In this document, we will discuss about Structural Design Patterns.

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.The structural design patterns simplifies the structure by identifying the relationships.These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

The sub-categories of Structural Design Patterns are as follows:

- Adapter Pattern.
- Bridge Pattern.
- Composite Pattern.
- Decorator Pattern.
- Facade Pattern.
- Flyweight Pattern.
- Proxy Pattern.

ADAPTER PATTERN

- Adapter pattern works as a bridge between two incompatible 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.
- An Adapter Pattern says that just "converts the interface of a class into another interface that a client wants".
- The Adapter Pattern is also known as Wrapper.

EXAMPLE

We are demonstrating use of Adapter pattern 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.

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.

```
public interface MediaPlayer { //MediaPlayer.java

  public void play(String audioType, String fileName);

}

public interface AdvancedMediaPlayer {  // AdvancedMediaPlayer.java

  public void playVlc(String fileName);
  public void playMp4(String fileName);

}
```

Step 2 : Create concrete classes implementing the AdvancedMediaPlayer interface.

```
public class VlcPlayer implements AdvancedMediaPlayer{    // VlcPlayer.java

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

  @Override
  public void playMp4(String fileName) {
    //do nothing
  }
}

public class Mp4Player implements AdvancedMediaPlayer{   // Mp4Player.java

  @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.

```
public class MediaAdapter implements MediaPlayer {   //MediaAdaper.java

  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.

```
public class AudioPlayer implements MediaPlayer {    //AudioPlayer.java
  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.

```
public class AdapterPatternDemo {   // AdapterPatternDemo.java
  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");
  }
}
```

OUTPUT :

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

BRIDGE DESIGN PATTERN

- There are 2 parts in Bridge design pattern : 1)Abstraction. 2)Implementation.
- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently.
- A Bridge Pattern says that just decouple the functional abstraction from the implementation so that the two can vary independently.
- The Bridge Pattern is also known as Handle or Body.

EXAMPLE:

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes. We have a DrawAPI interface which is acting as a bridge implementer and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.

Step 1 : Create bridge implementer interface.

```
public interface DrawAPI {  // DrawAPI.java
  public void drawCircle(int radius, int x, int y);
}
```

Step 2 : Create concrete bridge implementer classes implementing the *DrawAPI* interface.

```
public class RedCircle implements DrawAPI {   // RedCircle.java

  @Override
  public void drawCircle(int radius, int x, int y) {
    System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");
  }
}

public class GreenCircle implements DrawAPI {   // GreenCircle.java

  @Override
  public void drawCircle(int radius, int x, int y) {
    System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");
  }
}
```

Step 3 : Create an abstract class Shape using the DrawAPI interface.

```
public abstract class Shape {     // Shape.java
  protected DrawAPI drawAPI;

  protected Shape(DrawAPI drawAPI){
    this.drawAPI = drawAPI;
  }
  public abstract void draw();
}
```

Step 4 : Create concrete class implementing the Shape interface.

```
public class Circle extends Shape {   //Circle.java
  private int x, y, radius;

  public Circle(int x, int y, int radius, DrawAPI drawAPI) {

    super(drawAPI);
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  public void draw() {

    drawAPI.drawCircle(radius,x,y);
  }
}
```

Step 5 : Use the Shape and DrawAPI classes to draw different colored circles.

```
public class BridgePatternDemo {   // BridgePatternDemo.java
  public static void main(String[] args) {
    Shape redCircle = new Circle(100,100, 10, new RedCircle());
    Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

    redCircle.draw();
    greenCircle.draw();
  }
}
```

OUTPUT :

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[  color: green, radius: 10, x: 100, 100]
```

COMPOSITE DESIGN 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
- A Composite Pattern says that just allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects.
- This Pattern is used when the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.
- The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies.
- It allows you to have a tree structure and ask each node in the tree structure to perform a task.
- Composite design pattern treats each node in two ways:
  1) **Composite** – Composite means it can have other objects below it.
  2) **leaf** – leaf means it has no objects below it.

EXAMPLE

We are demonstrating use of composite pattern via following example in which we will show employees hierarchy of an organization.

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.

```
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.

```
public class CompositePatternDemo {   // CompositePatternDemo.java
  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);
      }
    }
  }
}
```

OUTPUT

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000  ]
```

DECORATOR PATTERN

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- A Decorator Pattern says that just **attach a flexible additional responsibilities to an object dynamically.**
- Decorator Pattern is used when we want to transparently and dynamically add responsibilities to objects without affecting other objects.
- The Decorator Pattern is also known as Wrapper.

EXAMPLE :

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. 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.

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

Step 2 : Create concrete classes implementing the same interface.

```
public class Rectangle implements Shape { //Rectangle.java

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

public class Circle implements Shape { //Circle.java

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

Step 3 : Create abstract decorator class implementing the *Shape* interface.

```
public abstract class ShapeDecorator implements Shape { //ShapeDecorator.java
  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.

```
public class RedShapeDecorator extends ShapeDecorator { // RedShapeDecorator.java

  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.

```
public class DecoratorPatternDemo {  // DecoratorPatternDemo.java
  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();  }  }
```

OUTPUT :

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

FAÇADE PATTERN

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.
- A Facade Pattern says that ***just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client.***
- Practically, every Abstract Factory is a type of Facade.

EXAMPLE

We are going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. FacadePatternDemo, our demo class, will use ShapeMaker class to show the results.

Step 1 : Create an interface.

```
public interface Shape {   //Shape.java
  void draw();
}
```

Step 2 : Create concrete classes implementing the same interface.

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

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

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

Step 3 : Create a facade class.

```
public class ShapeMaker {   // ShapeMaker.java
  private Shape circle;
  private Shape rectangle;
  private Shape square;

  public ShapeMaker() {
    circle = new Circle();
    rectangle = new Rectangle();
    square = new Square();
  }

  public void drawCircle(){
    circle.draw();
  }
  public void drawRectangle(){
    rectangle.draw();
  }
  public void drawSquare(){
    square.draw();
  }
}
```

Step 4 : Use the facade to draw various types of shapes.

```
public class FacadePatternDemo {   // FacadePatternDemo.java
  public static void main(String[] args) {
    ShapeMaker shapeMaker = new ShapeMaker();

    shapeMaker.drawCircle();
    shapeMaker.drawRectangle();
    shapeMaker.drawSquare();
  }
}
```

OUTPUT

```
Circle::draw()
Rectangle::draw()
Square::draw()
```

FLYWEIGHT PATTERN

- Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.
- A Flyweight Pattern says that just ***to reuse already existing similar kind of objects by storing them and create new object when no matching object is found***.

EXAMPLE

We will demonstrate this pattern by drawing 20 circles of different locations but we will create only 3 objects. Only 5 colors are available so color property is used to check already existing Circle objects.

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

ShapeFactory has a HashMap of Circle having key as color of the Circle object. Whenever a request comes to create a circle of particular color to ShapeFactory, it checks the circle object in its HashMap, if object of Circle found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

FlyWeightPatternDemo, our demo class, will use ShapeFactory to get a Shape object. It will pass information (red / green / blue/ black / white) to ShapeFactory to get the circle of desired color it needs.

Step 1: Create an interface.

```
public interface Shape {    // Shape.java
  void draw();
}
```

Step 2 : Create concrete class implementing the same interface.

```
public class Circle implements Shape {   // Circle.java
  private String color;
  private int x;
  private int y;
  private int radius;

  public Circle(String color){
    this.color = color;
  }
```

```
  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public void setRadius(int radius) {
    this.radius = radius;
  }

  @Override
  public void draw() {
    System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
  }
}
```

Step 3 : Create a factory to generate object of concrete class based on given information.

```
public class ShapeFactory {    // ShapeFactory.java

  private static final HashMap circleMap = new HashMap();
  public static Shape getCircle(String color) {
    Circle circle = (Circle)circleMap.get(color);

    if(circle == null) {
      circle = new Circle(color);
      circleMap.put(color, circle);
      System.out.println("Creating circle of color : " + color);
    }
    return circle;
  }
}
```

Step 4 : Use the factory to get object of concrete class by passing an information such as color.

```
public class FlyweightPatternDemo {     // FlyweightPatternDemo.java
  private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };
  public static void main(String[] args) {

    for(int i=0; i < 20; ++i) {
      Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
      circle.setX(getRandomX());
      circle.setY(getRandomY());
      circle.setRadius(100);
      circle.draw();
```

```
    }
  }

private static String getRandomColor() {
    return colors[(int)(Math.random()*colors.length)];
  }
  private static int getRandomX() {
    return (int)(Math.random()*100 );
  }
  private static int getRandomY() {
    return (int)(Math.random()*100);
  }
}
```

OUTPUT :

```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

PROXY PATTERN

- In proxy pattern, a class represents functionality of another class.
- In this pattern, we create object having original object to interface its functionality to outer world.
- A Proxy Pattern is that provides the control for accessing the original object.
- So, we can perform many operations like hiding the information of original object, on demand loading etc.
- Proxy pattern is also known as Surrogate or Placeholder.

This design pattern in used in 4 ways asw mentioned below.

- It can be used in **Virtual Proxy** scenario
  Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.

- It can be used in **Protective Proxy** scenario.
  It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.

- It can be used in **Remote Proxy** scenario.
  A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.

- It can be used in **Smart Proxy** scenario.
  A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

EXAMPLE

We are going to create an Image interface and concrete classes implementing the Image interface. ProxyImage is a a proxy class to reduce memory footprint of RealImage object loading.

ProxyPatternDemo, our demo class, will use ProxyImage to get an Image object to load and display as it needs.

Step 1 : Create an interface.

```
public interface Image  //Image.java
{
  void display();
}
```

Step 2 : Create concrete classes implementing the same interface.

```
public class RealImage implements Image {   //RealImage.java

  private String fileName;

  public RealImage(String fileName){
    this.fileName = fileName;
    loadFromDisk(fileName);
  }

  @Override
  public void display() {
    System.out.println("Displaying " + fileName);
  }

  private void loadFromDisk(String fileName){
    System.out.println("Loading " + fileName);
  }
}

public class ProxyImage implements Image{    //ProxyImage.java

  private RealImage realImage;
  private String fileName;

  public ProxyImage(String fileName){
    this.fileName = fileName;
  }

  @Override
  public void display() {
    if(realImage == null){
      realImage = new RealImage(fileName);
    }
```

```
    realImage.display();
  }
}
```

Step 3 : Use the *ProxyImage* to get object of *RealImage* class when required.

```
public class ProxyPatternDemo {    // ProxyPatternDemo.java

  public static void main(String[] args) {
    Image image = new ProxyImage("test_10mb.jpg");

    //image will be loaded from disk
    image.display();
    System.out.println("");

    //image will not be loaded from disk
    image.display();
  }
}
```

OUTPUT

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

----------End of Document----------