Lecture 14: Design Patterns

2. Structural Patterns

2.2. Decorator Pattern

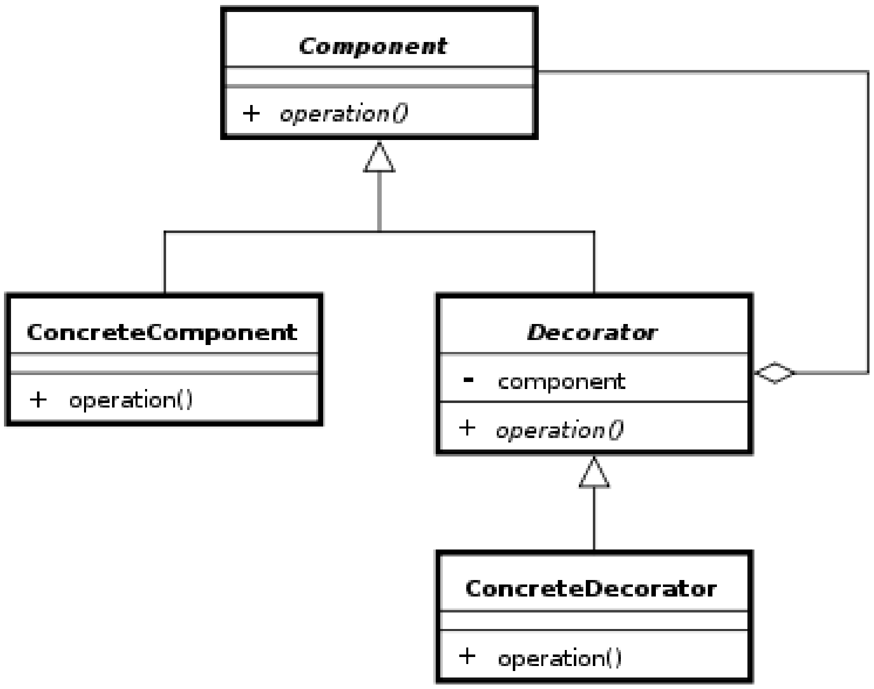
In object-oriented programming, the decorator pattern is a design pattern that allows behaviour to be added to an existing object dynamically.

The decorator pattern can be used to extend (decorate) the functionality of a certain object at run-time, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new decorator class that wraps the original class. This wrapping could be achieved by the following sequence of steps:

1. Subclass the original "Decorator" class into a "Component" class (see UML diagram);
2. In the Decorator class, add a Component pointer as a field;
3. Pass a Component to the Decorator constructor to initialize the Component pointer;
4. In the Decorator class, redirect all "Component" methods to the "Component" pointer; and
5. In the ConcreteDecorator class, override any Component method(s) whose behaviour needs to be modified.

This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time, and the change affects all instances of the original class; decorating can provide new behaviour at run-time for individual objects.



Decorator UML class diagram

Example:

```
//the Window insterface
interface Window{
        public void draw(); //draws the Window
        public String getDescription(); //returns a description of the Window
}
//implementation of a a simple Window without any scrollbars
class SimpleWindow implements Window{
        public void draw(){
                //draw window
        }
        public String getDescription(){
                return "simple window";
        }
}
```

The following classes contain the the decorators for all Window classes, including the decorator classes themselves.

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
   protected Window decoratedWindow; // the Window being decorated

   public WindowDecorator (Window decoratedWindow) {
      this.decoratedWindow = decoratedWindow;
   }
   public void draw() {
      decoratedWindow.draw();
   }
}

// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
   public VerticalScrollBarDecorator (Window decoratedWindow) {
      super(decoratedWindow);
   }

   public void draw() {
      decoratedWindow.draw();
      drawVerticalScrollBar();
   }

   private void drawVerticalScrollBar() {
      // draw the vertical scrollbar
   }

   public String getDescription() {
      return decoratedWindow.getDescription() + ", including vertical scrollbars";
   }
}

// the second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
   public HorizontalScrollBarDecorator (Window decoratedWindow) {
      super(decoratedWindow);
   }

   public void draw() {
      decoratedWindow.draw();
      drawHorizontalScrollBar();
   }
```

```
    private void drawHorizontalScrollBar() {
        // draw the horizontal scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including horizontal scrollbars";
    }
}
```

Here is a test program that creates a Window instance which is fully decorated (i.e., with vertical and horizontal scrollbars), and prints its description:

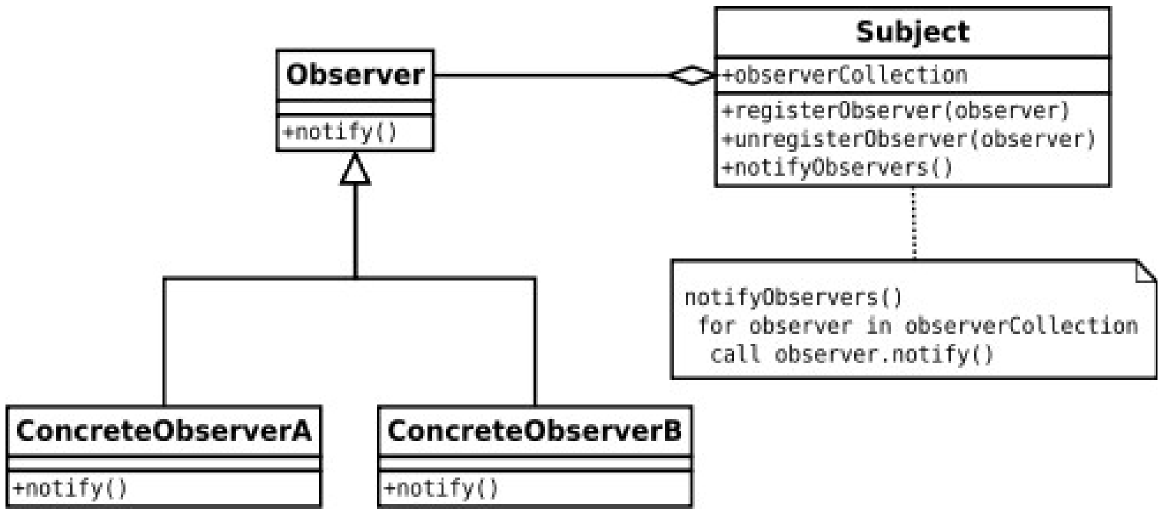
```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

3. Behavioural Pattern

3.1. Observer Pattern

The essence of the Observer Pattern is to "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
The observer pattern (aka. Dependents, publish/subscribe) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. Observer is also a key part in the familiar MVC architectural pattern. In fact the observer pattern was first implemented in Smalltalk's MVC based user interface framework.



UML diagram of Observer pattern

Example

Below is an example that takes keyboard input and treats each input line as an event. The example is built upon the library classes java.util.Observer and java.util.Observable. When a string is supplied from System.in, the method notifyObservers is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods - in our example, ResponseHandler.update(...).
The file MyApp.java contains a main() method that might be used in order to run the code.

```
/* File Name : EventSource.java */
package obs;

import java.util.Observable;          //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable {
   public void run() {
      try {
         final InputStreamReader isr = new InputStreamReader( System.in );
         final BufferedReader br = new BufferedReader( isr );
         while( true ) {
            String response = br.readLine();
            setChanged();
            notifyObservers( response );
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
/* File Name: ResponseHandler.java */

package obs;

import java.util.Observable;
import java.util.Observer;  /* this is Event Handler */

public class ResponseHandler implements Observer {
    private String resp;
    public void update (Observable obj, Object arg) {
        if (arg instanceof String) {
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}
/* Filename : MyApp.java */
/* This is the main program */

package obs;

public class MyApp {
    public static void main(String args[]) {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver( respHandler );

        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
}
```

3.2. Template Method Pattern

In software engineering, the template method pattern is a design pattern. It is a behavioural pattern, and is unrelated to C++ templates.
A template method defines the program skeleton of an algorithm. One or more of the algorithm steps can be overridden by subclasses to allow differing behaviours while ensuring that the overarching algorithm is still followed.
In object-oriented programming, first a class is created that provides the basic steps of an algorithm design. These steps are implemented using abstract methods. Later on, subclasses change the abstract methods to implement real actions. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.
The template method is used to:

1. let subclasses implement (through method overriding) behaviour that can vary
2. avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
3. control at what point(s) subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

The control structure (inversion of control) that is the result of the application of a template pattern is often referred to as the Hollywood Principle: "Don't call us, we'll call you." Using this principle, the template method in a parent class controls the overall process by calling subclass methods as required. This is shown in the following Java example:

Example:

```java
/**
* An abstract class that is common to several games in which players play against the others, but only
* one is playing at a given time.
 */
abstract class Game {
   protected int playersCount;
   abstract void initializeGame();
   abstract void makePlay(int player);
   abstract boolean endOfGame();
   abstract void printWinner();
   /* A template method : */
   public final void playOneGame(int playersCount) {
      this.playersCount = playersCount;
      initializeGame();
      int j = 0;
      while (!endOfGame()) {
         makePlay(j);
         j = (j + 1) % playersCount;
      }
      printWinner();
   }
}
```

```
//Now we can extend this class in order
//to implement actual games:
class Monopoly extends Game {
   /* Implementation of necessary concrete methods */
   void initializeGame() {
      // Initialize players
      // Initialize money
   }
   void makePlay(int player) {
      // Process one turn of player
   }
   boolean endOfGame() {
      // Return true if game is over
      // according to Monopoly rules
   }
   void printWinner() {
      // Display who won
   }
   /* Specific declarations for the Monopoly game. */

   // ...
}

class Chess extends Game {

   /* Implementation of necessary concrete methods */
   void initializeGame() {
      // Initialize players
      // Put the pieces on the board
   }
   void makePlay(int player) {
      // Process a turn for the player
   }
   boolean endOfGame() {
      // Return true if in Checkmate or
      // Stalemate has been reached
   }
   void printWinner() {
      // Display the winning player
   }
   /* Specific declarations for the chess game. */
   // ...
}
```