**WEEK-1**

**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**Java Project (Package)** :SingletonPatternExample

**Logger.java:**

public class Logger {

private static Logger singleobj;

private Logger(){

    System.out.println("Singleton Logger class's instance created");

}

public static Logger getinstance(){

    if (singleobj == null){

        singleobj= new Logger();

    }

    return singleobj ;

// ADD THIS | = new Logger() ;| IT CREATE NEW INSTANCES SO , IT WILL NOT FOLLOW SINGLETON DESIGN PATTERN

}

public void showmwsg(){

    System.out.println("hello from Logger");

}

}

**Test.java:**

public class Test {

    public static void main (String[] args) {

        Logger firstobj= Logger.getinstance();

        Logger secondobj= Logger.getinstance();

        firstobj.showmwsg();

        secondobj.showmwsg();

        if (firstobj==secondobj){

            System.out.println("FOLLOWS Singleton");

        }

        else{

            System.out.println("NOT FOLLOWS Singleton");

 }

}

}

**Output:**

**Output 1 Which Follows Singleton: -**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Output 2 Which Not Follows Singleton: -**

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 2: Implementing the Factory Method Pattern**

**Code:**

**Java Project (Package)** :FactoryPatternExample

**Document.java:**

public interface Document {

    void createAndOpen();

}

**Pdf.java:**

public class Pdf implements Document{

    @Override

    public void createAndOpen (){

        System.out.println("Created & Opening a Pdf Document" );

    }

}

**Word.java:**

public class Word implements Document {

    @Override

    public void createAndOpen (){

        System.out.println("Created & Opening a Word Document ");

    }

}

**Execl.java:**

public class Execl implements Document {

    @Override

    public void createAndOpen (){

        System.out.println("Created & Opening a Execl Document  ");

    }

}

**DocumentFactory.java:**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**PdfFactory.java:**

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new Pdf();

    }

}

**WordFactory.java:**

public class WordFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new Word();

    }

}

**ExeclFactory.java:**

public class ExeclFactory extends DocumentFactory{

    @Override

    public Document createDocument() {

        return new Execl();

    }

}

**Test.java:**

public class Test {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document WordDocument = wordFactory.createDocument();

        WordDocument.createAndOpen();

        DocumentFactory pdfFactory = new PdfFactory();

        Document PdfDocument = pdfFactory.createDocument();

        PdfDocument.createAndOpen();

        DocumentFactory excelFactory = new ExeclFactory();

        Document ExeclDocument = excelFactory.createDocument();

        ExeclDocument.createAndOpen();

    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**Java Project (Package)** :BuilderPatternExample

**Computer.java:**

public class Computer {

    private String CPU;

    private String RAM;

    private String storage;

    private String graphicsCard;

    private boolean hasBluetooth;

    private boolean hasWiFi;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.hasBluetooth = builder.hasBluetooth;

        this.hasWiFi = builder.hasWiFi;

    }

   public String getConfiguration() {

    return "CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage;

}

public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        private boolean hasBluetooth;

        private boolean hasWiFi;

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

            this.CPU = CPU;

            this.RAM = RAM;

            this.storage = storage;

        }

public Builder setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

public Builder enableBluetooth() {

            this.hasBluetooth = true;

            return this;

        }

public Builder enableWiFi() {

            this.hasWiFi = true;

            return this;

        }

public Computer build() {

            return new Computer(this);

        }

    }

}

**Test.java:**

public class Test {

    public static void main(String[] args) {

        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB", "1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .enableBluetooth()

                .enableWiFi()

                .build();

        Computer officeComputer = new Computer.Builder("AMDA Ryzen 3", "8GB", "500GB HDD")

                .build();

        System.out.println();

        System.out.println("Gaming PC:\n" + gamingComputer.getConfiguration());

        System.out.println();

        System.out.println("Office PC:\n" + officeComputer.getConfiguration());

    }

}

**Test.java:**

public class Test {

    public static void main(String[] args) {

        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB", "1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .enableBluetooth()

                .enableWiFi()

                .build();

        Computer officeComputer = new Computer.Builder("AMDA Ryzen 3", "8GB", "500GB HDD")

                .build();

        System.out.println();

        System.out.println("Gaming PC:\n" + gamingComputer.getConfiguration());

        System.out.println();

        System.out.println("Office PC:\n" + officeComputer.getConfiguration());

    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**Java Project (Package)** :AdapterPatternExample

**PaymentProcessor.java:**

public interface PaymentProcessor {

    void processPayment(String type, double amount);

}

**QrApPay.java:**

public class QrApPay {

    public void scanAndPay(double amount) {

        System.out.println("Paid Rs " + amount + " via QR Code.");

    }

}

**UpiPay.java:**

public class UpiPay {

   public void payUsingUPI(String upiId, double amount) {

        System.out.println("Paid Rs " + amount + " using UPI ID: " + upiId);

    }

}

**CryptoPay.java:**

public class CyrptoPay {

      public void makeTransaction(String cryptoType, double value) {

        System.out.println("Paid " + value + " in " + cryptoType + " using Crypto Wallet.");

    }

}

**PaymentAdapter.java:**

public class PaymentAdapter implements PaymentProcessor {

     private final String upiId = "dharan@cnrbank";

    private final String cryptoType = "Bitcoin";

    @Override

    public void processPayment(String type, double amount) {

        switch (type.toLowerCase()) {

            case "qr":

                QrApPay qr = new QrApPay();

                qr.scanAndPay(amount);

                break;

            case "upi":

                UpiPay upi = new UpiPay();

                upi.payUsingUPI(upiId, amount);

                break;

            case "crypto":

                CyrptoPay crypto = new CyrptoPay();

                crypto.makeTransaction(cryptoType, amount);

                break;

            default:

                System.out.println("Payment type not supported: " + type);

        }

    }

}

**TestPay.java:**

public class TestPay {

    public static void main(String[] args) {

        PaymentProcessor adapter = new PaymentAdapter();

System.out.println();System.out.println();

        adapter.processPayment("qr", 250.0);System.out.println();

        adapter.processPayment("upi", 500.0);System.out.println();

        adapter.processPayment("crypto", 0.01);System.out.println();

        adapter.processPayment("netbanking", 1000.0); System.out.println();

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**Java Project (Package)** :DecoratorPatternExample

**Notifier.java:**

public interface Notifier {

    String send();

}

**EmailNotifier.java:**

public class EmailNotifier implements Notifier {

    private String message;

    public EmailNotifier(String message) {

        this.message = message;

    }

    @Override

    public String send() {

         System.out.println();

        return "[ Email ]" + message;

    }

}

**NotifierDecorator.java:**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    @Override

    public String send() {

        return notifier.send();

    }

}

**SMSNotifierDecorator.java:**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public String send() {

        return super.send() + "\n [ SMS ]     Ding! ";

    }

}

**SlackNotifierDecorator.java:**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public String send() {

        return super.send() + "\n [ Slack ]  \*Beep\*  msg ";

    }

}

**PushNotifierDecorator.java:**

public class PushNotifierDecorator extends NotifierDecorator {

    public PushNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public String send() {

        return super.send() + "\n [  Push ]     Push Alert!";

    }

}

**TestNotifier.java:**

public class TestNotifier {

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier("Hello, Friends !");

        notifier = new SMSNotifierDecorator(notifier);

        notifier = new SlackNotifierDecorator(notifier);

        notifier = new PushNotifierDecorator(notifier);

        System.out.println(notifier.send());

    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**Java Project (Package)** :ProxyPatternExample

**Image.java:**

public interface Image {

    void display();

}

**RealImage.java:**

public class RealImage implements Image {

    private String fileName;

    private static int loadCount = 0;

    public RealImage(String fileName) {

        this.fileName = fileName;

        loadImageFromRemoteServer();

    }

    private void loadImageFromRemoteServer() {

        loadCount++;

        System.out.println();

        System.out.println("Loading image from remote server: " + fileName);

        try {

            Thread.sleep(1000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void display() {

        System.out.println("Displaying: " + fileName);

    }

    public static int getLoadCount() {

        return loadCount;

    }

}

**ProxyImage.java:**

import java.util.HashMap;

import java.util.Map;

public class ProxyImage implements Image {

    private String fileName;

    private static Map<String, RealImage> imageCache = new HashMap<>();

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        RealImage realImage = imageCache.get(fileName);

        if (realImage == null) {

            realImage = new RealImage(fileName);

            imageCache.put(fileName, realImage);

        } else {

             System.out.println();

            System.out.println("Using cached image: " + fileName);

        }

        realImage.display();

    }

}

**TestProxy.java:**

public class TestProxy {

    public static void main(String[] args) {

        Image img1 = new ProxyImage("sunset.jpg");

        Image img2 = new ProxyImage("sunset.jpg");

        Image img3 = new ProxyImage("mountain.png");

        img1.display();

        img2.display();

        img3.display();

         System.out.println();

        System.out.println("Total actual image loads: " + RealImage.getLoadCount());

    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Java Project (Package)** :ObserverPatternExample

**Stock.java:**

public interface Stock {

    void registerObserver(Observer o);

    void deregisterObserver(Observer o);

    void notifyObservers();

}

**StockMarket.java:**import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private double stockPrice;

    public void setStockPrice(double price) {

        this.stockPrice = price;

        notifyObservers();

    }

    public double getStockPrice() {

        return stockPrice;

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    @Override

    public void deregisterObserver(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockPrice);

        }

    }

}

**Observer.java:**

public interface Observer {

    void update(double stockPrice);

}

**MobileApp.java:**

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 + " - Mobile App: New stock price is USD " + stockPrice);

    }

}

**WebApp.java:**

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 + " - Web App: New stock price is USD " + stockPrice);

    }

}

**TestObserver.java:**

public class TestObserver {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileClient = new MobileApp("Client A");

        Observer webClient = new WebApp("Client B");

        stockMarket.registerObserver(mobileClient);

        stockMarket.registerObserver(webClient);

        System.out.println("\nSetting stock price to USD 100.50");

        stockMarket.setStockPrice(100.50);

        System.out.println("\nDeregistering Web Client...");

        stockMarket.deregisterObserver(webClient);

        System.out.println("Setting stock price to USD 110.75");

        stockMarket.setStockPrice(110.75);

    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**Java Project (Package)** :StrategyPatternExample

**PaymentStrategy.java:**

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java:**

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(double amount) {

        System.out.println("\nPaid Rs " + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

    }

}

**PayPalPayment.java:**

public class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("\nPaid Rs " + amount + " via PayPal account: " + email);

    }

}

**PaymentContext.java:**

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.paymentStrategy = strategy;

    }

    public void executePayment(double amount) {

        if (paymentStrategy == null) {

            System.out.println("No payment method selected.");

        } else {

            paymentStrategy.pay(amount);

        }

    }

}

**TestStrategy.java:**

public class TestStrategy {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1234567890123456"));

        context.executePayment(500.0);

        context.setPaymentStrategy(new PayPalPayment("userd@cnrbank.com"));

        context.executePayment(750.0);

    }

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Java Project (Package)** :CommandPatternExample

**Command.java:**

public interface Command {

    void execute();

}

**Light.java:**

public class Light {

    void on() {

        System.out.println("\nLight is ON\n");

    }

    void off() {

        System.out.println("Light is OFF\n");

    }

}

**LightOnCommand.java:**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.on();

    }

}

**LightOffCommand.java:**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.off();

    }

}

**RemoteContril.java:**

public class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        command.execute();

    }

}

**TestCommand.java:**

public class TestCommand {

    public static void main(String[] args) {

        Light light = new Light();

        Command on = new LightOnCommand(light);

        Command off = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(on);

        remote.pressButton();

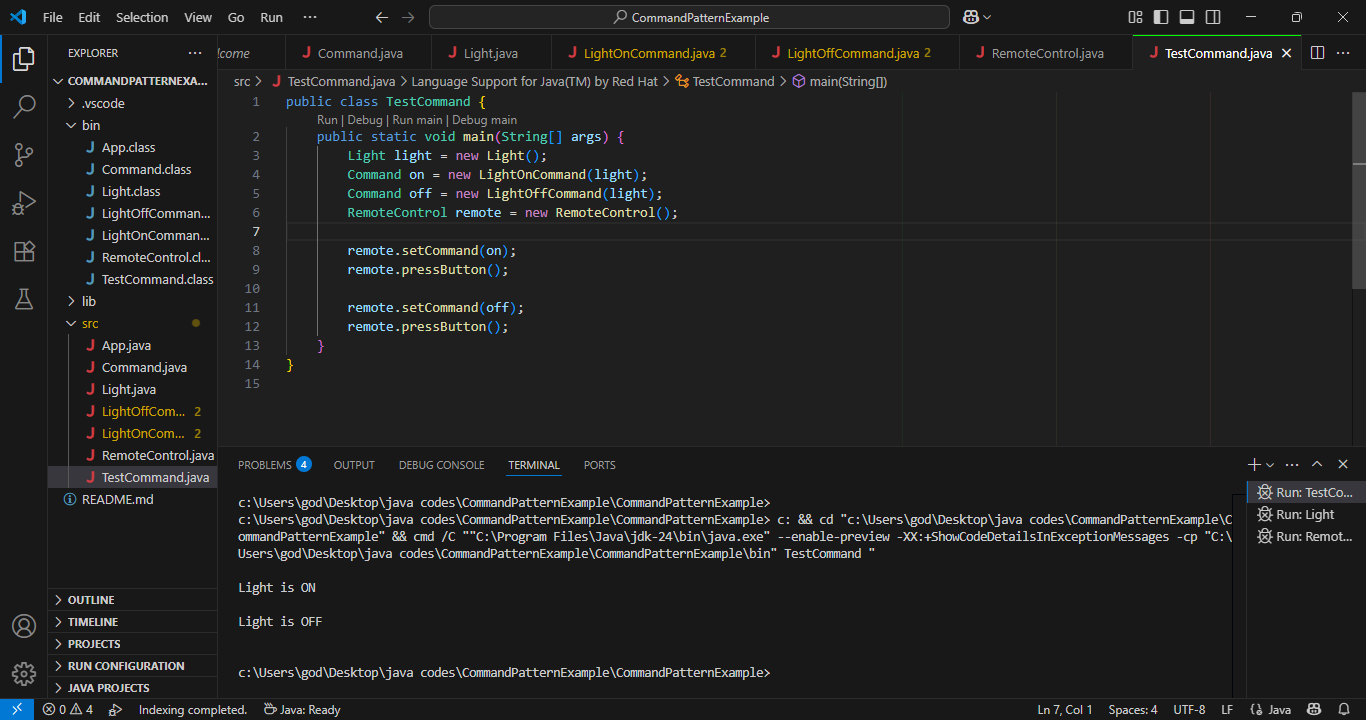
        remote.setCommand(off);

        remote.pressButton();

    }

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**Java Project (Package)** :MVCPatternExample

**Student.java:**

public class Student {

    private String name;

    private int rollNo;

    public Student(String name, int rollNo) {

        this.name = name;

        this.rollNo = rollNo;

    }

    public String getName() { return name; }

    public int getRollNo() { return rollNo; }

    public void setName(String name) { this.name = name; }

    public void setRollNo(int rollNo) { this.rollNo = rollNo; }

}

**StudentView.java:**

public class StudentView {

    public void showStudent(String name, int rollNo) {

        System.out.println("\nStudent: " + name);

        System.out.println("Roll No: " + rollNo+"\n");

    }

}

**StudentController.java:**

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 void setStudentRollNo(int rollNo) { model.setRollNo(rollNo); }

    public void updateView() {

        view.showStudent(model.getName(), model.getRollNo());

   }

}

**TestMvc.java:**

public class TestMvc {

    public static void main(String[] args) {

        Student student = new Student("John", 101);

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Alice");

        controller.setStudentRollNo(202);

        controller.updateView();

    }

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Exercise 11: Implementing Dependency Injection**

**Code:**

**Java Project (Package)** :DependencyInjectionPatternExample

**Customer.java:**

public class Customer {

    private int id;

    private String name;

    public Customer(int id, String name) {

        this.id = id;

        this.name = name;

    }

    public int getId() { return id; }

    public String getName() { return name; }

}

**CustomerRepository.java:**

public interface CustomerRepository {

    Customer findCustomerById(int id);

}

**CustomerRepositoryImpl.java:**

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

    private Map<Integer, Customer> database = new HashMap<>();

    public CustomerRepositoryImpl() {

        database.put(1, new Customer(1, "Alice"));

        database.put(2, new Customer(2, "Bob"));

    }

    public Customer findCustomerById(int id) {

        return database.get(id);

    }

}

**CustomerService.java:**

public class CustomerService {

    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository) {

        this.repository = repository;

    }

    public void printCustomer(int id) {

        Customer customer = repository.findCustomerById(id);

        if (customer != null) {

            System.out.println("\nCustomer Found: " + customer.getName());

        } else {

            System.out.println("\nCustomer not found.");

        }

    }

}

**TestDI.java:**

public class TestDI {

    public static void main(String[] args) {

        CustomerRepository repo = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repo);

        service.printCustomer(1);

        service.printCustomer(3);

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**