NAME: ARIF RAHAMAN
ROLL:002211001071
SEC:A1

CO5: Assignments on Design Patterns

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.

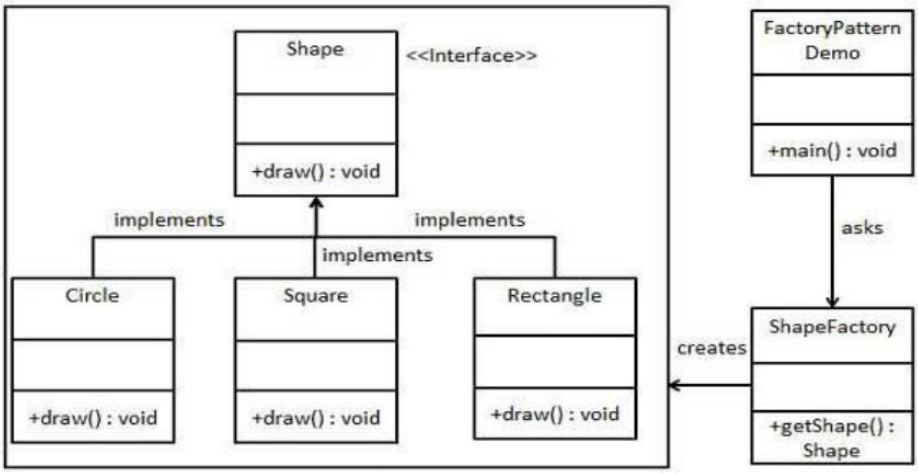
Intent:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

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:



Solution:

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


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


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

```
}

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

class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
}
```

```
public class FactoryPattern {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      Shape circle = shapeFactory.getShape("CIRCLE");
      if (circle != null) {
         circle.draw();
      }

      Shape square = shapeFactory.getShape("SQUARE");
      if (square != null) {
         square.draw();
      }

      Shape rectangle = shapeFactory.getShape("RECTANGLE");
      if (rectangle != null) {
         rectangle.draw();
      }
   }
}
```

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

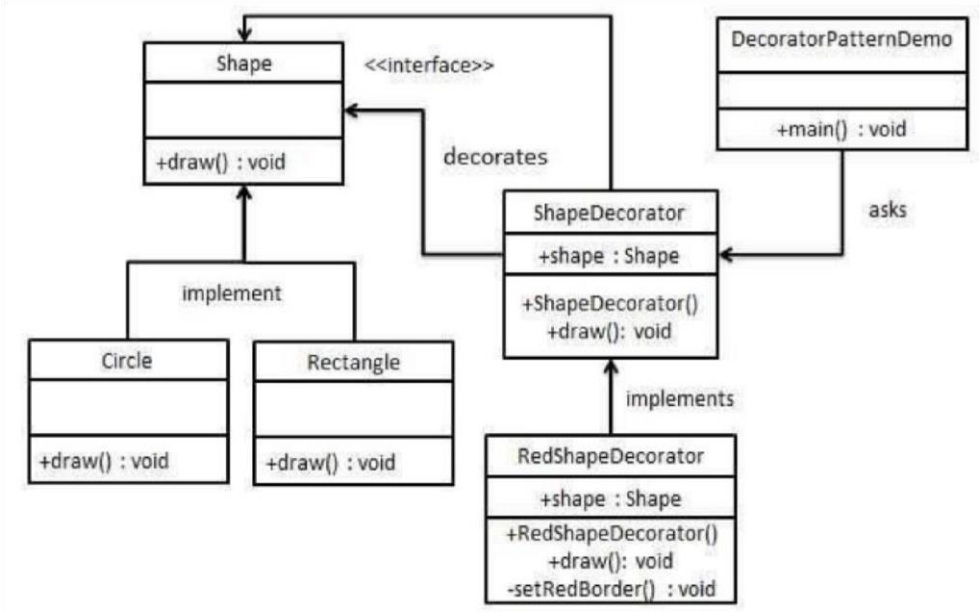
Wrapper

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:



Solution:

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

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

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


abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

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

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

class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

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

    private void addRedBorder(Shape decoratedShape) {
```

```
        System.out.println("Adding red border to the shape.");
    }
}

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("Normal Circle:");
        circle.draw();

        System.out.println("\nRed Circle with Red Border:");
        redCircle.draw();

        System.out.println("\nRed Rectangle with Red Border:");
        redRectangle.draw();
    }
}
```

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.

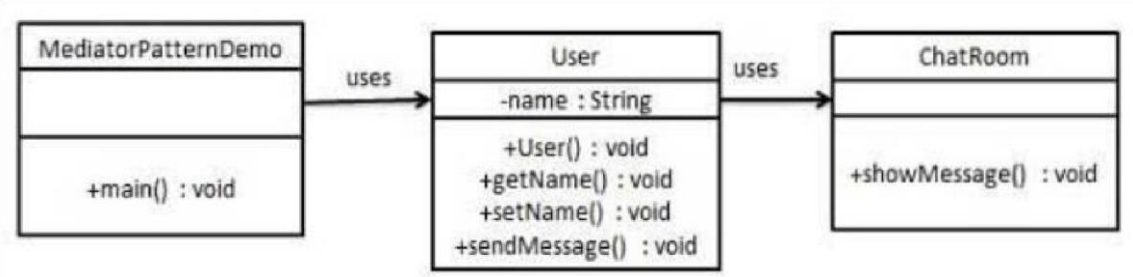
• **Applicability**

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:



Solution:

```
import java.util.ArrayList;
import java.util.List;

interface Mediator {
    void addUser(User user);
    void sendMessage(String message, User sender);
}

class ChatRoom implements Mediator {
    private List<User> users;

    public ChatRoom() {
        this.users = new ArrayList<>();
    }

    @Override
    public void addUser(User user) {
        users.add(user);
    }

    @Override
    public void sendMessage(String message, User sender) {
        for (User user : users) {
```

```
            if (user != sender) {
                user.receiveMessage(message);
            }
        }
    }
}

abstract class User {
    protected Mediator mediator;

    public User(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void sendMessage(String message);
    public abstract void receiveMessage(String message);
}

class BasicUser extends User {
    public BasicUser(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void sendMessage(String message) {
        System.out.println("Basic user sending message: " + message);
        mediator.sendMessage(message, this);
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println("Basic user received message: " + message);
    }
}

public class MediatorPatternDemo {
    public static void main(String[] args) {
        ChatRoom chatRoom = new ChatRoom();

        User user1 = new BasicUser(chatRoom);
        User user2 = new BasicUser(chatRoom);
        User user3 = new BasicUser(chatRoom);

        chatRoom.addUser(user1);
        chatRoom.addUser(user2);
        chatRoom.addUser(user3);

        user1.sendMessage("Hello, everyone!");
        user2.sendMessage("Hi, there!");
    }
```

}