**Week 1-Design Patterns and Principles Solution**

Program:

// Java Design Pattern Examples (Exercises 1–11)

import java.util.\*;

// ------------- Exercise 1: Singleton Pattern -------------

class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String msg) {

System.out.println("Log: " + msg);

}

}

// ------------- Exercise 2: Factory Method Pattern -------------

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("Opening Word Document");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening PDF Document");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel Document");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// ------------- Exercise 3: Builder Pattern -------------

class Computer {

private String CPU;

private String RAM;

private String Storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.Storage = builder.Storage;

}

public static class Builder {

private String CPU;

private String RAM;

private String Storage;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String Storage) {

this.Storage = Storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public String toString() {

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

}

}

// ------------- Exercise 4: Adapter Pattern -------------

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

void makePayment(double amount) {

System.out.println("PayPal processing Rs." + amount);

}

}

class PayPalAdapter implements PaymentProcessor {

PayPalGateway gateway = new PayPalGateway();

public void processPayment(double amount) {

gateway.makePayment(amount);

}

}

// ------------- Exercise 5: Decorator Pattern -------------

interface Notifier {

void send(String msg);

}

class EmailNotifier implements Notifier {

public void send(String msg) {

System.out.println("Email: " + msg);

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String msg) {

wrappee.send(msg);

}

}

class SMSNotifier extends NotifierDecorator {

SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send(String msg) {

super.send(msg);

System.out.println("SMS: " + msg);

}

}

// ------------- Exercise 6: Proxy Pattern -------------

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

void loadFromDisk() {

System.out.println("Loading " + filename);

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

// ------------- Exercise 7: Observer Pattern -------------

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private float price;

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

}

class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App: New price Rs." + price);

}

}

class WebApp implements Observer {

public void update(float price) {

System.out.println("Web App: New price Rs." + price);

}

}

// ------------- Exercise 8: Strategy Pattern -------------

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid Rs." + amount + " using Credit Card");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid Rs." + amount + " using PayPal");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(int amount) {

strategy.pay(amount);

}

}

// ------------- Exercise 9: Command Pattern -------------

interface Command {

void execute();

}

class Light {

public void on() {

System.out.println("Light is ON");

}

public void off() {

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

}

}

class LightOnCommand implements Command {

private Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

private Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

// ------------- Exercise 10: MVC Pattern -------------

class Student {

String name;

int id;

String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

}

class StudentView {

void displayStudentDetails(Student s) {

System.out.println("Student: " + s.name + ", ID: " + s.id + ", Grade: " + s.grade);

}

}

class StudentController {

Student model;

StudentView view;

StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

void updateView() {

view.displayStudentDetails(model);

}

}

// ------------- Exercise 11: Dependency Injection -------------

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id;

}

}

class CustomerService {

private CustomerRepository repo;

CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void getCustomer(int id) {

System.out.println("Found: " + repo.findCustomerById(id));

}

}

// ---------------- Main to run all exercises ----------------

public class Main {

public static void main(String[] args) {

Logger logger = Logger.getInstance();

logger.log("Singleton working");

DocumentFactory pdfFactory = new PdfFactory();

pdfFactory.createDocument().open();

Computer c = new Computer.Builder().setCPU("i5").setRAM("8GB").setStorage("512GB").build();

System.out.println("Built: " + c);

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment(500);

Notifier notifier = new SMSNotifier(new EmailNotifier());

notifier.send("Hello!");

Image image = new ProxyImage("cat.png");

image.display();

image.display();

StockMarket stock = new StockMarket();

stock.register(new MobileApp());

stock.register(new WebApp());

stock.setPrice(245.50f);

PaymentContext ctx = new PaymentContext(new CreditCardPayment());

ctx.pay(1000);

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

remote.setCommand(new LightOffCommand(light));

remote.pressButton();

Student student = new Student("Ravi", 101, "A+");

StudentController controller = new StudentController(student, new StudentView());

controller.updateView();

CustomerService service = new CustomerService(new CustomerRepositoryImpl());

service.getCustomer(10);

}

}

Output:

