**Exercise 1**

**3.**

**Use Case 1: Adapter Pattern**

**Problem:** In a payment processing system, we have a legacy payment gateway that uses a specific payment protocol, and we want to integrate it with a new payment system that uses a different protocol.

**Solution:** We can use the Adapter pattern to achieve this

// Legacy Payment Gateway

public interface LegacyPaymentGateway {

void processPayment(String paymentInfo);

}

public class LegacyPaymentGatewayImpl implements LegacyPaymentGateway {

public void processPayment(String paymentInfo) {

System.out.println("Processing payment using legacy protocol: " + paymentInfo);

}

}

// New Payment System

public interface NewPaymentSystem {

void processPayment(String paymentInfo);

}

public class NewPaymentSystemImpl implements NewPaymentSystem {

public void processPayment(String paymentInfo) {

System.out.println("Processing payment using new protocol: " + paymentInfo);

}

}

// Adapter class

public class PaymentAdapter implements NewPaymentSystem {

private LegacyPaymentGateway legacyGateway;

public PaymentAdapter(LegacyPaymentGateway legacyGateway) {

this.legacyGateway = legacyGateway;

}

@Override

public void processPayment(String paymentInfo) {

legacyGateway.processPayment(paymentInfo);

}

}

// Client code

public class PaymentProcessor {

public static void main(String[] args) {

LegacyPaymentGateway legacyGateway = new LegacyPaymentGatewayImpl();

NewPaymentSystem newSystem = new PaymentAdapter(legacyGateway);

newSystem.processPayment("123456"); // Output: Processing payment using legacy protocol: 123456

}

}

**Use Case 2: Bridge Pattern**

**Problem:** In a graphics rendering system, we want to render shapes using different algorithms (e.g., raster, vector) and different rendering engines (e.g., DirectX, OpenGL).

**Solution:** We can use the Bridge pattern to achieve this.

// Abstraction (Shape)

public abstract class Shape {

protected RenderingEngine engine;

public Shape(RenderingEngine engine) {

this.engine = engine;

}

public abstract void render();

}

// Refined Abstraction (Circle)

public class Circle extends Shape {

private int radius;

public Circle(int radius, RenderingEngine engine) {

super(engine);

this.radius = radius;

}

@Override

public void render() {

engine.renderCircle(radius);

}

}

// Implementor (RenderingEngine)

public interface RenderingEngine {

void renderCircle(int radius);

}

public class DirectXEngine implements RenderingEngine {

@Override

public void renderCircle(int radius) {

System.out.println("Rendering circle using DirectX: " + radius);

}

}

public class OpenGLEngine implements RenderingEngine {

@Override

public void renderCircle(int radius) {

System.out.println("Rendering circle using OpenGL: " + radius);

}

}

// Client code

public class GraphicsRenderer {

public static void main(String[] args) {

RenderingEngine directXEngine = new DirectXEngine();

Circle circle = new Circle(10, directXEngine);

circle.render(); // Output: Rendering circle using DirectX: 10

RenderingEngine openGLEngine = new OpenGLEngine();

circle = new Circle(10, openGLEngine);

circle.render(); // Output: Rendering circle using OpenGL: 10

}

}