Supplement: Design Patterns

For Introduction to Java Programming
By Y. Daniel Liang

1 Introduction

The most important benefit of object-oriented programming is to enable code reuse. For example, to create a button, you simply use the **JButton** class to create an instance of button. The **JButton** class is already defined in the Java API so you can use it without having to reinvent the wheel. Design patterns are neither classes nor objects. Rather, they are proven software strategies for designing classes. Applying design patterns is like reusing experience. You can apply successful patterns to develop new software without reinventing new solution strategies.

To apply design patterns effectively, you need to familiarize yourself with the most popular and effective patterns. This supplement introduces several popular design patterns.

2 History and Classifications of Design Patterns

Design patterns can be traced back to an article on the model-view-controller (MVC) architecture for building GUI in Smalltalk by Glenn Krasner and Stephen Pope in 1988. The MVC architecture separates data from presentation of data. The formal recognition of design patterns was presented in a groundbreaking book, *Design Patterns: Elements of Resuable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The book has a profound impact on object-oriented software development. The authors are admirably nicknamed the “Gang of Four” and their book is commonly known as the *gang-of-four book*.

The gang-of-four book argued that the expert designers do not solve every problem from scratches. Rather, they reuse solutions that have worked for them in the past. When they found a good solution, they use it over and over again. Design patterns can speed up the development process by providing almost ready-made solutions that have been used earlier and proved to be efficient.

The gang-of-four book recorded experiences in designing object-oriented software as design patterns, documented 23 most popular design patterns, and classified them into three categories:

- *Creational patterns* deal with object creation mechanisms. For example, you can use the Singleton pattern to ensure that only one object can be created from a class.

- *Structural design patterns* identify relationships among classes.
- *Behavioral design patterns* explore common communication patterns among objects.

Table 1 lists the patterns that are discussed in this supplement.

NOTE
The book will use the official name and definition for the patterns introduced in the Gang-of-Four book. It is important to stick with the same name and definition so the programmer can communicate using the same vocabularies.

3 The Singleton Pattern

The *Singleton pattern* declares a class that can be used to instantiate a unique instance. Occasionally, it is desirable to restrict instantiation of a class to exactly one object. For example, there may be many printers in a system, but there should be only one printer spooler. So, your system should ensure that only one printer spooler object is created.

The singleton pattern can be implemented by creating a class with a private constructor and a static constant that references an instance of the class. Listing 1 shows an example of such class.

Listing 1 Singleton.java (Singleton Class Example)

```
public final class Singleton {
  private static final Singleton uniqueInstance = new
Singleton();

  private Singleton() {
  }

  public static Singleton getInstance() {
    return uniqueInstance;
  }
}
```

An object of the **Singleton** class is created in Line 2. Since the constructor is private (Line 4), it prevents the client from creating objects using **new Singleton()**. The only way to access the object is through the **getInstance()** method (Line

7), which returns the reference of the object (Line 8). For example, the following code displays **true**, since **object1** and **object2** refers to the same object.

```
Singleton object1 = Singleton.getInstance();
Singleton object2 = Singleton.getInstance();
System.out.println(object1 == object2);
```

Will the **Singleton** class work in the presence of multiple threads? The answer is yes. Since the constant **uniqueInstance** is declared static (Line 2), the instance of **Singleton** is created when the class is loaded. So the unique object is created before any thread can access it. So, it is guaranteed that there is only one object created in the presence of multiple threads.

4 The Iterator Pattern

The *Iterator pattern* provides a unified way for traversing and processing the elements in an aggregate object such as an array, set, list, and map without exposing its underlying representation. An iterator is a separate object that contains the methods for retrieving and processing the elements in the aggregate object. Iterators are commonly defined using an interface. For example, the **Iterator** interface is defined in the Collections framework, as shown in Figure 21.3. You can use it to access any instance of **Collection**. Listing 2 shows an example of using iterators to access a set and a list.

Listing 2 TestIterator.java (Iterator in the Java API)

```
import java.util.*;

public class TestIterator {
  public static void main(String[] args) {
    Set<String> c1 = new HashSet<String>
      (Arrays.asList(new String[]{"London", "New York", "Chicago"}));
    Iterator iterator1 = c1.iterator();
    while (iterator1.hasNext())
      System.out.print(iterator1.next() + " ");

    System.out.println();
    List<String> c2 = new LinkedList<String>
      (Arrays.asList(new String[]{"London", "New York", "Chicago"}));
    Iterator iterator2 = c2.iterator();
    while (iterator2.hasNext())
      System.out.print(iterator2.next() + " ");
  }
}
```

A set of strings is created in Lines 5-6. Line 7 obtains an iterator for the set. Lines 8-9 is a loop that traverses the set using the iterator. A list of strings is created in Lines 12-13. Line 14 obtains an iterator for the list. Lines 15-16 is a loop that traverses the set using the iterator. As you see, the iterator allows you traverse the set and list without knowing how data are stored in the set and list.

The preceding example demonstrates how to use the iterators in the Java API. Invoking the **iterator()** method (Lines 7, 14) returns an **Iterator**. You may wonder how an iterator is created. To answer this question, let us create a simple class for a list of holidays (Listing 3) and an iterator class for traversing holidays (Listing 4).

Listing 3 Holidays.java (A Collection of Dates)

```
import java.util.*;

public class Holidays {
  private ArrayList<Date> dates = new ArrayList<Date>();

  public void add(Date date) {
    dates.add(date);
  }

  public HolidaysIterator iterator() {
    return new HolidaysIterator(dates);
  }
}
```

A **Holidays** object stores dates in an array list (Line 4). You can add a new date using the **add** method and create an iterator using the **iterator()** method for traversing and removing dates in a **Holidays** object.

Listing 4 HolidaysIterator.java (An Iterator for Holidays)

```
import java.util.*;

public class HolidaysIterator implements Iterator<Date> {
  private ArrayList<Date> dates = new ArrayList<Date>();
  private int position = 0;

  public HolidaysIterator(ArrayList<Date> dates) {
    this.dates = dates;
  }

  public boolean hasNext() {
    return (position < dates.size());
  }
```

```
    public Date next() {
      return hasNext()? dates.get(position++) : null;
    }

    public void remove() {
      if (hasNext()) dates.remove(position);
    }
  }
```

The **HolidaysIterator** class implements **Iterator<Date>** (Line 3). The internal data structure is an array list (Line 4) and is passed to the iterator when the iterator is constructed. The position variable records the current position in the array list. Invoking the **next()** method retrieves an element and moves the position forward (Line 16).

Listing 5 gives a test program that creates a **Holidays** object and using an iterator to traverse its elements.

Listing 5 TestHolidays.java (Test Iterator for Holidays)

```
import java.util.*;

public class TestHolidays {
  public static void main(String[] args) {
    Holidays holidays = new Holidays();
    holidays.add(new java.util.Date());
    holidays.add(new GregorianCalendar(2005, 9, 1).getTime());
    holidays.add(new java.util.Date(12345678900000L));

    Iterator iterator = holidays.iterator();
    while (iterator.hasNext())
      System.out.println(iterator.next());
  }
}
```

A **Holidays** object is created in Line 5. Three dates are added to the object (Lines 6-8). Line 10 creates an iterator for the **Holidays** object. Lines 11-12 uses a loop for traversing the iterator.

The iterator pattern enables you to encapsulate the internal storage structure for the data in the aggregate object. So, the design makes the classes flexible and easy to maintain. For example, if you decide to change the internal representation for the dates from an array list to an array, you need to change the **Holidays** and **HolidaysIterator** classes, but not the client program.

5 The Strategy Pattern

The *Strategy pattern* provides a *strategy object* that encapsulates an algorithm to implement a strategy. A typical outline of this design pattern is illustrated in Listing 6.

Listing 6 Outline of the Strategy Pattern

```
public interface StrategyInterface {
  public void performStrategy(optionalParameters);
}

public class Strategy1 implements StrategyInterface {
  public void performStrategy(optionalParameters) {
    ...
  }
}

public class Strategy2 implements StrategyInterface {
  public void performStrategy(optionalParameters) {
    ...
  }
}

public class Context {
  private StrategyInterface strategy;

  public void useStrategy(...) {
    strategy.performStrategy(optionParameters);
  }

  public void setStrategy(StrategyInterface strategy) {
    this.strategy = strategy;
  }
}
```

The Strategy pattern defines the strategy in **StrategyInterface** (Line 1). Concrete classes **Strategy1** (Line 4) and **Strategy2** (Line 10) implement **StrategyInterface** with specific algorithms on how to perform the strategy. The **Context** class can choose an appropriate strategy to achieve a desired purpose.

A well-known example of the Strategy pattern in the Java API is the layout manager. **LayoutManager** defines a strategy interface with the **layoutContainer** method for performing the strategy. A concrete layout manager such as **FlowLayout**, **GridLayout**, and **BorderLayout** implements **LayoutManager** using concrete algorithms on how to layout components in a container. A **Container** object can choose an appropriate layout manager and invoke the **doLayout()** method to layout the components. The **doLayout()** method invokes a layout manager object's **layoutContainer** method to produce a particular layout. Figure 1 illustrates the relationships among these classes.

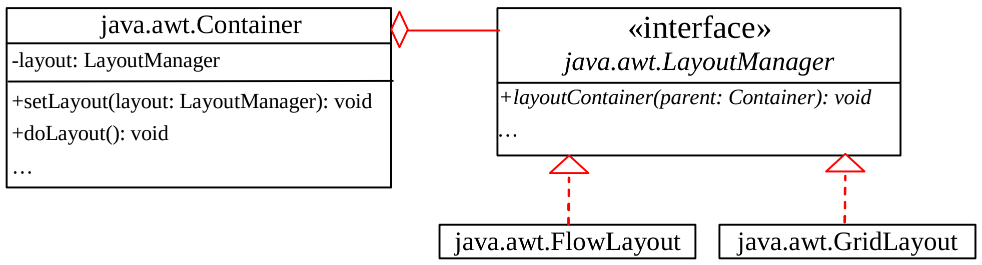


Figure 1

A container can choose a layout strategy to produce a desired layout.

In the Strategy pattern, strategies are encapsulated in the concrete strategy classes. The context class does not need to know how a strategy is implemented. Furthermore, the context can dynamically change a strategy to use a different algorithm.

Instead of inheriting strategies, the context class contains a strategy object. This is composition. Composition is more flexible than inheritance. As demonstrated in this pattern, composition allows you to change the strategy object at runtime.

6 The Adapter Pattern

The *Adapter pattern* defines a class, called *adapter*, which acts as the middleman between two classes with incompatible interfaces and make them work together. A common analogy for the Adapter pattern is the electric outlet adapter. The electric outlets have different shapes all over the world. For a US electric plug to connect an outlet in some Asian country, you need an adapter.

Suppose there are two interfaces **A** and **B**. You can create an adapter for **B** to be compatible with **A**. **B** is called an *adaptee* and **A** is called a target interface. The adapter should implement the target interface and contain an adaptee, as shown in Listing 7.

Listing 7 Outline of the Adapter Pattern

```
public interface A {
  public void doSomething(...);
}

public interface B {
  public void takeSomeAction(...);
```

```
    }

    public class Adapter implements A {
      private B b;

      public Adapter(B b) {
        this.b = b;
      }

      public void doSomething(...) {
        b.takeSomeAction(...);
      }
    }
```

The **Adapter** class implements **A**. So, an instance of **Adapter** can be used wherever a variable of **A** is required. Invoking **Adapter**'s **doSomething** method causes **B**'s **takeSomeAction** method to be invoked.

You learned how to create an event adapter in §25.7.1. Event adapters are examples of the Adapter pattern. Listing 8 gives an example to create an adapter for **ActionEvent**.

Listing 8 ActionEvent Adapter

```
public class Adapter implements ActionListener {
  private Adaptee adaptee;

  public Adapter(Adaptee adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.processActionEvent(e);
  }
}

public class Adaptee {
  public void processActionEvent(ActionEvent e) {
    ...
  }
}
```

Adapter implements the target interface **ActionListener** and contains the adaptee.

Now consider to write an adapter from the **Enumeration** interface to the **Iterator** interface, as shown in Figure 2. Both **Enumeration** and **Iterator** are used for traversing elements in a collection. **Iterator** was introduced in JDK 1.2 to supersede **Enumeartion**.

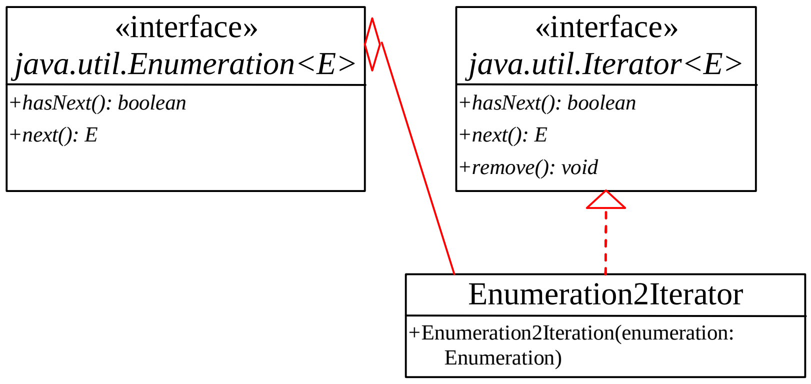


Figure 2

*You can write an adapter from **Enumeration** to **Iterator**.*

The adapter **Enumerator2Iterator**, given in Listing 9, implements **Iterator** and contains **Enumeration**. The **hasMoreElement()** and **nextElement()** methods in **Enumeration** are essentially same as the **hasNext()** and **next()** methods in **Iterator**. So, **hasNext()** in **Iterator** is delegated to **hasMoreElement()** in **Enumeration** and **next()** in **Iterator** is delegated to **nextElement()** in **Enumeration**. However, **Iterator** has the **remove()** method to remove an element, but **Enumeration** does not. So, an **UnsupportedOperationException** is thrown to indicate that the **remove** method is not supported by **Enumeration**.

Listing 9 Enumerator2Iterator.java

```
import java.util.*;

public class Enumeration2Iterator<E> implements Iterator<E> {
  Enumeration<E> enumeration;

  public Enumeration2Iterator(Enumeration<E> enumeration) {
    this.enumeration = enumeration;
  }

  public boolean hasNext() {
    return enumeration.hasMoreElements();
  }

  public E next() {
    return enumeration.nextElement();
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

7 The Decorator Pattern

The *Decorator pattern* uses a *decorator* class to attach additional functionalities to objects (called *decoratees*) dynamically through both inheritance and composition. Using this pattern, designers can easily add new functionalities to objects without creating separate classes. In general, both the decorator and decoratee are subclasses of the same interface and the decorator also contains a decoratee object, as shown in Figure 3. A typical outline of this design pattern is shown in Listing 10.

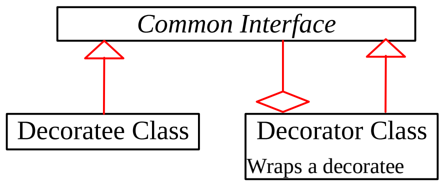


Figure 3

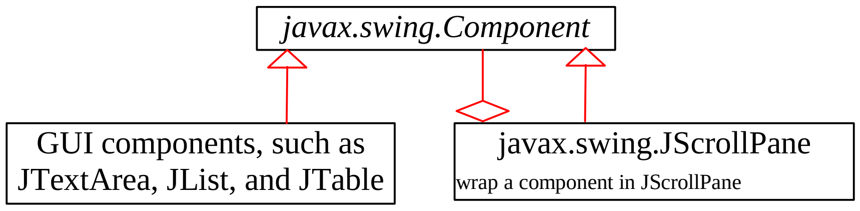
Decorator and decoratee are subclasses of a common interface. A decorator contains a decoratee with additional functions.

Listing 10 Outline of the Decorator Pattern

```
public interface CommonInterface {
  public void performSomeFunction(...);
  ...
}

public class Decoratee implements CommonInterface {
  public void performSomeFunction(...) {
    // perform it
  }
}

public class Decorator implements CommonInterface {
  private CommonInterface decoratee;

  public Decorator(CommonInterface decoratee) {
    this.decoratee = decoratee;
  }

  public void performSomeFunction(...) {
    // perform some new function
    ...

    // also perform the existing function in decoratee;
    decoratee.performSomeFunction(...);
  }
```

```
    }
```

<Side Remark: JScrollPane decorator>
JScrollPane is an example of the Decorator pattern. It adds the automatic scrolling ability to any instance of **Component**. Consider the scenario without **JScrollPane**, you have two options: (1) Create a separate class for each GUI component. For example, you may create a subclass of **JTextArea** named **ScrollableTextArea** for scrolling a text area, a subclass of **JList** named **ScrollableList** for scrolling a list, and a subclass of **JTable** named **ScrollableTable** for scrolling a table. (2) Code the scrolling function in the GUI components such as **JTextArea**, **JList**, and **JTable**. Neither is a good option. The first option creates many new classes with the same new function. The second option adds the same code in the classes. Both options present maintenance problems. The Java designer applied the Decorator pattern to design **JScrollPane**, as shown in Figure 4.



javax.swing.Component
GUI components, such as JTextArea, JList, and JTable
javax.swing.JScrollPane
wrap a component in JScrollPane

Figure 4

JScrollPane *subclasses* ***Component*** *and adds additional function to a component.*

In general, the decorator class (e.g., **JScrollPane**) and the decorate class (e.g., **JTextArea**) has the same common interface and the decorator class wraps a decoratee. Like **JTextArea**, **JList**, and **JTable**, **JScrollPane** is a subclass of **Component**. So, you can use **JScrollPane** wherever a **Component** is needed. **JScrollPane** is a decorator object that wraps any object of **JComponent** to add additional function to a component. A **ScrollPane** for a component seamlessly combines the new scrolling function with the existing function of the component into an object of **JScrollPane**. For example, the following statement creates a **JScrollPane** object that combines scrolling function with a text area. So, the resulting object is a scrollable text area.

```
       JScrollPane scrollPane = new JScrollPane(new JTextArea());
```

Another well-known example of the Decorator pattern is the design of I/O stream classes in the Java API. **BufferedInputStream** and **ObjectInputStream** are the decorator classes for **FileInputStream**, as shown in Figure 5. You can wrap a **BufferedInputStream** on a **FileInputStream** to add the buffering ability for an object of **FileInputStream** and wrap an **ObjectInputStream** on a **FileInputStream** to add the object input ability for an object of **FileInputStream**, as shown in the following statements:

```
       BufferedInputStream input1 = new BufferedInputStream(
         new FileInputStream("text.dat"));

       ObjectInputStream input2 = new ObjectInputStream(
         new FileInputStream("text.dat"));
```

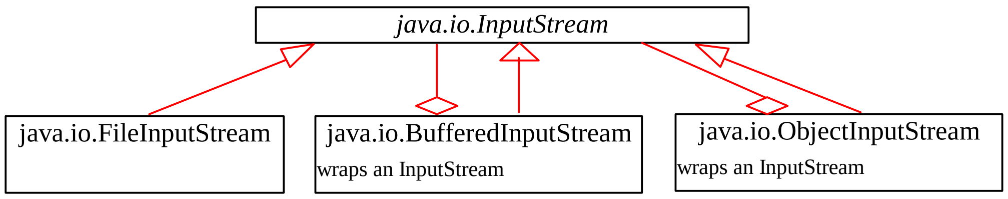


Figure 5

BufferedInputStream *and* ***ObjectInputStream*** *are the decorator classes for* ***FileInputStream****.*

Since **ObjectInputStream** can decorate any **InputStream**, you can also wrap an **ObjectInputStream** on a **BufferedInputStream** that wraps on a **FileInputStream** as follows,

```
       ObjectInputStream input3 = new ObjectInputStream(
         new BufferedInputStream(new FileInputStream("text.dat")));
```

You learned from the outline of the Decorator pattern and saw two of its examples. Now you can create your own example using the Decorator pattern. Suppose you can build a custom book by adding new chapters into a custom core. The custom core consists of a fixed set of chapters. The cost of each additional chapter is $2. You may declare a custom chapter decorator, as shown in Figure 6.

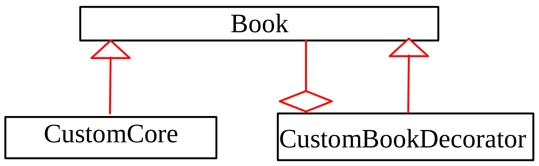


Figure 6

CustomBookDecorator *adds a new chapter to the book.*

Listing 11 gives the source code of these three classes: **Book**, **CustomCore**, and **CustomBookDecorator**.

Listing 11 Decorator Pattern Example

```
public class TestDecorator {
  public static void main(String[] args) {
    Book book = new CustomCore();
    book = new CustomBookDecorator(19, book);
    book = new CustomBookDecorator(24, book);
    book = new CustomBookDecorator(25, book);

    System.out.println("contents are " + book.toString());
    System.out.println("cost is " + book.cost());
  }
}

interface Book {
  public double cost();
}

class CustomCore implements Book {
  public double cost() {
    return 50;
  }

  public String toString() {
    return "Custom Core";
  }
}

class CustomBookDecorator implements Book {
  private Book book;
  int chapterNo;

  public CustomBookDecorator(int chapterNo, Book book) {
    this.book = book;
    this.chapterNo = chapterNo;
  }

  public double cost() {
    return book.cost() + 2;
  }

  public String toString() {
    return book.toString() + ", Chapter " + chapterNo;
  }
}
```

The **Book** interface (Lines 13-15) defines the **cost** method that returns the cost of the book. The **CustomCore** class (Lines 17-25) is a concrete class for **Book** that defines a custom core. The **CustomBookDecorator** class (Lines 27-43) is a decorator class for a book, which adds a new chapter to an existing book. The **cost()** and **toString()** methods return the cost and contents of the new book.

TestDecorator is a test class that creates a book from custom core (Line 3), adds chapters 19, 24, and 25 to the book (Lines 4-5). So the final contents for the book are "Custom Core, Chapter 19, Chapter 24, Chapter 25" and the cost is 56.0.

8 The Observer Pattern

The *Observer pattern* defines a dependency relationship between an object (called *subject*) and its dependent objects (called *observers*). When the subject object changes its state, all the observer objects are notified. The GUI event model is designed using the Observer pattern. The listeners are registered with the source to listen for an event. When the event occurs, the source notifies the listeners. The model-view-controller architecture is also an example of the Observer pattern. Whenever the data in the model is changed, its dependent views are notified and updated.

Prior to JDK 1.1, you had to declare a subject class by extending the **java.util.Observable** class and declare an observer class by implementing the **java.util.Observer** interface. **Observable** and **Observer** were introduced in JDK 1.0. With the arrival of the new Java event delegation model, using **Observable** and **Observer** is obsolete. The JDK event delegation model provides a superior architecture for supporting the Observer pattern component development. The subject can be implemented as a source with appropriate event and event listener registration methods. The observer can be implemented as a listener. So, if data are changed in the subject, the observer will be notified.

A typical outline of this design pattern is shown in Listing 12.

Listing 12 Outline of the Observer Pattern

```
public class SubjectClass {
  public void stateChange(...) {
    // fire an ActionEvent to notify the observers
    processEvent(new ActionEvent(this,
      ActionEvent.ACTION_PERFORMED, null));
  }

  /** implement addActionListener, removeActionListener,
   *  and processEvent. Same as in Listing 25.2 on page XXX.
   */
}
```

```
public class Observer1Class implements ActionListener {
  ...

   public void actionPerformed(ActionEvent e) {
      // process the notification
   }
}

public class Observer2Class implements ActionListener {
  ...

public void actionPerformed(ActionEvent e) {
     // process the notification
  }
}

public class Test {
  public static void main(String[] args) {
    SubjectClass subject = new SubjectClass();
    Observer1Class Observer1 = new Observer1Class();
    Observer2Class Observer2 = new Observer2Class();
    subject.addActionListener(Observer1);
    subject.addActionListener(Observer2);

    ...
  }
}
```

The observers are interested in the changes of the subject. The subject fires an event (e.g., **ActionEvent**) to notify the observers by invoking the observer's handling method (e.g., **actionPerformed**) when the changes are made on the subject. The observers are registered with the subject in order to receive the notification. Using the event delegation model to implement the Observer pattern makes the classes easy to maintain. The subject and observer classes are decoupled. You can plug an observer to observe any subject that fires the event that the observer is capable of processing.

9 The Template Method Pattern

The *Template Method pattern* defines the skeleton of an algorithm in a method, deferring the detailed implementation to subclasses. This pattern is widely used in the Java API. Consider the following four examples of the Template Method pattern.

The **Comparable** interface defines the template method **compareTo** for comparing this object with a specified object. If a class implements **Comparable**, its objects can be compared using the **compareTo** method. The method returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object. Many classes in the Java API implement **Comparable**. For example, the wrapper classes **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, and **Character** implement **Comparable**, and the **Date** and **Calendar** classes implement **Comparable**. So, you can compare two **Integer** objects or two **Date** objects using the **compareTo** method. The **java.util.Arrays.sort(Object[])** method sorts an array of comparable objects using the **compareTo** method. To use this **sort** method, the objects must be instances of **Comparable**.

The **paintComponent** method defined in **javax.swing.JComponent** is used to paint a graphics context in a Swing GUI component. This is a template method. To paint something, you override this method with specific instructions. Your program never directly invokes the **paintComponent** method. This method is invoked by the JVM whenever the component needs to be painted or repainted.

The **Applet** class defines the template methods **init()**, **start()**, **stop()**, and **destroy()**. These methods are called by the JVM to control an applet from a Web browser. By default, these methods do nothing. To perform specific functions, you create a subclass of **Applet** to override these methods. The **Applet** class defines a framework for creating applets.

In the Java event delegation model, the listener must be an instance of a listener interface. A listener interface defines the handlers for processing the event. These methods are the template methods that are invoked by the source component. For example, to be a listener for **ActionEvent**, the listener class must implement **ActionListener**. The **actionPerformed** method in **ActionListener** is a template method. The listener class provides the detailed instruction on how to handle the event in the **actionPerformed** method. The source component invokes the listener’s **actionPerformed** method to process the event.

The Strategy pattern and the Template Method pattern are very similar. Both patterns implement specific algorithms in the template methods. For example, you override the **init()** method in the **Applet** class to specify how to initialize an applet using the Template Method pattern and implement the **layoutContainer** method in the **LayoutManager** interface to specify how to layout components using the Strategy pattern. The difference, however, lies in whether the algorithms can be dynamically plugged. The Strategy pattern defines a strategy object that contains the algorithm in the template method. The strategy object can be dynamically set in an application, commonly using a set method. Using the Template

Method pattern, the algorithms can not be changed dynamically. For example, for an applet, you can change its layout manager using the **setLayout** method, but you cannot change its **init** method with a new algorithm at runtime.

10 The Command Pattern

The *Command pattern* defines a command to be executed. The command is encapsulated into an object, called the *command object*. The command object is used by a sender object to send requests to a receiver object. The sender does not need to know the interface of the receiver. This pattern decouples the sender from the receiver and provides flexibility and extensibility for both senders and receivers.

Let us use a simple example to demonstrate this pattern. The example designs a remote controller (i.e., sender) to turn on/off a TV and VCR (i.e., receiver). The TV and VCR have different interfaces, as shown in the **TV** and **VCR** classes in Listing 13. To send a command from the sender to the receiver, define the **Command** interface with the **execute()** method and four concrete classes **TVOnCommand**, **TVOffCommand**, **VRCOnCommand**, and **VCROffCommand** of the **Command** interface for turning on/of TV and VRC, as shown in Listing 14.

Listing 13 Receiver Classes

```
class TV {
  public void turnOn() {
    System.out.println("Turn on the TV");
  }

  public void turnOff() {
    System.out.println("Turn off the TV");
  }
}

class VCR {
  public void start() {
    System.out.println("Start the VCR");
  }

  public void stop() {
    System.out.println("Stop the VCR");
  }
}
```

Listing 14 Command Interface and Command Classes

```
interface Command {
  public void execute();
}

class TVOnCommand implements Command {
  private TV tv;

  public TVOnCommand(TV tv) {
    this.tv = tv;
  }

  public void execute() {
    tv.turnOn();
  }
}

class TVOffCommand implements Command {
  private TV tv;

  public TVOffCommand(TV tv) {
    this.tv = tv;
  }

  public void execute() {
    tv.turnOff();
  }
}

class VCROnCommand implements Command {
  private VCR vcr;

  public VCROnCommand(VCR vcr) {
    this.vcr = vcr;
  }

  public void execute() {
    vcr.start();
  }
}

class VCROffCommand implements Command {
  private VCR vcr;

  public VCROffCommand(VCR vcr) {
    this.vcr = vcr;
  }

  public void execute() {
    vcr.stop();
  }
}
```

The controller uses two methods **clickOn** and **clickOff** to request a device to turn on and off. The controller has no knowledge of the receiver’s interface. The controller sends a request to a receiver through a **Command** object by invoking the command’s **execute()** method. One **Command** object is associated with one request. Since there are two requests in the controller, the controller needs to contain two **Command** objects. The command can be dynamically set in a controller or created in the constructor of the controller class, as shown in Listing 15.

Listing 15 Sender Class

```
class RemoteController {
  private Command onCommand, offCommand;

  public RemoteController(Command onCommand, Command
offCommand) {
    this.onCommand = onCommand;
    this.offCommand = offCommand;
  }

  public void clickOn() {
    onCommand.execute();
  }

  public void clickOff() {
    offCommand.execute();
  }

  public void setOnCommand(Command onCommand) {
    this.onCommand = onCommand;
  }

  public void setOffCommand(Command offCommand) {
    this.offCommand = offCommand;
  }
}
```

Listing 16 gives a test program that creates a sender and two receivers. The program creates a TV as a receiver (Line 3), two commands on the TV (Lines 4-5), a **RemoteController** for the two commands as a sender (Lines 6-7). The controller invokes the **clickOn()** method (Line 8). The **clickOn()** method then invokes the **execute()** method on **tvOnCommand**. The **execute()** method finally invokes the **turnOn()** method on **tv**. Notice that the sender and the receiver are completely decoupled. The sender issues the request, but does not know how the request will be carried out by the receiver.

Listing 16 TestCommandPattern.java

```
public class TestCommandPattern {
  public static void main(String[] args) {
    TV tv = new TV();
    Command tvOnCommand = new TVOnCommand(tv);
    Command tvOffCommand = new TVOffCommand(tv);
    RemoteController controller =
      new RemoteController(tvOnCommand, tvOffCommand);
    controller.clickOn();
    controller.clickOff();

    VCR vcr = new VCR();
    Command vcrOnCommand = new VCROnCommand(vcr);
    Command vcrOffCommand = new VCROffCommand(vcr);
    controller.setOnCommand(vcrOnCommand);
    controller.setOffCommand(vcrOffCommand);
    controller.clickOn();
    controller.clickOff();
  }
}
```

The controller can also be used to control a VCR. The program creates a VCR (Line 11) and two commands on the VCR (Lines 12-13). These two new commands are set in the controller (Lines 14-15). The controller now can invoke the **clickOn()** and **clickOff()** methods to control the VCR (Lines 16-17). So, the output of the program is:

```
Turn on the TV
Turn off the TV
Start the VCR
Stop the VCR
```

As seen from this example, one design may use several design patterns. The example uses the Command pattern. Since the commands can be dynamically set in a controller, the example also utilizes the Strategy pattern.

11 The Factory Method Pattern

The *Factory Method pattern* defines an abstract method for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. A typical outline of this design pattern is shown in Listing 17.

Listing 17 Outline of Factory Method Pattern

```
abstract class or interface Base {
  ...

  public abstract SomeSuperType factoryMethod(...);
```

```
    }

    class SubClass extends or implements Base {
      ...

      public SomeSuperType factoryMethod(...) {
        SomeSuperType object = new ConcreteSubtype(...);
        ...
        return object;
      }
    }
```

A factory method for creating an object is defined in an abstract class or an interface (Line 4). The method is abstract because it does not have enough information for implementation in the base class (Line 1). The subclasses implement the factory method. The return type **SomeSuperType** of the factory method is usually an abstract class or an interface. **ConcreteSubtype** is a subtype of **SomeSuperType**. Consider the following two examples for the Factory Method pattern.

Consider three examples. The first example creates a base class **HosuePlan** with factory methods **createWindow** and **createDoor** (Lines 2-3), as shown in Listing 18. For different type of houses, these two methods are implemented differently. So, their implementation is deferred to subclasses of **HousePlan**. **LuxuryHousePlan** extends **HousePlan** and implements **createWindow** and **createDoor** by creating concrete objects.

Listing 18 Factory Method Pattern Example

```
abstract class HousePlan {
  public abstract Window createWindow(...);
  public abstract Door createDoor(...);

  public House createHouse() {
    return new House(createWindow(), createDoor());
  }
}

class LuxuryHousePlan extends HousePlan {
  public Window createWindow() {
    return  new LuxuryWindow(...);
  }

  public Door createDoor();
    return  new LuxuryDoor(...);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    HousePlan housePlan = new LuxuryHousePlan();
    housePlan.createHouse();
  }
}
```

For the second example, let us review **GenericMatrix** presented in §20.9, "Case Study: Generic Matrix Class." This class defines the abstract method **zero** to return an object. The instantiation of this object must be deferred to subclasses **IntegerMatrix** and **RationalMatrix**. The **zero()** method returns **new Integer(0)** for the **Integer** type and **new Rational(0, 1)** for the **Rational** type. This method cannot be implemented in **GenericMatrix** and must be implemented in the subclasses **IntegerMatrix** and **RationalMatrix**.

For the third example, let us review the **Collection** interface in the Java API. This class defines the **iterator()** method that creates an **Iterator** for a collection such as a set and a list. The implementation of **iterator()** depends on the concrete data structure that supports a specific type of collection. An iterator for a hash set is different from the one for a linked list. So, the **iterator()** method is implemented in a concrete subclass.

12 The Abstract Factory Pattern

The *Abstract Factory pattern* provides an abstract class or interface for creating a family of related or dependent objects without specifying their concrete classes. A typical outline of this pattern is shown in Listing 19.

Listing 19 Outline of Abstract Factory Pattern

```
abstract class or interface AbstractFactory {
  ...

  public abstract SomeSuperType1 createObject1(...);
  public abstract SomeSuperType2 createObject2(...);
}

class ConcreteFactory1 extends or implements AbstractFactory {
  ...

  public SomeSuperType1 createObject1(...) {
    SomeSuperType1 object = new ConcreteSubtype1(...);
    ...
    return object;
  }
  public SomeSuperType2 createObject2(...) {
    SomeSuperType2 object = new ConcreteSubtype2(...);
    ...
    return object;
  }
}

class ConcreteFactory2 extends or implements AbstractFactory {
  ...

  public SomeSuperType1 createObject1(...) {
    SomeSuperType1 object = new ConcreteSubtypeA(...);
    ...
    return object;
  }

  public SomeSuperType2 createObject2(...) {
    SomeSuperType2 object = new ConcreteSubtypeB(...);
    ...
    return object;
  }
}

class ClientClass {
  ...

  public void doSomething(AbstractFactory factory) {
    SomeSuperType1 object1 = factory.createObject1(...);
    SomeSuperType2 object1 = factory.createObject2(...);
    ...
  }
}
```

An abstract factory defines two abstract methods for creating objects (Lines 1-7). A concrete factory class implements these methods to create objects (Lines 8-22). A client class may pass a concrete factory object to invoke the **doSomething** method in Line 27.

The **HosuePlan** example in the preceding section was designed using the Factory Method pattern. The Abstract Factory pattern can be applied to the same problem, as shown in Listing 20. A factory object is used to create products in the **createHouse** method (Line 4). The **HouseFactory** interface defines the methods for creating windows and doors. **LuxuryHouseFactory** is a concrete class that implements the methods for creating windows and doors.

Listing 20 Abstract Factory Pattern Example

```
class HousePlan {
  ...

  public House createHouse(HouseFactory factory) {
    return new House(factory.createWindow(),
factory.createDoor());
  }
}

interface HouseFactory {
  public abstract Window createWindow(...);
  public abstract Door createDoor(...);
}

class LuxuryHouseFactory implements HouseFactory {
  public Window createWindow() {
    return  new LuxuryWindow(...);
  }

  public Door createDoor();
    return  new LuxuryDoor(...);
  }
}

public class Test {
  public static void main(String[] args) {
    HousePlan housePlan = new HousePlan();
    LuxuryHouseFactory factory = new HousePlan();
    housePlan.createHouse(factory);
  }
}
```

The Abstract Factory pattern is very similar to the Factory Method pattern. Both patterns defer object creation to concrete subclasses. However, the Factory Method pattern uses polymorphism to create objects while the Abstract Factory pattern uses delegation to create objects. In the **Test** class in Listing 18 for the Factory Method pattern, invoking **creatHouse** in Line 4 causes the concrete methods **createWindow** and **createDoor** defined in the **LuxuryHousePlan** to be invoked at runtime. In the **Test** class in Listing 20 for the Abstract Factory pattern, **createWindow** and **createDoor** are invoked from a factory object. Creating objects are delegated to the factory object.

12 The Facade Pattern

The *Facade pattern* provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Before applying the Facade pattern, a subsystem may look like the one in Figure 6(a) where the components are accessed by many clients. Applying the Facade pattern, you can create a unified interface called Facade, as shown in Figure 6(b), where the clients access the components in the subsystem through the Facade.

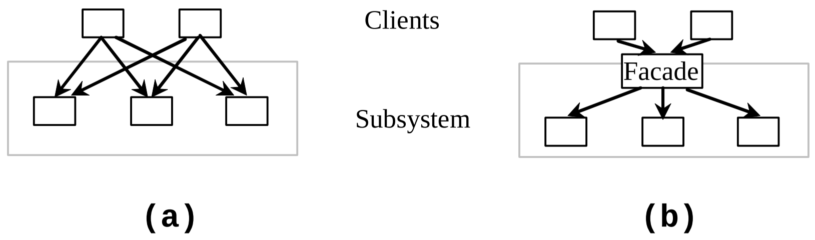


Figure 6

The Facade pattern provides a unified interface for accessing the components in a subsystem.

To demonstrate this pattern, consider the example in Listing 21. The program creates three objects light, stove, and air conditioner. When a person leaves the house, turn off the light, check the stove, and turn off the air conditioner. When a person enters the house, turn on the light and turn on the air conditioner.

Listing 21 HouseControlSimulator.java

```
class HouseControlSimulator {
  Light light = new Light();
  Stove stove = new Stove();
  AirConditioner air = new AirConditioner();

  public static void main(String[] args) {
    // Leave the house
    light.turnOff();
    stove.check();
    air.setTemparature(80);

    // Enter the house
    light.turnOn();
    air.setTemparature(75);
  }
}
```

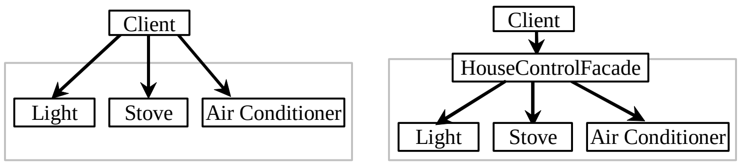
This design can be improved using the Facade pattern as shown in Listing 22.

Listing 22 HouseControlSimulatorUsingFacade.java

```
class HouseControlSimulatorUsingFacade {
  Light light = new Light();
```

```
          Stove stove = new Stove();
          AirConditioner air = new AirConditioner();

          public static void main(String[] args) {
            HouseControlFacade façade = new HouseControlFacade(
              light, stove, air);

            façade.leaveHouse();
            façade.enterHouse();
          }
        }

        class HouseControlFacade {
          Light light;
          Stove stove;
          AirConditioner air;

          public HouseControlFacade(
              Light light, Stove stove, AirConditioner air) {
            this.light = light;
            this.stove = sotve;
            this.air = air;
          }

          public void leaveHouse() {
            light.turnOff();
            stove.check();
            air.turnOff();
          }

          public void enterHouse() {
            light.turnOn();
            air.turnOn();
          }
        }
```

The light, stove and air conditioner are the components in the house subsystem. The Facade pattern simplifies the interface for accessing these components. In Listing 21, the client program directly accesses each individual component, as shown in Figure 7(a). In Listing 22, the client program accesses each individual component through the facade, as shown in Figure 7(b).



(a) **(b)**

Figure 7

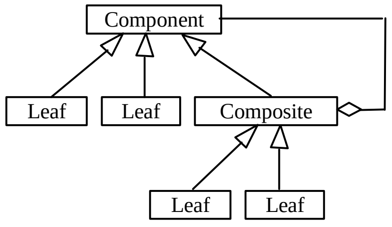
The Facade pattern provides a unified interface for accessing light, stove, and air conditioner.

The Facade pattern helps reduce complexity by structuring a system into subsystems. It hides the implementation of the subsystem from clients, making the subsystem easy to use.

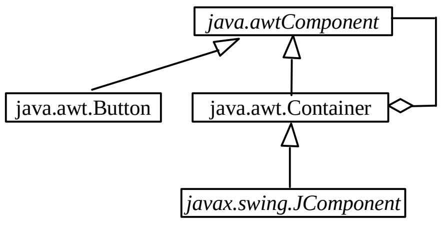
> NOTE
>
> The Facade pattern decouples the client from the classes in the subsystem. However, a facade does not encapsulate the subsystem; it merely provides a simplified interface for accessing the subsystem. The client can still access the components in the subsystem directly if necessary. For example, in Listing 21, the client can directly access the components if needed.

12 The Composite Pattern

The *Composite pattern* composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. A typical outline of this pattern is shown in Figure 8(a), where composite contains components. Since composite is a component itself, composites can be embedded inside composites. The **Composite** class should contain the methods for adding or removing components and also a method for obtaining its children. The components in the **Composite** class can be stored a list.



(a)



(b)

Figure 8

The Composite pattern treats individual objects and composite objects uniformly.

A familiar example of the *Composite pattern* is the **Component** and **Container** hierarchy in the Java GUI API, as shown in Figure 8(b). **Component** is the root class for all GUI components. A container can contain components. A container is also a component. A container is treated just like a component. **JComponent** is a subclass of **Container**. So an instance of **JComponent** such as **JPanel** can function as a container. A **JPanel** can contain components including another **JPanel**.

Consider another example with **Product** and **CompositeProduct**, as shown in Figure 9. **Book** and **Pen** are individual products. **StationaryBundle** is a composite product. Whenever a product is added to or removed from a composite product, the price of the composite product is changed. Listing 23 gives an implementation of the **CompositeProduct** class.

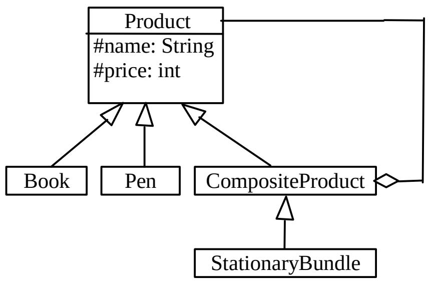


Figure 9

The Composite class contains the methods for adding and removing components.

Listing 23 CompositeProduct.java (Composite Class)

```
class CompositeProduct extends Product {
  private ArrayList products = new ArrayList();

  public void add(Product product) {
    products.add(product);
    price += produce.price;
  }

  public void remove(Product product) {
    if (products.contains(product)) {
      products.remove(product);
      price += produce.price;
    }
  }

  public Product[] getChild() {
    return products.toArray();
  }
}
```

The Composite pattern allows you to add new type of components to the system and makes the system easy to extend. Since the leaf and composite components are treated uniformly, it makes clients simpler.

13 The Chain of Responsibility Pattern

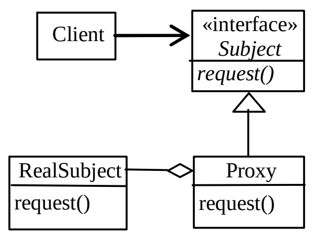
The *Chain of Responsibility pattern* avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. Normally, all the receiver objects on the chain share a common interface for handling requests.

For example, a handler often partially handles an event and then passes the event to another handler for further processing. §14.4, "Radio Buttons," presented the **RadioButtonDemo** class which contains the **actionPerformed** method. When an **ActionEvent** occurs, the handler first checks whether the action is from a radio button. If so, the event is processed. If not, it invokes the superclass's **actionPerformed** method to check whether the event is from a check box. So, the event is processed through a chain of responsibility.

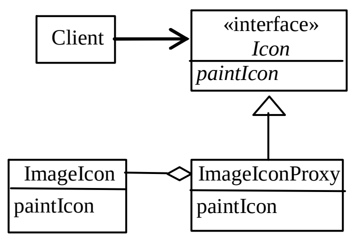
The Chain of Responsibility pattern decouples the sender and the receiver and enables a request to be processed by multiple objects. The chain of handlers may be modified dynamically by setting a receiver along the chain.

14 The Proxy Pattern

The *Proxy pattern* provides a surrogate or placeholder for another object to control access to it. There are situations in which a client does not or cannot reference an object directly, but wants to still interact with the object. A proxy object can act as the intermediary between the client and the target object. A proxy has the authority to act for another object. The proxy object has the same interface as the target object. The proxy object holds a reference to the target object and can forward requests to the target as needed, as shown in Figure 10(a).



(a) Outline



(b) Example

Figure 10

The Proxy pattern provides a proxy object for a real subject.

There are many reasons to use proxies. One reason is to defer the instantiation of an object. For example, it is expensive to load a large image file. If a program loads many large images at once, it could cause a significant performance hit. Also, if an application does not use all of its images, it is a waste to create them upfront. A better solution is to delay the creation of the image until it is needed. Normally, you create an image icon using the following statement,

```
  ImageIcon icon = new ImageIcon(imagefileURL);
```

To defer loading image, you need to use a proxy object, as follows:

```
  ImageIcon icon = new ProxyImageIcon(imagefileURL);
```

ImageIconProxy is a proxy for **ImageIcon**, as shown in Figure 10(b). You can declare **ImageIconProxy** by either implementing **javax.swing.Icon** or extends **javax.swing.ImageIcon**. As shown in Listing 24, **ImageIconProxy** is a subclass of **ImageIcon**. You can create an **ImageIconProxy** using one of the two constructors by passing either a file name (Line 10) or a file URL (Line 15). Since **ImageIconProxy** is a subclass of **ImageIcon**, an **ImageIcon** can be replaced by an **ImageIconProxy** in a client program. No **ImageIcon** object is created when an instance of **ImageIconProxy** is created. When the component that contains an **ImageIconProxy** is displayed, the **paintIcon** method (Line 20) is invoked, which then creates an **ImageIcon** (Lines 25, 29). The **ImageIconProxy** forwards the request to an **ImageIcon** for displaying the image (Line 33).

Listing 24 ImageIconProxy.java (Proxy Class)

```
import java.awt.*;
import javax.swing.ImageIcon;

public class ImageIconProxy extends ImageIcon {
  private java.net.URL url;
  private String filename;
  private ImageIcon image;

  /** Constructs a proxy for delayed loading of an image file */
  public ImageIconProxy(String filename) {
    this.filename = filename;
  }
  /** Constructs a proxy for delayed loading of an image from URL */
  public ImageIconProxy(java.net.URL url) {
    this.url = url;
  }

  /** Override paintIcon to display image icon */
  public void paintIcon(final Component c, Graphics g, int x, int y)   {
    // Load the image if it hasn't been loaded yet.
    if (image == null) {
      if (filename != null) {
        System.out.println("Loading " + filename);
        image = new ImageIcon(filename);
      }
      else if (url != null) {
        System.out.println("Loading " + url);
        image = new ImageIcon(url);
      }
    }

    image.paintIcon(c, g, x, y);
  }

  /** Override getIconWidth to return icon width */
  public int getIconWidth() {
    return (image == null) ? 200 : image.getIconWidth();
  }

  /** Override getIconHeight to return icon height */
  public int getIconHeight() {
    return (image == null) ? 100 : image.getIconHeight();
  }
}
```

Listing 25 gives a test program that uses **ImageIconProxy** to delay loading images. The program creates a tabbed pane (Line 6) and adds two labels to the tabbed pane (Lines 7-10), as shown in Figure 11. The labels are created with image icons. The image icons are controlled by proxies. When the label is displayed for the first time, the proxy's **paintIcon** method is invoked. If no image icon has been created, the image icon will be created in the proxy. The proxy controls this real image icon to display the image.

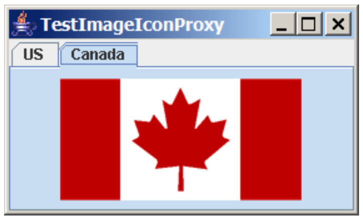


Figure 11

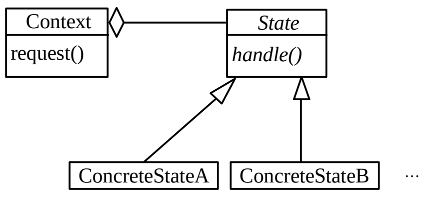
The image icons are controlled by proxies.

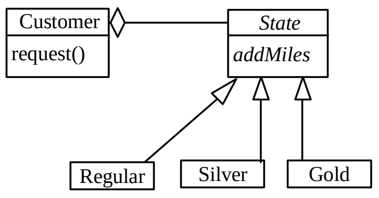
Listing 24 TestImageIconProxy.java (Test Proxy Class)

```
import java.awt.*;
import javax.swing.*;

public class TestImageIconProxy extends JApplet {
  public TestImageIconProxy() {
    JTabbedPane tabbedPane = new JTabbedPane();
    tabbedPane.add(new JLabel(new ImageIconProxy(
      getClass().getResource("image/us.gif"))), "US");
    tabbedPane.add(new JLabel(new ImageIconProxy(
      getClass().getResource("image/ca.gif"))), "Canada");

    add(tabbedPane, BorderLayout.CENTER);
  }
}
```

14 The State Pattern

The *State pattern* allows an object to alter its behavior when its internal state changes. The object will appear to change its class. The object represents a state. It is automatically changed to a new state object when the state changes. The State pattern structure can be outlined as shown in Figure 12(a). A sample implementation is given in Listing 25.





(a) structure **(b) example**

Figure 12

The State pattern uses objects to represent states.

Listing 25 Outline of State Pattern

```
public class Context {
  State state1 = new ConcreteStateA();
  State state2 = new ConcreteStateB();
  State state3 = new ConcreteStateC();
  State state4 = new ConcreteStateD();
  State state = state1;

  public void setState(State state) {
    this.state = state;
  }
```

```
   public void request1(...) {
     ...
     state.handle1(...);
   }

   public void request2(...) {
     ...
     state.handle2(...);
   }

  ...
}

public interface State {
  public void handle1(...);
  public void handle2(...);
}

public class ConcreteStateA implements State {
  Context context = new Context();

  public ConcreteStateA(Context context) {
    this.context = context;
  }

  public void handle1(...) {
    ...
    context.setState(context.state1); // state1 could state2 or else
  }

  public void handle2(...) {
    ...
    context.setState(context.state2); // state2 could state1 or else
  }
}
```

To demonstrate the State pattern, consider the following example. Suppose an airline classifies the customers into three categories: regular, silver, and gold, based on their award miles. Initially, a customer obtains the regular status. If the customer has more than 20000 award miles, the customer is upgraded to the silver status; if the customer has more than 60000 award miles, the customer is upgraded to the gold status. For every actual mile, a gold customer receives three award miles, a silver customer receives two award miles, and a regular customer receives one award mile. You can use the State pattern to design the classes: Customer, State, Regular, Silver, and Gold, as shown in Figure 12(b). Listing 26 implements these classes.

Listing 26 Example of State Pattern

```
class Customer {
  State goldState = new Gold(this);
  State silverState = new Silver(this);
  State regularState = new Regular(this);
  State state = regularState;

  public void setState(State state) {
    this.state = state;
  }

  public State getState() {
    return state;
  }

  public void addMiles(int miles) {
    state.addMiles(miles);
  }

  public String toString() {
    return state.toString();
  }
}

abstract class State {
  int awardMiles;
  String status;
  final static int SILVER_THRESHOLD = 20000;
  final static int GOLD_THRESHOLD = 60000;

  public abstract void addMiles(int miles);

  public String getStatus() {
    return status;
  }

  public int getAwardMiles() {
    return awardMiles;
  }

  public void setAwardMiles(int awardMiles) {
    this.awardMiles = awardMiles;
  }

  public String toString() {
    return "Status: " + status + " and award miles: " +
awardMiles;
  }
}
```

```
class Regular extends State {
  Customer customer;

  public Regular(Customer customer) {
    this.customer = customer;
    status = "Regualar";
  }

  public void addMiles(int miles) {
    awardMiles += miles;
    if (awardMiles > State.GOLD_THRESHOLD) {
      customer.goldState.setAwardMiles(awardMiles);
      customer.setState(customer.goldState);
    }
    else if (awardMiles > State.SILVER_THRESHOLD) {
      customer.silverState.setAwardMiles(awardMiles);
      customer.setState(customer.silverState);
    }
  }
}

class Silver extends State {
  Customer customer;

  public Silver(Customer customer) {
    this.customer = customer;
    status = "Silver";
  }

  public void addMiles(int miles) {
    awardMiles += 2 * miles;
    if (awardMiles > State.GOLD_THRESHOLD) {
      customer.goldState.setAwardMiles(awardMiles);
      customer.setState(customer.goldState);
    }
  }
}

class Gold extends State {
  Customer customer;

  public Gold(Customer customer) {
    this.customer = customer;
    status = "Gold";
  }

  public void addMiles(int miles) {
    awardMiles += 3 * miles;
  }
}
```

```
public class TestCustomer {
  public static void main(String[] args) {
    Customer customer = new Customer();
    customer.addMiles(20000);
    System.out.println(customer.toString());

    customer.addMiles(100);
    System.out.println(customer.toString());

    customer.addMiles(100);
    System.out.println(customer.toString());

    customer.addMiles(20000);
    System.out.println(customer.toString());
  }
}
```

The **Customer** class contains all three states **regularState**, **silverState**, and **goldState**. A customer is in regular state initially (Line 5). The state changes automatically when new award miles are added for the customer using the **addMiles** method (Line 15).

The abstract **State** class defines the data and operations for the state. The **addMiles** method (Line 30) is abstract and it is left for concrete subclasses **Regular**, **Silver**, and **Gold** to override. The **addMiles** method (Line 57) in the **Regular** class first adds the actual miles to the award mile and then checks whether the customer could be upgraded to gold or silver status (Lines 59-66). If so, set a new state in the customer.

The **TestCustomer** class creates a **Customer** with an initial regular state. As the award miles increase, the state is automatically changed to silver and gold.

The State pattern encapsulates all information for a state into one object and this object also incorporates state transition logic. It helps avoid inconsistent state since state changes occurs using just the one state object and not several objects or attributes.

The State pattern and the Strategy pattern are very similar in the sense that both patterns enable the object to change the behavior at runtime. The State pattern allows a state object to be changed in a context object and the Strategy pattern allows a strategy object to be plugged into an object. Using the State pattern, you create a context object with an initial state. The state object is automatically changed in the context according to predefined transition logics. Using the Strategy pattern, you control what strategy to use on an object.

15 The Visitor Pattern

The *Visitor pattern* represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. Figure 13 illustrates the structure of the Visitor pattern.

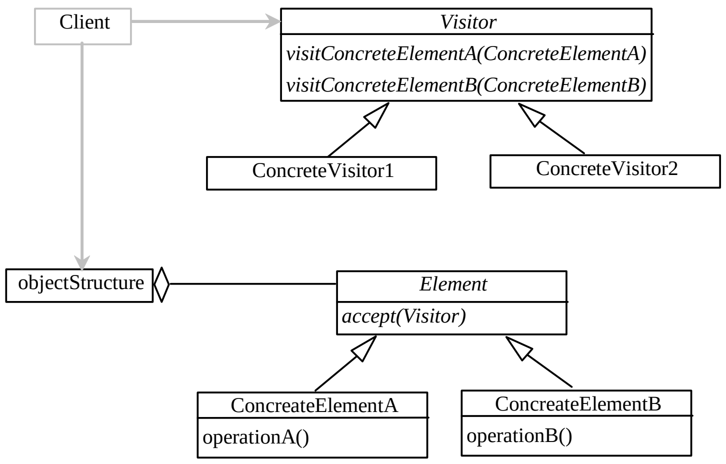


Figure 13

The Visitor pattern defines operations to be performed on elements of various types.

Visitor is usually an interface or abstract class that defines the various visit methods for processing the elements. A concrete visitor class defines a particular process when visiting the elements. **Element** defines the **accept** method that sets a visitor for the element. When an element invokes the **accept** method, the processing of the element is delegated to the visitor. Listing 27 gives a sample implementation of the structure.

Listing 27 Sample Implementation of a Visitor Pattern

```
interface Visitor {
  public void visitConcreteElementA(ConcreteElementA element);
  public void visitConcreteElementB(ConcreteElementB element);
}

class ConcreteVisitor1 implements Visitor {
  public void visitConcreteElementA(ConcreteElementA element) {
    System.out.println("Visitor 1 for element type A");
  }
  public void visitConcreteElementB(ConcreteElementB element) {
    System.out.println("Visitor 1 for element type B");
  }
}

class ConcreteVisitor2 implements Visitor {
  public void visitConcreteElementA(ConcreteElementA element) {
    System.out.println("Visitor 2 for element type A");
  }

  public void visitConcreteElementB(ConcreteElementB element) {
    System.out.println("Visitor 2 for element type B");
  }
}

interface Element {
  public abstract void accept(Visitor visitor);
}

class ConcreteElementA implements Element {
  // Some operations

  public void accept(Visitor visitor) {
    visitor.visitConcreteElementA(this);
  }
}

class ConcreteElementB implements Element {
  // Some operations

  public void accept(Visitor visitor) {
    visitor.visitConcreteElementB(this);
  }
}

public class TestVisitor {
  public static void main(String[] args) {
    Visitor visitor1 = new ConcreteVisitor1();
    Visitor visitor2 = new ConcreteVisitor2();
    Element element1 = new ConcreteElementA();
    Element element2 = new ConcreteElementB();

    element1.accept(visitor1);
    element1.accept(visitor2);
    element2.accept(visitor1);
    element2.accept(visitor2);
  }
}
```

To demonstrate the Visitor pattern, consider an example of visiting the elements in an array. Suppose that the element types are **Integer** and **Date**. For an integer, the visitor uses the **NumberFomat** class to format the number. For a date, the visitor uses the **DateFormat** class to format the date, as shown in Listing 28.

Listing 28 Example of a Visitor Pattern

```
import java.util.*;
import java.text.*;

interface Visitor {
  public void visitInteger(IntegerElement element);
  public void visitDate(DateElement element);
}

class ConcreteVisitor implements Visitor {
  public void visitInteger(IntegerElement element) {
    NumberFormat formatter = NumberFormat.getNumberInstance();
    System.out.println(formatter.format(element.getValue()));
  }

  public void visitDate(DateElement element) {
    DateFormat formatter = DateFormat.getDateTimeInstance();
    System.out.println(formatter.format(element.getValue()));
  }
}

interface Element {
  public abstract void accept(Visitor visitor);
}

class IntegerElement implements Element {
  private Integer value;

  public IntegerElement(Integer value) {
    this.value = value;
  }

  public int getValue() {
    return value;
  }

  public void accept(Visitor visitor) {
     visitor.visitInteger(this);
  }
}

class DateElement implements Element {
  private Date value;

  public DateElement(Date value) {
    this.value = value;
  }

  public Date getValue() {
    return value;
  }

  public void accept(Visitor visitor) {
     visitor.visitDate(this);
  }
}

public class VisitorDemo {
  public static void main(String[] args) {
    Element[] list = {new IntegerElement(1000),
      new DateElement(new Date())};
    for (int i = 0; i < list.length; i++)
      list[i].accept(new ConcreteVisitor());
  }
}
```

The **Visitor** interface (Lines 4-7) defines the methods for visiting **Integer** and **Date** objects. **ConcreteVisitor** is a concrete implementation of the interface. The **Element** interface (Lines 21-23) defines the **accept** method for an element to accept a visitor. The **IntegerElement** (Lines 25-39) wraps an **Integer** value and implements the **accept** method to let a visitor process the integer value. The **DateElement** (Lines 41-55) wraps a **Date** value and implements the **accept** method to let a visitor process the date value.

You may rewrite the program in Listing 27 as follows:

```
    public class VisitorDemo {
      public static void main(String[] args) {
        Element[] list = {new IntegerElement(1000),
          new DateElement(new Date())};
        NumberFormat formatter1 =
    NumberFormat.getNumberInstance();
        DateFormat formatter2 = DateFormat.getDateTimeInstance();

        for (int i = 0; i < list.length; i++) {
          if (list[i] instanceof Integer)
            System.out.println(formatter1.format(list[i]));
          else if (list[i] instanceof Date)
            System.out.println(formatter2.format(list[i]));
        }
      }
    }
```

This revised program appears simpler than the program in Listing 27. But the program developed using the Visitor pattern is more flexible than this revised program. The Visitor pattern allows you to add new operations by simply declaring a new concrete visitor class and these operations are centralized in a visitor. If the elements are in a composite structure, The Visitor pattern allows you add operations to the structure without changing the structure itself.

This Visitor pattern uses a technical called *double dispatch* to perform the operation on the element. When an element invokes the **accept** method with a visitor, which **accept** method is invoked is dynamically decided, depending on the actual element type. This is a single dispatch using polymorphism. When a visitor invokes the **visitInteger** method with an element, which **visitInteger** method is invoked is dynamically decided, depending on the actual visitor type. This is another single dispatch using polymorphism. So the Visitor pattern uses double dispatch.

16 The Memento Pattern
The *Memento pattern* can be used to return an object to one of its previous states. This pattern can be implemented using serialization in Java to store and restore an object.

17 The Prototype Pattern
The *Prototype pattern* specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. The prototype pattern allows you to make new instances by copying existing instances. This pattern is supported in Java using the **clone** method. To enable an object to be cloned, the class must implement the **Cloneable** interface and override the **clone()** method.

18 The Mediator Pattern
The *Mediator pattern* can be used to centralize complex communications and control between related objects. The objects notify the mediator when their state changes and respond to requests from the mediator. With the mediator in place, the objects are completely decoupled from each other.

19 The Interpreter Pattern
The *Interpreter pattern* can be used to build an interpreter for a language according to its grammar rules. You can use a class hierarchy to represent grammar rules. The interpreter recursively evaluate a parse tree of rule objects.

20 The Flyweight Pattern
The *Flyweight pattern* can be used to share objects rather than creating separate objects with identical states. For

example, an icon object can be shared in a menu, a tool box, or a button.

21 The Bridge Pattern

The *Bridge pattern* allows you to vary the implementation and the abstraction by placing the two in separate class hierarchies.