Java Methods

A & AB

Object-Oriented Programming
and
Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

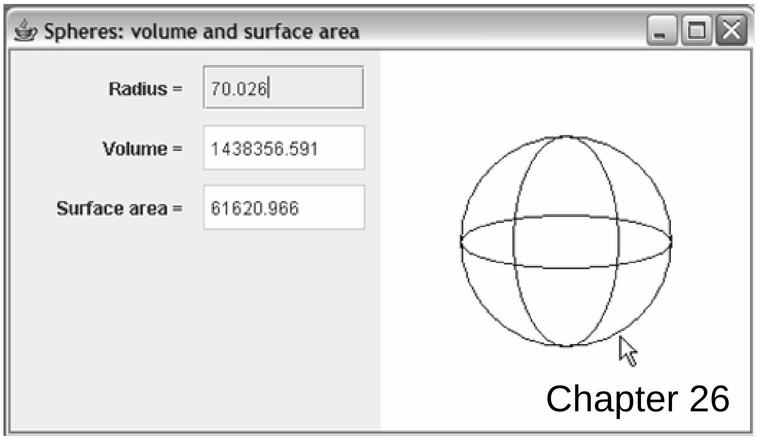
Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2005910949

ISBN-10: 0-9727055-7-0
ISBN-13: 978-0-9727055-7-8



Radius = 70.026

Volume = 1438356.591

Surface area = 61620.966

Chapter 26

Design Patterns

26.1 Prologue

Object-oriented design is not easy — designing a software application often takes more time than coding it, and design errors may be more costly than errors in the code. *Design patterns* represent an attempt by experienced designers to formalize their experience and share it with novices. Design patterns help solve common problems and avoid common mistakes.

The idea of design patterns came to OOP from an influential writer on architecture, Christopher Alexander [1]. In his books, *The Timeless Way of Building*[*] and *A Pattern Language*,[**] Alexander introduced design patterns as a way to bring some order into the chaotic universe of arbitrary architectural design decisions: how rooms should be connected or where windows should be placed. In *A Pattern Language*, Alexander and his co-authors catalogued 253 patterns that helped solve specific architectural problems and offered standard ideas for better designs.

No one has a good formal definition of a "pattern" — somehow we recognize a pattern when we see one. In fact, we humans are very good at pattern recognition. We recognize a pattern as some recurring idea manifested in diverse situations. We talk about organizational patterns and patterns of behavior, grammatical patterns, musical patterns, speech patterns, and ornament patterns. Recognizing patterns helps us structure our thoughts about a situation and draw on past experiences of similar situations.

Experts, researchers, and volunteers have published some OO software design patterns in books, magazine articles, and on the Internet. The first and most famous book on the subject, *Design Patterns*, was published in 1995.[***] Since then, hundreds of patterns have been published, some rather general, others specialized for particular types of applications. Apparently many people enjoy discovering and publishing new design patterns. The great interest in design patterns is evident from the numerous conferences, workshops, discussion groups, and web sites dedicated to collecting and cataloguing patterns [1, 2].

A more or less standard format for describing patterns has emerged. A typical description includes the pattern name; a brief statement of its intent or the problem it

[*] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.
[**] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*. Oxford University Press, 1977.
[***] The famous "Gang of Four" book. *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley, 1995.

solves; a description of the pattern and the types of classes and objects involved; perhaps a structural diagram; and an example, sometimes with sample code.

With too many patterns around, there is a danger that a novice may get lost. It is a good idea to start with only a handful of the most commonly used patterns and to understand exactly when and how they are used. Don't feel inadequate if your initial design doesn't follow an "officially" named pattern. But if the design runs into a problem, you can try to find a pattern that deals with that kind of problem, fix the design, and follow the pattern in the future. Fortunately, unlike buildings, programs are not set in concrete — it is often possible to change the design through minor restructuring of classes while keeping most of the code intact. If you find a standard pattern that fits your situation really well, it may bring you great satisfaction.

Being aware that OO design patterns exist helps you pay more attention to the design stage of your project before rushing to write code.

In the following sections we briefly review and illustrate six common design patterns: Façade, Strategy, Singleton, Decorator, Composite, and MVC (Model-View-Controller).

26.2 Façade

When you design a complex software system, it is often split into subsystems, each implemented in several classes. The Façade design pattern solves the problem of a complicated interface to a subsystem, replacing complex interfaces to several classes with one simpler interface to the whole subsystem. This is achieved by hiding the functionality of the subsystem's classes and their interactions in one "black-box" class.

On example of Façade is our `EasyReader` class, described in Appendix E. If you look at `EasyReader`'s code, you will see that this class hardly does any work — it simply delegates its responsibilities to the library classes. For example:

```
public EasyReader(String fileName)
{
  ///
  try
  {
    inputFile = new BufferedReader(new FileReader(fileName), 1024);
  }
  catch (FileNotFoundException ex)
  {
    ...
  }
}

public String readLine()
{
  String s = null;

  try
  {
    s = inputFile.readLine();
  }
  catch (IOException ex)
  {
    ...
  }
  ...
}
```

A beginner does not necessarily want to deal with different types of Java I/O classes or exceptions. `EasyReader` provides an adequate façade. In Java 5.0, Java developers have created their own façade for their I/O classes, the `Scanner` class. (`EasyReader` is still easier to use, because its constructor takes a pathname as a parameter and does not throw exceptions, and because it has a `readChar` method.)

The Façade design pattern can also be used for encapsulating a process that involves several steps. Suppose we are designing an application for storage and automatic retrieval of scanned documents (such as parking tickets). A document has a number printed on it, and we want to use OCR (Optical Character Recognition) to read that number and use it as a key to the document. The OCR subsystem may include several components and classes: `ImageEditor`, `TextLocator`, `OCRReader`, and so on. For example:

```
    ...
    Rectangle ocrArea = new Rectangle(200, 20, 120, 30);
    ImageEditor imageEditor = new ImageEditor();
    image = imageEditor.cut(image, ocrArea);
    TextLocator locator = new TextLocator();
    ocrArea = locator.findTextField(image);
    String charSet = "0123456789";
    OCRReader reader = new OCRReader(charSet);
    String result = reader.ocr(image, ocrArea);
    ...
```

A client of the OCR subsystem does not need to know all the detailed steps — all it wants from the OCR subsystem is the result. The OCR subsystem should provide a simple façade class, something like this:

```
public class OCR
{
  public static String read(Image image, Rectangle ocrArea)
  {
    ...
  }
}
```

Java library classes and interfaces involved in playing an audio clip from a `.wav` or `.au` file include `File`, `IOException`, `AudioFormat`, `AudioInputStream`, `AudioSystem`, `DataLine`, `DataLine.Info`, `SourceDataLine`, and `LineUnavailableException`, Our `EasySound` class, described in Appendix E, provides a façade for that, with one class, one constructor, and one method.

26.3 Strategy

The Strategy design pattern is simply common sense. If you expect that an object may eventually use different strategies for accomplishing a task, make the strategy module "pluggable" rather than hard-coded in the object. For example, in the design of our *Chomp* project in Chapter 12 we made provisions for choosing different strategies for the `ComputerPlayer`. If we need to support different levels of play or different board sizes or even switch the strategy in the middle of a game, all we need to do is pass a new `Strategy` parameter to `ComputerPlayer`'s `setStrategy` method. To accomplish this separation, we defined an interface `Strategy`, so that different strategy classes can implement it. For example:

```
public class Chomp4by7Strategy implements Strategy
{
  ...
}
```

We then can pass a particular type of `Strategy` object to the `ComputerPlayer`'s `setStrategy` method. For example:

```
ComputerPlayer computer = new ComputerPlayer(this, game, board);
computer.setStrategy(new Chomp4by7Strategy());
```

We also used the `Strategy` design pattern in the *Dance Studio* project in Chapter 11: a `Dance` was treated as a "strategy" for a `Dancer`, and the `Dancer`'s `learn(Dance d)` method played the role of a "set strategy" method.

The Java Swing package follows the Strategy pattern for laying out components in a container. Different strategies are implemented as different types of `Layout` objects. A particular layout is chosen for a container by calling its `setLayout` method. For example:

```
JPanel panel = new JPanel();
GridBagLayout gbLayout = new GridBagLayout();
panel.setLayout(gbLayout);
```

26.4 Singleton

Suppose we want to have a "log" file in our program and we want to write messages to it from methods in different classes. We can create a `PrintWriter` object `log` in `main`. The problem is: How do we give all the other classes access to `log`? We only ever need one log file open at a time, and it is tedious to pass references to this file in various constructors and methods.

The first solution that comes to mind is to define a special `LogFile` class with a static `PrintWriter` field embedded in it. `LogFile`'s methods are all static and simply delegate their responsibilities to the corresponding `PrintWriter` methods. For example:

```
public class LogFile  // Does not follow Singleton pattern
{
  private static PrintWriter myLog;

  private LogFile()  // Can't instantiate this class
  {
  }

  public static void createLogFile(String fileName)
  {
    if (myLog == null)
      myLog = new PrintWriter(new FileWriter(pathName, false));
  }

  public static void println(String line)
  {
    if (myLog != null)
      myLog.println(line);
  }

  public static void closeLog()
  {
    if (myLog != null)
      myLog.close();
  }
}
```

Now `main` can call `LogFile.createLogFile("log.txt")`, and client classes can call `LogFile.println(msg)`.

This solution works, but it has two flaws. First, `LogFile`'s methods are limited to those that we have specifically defined for it. We won't be able to use all `PrintWriter`'s methods unless we implement all of them in `LogFile`, too. Second, `LogFile` is a class, not an object, and we have no access to the instance of `PrintWriter` embedded in it. So we cannot do things with it that we usually do with an object, such as pass it to constructors or methods, add it to collections, or "decorate" it as explained in the next section.

The Singleton design pattern offers a better solution: rather than channeling method calls to the `PrintWriter` instance in the `LogFile` class, make that instance accessible to clients. Provide one static method in `LogFile` that initializes and returns the `PrintWriter` field embedded in `LogFile`, but make sure that that field is initialized only once and that the same reference is returned in all calls. For example:

```
public class LogFile
{
  private static EasyWriter myLog;

  protected LogFile()  // Can't instantiate this class
  {
  }

  public static PrintWriter getLogFile(String fileName)
  {
    if (myLog == null)
      myLog = new PrintWriter(new FileWriter(pathName, false));
    return myLog;
  }

  public static void closeLog()
  {
    if (myLog != null)
      myLog.close();
  }
}
```

Now any method can get hold of the log file and use all `PrintWriter`'s methods. For example:

```
    PrintWriter log = LogFile.getLogFile("log.txt");
    log.print(lineNum);
    log.print(' ');
    log.println(msg);
```

This design is better, but not perfect. When the `get...` method takes an argument, as above, it gives the impression that we can construct different objects (for example, log files with different names). In fact, the parameter has an effect only when the `get...` method is called for the first time. All the subsequent calls ignore the parameter. We could provide an overloaded no-args version of `get...` and call it after the log file is created.

In a slightly more sophisticated version, our `LogFile` class could keep track of the file names passed to its `getLogFile` method and report an error if they disagree. Or perhaps it could keep a set of all the different log files and return the file that matches the name. But then it wouldn't be a Singleton any more... Have we just discovered a new design pattern?

26.5 Decorator

Suppose you are designing a geometry package. You have started a little class hierarchy for triangles:

```
public abstract class Triangle
{
  ...
  public abstract void draw(Graphics g);
}

public class RightTriangle extends Triangle
{
  ...
  public void draw(Graphics g)
  {
    g.drawLine(x, y, x + a, y);
    g.drawLine(x, y, x, y - b);
    g.drawLine(x+a, y, x, y - b);
  }
}

public class IsoscelesTriangle extends Triangle
{
  ...
  public void draw(Graphics g)
  {
    g.drawLine(x, y, x - c/2, y + h);
    g.drawLine(x, y, x + c/2, y + h);
    g.drawLine(x - c/2, y + h, x + c/2, y + h);
  }
}
```

So far, so good. Now you want to add the letters *A*, *B*, *C* to denote vertices in your drawings. And sometimes to draw a median. Or a bisector. Or just a median but no letters. Or three medians. This is beginning to look like a nightmare. What do you do? Do you extend each of the classes to satisfy all these multiple demands? Like this:

```
public class RightTriangleWithMedian extends RightTriangle
{
  ...
  public void draw(Graphics g)
  {
    super.draw(g);
    g.drawLine(x, y, x + a/2, y - b/2);
  }
}

public class IsoscelesTriangleWithMedian extends IsoscelesTriangle
{
  ...
  public void draw(Graphics g)
  {
    super.draw(g);
    g.drawLine(x, y, x, y + h);
  }
}
```

And so on. And what if you want to construct a triangle object and sometimes show it with letters and sometimes without? Polymorphism won't allow you to show the same object in different ways.

The Decorator design pattern helps you solve these two problems: (1) the lack of multiple inheritance in Java for adding the same functionality to classes on diverging inheritance branches (such as adding letters to drawings of all kinds of triangles) and (2) the difficulty of changing or extending the behavior of an individual object at run time (such as drawing the same triangle object, sometimes with letters and other times without), as opposed to changing or extending the behavior of the whole class through inheritance.

The idea of the Decorator design pattern is to define a specialized "decorator" class (also called a *wrapper* class) that modifies the behavior of or adds a feature to the objects of the "decorated" class. Decorator uses real inheritance only for inheriting the decorated object's data type. At the same time, decorator uses a kind of do-it-yourself "inheritance," actually modeled through embedding. It has a field of the decorated class type and redefines all the methods of the decorated class, delegating them to that embedded field. The decorator adds code to methods where necessary. The decorator's constructor initializes the embedded field to an object to be decorated.

This is how the code might look in our `Triangle` example:

```
public class TriangleWithABC extends Triangle
{
  private Triangle myTriangle;

  public TriangleWithABC(Triangle t)
  {
    myTriangle = t;
  }

  public void draw(Graphics g)
  {
    myTriangle.draw(g);
    drawABC(g);
  }

  ...

  private drawABC(Graphics g)
  {
    ...
  }
}
```

A more formal decorator diagram is shown in Figure 26-1.

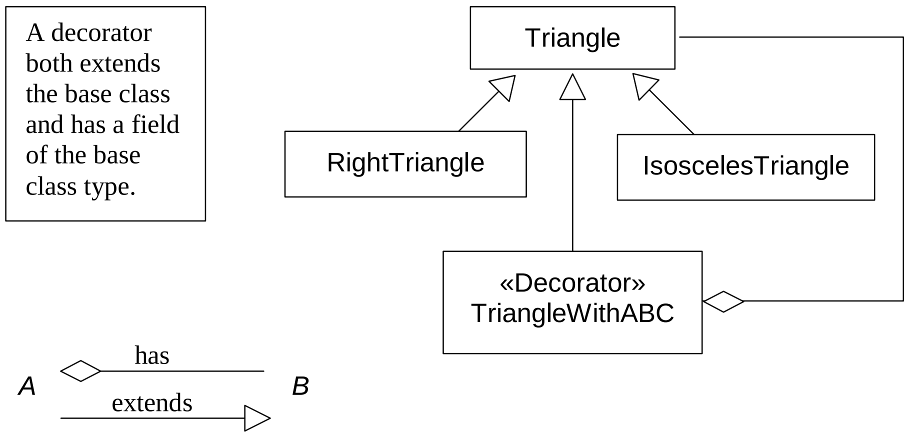

A decorator both extends the base class and has a field of the base class type.
Triangle
RightTriangle
IsoscelesTriangle
«Decorator» TriangleWithABC
A
has
extends
B

Figure 26-1. The Decorator design pattern

Now you can pass any type of triangle to your decorator. Polymorphism takes care of the rest. For example:

```
Triangle rightT = new RightTriangle(...);
Triangle isosT = new IsoscelesTriangle(...);
Triangle rightTwABC = new TriangleWithABC(rightT);
Triangle isosTwABC = new TriangleWithABC(isosT);
...
rightTwABC.draw(g);
isosTwABC.draw(g);
```

Or, for short:

```
Triangle rightTwABC =
       new TriangleWithABC(new RightTriangle(...));
Triangle isosTwABC =
       new TriangleWithABC(new IsoscelesTriangle(...));
rightTwABC.draw(g);
isosTwABC.draw(g);
```

This solves the first problem — adding the same functionality to classes on diverging inheritance branches.

You can also change the behavior of an object at run time by using its decorated stand-ins when necessary. For example:

```
Triangle rightT = new RightTriangle(...);
Triangle rightTwABC = new TriangleWithABC(rightT);
...
rightT.move(100, 50);     // moves both rightT and rightTwABC
...
rightT.draw(g);           // draw without A,B,C
...
rightTwABC.draw(g);       // draw THE SAME triangle with A,B,C
...
```

Since a decorated triangle is still a `Triangle`, we can pass it to another decorator. For example:

```
Triangle rightT = new RightTriangle(...);
Triangle rightTwABCwMedian =
    new TriangleWithMedian(new TriangleWithABC(rightT));
```

Exactly the same structure would work if the base type to be decorated (for example, `Triangle`) were an interface, rather than a class.

One problem with decorators is that we need to redefine all the methods of the base class (or define all the methods of the interface) in each decorator. This is repetitive and laborious. A better solution is to define a kind of abstract “decorator adapter” class, a generic decorator for a particular type of objects, and then derive all decorators from it. This structure is shown in Figure 26-2.

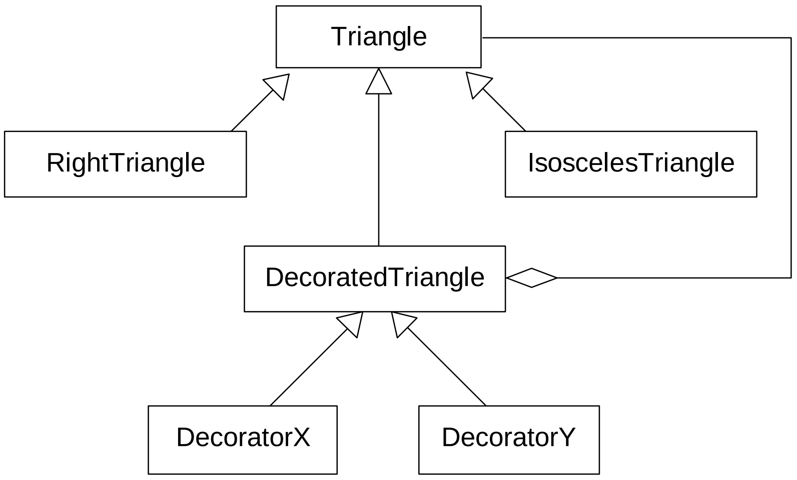


Figure 26-2. The Decorator design pattern with the intermediate abstract decorator class

The code may look as follows:

```
public abstract class DecoratedTriangle extends Triangle
{
  protected Triangle myTriangle;

  protected DecoratedTriangle(Triangle t)
  {
    myTriangle = t;
  }

  public void move(int x, int y)
  {
    myTriangle.move(x, y);
  }

  public void draw(Graphics g)
  {
    myTriangle.draw(g);
  }

  ...  // redefine ALL other methods of Triangle
}
```

Now you can derive a specific decorator from `DecoratedTriangle`, redefining only the necessary methods:

```
public class TriangleWithABC extends DecoratedTriangle
{
  public void TriangleWithABC(Triangle t)
  {
    super(t);
  }

  public void draw(Graphics g)
  {
    super.draw(g);
    drawABC(g);
  }

  private drawABC(Graphics g)
  {
    ...
  }
}
```

Decorators are a potential source of subtle bugs. An object of a decorator class possesses the same fields as the decorated object (because a wrapper class extends the wrapped class) but they are not used and remain uninitialized. This may cause problems when a wrapped object is passed to a method that expects an unwrapped object and refers directly to its fields. That is why it's a good idea not to refer directly to fields of other objects passed to methods, even if they appear to be objects of the same class. Before you decorate a class, make sure it follows this convention.

In our own projects, we benefited from the Decorator design pattern in the *Dance Studio* program (Chapter 11). The `ReversedDance` class that you wrote is a decorator for any dance (any class that implements the `Dance` interface).

Decorators must be used with caution: too many decorative layers make code unreadable. The Java stream I/O package, for example, uses decorators extensively, but the Java API documentation does not readily list wrapper classes for a given class. As a result, we felt it was necessary to put a reasonable façade on it, with our `EasyReader` and `EasyWriter` classes.

26.6 Composite

The Composite design pattern is a recursive pattern useful for nested structures. It applies when we want a list or a set of things of a certain type to also be a thing of that type.

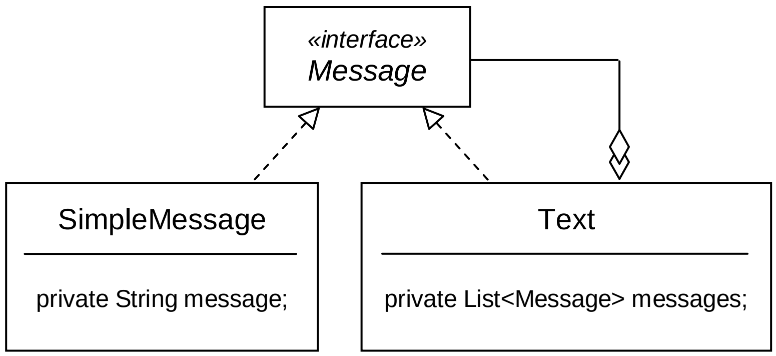
Consider, for example the following two classes:

```
public class SimpleMessage implements Message
{
  private String message;

  public SimpleMessage()
  {
    message = "";
  }

  public SimpleMessage(String str)
  {
    message = str;
  }

  public void print()
  {
    System.out.print(message + " ");
  }
}

public class Text implements Message
{
  private List<Message> messages;

  public Text()
  {
    messages = new LinkedList<Message>();
  }

  public void add(Message msg)
  {
    messages.add(msg);
  }

  public void print()
  {
    for (Message msg : messages)
      msg.print();      // works polymorphically both for a Message
                        // object and for a composite Text object
  }
}
```

Both implement the interface `Message`:

```
public interface Message
{
  void print();
}
```

A `Text` object both IS-A `Message` and HAS-A list of `Messages`:



Therefore, we can add `Message` objects to a `Text` object without worrying whether they are simple "one-liners" or other "texts." For example:

```
     Text fruits = new Text();
     fruits.add(new SimpleMessage("apples"));
     fruits.add(new SimpleMessage("and"));
     fruits.add(new SimpleMessage("bananas"));
     Text song = new Text();
     song.add(new SimpleMessage("I like to eat"));
     song.add(fruits);
     song.print();
     System.out.println();
```

The output will be

```
I like to eat apples and bananas
```

When we call `print` for a `Text` object, polymorphism makes sure that all the `Messages` in its internal list are printed properly. In this example, `Message` could be an abstract class rather than an interface.

The `java.awt` package uses the Composite pattern for handling GUI components: a `Container` is a `Component` and we can also add `Components` to it. Polymorphism makes sure that all components, both "simple" and "composite," are correctly displayed.

26.7 MVC (Model-View-Controller)

Suppose you are designing a 3-D geometry package. You want to demonstrate visually that the surface area of a sphere is proportional to its radius squared and that the volume is proportional to the radius cubed. When the user enters a new radius, your program displays a scaled picture of the sphere, the numbers for the surface area and volume, and perhaps a bar chart that compares them. The user can also stretch the model sphere with a mouse; the numbers change automatically (Figure 26-3).

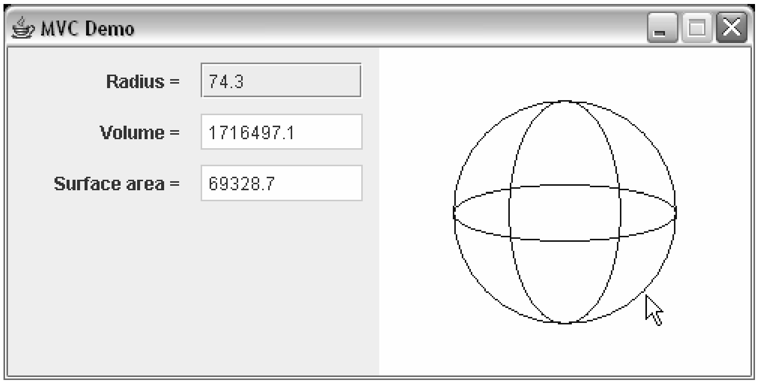


Figure 26-3. Sphere model with text and mouse input

As you know, the first rule in a program of this kind is to isolate the model part (representation of a sphere) and separate it from the GUI part (control and display functions). In this example, we can discern several different "views" of the model: the graphics view that shows a picture of the sphere, the text view that displays the numbers for the surface area and volume, and perhaps other views. It is reasonable to implement these different views as different classes.

There are also a couple of different "controllers": one lets the user type in the radius of the sphere, the other lets the user change the radius with the mouse. Again we may want to implement them as different classes. The problem is how to structure the interactions between all these classes. The MVC design pattern offers a flexible solution.

MVC applies to situations where you have a “model” class that represents a system, a situation, or a real-world or mathematical object. The model’s fields describe the state of the model. The model class is isolated from the user interface, but there are one or several “views” of the model that reflect its state. When the state changes, all the views have to be updated.

You might be tempted to set up a “totalitarian system” in which one central controller updates the model and manages all the views. In this approach, the model is not aware that it is being watched and by whom. The controller changes the model by calling its modifier methods, gets information about its state by calling its accessor methods, and then passes this information to all the views (Figure 26-4). For example:

```
//  "Totalitarian system":

public class SphereController implements ActionListener
{
  private Sphere sphereModel;
  private TextView view1;
  private GraphicsView view2;
  ...

  public SphereController()
  {
    sphereModel = new Sphere(100);
    view1 = new TextView();
    view2 = new GraphicsView();
    ...
  }

  private void updateViews()
  {
    double r = sphereModel.getRadius();
    view1.update(r);
    view2.update(r);
  }

  public void actionPerformed(ActionEvent e)
  {
    String s = ((JTextField)e.getSource()).getText();
    double r = Double.parseDouble(s);
    sphereModel.setRadius(r);  // update the model
    updateViews();             // update the views
  }
  ...
}
```

This setup becomes problematic if requests to change the model come from many different sources: GUI components, keyboard and mouse event listeners, timers, even the model itself. All the different requests would have to go through the cumbersome central bureaucracy.

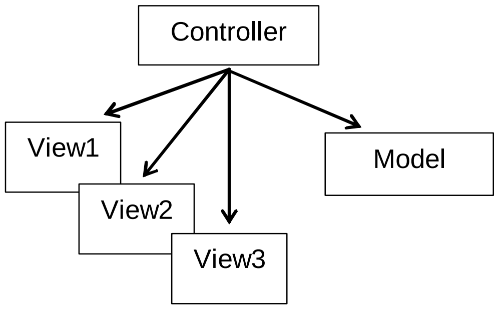


Figure 26-4. "Totalitarian" control of the model and the views

MVC offers a decentralized solution where the views are attached to the model itself. The model knows when its state changes and updates all the views when necessary (Figure 26-5). An MVC design can support several independent views and controllers and makes it easy to add more views and controllers.

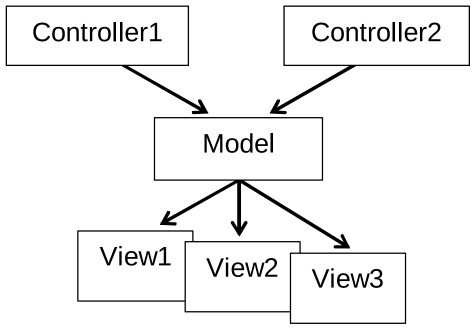


Figure 26-5. The MVC design pattern

In our example, the MVC code with one controller might look like this:

```
public class SphereController implements ActionListener
{
  private Sphere sphereModel;
  private TextView view1;
  private GraphicsView view2;
  ...

  public SphereController()
  {
    sphereModel = new Sphere(100);
    sphereModel.addView(new TextView());
    sphereModel.addView(new GraphicsView());
    ...
  }

  public void actionPerformed(ActionEvent e)
  {
    String s = ((JTextField)e.getSource()).getText();
    double r = Double.parseDouble(s);
    sphereModel.setRadius(r);
  }
  ...
}

public class Sphere
{
  ...
  public setRadius(double r)
  {
    myRadius = r;
    updateAllViews();
  }
  ...
}
```

The model keeps all the views attached to it in a set or list.

❖ ❖ ❖

The Java library supports the MVC design pattern by providing the `Observable` class and the `Observer` interface in its `java.util` package. If your “model” class extends `Observable`, it inherits a reference to an initially empty list of observers and the `addObserver` method for adding an observer to the list. Your class also inherits the `setChanged` method for raising the “model changed” flag and the `notifyObservers` method for notifying all the registered observers.

Each view registered with the model must implement the `Observer` interface and provide its one required public method:

```
void update(Observable model, Object arg);
```

The first argument is the observed model; the second argument is any object passed from the model to the view as a parameter.

Figure 26-6 shows how it all might look in our example.

```
public class SphereController implements ActionListener
{
  private Sphere sphereModel;
  ...

  public SphereController()
  {
    sphereModel = new Sphere(100);
    sphereModel.addObserver(new TextView());
    sphereModel.addObserver(new GraphicsView());
    ...
  }

  public void actionPerformed(ActionEvent e)
  {
    String s = ((JTextField)e.getSource()).getText();
    double r = Double.parseDouble(s);
    sphereModel.setRadius(r);
  }
  ...
}

public class TextView
    implements java.util.Observer
{
  ...
  public void update(Observable model, Object arg)
  {
    Sphere sphere = (Sphere)model;
    ...
  }
}
```

Figure 26-6 continued ⮫

```
public class GraphicsView
    implements java.util.Observer
{
  ...
  public void update(Observable model, Object arg)
  {
    Sphere sphere = (Sphere)model;
    ...
  }
}

class Sphere extends java.util.Observable
{
  ...
  public void setRadius(double r)
  {
    myRadius = r;
    setChanged();         // Indicates that the model has changed
    notifyObservers();    // Or: notifyObservers(someObjectParameter)
  }
  ...
}
```

Figure 26-6. A sketch for the "Sphere" classes with `Observer`/`Observable`

The `notifyObservers` method in the model class calls `update` for each registered observer. If `notifyObservers` is called with some object as a parameter, that parameter is passed to `update` (as its second argument); if the model calls the overloaded no-args version of `notifyObservers`, then the parameter passed to `update` is `null`.

The complete code for our MVC sphere example with two controllers and two views is included in `JM\Ch26\MVC\source.zip`.

❖ ❖ ❖

In our case studies and labs we tried to follow the MVC design pattern where possible. For example, our *Craps*, *Chomp*, and *Dance Studio* designs all follow the MVC pattern. We did not use `Observer-Observable` there because the model had only one view and it was easier to handle it directly.

Java's Swing follows MVC in implementing GUI components. For example, the `JButton` class is actually a façade for the `DefaultButtonModel` (model), `BasicButtonUI` (view), and `ButtonUIListener` (controller) classes. This design makes it possible to implement "pluggable look and feel" by changing only the views.

MVC can be applied to more than just visual display: a "view" class can play sounds, update files, and so on.

The MVC design pattern is not universal. While it is always a good idea to isolate the "model" from the user interface, the "view" and "controller" may sometimes be intertwined too closely for separation. When you are filling out a form on the screen, for example, the viewing and controlling functions go together. There is no need to stretch a design pattern to fit a situation where it really does not apply.

26.8 Summary

We have described half a dozen design patterns to give you an idea of what design patterns are about and to help you get started. Many more patterns have been published. Being generally aware of their existence brings the issue of sound OO design into focus. Following specific design patterns helps beginners avoid common mistakes. An OOP designer should gradually become familiar with the more famous patterns and learn how and when to apply them.

The Façade design pattern is used to shield clients from the complexities of a subsystem of classes by providing a simplified interface to them in one "façade" class. A façade may serve as a "black box" for a process or a function.

The Strategy design pattern deals with attaching strategies (algorithms, solutions, decisions) to objects. If you want your design to be flexible, isolate each strategy in its own class. Have a higher-level class or `main` choose a strategy and attach it to a client. Do not allow a client to create its own strategy, because that will make it harder to change the client's strategy.

The Singleton design pattern deals with situations where you need to have only one object of a certain class in your program and you want to make that object readily available in different classes. A "Singleton" class embeds that object as a static field and provides a static accessor to it. The accessor initializes the object on the first call but not on subsequent calls.

The Decorator design pattern solves the problem of adding the same functionality to classes on diverging inheritance branches without duplication of code. It also offers an elegant solution for modifying an object's behavior at run time. A decorator both extends the base class and embeds an instance of the base class as a field. The decorator's methods delegate their responsibilities to the embedded field's methods but can also enhance them. A decorator provides a constructor that takes an object of the base class (or any class derived from it) as an argument. That is why decorator classes are also called wrapper classes.

The Composite design pattern is used for implementing nested structures. A “composite” is a class that both inherits from a base class and contains a set or a list of objects of the base class type. This way a composite can hold both basic objects and other composites, nested to any level.

The MVC design pattern offers a flexible way to arrange interactions between a “model” object, one or more “controller” objects, and one or more “view” objects. The views are attached to the model, which updates all the views when its state changes. Java supports MVC design by providing the `Observable` class and the `Observer` interface in its `java.util` package. The “model” class extends `Observable`, which provides methods to add an observer to the model and to notify all observers when the model needs to update them. A view class implements the `Observer` interface and must provide the `update` method, which is called when observers are notified.

Design patterns offer general ideas but should not be followed slavishly. Don’t try to stretch a pattern to fit a situation where it does not apply.

❖ ❖ ❖

OO design patterns and tools represent a bold attempt to turn the art of software design into something more precise, scientific, and teachable. But behind all the technical terms and fancy diagrams, some art remains an essential ingredient. As we said in the introduction, software structures are largely hidden from the world. Still, something — not only the need to finish a project on time — compels a designer to look for what Christopher Alexander called “the quality without a name”: order, balance, economy, fit to the purpose, and, in Alexander’s words, “a subtle kind of freedom from inner contradictions.” Let us join this search — welcome, and good luck!

🕮 Exercises 🕮

The exercises for this chapter are in the book (*Java Methods A and AB: Object-Oriented Programming and Data Structures*, *AP Edition*, ISBN 0-9727055-7-0, Skylight Publishing, 2006 [1]).