1 Introduction

As object-oriented programming became more common, programmers found that certain
problems or issues recurred. A collection of solutions emerged to address these common
problems when writing reusable object-oriented software; however, these solutions were
not available to the majority of programmers. In the mid 1990s, the book Design Patterns
by Gamma, Helm, Johnson, and Vlissides was published. It was received enthusiastically
because it made these high quality solutions widely available. A design pattern includes
a description of a problem, its solution, and commentary.

We will see that some of these patterns are implemented explicitly in Java. Others are
not. Some argue that all patterns should be recast as programming language features.
Nevertheless, even if a pattern is not implemented in Java, it can be quite useful.

2 Iterator

The iterator pattern is about providing a mechanism for moving about in an aggregate
data structure without exposing the details of the implementation. This separation of the
data structure and the iteration object has the following advantages.

• The iterator can provide more methods for movement than what would naturally be
  provided with a data structure. For example, an iterator can allow both forward and
  backward movement in a singly linked list. In Java, we see this extension realized
  in the ListIterator interface.

• It is possible to have different iterators for the same data structure. For example,
  a tree data structure might have iterators for in-order, pre-order, and post-order
  traversals; an array-like data structure might have an iterator that starts at the
  beginning and one that starts at the end.

• It is possible to have multiple iterator instances for the same data structure. If
  an iterator needs state, multiple instances can keep track of multiple states. If the
  iterator and the data structure were not separate, keeping track of multiple states
  would be significantly more complicated.

• Iterators provide a convenient mechanism for polymorphic iteration operators. In
  Java, we see that the for-each loop works on different collections.

Java incorporates the iterator pattern into the language explicitly with the Iterator and
ListIterator interfaces.

Example:

private static double sum(List<Double> dList){
    double total = 0.0;
for(Iterator<Double> it = dList.iterator(); it.hasNext();){
    Double d = it.next();
    total = total + d;
}
return total;

Another example, where this time we create our own iterator:

final ArrayList<Integer> is = new ArrayList<Integer>(Arrays.asList(1,2,3));
Iterator<Integer> it = new Iterator<Integer>() {
    private int pos = is.size() - 1;
    public boolean hasNext(){
        return pos > -1;
    }
    public Integer next(){
        return is.get(pos--);
    }
    public void remove(){ }
};

for(;it.hasNext();){ // prints the array list backwards
    Integer n = it.next();
    System.out.println(n);
}

3 MVC

Model-View-Controller, or MVC\textsuperscript{1}, is about creating a modular and maintainable user interface by putting different aspects of the GUI into separate objects.

The object that contains the data central for solving the problem is called the model. Typically, it also has methods to manipulate the data. For example, a calculator model might contain the running total, memory cells, and methods to get these values, and to do arithmetic based on the state; a clock model might contain hours and minutes, with methods to read, set, and advance the time; and a temperature model might contain the temperature, and methods to read and set the temperature.

\textsuperscript{1} MVC is not universally regarded as a design pattern; and while it is mentioned in the Design Patterns book, it is not in its catalog of design patterns.
The object that generates a graphical display is called the view. For example, a calculator view might be a window with buttons with numbers and operations, or simply a text prompt. A clock view might be a window that contained a circle together with lines for hands. Another view might be a window with text to display the time. Similarly, a temperature view might be a window with an image of a thermometer with a slider, or with a text display.

The object that detects and handles events, which might involve modifying the data in the model, is called the controller. For example, a calculator controller might respond to numeric button clicks, building up a number in one of the model’s memory cells. A clock controller might respond to a timer by advancing the model’s time representation every time it detects a tick. A temperature controller might respond to changes in the slider position by increasing or decreasing the temperature value in the model.

The model is mostly independent of the other two. It will typically let the views know of any changes, but the views are in charge of registering themselves with the model. The view is a function of the model, so any view instance must be supplied with a model. The controller needs to know about both the view and the model; it needs to know about the view in order to process events, and it needs to know about the model so as to modify or query it in response to an event.

The MVC separation has the following advantages.

- There can be multiple views of the same model. For example, a numerical model could have a spreadsheet view, a bar-graph view, and a pie-chart view.
- It is easier to test the model independent of the view and controller.
- The view can evolve independently of the model.

In Java, the view is implemented using the AWT or Swing components to create windows, buttons, text fields, etc. However, in contrast to the usual MVC pattern, the view and the controller are tightly coupled. In particular, the controller is implemented as listener objects attached to view objects such as buttons or text fields.

4 Observer

Frequently, a model needs to inform its views that its state has changed so that the views can update so as to properly reflect the model. The observer pattern is about providing a mechanism that facilitates one object (the observable) transparently informing others (the observers) that its state has changed. This mechanism involves the observable inheriting hidden state to record the observers as well as methods to register and notify the observers. The observers need a standard way to react to a state change and so they must implement an update method.

Java supplies an implementation of the observer pattern. The class Observable provides methods that the observable object needs; the interface Observer provides the method that the observable object uses to let the observers know a change has occurred.
The class **Observable** includes the following methods.

- **addObserver** This method takes an **Observer** as a parameter. It is used by an observer to register itself with the observable object.
- **setChanged** The observable object explicitly indicates a change has occurred by invoking this method.
- **hasChanged** The observers can invoke this method to see if a change has occurred, and then update accordingly.
- **notifyObservers** This method is invoked by the observable object to signal all observers to check for a model change, and update accordingly.

The **Observer** interface requires the method **update(Observable, Object)**. It is the method that **notifyObservers** invokes.

### 5 Decorator

The decorator pattern is about using classes to add functionality via composition rather than inheritance. It has the following advantages over inheritance.

- It has more flexibility than inheritance. The inheritance hierarchy is specified statically. A decorator can be applied dynamically. It is also possible to compose a decorators recursively.
- One can avoid putting an excessive number of methods in a single class, by placing additional functionality in decorators.

Given a **Writer** instance \( w \), it is possible to modify its behavior by providing it to a decorating **Writer** that inspects and modifies the input before sending it to \( w \).

Example:

```java
import java.io.*;

public class Hacker extends Writer {

private Writer w;

public Hacker(Writer w) {
    this.w = w;
}

public void write(int c) throws IOException {
    if (c == 'a') {
        w.write('@');
    } else if (c == 's') {
        w.write('$');
    } else {
        w.write(c);
    }
}
```
public void write(char[] cbuf, int off, int len) throws IOException {
    for(int i = 0; i< len; i++) {
        write(cbuf[off + i]);
    }
}

public void close() throws IOException {
    w.close();
}

public void flush() throws IOException {
    w.flush();
}

public static void main(String[] args){
    BufferedReader brd = null;
    BufferedWriter bwr = null;
    try {
        brd = new BufferedReader(new InputStreamReader(System.in));
        bwr = new BufferedWriter(new Hacker(new OutputStreamWriter(System.out)));
        String str;
        while((str = brd.readLine()) != null){
            bwr.write(str);
            bwr.newLine();
        }
    } catch(IOException e) {
    } finally {
        try{
            if (brd != null) {brd.close();}
            if (bwr != null) {bwr.close();}
        } catch(IOException e) {
        }
    }
}

6 Factory Method

The factory method pattern is about using a method to create appropriate instances of sub-classes of some class. It may be that the method takes parameters that are used to determine which sub-class should be created.

A createBook method might take parameters, including a media string, and based on that string, the method could return either an instance of HardCover or AudioBook.
Example:

```java
public class BookFactory {
    public static Book createBook(String[] bd) {
        int bookType = determineBookType(bd);

        switch (bookType) {
            case BookFactory.HARD:
                return new HardCover(bd);
            case BookFactory.AUDIO:
                return new AudioBook(bd);
            // etc.
        }
    }
}
```

Factory methods can also be used to give different names to different construction methods. When creating a complex number, we might want the option of either creating a complex number by supplying Cartesian coordinates or polar coordinates. We cannot use an ordinary constructor, because the types cannot distinguish the two; rather the names of the methods must. Since Java constructors have only one name, we hide the actual constructor and use factory methods.

Example:

```java
public class Complex {
    public static Complex fromCartesian(double x, double y) {
        return new Complex(x, y);
    }

    public static Complex fromPolar(double r, double theta) {
        return new Complex(r * cos(theta), r * sin(theta));
    }

    private Complex(double a, double b) {
        //...
    }
}
```

7 Command

The command pattern is about representing actions as objects rather than implicit method calls. For example, a listener in an ATM GUI might directly call a withdraw method in the bank model. The command pattern approach would realize the withdraw request as

---

a withdraw object. The bank model would then respond to that command object by performing the withdrawal. Making the request an object has the following advantages.

- Commands can be naturally recorded in a log.
- Having a sequence of recorded commands makes in simpler to undo the operations.
- Adding additional commands becomes simpler. There no longer needs to be a public method for every command. Further, high level commands can be interpreted as sequences of lower-level commands.

Extended example:

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

import java.util.Stack;

public class Counter {
    private Stack<Command> history = new Stack<Command>();
    private int state = 0;

    public class Increment implements Command {
        private int oldState;
        private boolean canUndo;

        public void execute(){
            oldState = state;
            canUndo = true;
            state = oldState + 1;
        }

        public void undo(){
            if (canUndo) {
                canUndo = false;
                state = oldState;
            } else {
                System.err.println("Cannot undo!");
            }
        }
    }

    public class Display implements Command {

        public void execute(){
            System.out.println("Value is " + state);
        }
    }
}
public void undo()
{
    System.out.println("Cannot undo display.");
}

public void process(Command c)
{
    history.push(c);
    c.execute();
}

public void undoLast()
{
    if (!history.empty())
    {
        Command c = history.pop();
        c.undo();
    }
    else {
        System.err.println("Nothing to undo!");
    }
}

public static void main(String[] args)
{
    Counter count = new Counter();

    count.process(count.new Increment());
    count.process(count.new Increment());
    count.process(count.new Increment());
    count.undoLast();
    count.process(count.new Display());
}