a new interface that adapts to another object’s interface, allowing both objects to collaborate with one another. We might liken the adapter in this pattern to an adapter for a plug on an electrical device—electrical sockets in Europe are shaped differently from those in the United States, so an adapter is needed to plug an American device into a European socket and vice versa.

Java provides several classes that use the Adapter design pattern. Objects of the concrete subclasses of these classes act as adapters between objects that generate certain events and objects that handle the events. For example, a `MouseEventAdapter`, which we explained in
Section 11.14, adapts an object that generates MouseEvents to an object that handles MouseEvents.

**Bridge**

Suppose that we are designing class Button for both the Windows and Macintosh operating systems. Class Button contains specific button information such as an ActionListener and a label. We design classes Win32Button and MacButton to extend class Button. Class Win32Button contains “look-and-feel” information on how to display a Button on the Windows operating system, and class MacButton contains “look-and-feel” information on how to display a Button on the Macintosh operating system.

Two problems arise here. First, if we create new Button subclasses, we must create corresponding Win32Button and MacButton subclasses. For example, if we create class ImageButton (a Button with an overlapping Image) that extends class Button, we must create additional subclasses Win32ImageButton and MacImageButton. In fact, we must create Button subclasses for every operating system we wish to support, which increases development time. Second, when a new operating system enters the market, we must create additional Button subclasses specific to it.

The Bridge design pattern avoids these problems by dividing an abstraction (e.g., a Button) and its implementations (e.g., Win32Button, MacButton, etc.) into separate class hierarchies. For example, the Java AWT classes use the Bridge design pattern to enable designers to create AWT Button subclasses without needing to create additional operating-system specific subclasses. Each AWT Button maintains a reference to a ButtonPeer, which is the superclass for platform-specific implementations, such as Win32ButtonPeer, MacButtonPeer, etc. When a programmer creates a Button object, class Button calls factory method createButton of class Toolkit to create the platform-specific ButtonPeer object. The Button object stores a reference to its ButtonPeer—this reference is the “bridge” in the Bridge design pattern. When the programmer invokes methods on the Button object, the Button object delegates the work to the appropriate lower-level method on its ButtonPeer to fulfill the request. A designer who creates a Button subclass called, e.g., ImageButton, does not need to create a corresponding Win32ImageButton or MacImageButton with platform-specific image-drawing capabilities. An ImageButton is a Button. Therefore, when an ImageButton needs to display its image, the ImageButton uses its ButtonPeer’s Graphics object to render the image on each platform. This design pattern enables designers to create new cross-platform GUI components using a “bridge” to hide platform-specific details.

**Portability Tip M.1**

Designers often use the Bridge design pattern to enhance the platform independence of their systems. This design pattern enables designers to create new cross-platform components using a “bridge” to hide platform-specific details.

**Composite**

Designers often organize components into hierarchical structures (e.g., a hierarchy of directories and files in a file system)—each node in the structure represents a component (e.g., a file or directory). Each node can contain references to one or more other nodes, and if it does so, it is called a branch (e.g., a directory containing files); otherwise, it is called a leaf (e.g., a file). Occasionally, a structure contains objects from several different classes (e.g., a
directory can contain files and directories). An object—called a client—that wants to traverse the structure must determine the particular class for each node. Making this determination can be time consuming, and the structure can become hard to maintain.

In the Composite design pattern, each component in a hierarchical structure implements the same interface or extends a common superclass. This polymorphism (introduced in Chapter 10) ensures that clients can traverse all elements—branch or leaf—uniformly in the structure and does not have to determine each component type, because all components implement the same interface or extend the same superclass.

Java GUI components use the Composite design pattern. Consider the Swing component class JPanel, which extends class JComponent. Class JComponent extends class java.awt.Container, which extends class java.awt.Component (Fig. M.5). Class Container provides method add, which appends a Component object (or Component subclass object) to that Container object. Therefore, a JPanel object may be added to any object of a Component subclass, and any object from a Component subclass may be added to that JPanel object. A JPanel object can contain any GUI component while remaining unaware of its specific type. Nearly all GUI classes are both containers and components, enabling arbitrarily complex nesting and structuring of GUIs.

A client, such as a JPanel object, can traverse all components uniformly in the hierarchy. For example, if the JPanel object calls method repaint of superclass Container, method repaint displays the JPanel object and all components added to the JPanel object. Method repaint does not have to determine each component’s type, because all components inherit from superclass Container, which contains method repaint.

M.3.3 Behavioral Design Patterns

This section continues our discussion on behavioral design patterns. We discuss the Chain of Responsibility, Command, Observer, Strategy and Template Method design patterns.

*Chain of Responsibility*

In object-oriented systems, objects interact by sending messages to one another. Often, a system needs to determine at runtime the object that will handle a particular message. For example, consider the design of a three-line office phone system. When a person calls the office, the first line handles the call—if the first line is busy, the second line handles the call, and if the second line is busy, the third line handles the call. If all lines in the system
are busy, an automated speaker instructs the caller to wait for the next available line. When a line becomes available, that line handles the call.

The Chain of Responsibility design pattern enables a system to determine at runtime the object that will handle a message. This pattern allows an object to send a message to several objects in a chain. Each object in the chain either may handle the message or pass it to the next object. For instance, the first line in the phone system is the first object in the chain of responsibility, the second line is the second object, the third line is the third object and the automated speaker is the fourth object. The final object in the chain is the next available line that handles the message. The chain is created dynamically in response to the presence or absence of specific message handlers.

Several Java AWT GUI components use the Chain of Responsibility design pattern to handle certain events. For example, class java.awt.Button overrides method processEvent of class java.awt.Component to process AWTEvent objects. Method processEvent attempts to handle the AWTEvent upon receiving it as an argument. If method processEvent determines that the AWTEvent is an ActionEvent (i.e., the Button has been pressed), it handles the event by invoking method processActionEvent, which informs any ActionListener registered with the Button that the Button has been pressed. If method processEvent determines that the AWTEvent is not an ActionEvent, the method is unable to handle it and passes it to method processEvent of superclass Component (the next listener in the chain).

Command
Applications often provide users with several ways to perform a given task. For example, in a word processor there might be an Edit menu with menu items for cutting, copying and pasting text. A toolbar or a popup menu could also offer the same items. The functionality the application provides is the same in each case—the different interface components for invoking the functionality are provided for the user’s convenience. However, the same GUI component instance (e.g., JButton) cannot be used for menus, toolbars and popup menus, so the developer must code the same functionality three times. If there were many such interface items, repeating this functionality would become tedious and error prone.

The Command design pattern solves this problem by enabling developers to encapsulate the desired functionality (e.g., copying text) once in a reusable object; that functionality can then be added to a menu, toolbar, popup menu or other mechanism. This design pattern is called Command because it defines a command, or instruction, to be executed. It allows a designer to encapsulate a command, so that it may be used among several objects.

Observer
Suppose that we want to design a program for viewing bank account information. This system includes class BankStatementData to store data pertaining to bank statements and classes TextDisplay, BarGraphDisplay and PieChartDisplay to display the data. [Note: This approach is the basis for the Model-View-Controller architecture pattern, discussed in Section M.5.3.] Figure M.6 shows the design for our system. The data is displayed by class TextDisplay in text format, by class BarGraphDisplay in bar-graph format and by class PieChartDisplay as a pie chart. We want to design the system so that the BankStatementData object notifies the objects displaying the data of a change in the data. We also want to design the system to loosen coupling—the degree to which classes depend on each other in a system.
Fig. M.6 | Basis for the Observer design pattern.

Software Engineering Observation M.1
Loosely coupled classes are easier to reuse and modify than are tightly coupled classes, which depend heavily on each other. A modification in a class in a tightly coupled system usually results in modifying other classes in that system. A modification to one of a group of loosely coupled classes would require little or no modification to the other classes.

The Observer design pattern is appropriate for systems like that of Fig. M.6. This pattern promotes loose coupling between a subject object and observer objects—a subject notifies the observers when the subject changes state. When notified by the subject, the observers change in response. In our example, the BankStatementData object is the subject, and the objects displaying the data are the observers. A subject can notify several observers; therefore, the subject has a one-to-many relationship with the observers.

The Java API contains classes that use the Observer design pattern. Class java.util.Observable represents a subject. Class Observable provides method addObserver, which takes a java.util.Observable argument. Interface Observer allows the Observable object to notify the Observer when the Observable object changes state. The Observer can be an instance of any class that implements interface Observer; because the Observable object invokes methods declared in interface Observer, the objects remain loosely coupled. If a developer changes the way in which a particular Observer responds to changes in the Observable object, the developer does not need to change the object. The Observable object interacts with its Observers only through interface Observer, which enables the loose coupling.

The Swing GUI components use the Observer design pattern. GUI components collaborate with their listeners to respond to user interactions. For example, an ActionListener observes state changes in a JButton (the subject) by registering to handle that JButton’s events. When pressed by the user, the JButton notifies its ActionListener objects (the observers) that the JButton’s state has changed (i.e., the JButton has been pressed).

Strategy
The Strategy design pattern is similar to the State design pattern (discussed in Section M.2.3). We mentioned that the State design pattern contains a state object, which encapsulates the state of a context object. The Strategy design pattern contains a strategy...
object, which is analogous to the State design pattern's state object. The key difference is that the strategy object encapsulates an algorithm rather than state information.

For example, java.awt.Container components implement the Strategy design pattern using LayoutManagers (discussed in Section 11.17) as strategy objects. In package java.awt, classesFlowLayout, BorderLayout and GridLayout implement interface LayoutManager. Each class uses method addLayoutComponent to add GUI components to a Container object. However, each method uses a different algorithm to display these GUI components: A FlowLayout displays them in a left-to-right sequence, a BorderLayout displays them in five regions and a GridLayout displays them in row-column format.

Class Container contains a reference to a LayoutManager object (the strategy object). An interface reference (i.e., the reference to the LayoutManager object) can hold references to objects of classes that implement that interface (i.e., the FlowLayout, BorderLayout or GridLayout objects), so the LayoutManager object can reference a FlowLayout, BorderLayout or GridLayout at any time. Class Container can change this reference through method setLayout to select different layouts at runtime.

Class FlowLayoutFrame (Fig. 11.39) demonstrates the application of the Strategy pattern—line 23 declares a new FlowLayout object and line 25 invokes the Container object's method setLayout to assign the FlowLayout object to the Container object. In this example, the FlowLayout provides the strategy for laying out the components.

Template Method
The Template Method design pattern also deals with algorithms. The Strategy design pattern allows several objects to contain distinct algorithms. However, the Template Method design pattern requires all objects to share a single algorithm defined by a superclass.

For example, consider the design of Fig. M.6, which we presented in the Observer design pattern discussion earlier in this section. Objects of classes TextDisplay, BarGraphDisplay and PieChartDisplay use the same basic algorithm for acquiring and displaying the data—get all statements from the BankStatementData object, parse the statements, then display the statements. The Template Method design pattern allows us to create an abstract superclass called BankStatementDisplay that provides the common algorithm for displaying the data. In this example, the algorithm invokes abstract methods getData, parseData and displayData. Classes TextDisplay, BarGraphDisplay and PieChartDisplay extend class BankStatementDisplay to inherit the algorithm, so each object can use the same algorithm. Each BankStatementDisplay subclass then overrides each method in a way specific to that subclass, because each class implements the algorithm differently. For example, classes TextDisplay, BarGraphDisplay and PieChartDisplay might get and parse the data identically, but each displays that data differently.

The Template Method design pattern allows us to extend the algorithm to other BankStatementDisplay subclasses—e.g., we could create classes, such as LineGraphDisplay or class 3DimensionalDisplay, that use the same algorithm inherited from class BankStatementDisplay and provide different implementations of the abstract methods the algorithm calls.

**M.3.4 Conclusion**

In this section, we discussed how Swing components take advantage of design patterns and how developers can integrate design patterns with GUI applications in Java. In the next
section, we discuss concurrency design patterns, which are particularly useful for developing multithreaded systems.

### M.4 Concurrency Design Patterns

Many additional design patterns have been discovered since the publication of the Gang of Four book, which introduced patterns involving object-oriented systems. Some of these new patterns involve specific types of object-oriented systems, such as concurrent, distributed or parallel systems. In this section, we discuss concurrency patterns to complement our discussion of multithreaded programming in Chapter 23.

**Concurrency Design Patterns**

Multithreaded programming languages such as Java allow designers to specify concurrent activities—that is, those that operate in parallel with one another. Designing concurrent systems improperly can introduce concurrency problems. For example, two objects attempting to alter shared data at the same time could corrupt that data. In addition, if two objects wait for one another to finish tasks, and if neither can complete their task, these objects could potentially wait forever—a situation called deadlock. Using Java, Doug Lea\(^2\) and Mark Grand\(^3\) documented concurrency patterns for multithreaded design architectures to prevent various problems associated with multithreading. We provide a partial list of these design patterns:

- **The Single-Threaded Execution design pattern** (Grand, 2002) prevents several threads from executing the same method of another object concurrently. Chapter 23 discusses various techniques that can be used to apply this pattern.

- **The Guarded Suspension design pattern** (Lea, 2000) suspends a thread's activity and resumes that thread's activity when some condition is satisfied. Lines 87–90 and lines 41–44 of class RunnableObject (Fig. 23.17) use this design pattern—methods await and signal suspend and resume the program threads, and line 72 of class RandomCharacters (Fig. 23.18) toggles the guard variable that the condition evaluates.

- **The Balk ing design pattern** (Lea, 2000) ensures that a method will balk—that is, return without performing any actions—if an object occupies a state that cannot execute that method. A variation of this pattern is that the method throws an exception describing why that method is unable to execute—for example, a method throwing an exception when accessing a data structure that does not exist.

- **The Read/Write Lock design pattern** (Lea, 2000) allows multiple threads to obtain concurrent read access on an object but prevents multiple threads from obtaining concurrent write access on that object. Only one thread at a time may obtain write access to an object—when that thread obtains write access, the object is locked to all other threads.

---


The Two-Phase Termination design pattern (Grand, 98) uses a two-phase termination process for a thread to ensure that a thread has the opportunity to free resources—such as other spawned threads—in memory (phase one) before termination (phase two). In Java, a Runnable object can use this pattern in method run. For instance, method run can contain an infinite loop that is terminated by some state change—upon termination, method run can invoke a private method responsible for stopping any other spawned threads (phase one). The thread then terminates after method run terminates (phase two).

In the next section, we return to the Gang of Four design patterns. Using the material introduced in Chapter 14 and Chapter 24, we identify those classes in package java.io and java.net that use design patterns.

### M.5 Design Patterns Used in Packages java.io and java.net

This section introduces those design patterns associated with the Java file, streams and networking packages.

#### M.5.1 Creational Design Patterns

We now continue our discussion of creational design patterns.

**Abstract Factory**

Like the Factory Method design pattern, the Abstract Factory design pattern allows a system to determine the subclass from which to instantiate an object at runtime. Often, this subclass is unknown during development. However, Abstract Factory uses an object known as a factory that uses an interface to instantiate objects. A factory creates a product, which in this case is an object of a subclass determined at runtime.

The Java socket library in package java.net uses the Abstract Factory design pattern. A socket describes a connection, or a stream of data, between two processes. Class Socket references an object of a SocketImpl subclass (Section 24.5). Class Socket also contains a static reference to an object implementing interface SocketImplFactory. The Socket constructor invokes method createSocketImpl of interface SocketImplFactory to create the SocketImpl object. The object that implements interface SocketImplFactory is the factory, and an object of a SocketImpl subclass is the product of that factory. The system cannot specify the SocketImpl subclass from which to instantiate until runtime, because the system has no knowledge of what type of Socket implementation is required (e.g., a socket configured to the local network’s security requirements). Method createSocketImpl decides the SocketImpl subclass from which to instantiate the object at runtime.

#### M.5.2 Structural Design Patterns

This section concludes our discussion of structural design patterns.

**Decorator**

Let us reexamine class CreateSequentialFile (Fig. 14.18). Lines 20–21 of this class allow a FileOutputStream object, which writes bytes to a file, to gain the functionality of an ObjectOutputStream, which provides methods for writing entire objects to an Output-
Stream. Class CreateSequentialFile appears to “wrap” an ObjectOutputStream object around a FileOutputStream object. The fact that we can dynamically add the behavior of an ObjectOutputStream to a FileOutputStream obviates the need for a separate class called ObjectOutputStream, which would implement the behaviors of both classes.

Lines 20–21 of class CreateSequentialFile show an example of the Decorator design pattern, which allows an object to gain additional functionality dynamically. Using this pattern, designers do not have to create separate, unnecessary classes to add responsibilities to objects of a given class.

Let us consider a more complex example to discover how the Decorator design pattern can simplify a system’s structure. Suppose that we wanted to enhance the I/O performance of the previous example by using a BufferedOutputStream. Using the Decorator design pattern, we would write

```java
output = new ObjectOutputStream(
    new BufferedOutputStream(
        new FileOutputStream( fileName )));
```

We can combine objects in this manner, because ObjectOutputStream, BufferedOutputStream and FileOutputStream extend abstract superclass OutputStream, and each subclass constructor takes an OutputStream object as a parameter. If the stream objects in package java.io did not use the Decorator pattern (i.e., did not satisfy these two requirements), package java.io would have to provide classes BufferedFileOutputStream, ObjectBufferedOutputStream, ObjectBufferedFileOutputStream and ObjectFileOutputStream. Consider how many classes we would have to create if we combined even more stream objects without applying the Decorator pattern.

**Facade**

When driving, you know that pressing the gas pedal accelerates your car, but you are unaware of exactly how it does so. This principle is the foundation of the Facade design pattern, which allows an object—a called a facade object—to provide a simple interface for the behaviors of a subsystem (an aggregate of objects that comprise collectively a major system responsibility). The gas pedal, for example, is the facade object for the car’s acceleration subsystem, the steering wheel is the facade object for the car’s steering subsystem and the brake is the facade object for the car’s deceleration subsystem. A client object uses the facade object to access the objects behind the facade. The client remains unaware of how the objects behind the facade fulfill responsibilities, so the subsystem complexity is hidden from the client. When you press the gas pedal, you act as a client object. The Facade design pattern reduces system complexity, because a client interacts with only one object (the facade) to access the behaviors of the subsystem the facade represents. This pattern shields applications developers from subsystem complexities. Developers need to be familiar with only the operations of the facade object, rather than with the more detailed operations of the entire subsystem. The implementation behind the facade may be changed without changes to the clients.

In package java.net, an object of class URL is a facade object. This object contains a reference to an InetAddress object that specifies the host computer’s IP address. The URL facade object also references an object from class URLStreamHandler, which opens the URL connection. The client object that uses the URL facade object accesses the Inet-
Address object and the URLStreamHandler object through the facade object. However, the client object does not know how the objects behind the URL facade object accomplish their responsibilities.

### M.5.3 Architectural Patterns

Design patterns allow developers to design specific parts of systems, such as abstracting object instantiations or aggregating classes into larger structures. Design patterns also promote loose coupling among objects. Architectural patterns promote loose coupling among subsystems. These patterns specify how subsystems interact with one another.\(^4\) We introduce the popular Model-View-Controller and Layers architectural patterns.

#### MVC

Consider the design of a simple text editor. In this program, the user inputs text from the keyboard and formats it using the mouse. Our program stores this text and format information into a series of data structures, then displays this information on screen for the user to read what has been inputted.

This program adheres to the Model-View-Controller (MVC) architectural pattern, which separates application data (contained in the model) from graphical presentation components (the view) and input-processing logic (the controller). Figure M.7 shows the relationships between components in MVC.

The controller implements logic for processing user inputs. The model contains application data, and the view presents the data stored in the model. When a user provides some input, the controller modifies the model with the given input. With regard to the text-editor example, the model might contain only the characters that make up the document. When the model changes, it notifies the view of the change so that it can update its presentation with the changed data. The view in a word processor might display characters using a particular font, with a particular size, etc.

MVC does not restrict an application to a single view and a single controller. In a more sophisticated program (such as a word processor), there might be two views of a document model. One view might display an outline of the document and the other might display the complete document. The word processor also might implement multiple controllers—one for handling keyboard input and another for handling mouse selections. If either controller makes a change in the model, both the outline view and the print-preview window will show the change immediately when the model notifies all views of changes.

Another key benefit to the MVC architectural pattern is that developers can modify each component individually without having to modify the others. For example, devel-

---

operators could modify the view that displays the document outline without having to modify either the model or other views or controllers.

**Layers**
Consider the design in Fig. M.8, which presents the basic structure of a three-tier application, in which each tier contains a unique system component.

The information tier (also called the “bottom tier”) maintains data for the application, typically storing it in a database. The information tier for an online store may contain product information, such as descriptions, prices, and quantities in stock, and customer information, such as user names, billing addresses and credit-card numbers.

The middle tier acts as an intermediary between the information tier and the client tier. The middle tier processes client-tier requests, and reads data from and writes data to the database. It then processes data from the information tier and presents the content to the client tier. This processing is the application’s business logic, which handles such tasks as retrieving data from the information tier, ensuring that data is reliable before updating the database and presenting data to the client tier. For example, the business logic associated with the middle tier for the online store can verify a customer’s credit card with the credit-card issuer before the warehouse ships the customer’s order. This business logic could then store (or retrieve) the credit information in the database and notify the client tier that the verification was successful.

The client tier (also called the “top tier”) is the application’s user interface, such as a standard Web browser. Users interact directly with the application through the user interface. The client tier interacts with the middle tier to make requests and retrieve data from the information tier. The client tier then displays data retrieved from the middle tier.

![Three-tier application model](image)

**Fig. M.8** | Three-tier application model.
Figure M.8 is an implementation of the Layers architectural pattern, which divides functionality into separate layers. Each layer contains a set of system responsibilities and depends on the services of only the next lower layer. In Fig. M.8, each tier corresponds to a layer. This architectural pattern is useful, because a designer can modify one layer without having to modify the others. For example, a designer could modify the information tier in Fig. M.8 to accommodate a particular database product but would not have to modify either the client tier or the middle tier.

M.5.4 Conclusion
In this section, we discussed how packages java.io and java.net take advantage of specific design patterns and how developers can integrate design patterns with networking/file-processing applications in Java. We also introduced the Model-View-Controller and Layers architectural patterns, which both assign system functionality to separate subsystems. These patterns make designing a system easier for developers. In the next section, we conclude our presentation of design patterns by discussing those patterns used in package java.util.

M.6 Design Patterns Used in Package java.util
In this section, we use the material on data structures and collections discussed in Chapters 17–19 to identify classes from package java.util that use design patterns.

M.6.1 Creational Design Patterns
We conclude our discussion of creational design patterns by presenting the Prototype design pattern.

Prototype
Sometimes a system must make a copy of an object but will not “know” that object’s class until execution time. For example, consider the drawing program design of Exercise 10.1 in the optional GUI and Graphics Case Study—classes MyLine, MyOval and MyRect represent “shape” classes that extend abstract superclass MyShape. We could modify this exercise to allow the user to create, copy and paste new instances of class MyLine into the program. The Prototype design pattern allows an object—called a prototype—to return a copy of that prototype to a requesting object—called a client. Every prototype must belong to a class that implements a common interface that allows the prototype to clone itself. For example, the Java API provides method clone from class java.lang.Object and interface java.lang.Cloneable—any object from a class implementing Cloneable can use method clone to copy itself. Specifically, method clone creates a copy of an object, then returns a reference to that object. If we designate class MyLine as the prototype for Exercise 10.1, then class MyLine must implement interface Cloneable. To create a new line in our drawing, we clone the MyLine prototype. To copy a preexisting line, we clone that object. Method clone also is useful in methods that return a reference to an object, but the developer does not want that object to be altered through that reference—method clone returns a reference to the copy of the object instead of returning that object’s reference. For more information on interface Cloneable, visit java.sun.com/j2se/5.0/docs/api/java/lang/Cloneable.html
M.6.2 Behavioral Design Patterns

We conclude our discussion of behavioral design patterns by discussing the Iterator design pattern.

**Iterator**

Designers use data structures such as arrays, linked lists and hash tables to organize data in a program. The *Iterator design pattern* allows objects to access individual objects from any data structure without “knowing” the data structure’s behavior (such as traversing the structure or removing an element from that structure) or how that data structure stores objects. Instructions for traversing the data structure and accessing its elements are stored in a separate object called an *iterator*. Each data structure can create an iterator—each iterator implements methods of a common interface to traverse the data structure and access its data. A client can traverse two differently structured data structures—such as a linked list and a hash table—in the same manner, because both data structures provide an iterator object that belongs to a class implementing a common interface. Java provides interface *Iterator* from package `java.util`, which we discussed in Section 19.3—class *CollectionTest* (Fig. 19.3) uses an *Iterator* object.

M.7 Wrap-Up

In this appendix, we have introduced the importance, usefulness and prevalence of design patterns. In their book *Design Patterns, Elements of Reusable Object-Oriented Software*, the “Gang of Four” described 23 design patterns that provide proven strategies for building systems. Each pattern belongs to one of three pattern categories—creational patterns address issues related to object creation; structural patterns provide ways to organize classes and objects in a system; and behavioral patterns offer strategies for modeling how objects collaborate with one another in a system.

Of the 23 design patterns, we discussed 18 of the more popular ones used by the Java community. The discussion was divided according to how certain packages of the Java API—such as package `java.awt`, `javax.swing`, `java.io`, `java.net` and `java.util`—use these design patterns. Also discussed were patterns not described by the “Gang of Four,” such as concurrency patterns, which are useful in multithreaded systems, and architectural patterns, which help designers assign functionality to various subsystems in a system. We motivated each pattern—explained why it is important and explained how it may be used. When appropriate, we supplied several examples in the form of real-world analogies (e.g., the adapter in the Adapter design pattern is similar to an adapter for a plug on an electrical device). You learned also how Java API packages take advantage of design patterns (e.g., Swing GUI components use the Observer design pattern to collaborate with their listeners to respond to user interactions). We provided examples of how certain programs in this book used design patterns.

We hope that you view this appendix as a beginning to further study of design patterns. Design patterns are used most prevalently in the J2EE (Java 2 Platform, Enterprise Edition) community, where systems tend to be exceedingly large and complex, and where robustness, portability and performance are so critical. However, even beginner programmers can benefit from early exposure to design patterns. We recommend that you visit the many URLs we have provided in Section M.8, and that you then read the Gang of Four
book. This information will help you build better systems using the collective wisdom of
the object-technology industry.

We hope you continue your study of design patterns. Be sure to send your comments,
criticisms and suggestions for improvement of *Java How to Program* to
deitel@deitel.com. Good luck!

### M.8 Web Resources

The following URLs provide further information on the nature, importance and applica-
tions of design patterns.

**Design Patterns**

www.hillside.net/patterns
Displays links to information on design patterns and languages.

www.hillside.net/patterns/books/
Lists books on design patterns.

www.netobjectives.com/design.htm
Introduces the importance of design patterns.

umbc7.umbc.edu/~tarr/dp/dp.html
Links to design patterns Web sites, tutorials and papers.

www.c2.com/ppr/
Discusses recent advances in design patterns and ideas for future projects.

www.dofactory.com/patterns/Patters.aspx
Provides UML class diagrams illustrating each of the 23 “Gang of Four” design patterns.

**Design Patterns in Java**

java.sun.com/blueprints/patterns/index.html
Sun Microsystems’ resource page describing design patterns applicable to Java 2 Platform, Enter-
prise Edition (J2EE) applications.

www.javaworld.com/channel_content/jw-patterns-index.shtml
Contains articles discussing when to use and how to implement popular design patterns in Java, il-
lustrating them with UML class diagrams.

www.fluffycat.com/java/patterns.html
Provides example Java code and UML class diagrams to illustrate each of the 23 “Gang of Four” de-
sign patterns.

www.cmcrossroads.com/bradapp/javapats.html
Discusses Java design patterns and presents design patterns in distributed computing.

www.javacamp.org/designPattern/
Provides definitions and example code for several design patterns, describing where each pattern
should be used and its benefits.

**Architectural Patterns**

Contains an article discussing how Swing components use Model-View-Controller architecture.

www.oottips.org/mvc-pattern.html
Provides information and tips on using MVC.

Provides information on architectural and design pattern and idioms (patterns targeting a specific
language).