**Week 1:Design Patterns**

**Exercise 3:Implementing Builder Pattern**

**Computer Class:**

public class Computer {

private String CPU;

private String RAM;

private String storage;

private boolean isGraphicsCardEnabled;

private boolean isBluetoothEnabled;

private Computer(ComputerBuilder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

}

public String getCPU() { return CPU; }

public String getRAM() { return RAM; }

public String getStorage() { return storage; }

public boolean isGraphicsCardEnabled() { return isGraphicsCardEnabled; }

public boolean isBluetoothEnabled() { return isBluetoothEnabled; }

}

**Builder Class:**

public static class ComputerBuilder {

private String CPU;

private String RAM;

private String storage;

private boolean isGraphicsCardEnabled = false;

private boolean isBluetoothEnabled = false;

public ComputerBuilder(String CPU, String RAM, String storage) {

this.CPU = CPU;

this.RAM = RAM;

this.storage = storage;

}

public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {

this.isGraphicsCardEnabled = isGraphicsCardEnabled;

return this;

}

public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Computer build() {

return new Computer(this);

}

}

**Test Builder Implementation:**

public class BuilderPatternTest {

public static void main(String[] args) {

Computer myComputer = new Computer.ComputerBuilder("Intel i7", "16GB", "1TB")

.setGraphicsCardEnabled(true)

.setBluetoothEnabled(true)

.build();

System.out.println("CPU: " + myComputer.getCPU());

System.out.println("RAM: " + myComputer.getRAM());

System.out.println("Storage: " + myComputer.getStorage());

System.out.println("Graphics Card Enabled: " + myComputer.isGraphicsCardEnabled());

System.out.println("Bluetooth Enabled: " + myComputer.isBluetoothEnabled());

}

}

**Exercise 4:Implement Adapter Pattern**

**Target Interface:**

public interface PaymentProcessor {

void processPayment(double amount);

}

Adaptee Classes:

Adaptee 1: PayPal

public class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through PayPal.");

}

}

Adaptee 2: Stripe

public class StripeGateway {

public void makePayment(double amount) {

System.out.println("Processing payment of $" + amount + " through Stripe.");

}

}

Adaptee 3: Square

public class SquareGateway {

public void performPayment(double amount) {

System.out.println("Processing payment of $" + amount + " through Square.");

}

}

**Adapter Classes:**

Adapter for PayPal

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.sendPayment(amount);

}

}

Adapter for Stripe

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.makePayment(amount);

}

}

Adapter for Square

public class SquareAdapter implements PaymentProcessor {

private SquareGateway squareGateway;

public SquareAdapter(SquareGateway squareGateway) {

this.squareGateway = squareGateway;

}

@Override

public void processPayment(double amount) {

squareGateway.performPayment(amount);

}

}

**Test Adapter Iplementation:**

public class AdapterPatternTest {

public static void main(String[] args) {

PayPalGateway payPal = new PayPalGateway();

StripeGateway stripe = new StripeGateway();

SquareGateway square = new SquareGateway();

PaymentProcessor payPalAdapter = new PayPalAdapter(payPal);

PaymentProcessor stripeAdapter = new StripeAdapter(stripe);

PaymentProcessor squareAdapter = new SquareAdapter(square);

payPalAdapter.processPayment(100.0);

stripeAdapter.processPayment(200.0);

squareAdapter.processPayment(300.0);

}

}

**Exercise 5:Implementing Decorator Pattern**

**Component Interface:**

public interface Notifier {

void send(String message);

}

**Concrete Component:**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

System.out.println("Sending Email: " + message);

}

}

**Decorator Class:**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

System.out.println("Sending SMS: " + message);

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlackMessage(message);

}

private void sendSlackMessage(String message) {

System.out.println("Sending Slack message: " + message);

}

}

**Testing Implementation:**

public class DecoratorPatternTest {

public static void main(String[] args) {

// Basic notifier

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

notifier.send("This is a test notification.");

}

}

**Exercise 6: Proxy Pattern**

**Subject Interface:**

public interface Image {

void display();

}

**Real Image Class:**

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadImageFromDisk();

}

private void loadImageFromDisk() {

System.out.println("Loading image from disk: " + filename);

}

@Override

public void display() {

System.out.println("Displaying image: " + filename);

}

}

**Proxy Class:**

public class ProxyImage implements Image {

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();

}

}

**Testing Implementation:**

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

image1.display();

System.out.println("");

image1.display();

System.out.println("");

image2.display();

}

}

**Exercise 7:Observe Pattern**

**Subject Interface:**

import java.util.List;

public interface Stock {

void registerObserver(Observer observer);

void deregisterObserver(Observer observer);

void notifyObservers();

}

**Stock Market Class:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

this.observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void deregisterObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockPrice);

}

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

}

Observer Interface:

public interface Observer {

void update(double stockPrice);

}

**Implement Concrete Observers:**

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println(name + " received stock price update: " + stockPrice);

}

}

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println(name + " received stock price update: " + stockPrice);

}

}

**Testing Implementation:**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("Mobile App 1");

Observer webApp = new WebApp("Web App 1");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockPrice(100.50);

stockMarket.setStockPrice(102.75);

stockMarket.deregisterObserver(mobileApp);

stockMarket.setStockPrice(98.60);

}

}

**Exercise 8: Strategy Pattern**

**Strategy Interface:**

public interface PaymentStrategy {

void pay(double amount);

}

**Concrete Strategies:**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

System.out.println("Paying $" + amount + " using Credit Card.");

// Logic to process credit card payment

}

}

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.out.println("Paying $" + amount + " using PayPal.");

}

}

**Context Class:**

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(double amount) {

if (paymentStrategy != null) {

paymentStrategy.pay(amount);

} else {

System.out.println("Payment strategy not set.");

}

}

}

**Testing Implementation:**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

paymentContext.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

paymentContext.pay(250.00);

paymentContext.setPaymentStrategy(new PayPalPayment("john.doe@example.com"));

paymentContext.pay(75.00);

}

}

**Exercise 10 : MVC Pattern**

**Model Class:**

public class Student {

private String id;

private String name;

private String grade;

public Student(String id, String name, String grade) {

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

**View Class:**

public class StudentView {

public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

System.out.println("Student:");

System.out.println("Name: " + studentName);

System.out.println("ID: " + studentId);

System.out.println("Grade: " + studentGrade);

}

}

**Controller Class:**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

**Test Implementation:**

public class MVCPatternExample {

public static void main(String[] args) {

Student student = new Student("1", "John Doe", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Jane Doe");

controller.setStudentGrade("A+");

controller.updateView();

}

}