ASSIGNMENT 5

Name : Hrithvik Kondalkar
Roll : 002211001088

Problem No. 1: Write a java program to implement Factory pattern.

Theory:

• Defines an interface for creating objects but let sub-classes decide which of those instantiate. • Enables the creator to defer Product creation to a sub-class.

• Factory pattern is one of the most used design pattern 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.

Intent:

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As:

• Virtual Constructor.

Applicability:

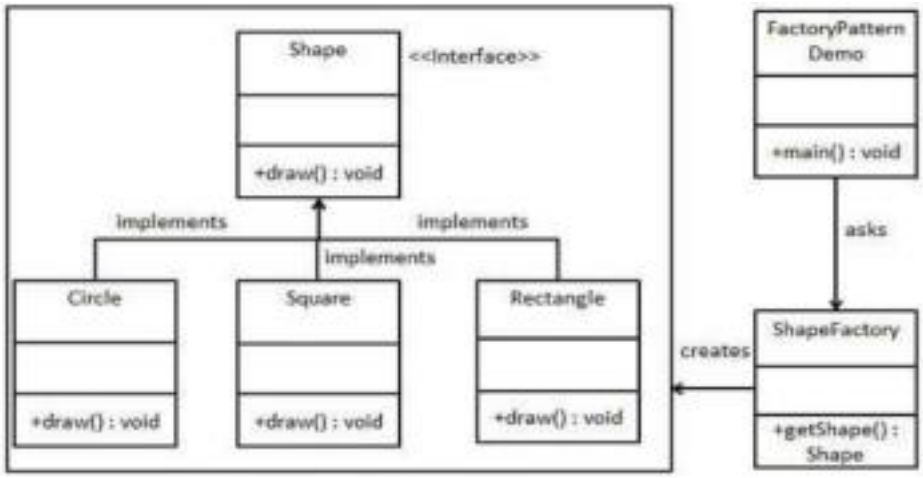
Use the Factory Method pattern when

· a class can't anticipate the class of objects it must create.

· a class wants its subclasses to specify the objects it creates.

· classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Class Diagram:



```java
interface Shape{

    void draw();
}

class Circle implements Shape{
    private int radius;
    Circle(int r){
        radius  = r;
    }
    public void draw(){
        System.out.println("circle");
    }
}

class Square implements Shape{
    private int side;
    Square(int s){
        side = s;
    }
    public void draw(){
        System.out.println("Square");
    }
}

class Rectangle implements Shape{
    private int length,breadth;
    Rectangle(int l,int b){
        length = l;
        breadth = b;
    }
    public void draw(){
        System.out.println("Rectangle");
    }
}

class ShapeFactory{
    Shape getShape(String type){
        if(type.equalsIgnoreCase("circle")){
            return new Circle(1);
        }
        else if(type.equalsIgnoreCase("square")){
            return new Square(1);
        }
        else if(type.equalsIgnoreCase("rectangle")){
            return new Rectangle(1,2);
        }
```

```
        else{
            System.out.println("invalid");
            return null;
        }
    }
}

class Main{
    public static void main(String[] args){
        ShapeFactory generator = new ShapeFactory();
        Shape shape1 = generator.getShape("circle");
        Shape shape2 = generator.getShape("rectangle");
        Shape shape3 = generator.getShape("Square");

        shape1.draw();
        shape2.draw();
        shape3.draw();
    }
}
```

```
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ javac 1.java
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ java Main
circle
Rectangle
Square
```

Problem No. 2: Write a java program to implement decorator pattern.

Theory:

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.

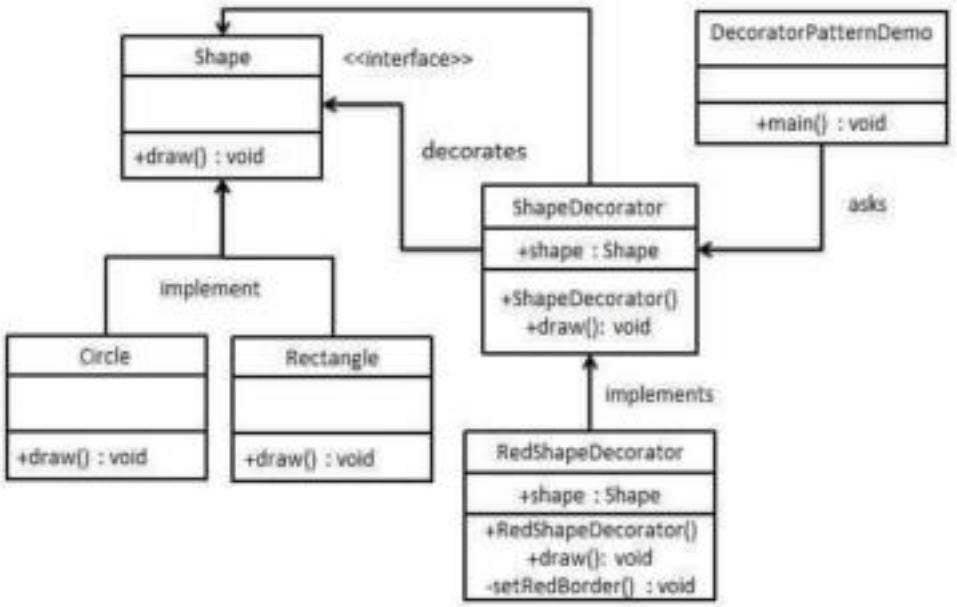
Intent :
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As
Wrapper

Applicability :
Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by sub classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub classing.

Class Diagram:



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

class Circle implements Shape{
    private int radius;
    Circle(int r){
        radius  = r;
    }
    public void draw(){
        System.out.println("circle");
    }
}

class Square implements Shape{
    private int side;
    Square(int s){
        side = s;
    }
    public void draw(){
        System.out.println("Square");
    }
}

abstract class ShapeDecorator implements Shape{
    Shape decoratingthis;
    ShapeDecorator(Shape originalShape){
        decoratingthis = originalShape;
    }
    public void draw(){
        decoratingthis.draw();
    }
    public void newFunctionality(){
        System.out.println("new functionality");
    }
}

class NewShape extends ShapeDecorator{
    NewShape(Shape olderShape){
        super(olderShape);
    }
    public void draw(){
        super.decoratingthis.draw();
        super.newFunctionality();
    }
}
```

```java
class Main{
    public static void main(String[] args){
        Shape oldshape = new Circle(1);
        Shape newshape = new NewShape(oldshape);

        oldshape.draw();
        System.out.println("--------------");
        newshape.draw();

    }
}
```

```
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ javac 2.java
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ java Main
circle
--------------
circle
new functionality
```

Problem No. 3: Write a java program to design mediator pattern.

Theory:

• Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

• Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

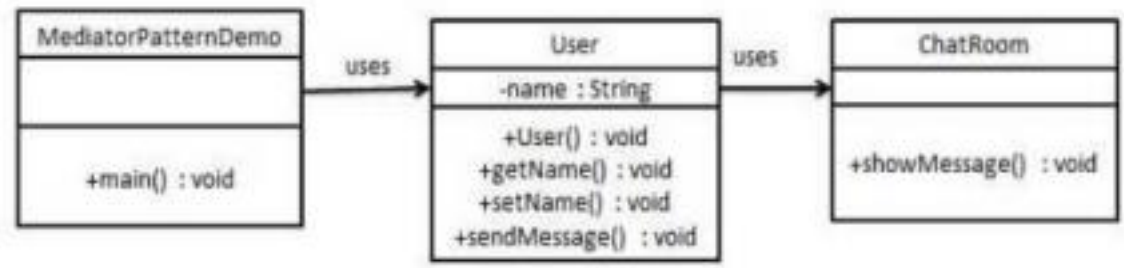
• Applicability

Use the Mediator pattern when

• a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.

• reusing an object is difficult because it refers to and communicates with many other objects.

Class Diagram:




```
import java.util.Date;

 class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString() + " [" + user.getName() + "]
: " + message);
   }
}

 class User {
   private String name;
   public User(String name){
      this.name  = name;
   }
   public String getName(){return name;}
```

```
    public void sendMessage(String message){
        ChatRoom.showMessage(this,message);
    }
}

 class Main {
    public static void main(String[] args) {
        User u1 = new User("user1");
        User u2 = new User("user2");

        u1.sendMessage("Hi!");
        u2.sendMessage("Hello!");
    }
}
```

```
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ javac 3.java
@Frozha ➜.../jadavpur-labs-IT/2ndyear/oos/assignment5 (main) $ java Main
Sun Apr 21 12:49:33 UTC 2024 [user1] : Hi!
Sun Apr 21 12:49:34 UTC 2024 [user2] : Hello!
```