**Week 1 – Day 1**

**Assignment 2 :-**

**Q. Name five design patterns and the advantages of using that patterns.**

Five common design patterns with their benefits are listed as follows:-

1. The **singleton pattern** is one of the simplest **design patterns**: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance.
2. The **factory pattern** uses factory methods to deal with the problem of creating objects. The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern.
3. The **Strategy design pattern** is based on collecting the basic idea of complete application development at once. This design pattern allows you to identify the objects you need in development and interface you will require in the entire application development.
4. The **Observer design pattern** is simply based on its name. It gives you liberty to describe an observed and observer relationship in the application. The rules and conventions for implementing an observed and observer class are written explicitly in this design pattern.
5. The **Cascading Bridge design pattern** is a popular design pattern that describes seamless ways to implement classes. It helps developers while dealing with the API variability.

Singleton Pattern

Singleton pattern is one of the simplest 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.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

## **Implementation**

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



## **Step 1**

Create a Singleton Class.

*SingleObject.java*

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

## **Step 2**

Get the only object from the singleton class.

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message

object.showMessage();

}

}

## **Step 3**

Verify the output.

Hello World!

# 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 square

shape3.draw();

}

}

## **Step 5**

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

# Strategy Pattern

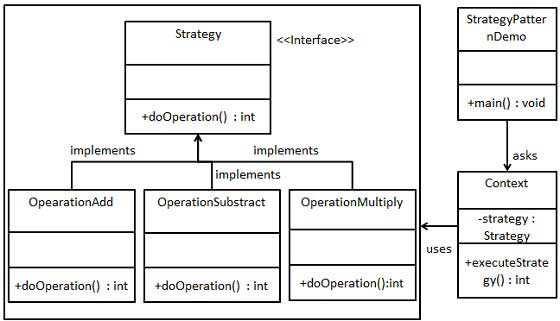
In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

## **Implementation**

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.

*StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



## **Step 1**

Create an interface.

*Strategy.java*

public interface Strategy {

public int doOperation(int num1, int num2);

}

## **Step 2**

Create concrete classes implementing the same interface.

*OperationAdd.java*

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

*OperationSubstract.java*

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

*OperationMultiply.java*

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

## **Step 3**

Create *Context* Class.

*Context.java*

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

## **Step 4**

Use the *Context* to see change in behaviour when it changes its *Strategy*.

*StrategyPatternDemo.java*

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

## **Step 5**

Verify the output.

10 + 5 = 15

10 - 5 = 5

10 \* 5 = 50

# Observer Pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

## **Implementation**

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.

*ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



## **Step 1**

Create Subject class.

*Subject.java*

import java.util.ArrayList;

import java.util.List;

public class Subject {

private List<Observer> observers = new ArrayList<Observer>();

private int state;

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

notifyAllObservers();

}

public void attach(Observer observer){

observers.add(observer);

}

public void notifyAllObservers(){

for (Observer observer : observers) {

observer.update();

}

}

}

## **Step 2**

Create Observer class.

*Observer.java*

public abstract class Observer {

protected Subject subject;

public abstract void update();

}

## **Step 3**

Create concrete observer classes

*BinaryObserver.java*

public class BinaryObserver extends Observer{

public BinaryObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );

}

}

*OctalObserver.java*

public class OctalObserver extends Observer{

public OctalObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );

}

}

*HexaObserver.java*

public class HexaObserver extends Observer{

public HexaObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );

}

}

## **Step 4**

Use *Subject* and concrete observer objects.

*ObserverPatternDemo.java*

public class ObserverPatternDemo {

public static void main(String[] args) {

Subject subject = new Subject();

new HexaObserver(subject);

new OctalObserver(subject);

new BinaryObserver(subject);

System.out.println("First state change: 15");

subject.setState(15);

System.out.println("Second state change: 10");

subject.setState(10);

}

}

## **Step 5**

Verify the output.

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

Octal String: 12

Binary String: 1010

# Bridge Pattern

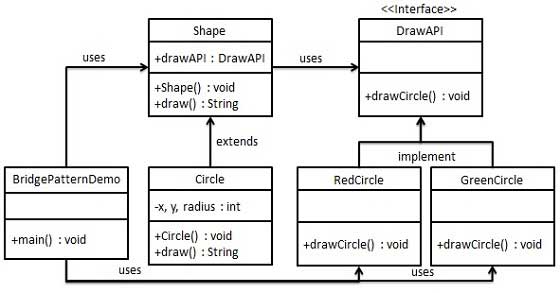
Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

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.

## **Implementation**

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.

*DrawAPI.java*

public interface DrawAPI {

public void drawCircle(int radius, int x, int y);

}

## **Step 2**

Create concrete bridge implementer classes implementing the *DrawAPI* interface.

*RedCircle.java*

public class RedCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

*GreenCircle.java*

public class GreenCircle implements DrawAPI {

@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.

*Shape.java*

public abstract class Shape {

protected DrawAPI drawAPI;

protected Shape(DrawAPI drawAPI){

this.drawAPI = drawAPI;

}

public abstract void draw();

}

## **Step 4**

Create concrete class implementing the *Shape* interface.

*Circle.java*

public class Circle extends Shape {

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.

*BridgePatternDemo.java*

public class BridgePatternDemo {

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();

}

}

## **Step 6**

Verify the output.

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

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