BIRLA INSTITIUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB-10 [Design Pattern]

DATE: 20/11/2017 **TIME: 02 Hours**

1. **Factory Pattern**
2. **Adapter Pattern**
3. **Composite Pattern**
4. **Decorator Pattern**

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

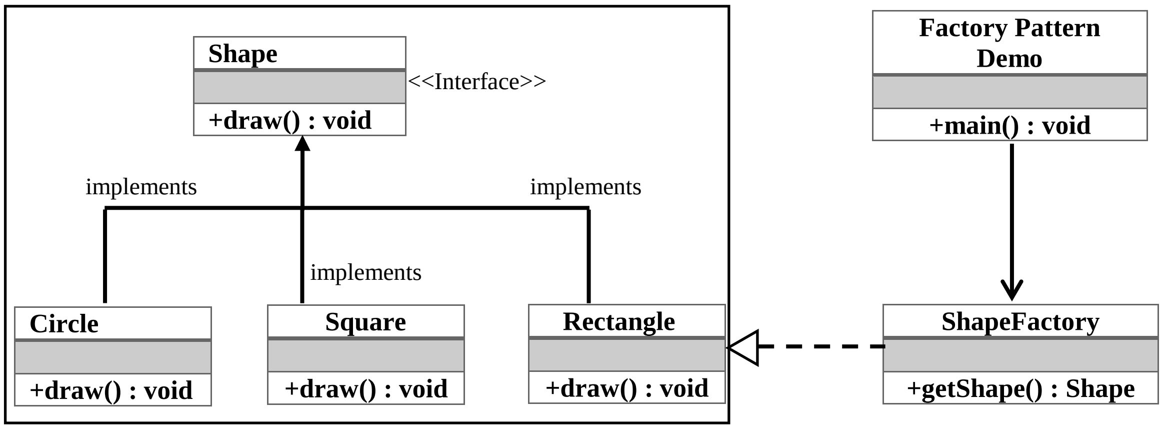
Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

1. Factory Pattern:

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation



We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

Step 1

Create an interface.

Shape.java

```
public interface Shape {
void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {
 @Override
 public void draw() {
 System.out.println("Inside Rectangle::draw() method.");
 }
}
```

Square.java

```
public class Square implements Shape {
 @Override
 public void draw() {
 System.out.println("Inside Square::draw() method.");
 }
}
```

Circle.java

```
public class Circle implements Shape {
 @Override
 public void draw() {
 System.out.println("Inside Circle::draw() method.");
 }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {
 //use getShape method to get object of type shape
 public Shape getShape(String shapeType){
 if(shapeType == null){
         return null;
 }
 if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
 } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
} else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
}
        return null;
}
}
```

Step 4
Use the Factory to get object of concrete class by passing an information such as type.
FactoryPatternDemo.java

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      //call draw method of Circle
      shape1.draw();
      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Rectangle
      shape2.draw();
      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");
      //call draw method of circle
      shape3.draw();
   }
}
```

2. Adapter Pattern:

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Adapter pattern is demonstrated via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

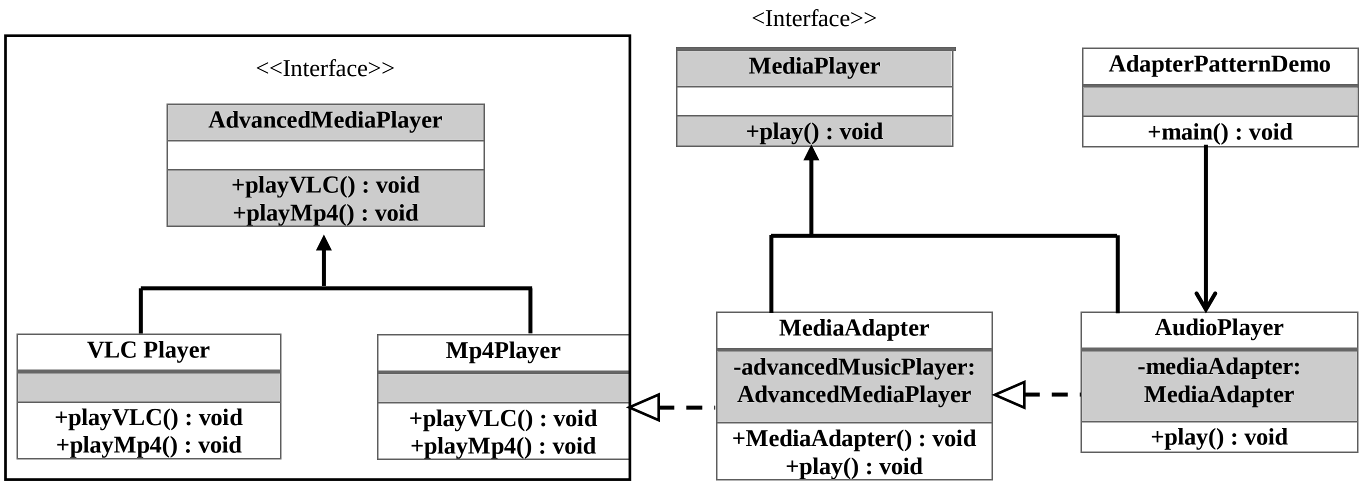
Implementation

We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

We want to make AudioPlayer to play other formats as well. To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.

AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.



Step 1
Create interfaces for Media Player and Advanced Media Player.
MediaPlayer.java

```
public interface MediaPlayer {
 public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Step 2
Create concrete classes implementing the AdvancedMediaPlayer interface.
VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }
   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

Step 3
Create adapter class implementing the MediaPlayer interface.
MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {
   AdvancedMediaPlayer advancedMusicPlayer;
   public MediaAdapter(String audioType){
         if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
         if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Step 4
Create concrete class implementing the MediaPlayer interface.
AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }

      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format not supported");
      }
   }
}
```

Step 5
Use the AudioPlayer to play different types of audio formats.
AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

3. Composite Pattern:

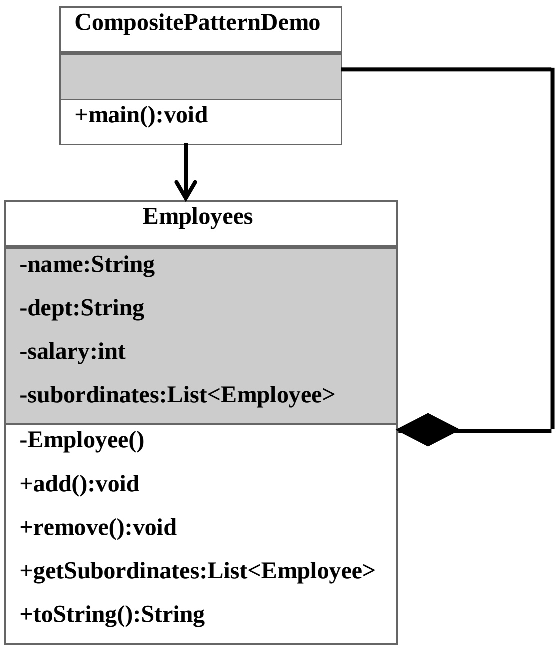
Composite pattern is used where we need to treat a group of objects in similar way as a single object. **Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy**. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

Composite pattern is demonstrated via following example in which we will show employees hierarchy of an organization.

Implementation

We have a class Employee which acts as composite pattern actor class. CompositePatternDemo, our demo class will use Employee class to add department level hierarchy and print all employees.



Step 1
Create Employee class having list of Employee objects.
Employee.java

```
import java.util.ArrayList;
import java.util.List;
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;
   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }
   public void add(Employee e) {
      subordinates.add(e);
   }
   public void remove(Employee e) {
      subordinates.remove(e);
   }
   public List<Employee> getSubordinates(){
     return subordinates;
   }
   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+"
]");
```

```
   }
}
```

Step 2
Use the Employee class to create and print employee hierarchy.
CompositePatternDemo.java

```
public class CompositePatternDemo {
   public static void main(String[] args) {
      Employee CEO = new Employee("John","CEO", 30000);
      Employee headSales = new Employee("Robert","Head Sales", 20000);
      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);
      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);
      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //print all employees of the organization
      System.out.println(CEO);

      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);

         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
}
```

4. Decorator Pattern:

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

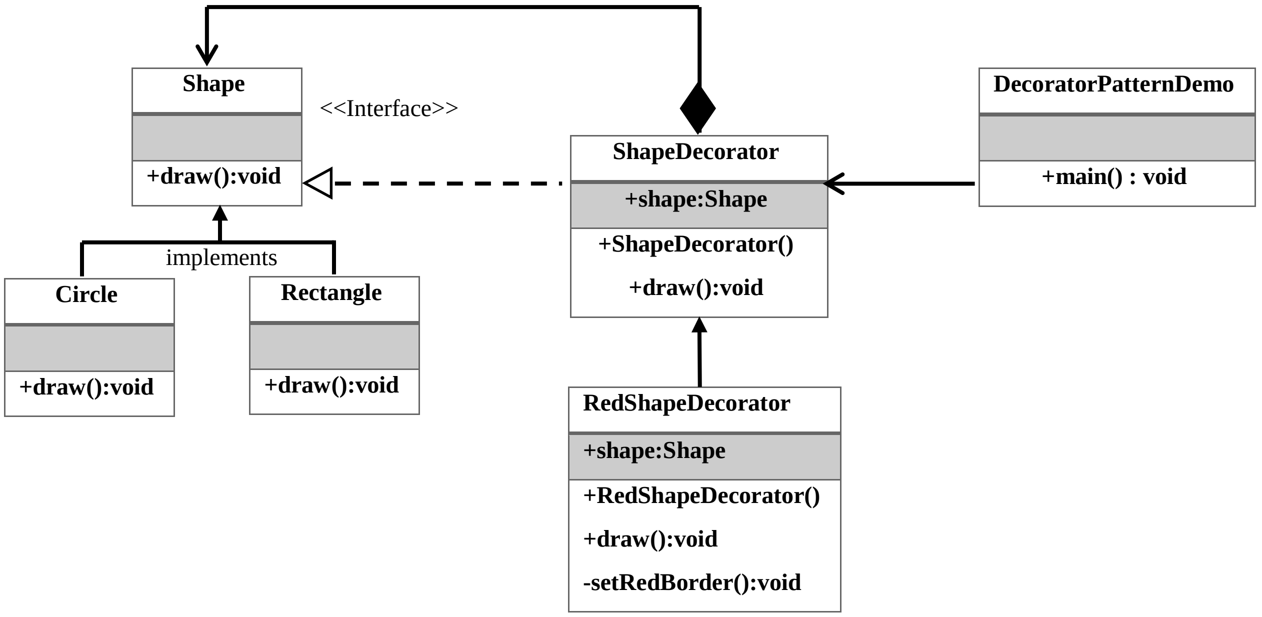
We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We will then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable.

RedShapeDecorator is concrete class implementing *ShapeDecorator*.

DecoratorPatternDemo, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.



Step 1
Create an interface.
Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2
Create concrete classes implementing the same interface.
Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

Step 3
Create abstract decorator class implementing the Shape interface.
ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

Step 4
Create concrete decorator class extending the ShapeDecorator class.
RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

Step 5
Use the RedShapeDecorator to decorate Shape objects.
DecoratorPatternDemo.java

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

Case Study

1.

Let’s say, we need Currency objects based on country. Now you define an interface called Currency, and specify your functionality in the form of method. Now implement this interface for different countries, and each method implementation will be specific to the country.

Now how can we create objects? Create a factory class and provide a static method to get required object. This method should take input, here input can be country name. Based on the country value, create specific country object, and return it.

Here is the Currency Interface

```
public interface Currency {

    public String getCurrency();

    public String getSymbol();
}
```

Implementation class for Indian Currency

```
public class India implements Currency{

    @Override
    public String getCurrency() {

        return "Rupee";
    }

    @Override
    public String getSymbol() {

        return "Rs";
    }

    public static void main(String a[]){

        India in = new India();
        System.out.println(in.getSymbol());
    }
}
```

Implementation class for USA Currency

```
public class USA implements Currency{

    @Override
    public String getCurrency() {

        return "Dollar";
    }

    @Override
    public String getSymbol() {

        return "$";
    }
}
```

Factory class to create objects

```
public class CurrencyFactory {

    public static Currency getCurrencyByCountry(String cnty) throws Exception{
//Complete this method.
}
public static void main(String a[]){
        //call above method from here.
}
```

2.

Implement the following Scenario

Design Pattern: Decorator

Let's say, you want to book a flight from Delhi to Toronto. There are 4 types of seats available Economy class, Premium Economy class, Business class and First class each having an estimated price of $2500, $3500, $4500 and $5500 respectively.

The fares given above are just the base fares without any extra amenities. There are few available amenities like WiFi, Live TV and Wine which will be provided only if we opt for them. They cost $10, $50 and $30 respectively.

Once, you opt for any of the available amenities the cost will be added correspondingly and the final cost will be shown which is to be paid.