**WEEK 1 - DESIGN PATTERNS AND PRINCIPLES**

Super Set ID: 6427473

Name: SANJAI JAIVARDHAN B

1. **Implementing the Singleton Pattern**

**Program:**

* Logger.java:

package LoggerManagement;

public interface Logger {

void log();

}

* SystemLogger.java:

**package** LoggerManagement;

**public** **class** SystemLogger **implements** Logger {

**private** **static** SystemLogger *instance*;

**private** SystemLogger() {}

**public** **static** SystemLogger getInstance() {

**if**(*instance*==**null**) {

*instance* = **new** SystemLogger();

}

**return** *instance*;

}

@Override

**public** **void** log() {

System.***out***.println("System log recorded.");

}

}

* UserLogger.java:

**package** LoggerManagement;

**public** **class** UserLogger **implements** Logger {

@Override

**public** **void** log() {

System.***out***.println("User log recorded.");

}

}

* Main.java:

package LoggerManagement;

public class Main {

public static void main(String[] args) {

Logger systemLogger = SystemLogger.*getInstance*();

Logger anotherLogger = SystemLogger.*getInstance*();

Logger userLogger = new UserLogger();

System.*out*.println("System Logger:"+systemLogger);

System.*out*.println("Another Logger:"+anotherLogger);

systemLogger.log();

userLogger.log();

}

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Result:**

The output confirms that both the loggers (System Logger and Another Logger) are using one global instance of the SystemLogger class. This demonstrates that the Singleton pattern has been correctly implemented, ensuring a single, consistent logger instance is used across the entire application for centralized logging.

1. **Implementing the Factory Method Pattern**

**Program:**

* Manager.java:

package DocumentManagement;

interface Document{

void view();

}

class Pdf implements Document{

public void view() {

System.*out*.println("Pdf Document is opened");

}

}

class Word implements Document{

public void view() {

System.*out*.println("Word Document is opened");

}

}

class Excel implements Document{

public void view() {

System.*out*.println("Excel Document is opened");

}

}

abstract class DocumentFactory{

public abstract Document createDocument();

}

class WordDocument extends DocumentFactory{

public Document createDocument() {

return new Word();

}

}

class PdfDocument extends DocumentFactory{

public Document createDocument() {

return new Pdf();

}

}

class ExcelDocument extends DocumentFactory{

public Document createDocument() {

return new Excel();

}

}

public class Manager {

public static void main(String args[]) {

DocumentFactory factory;

factory = new PdfDocument();

Document pdf = factory.createDocument();

pdf.view();

factory = new WordDocument();

Document word = factory.createDocument();

word.view();

factory = new PdfDocument();

Document excel = factory.createDocument();

excel.view();

}

}

**Output:**

A close-up of a computer screen

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Factory Design Pattern** is used to create different types of documents (PDF, Word, Excel) without specifying the exact class that needs to be instantiated in the client code. The goal is to **abstract the instantiation process** and allow **object creation through a common interface**, promoting loose coupling and easier scalability in the document management system.

1. **Implementing the Builder Pattern**
   * **Program:**
   * Computer.java:

package ComputerManagement;

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isWiFiEnabled;

private boolean isBluetoothEnabled;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.isWiFiEnabled = builder.isWiFiEnabled;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

}

public static class Builder{

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isWiFiEnabled;

private boolean isBluetoothEnabled;

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 Builder setGraphicsCard(String graphics) {

this.graphicsCard = graphics;

return this;

}

public Builder enableWiFi(boolean value) {

this.isWiFiEnabled = value;

return this;

}

public Builder enableBluetooth(boolean value) {

this.isBluetoothEnabled = value;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

* + BuilderPatternTest.java:

package ComputerManagement;

public class BuilderPatternTest {

public static void main(String[] args) {

Computer gamingPC = new Computer

.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.enableWiFi(true)

.enableBluetooth(true)

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.enableWiFi(false)

.enableBluetooth(false)

.build();

System.*out*.println("Gaming PC: " + gamingPC);

System.*out*.println("Office PC: " + officePC);

}

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Builder Design Pattern** is used to construct complex objects such as different configurations of a computer (e.g., Gaming PC, Office PC) step-by-step. It allows the client to create objects with varying combinations of attributes without directly instantiating the class with a telescoping constructor. This promotes code **readability**, **flexibility**, and **maintainability**, especially when many optional parameters are involved. The output confirms that two different Computer objects with unique configurations were successfully created using the builder.

1. **Implementing the Adapter Pattern**

**Program:**

* PaymentProcessor.java:

**package** PaymentManagement;

**public** **interface** PaymentProcessor {

**void** processPayment(**double** amount);

}

* PayPal.java:

**package** PaymentManagement;

**public** **class** PayPal {

**public** **void** makePayPalPayment(**double** amount) {

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

}

}

* Stripe.java:

**package** PaymentManagement;

**public** **class** Stripe {

**public** **void** makeStripePayment(**double** amount,String currency) {

System.***out***.println("Paid "+amount +" "+ currency+" using Stripe.");

}

}

* PayPalAdapter.java:

**package** PaymentManagement;

**public** **class** PayPalAdapter **implements** PaymentProcessor {

**private** PayPal payPal;

**public** PayPalAdapter(PayPal payPal) {

**this**.payPal = payPal;

}

@Override

**public** **void** processPayment(**double** amount) {

payPal.makePayPalPayment(amount);

}

}

* StripeAdapter.java:

**package** PaymentManagement;

**public** **class** StripeAdapter **implements** PaymentProcessor{

**private** Stripe stripe;

**public** StripeAdapter(Stripe stripe) {

**this**.stripe = stripe;

}

@Override

**public** **void** processPayment(**double** amount) {

stripe.makeStripePayment(amount, "INR");

}

}

* Main.java:

**package** PaymentManagement;

**public** **class** Main {

**public** **static** **void** main(String args[]) {

PaymentProcessor payPalPayment = **new** PayPalAdapter(**new** PayPal());

PaymentProcessor stripePayment = **new** StripeAdapter(**new** Stripe());

payPalPayment.processPayment(500.0);

stripePayment.processPayment(1000.0);

}

}

**Output:**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**Result:**

In this implementation, the **Adapter Design Pattern** is used to allow two incompatible interfaces PayPal and Stripe to work seamlessly with a common payment interface. The Stripe class has a different method signature than PayPal, so an adapter class bridges this difference by converting the Stripe payment method into a compatible form. This ensures both payment systems can be used interchangeably without modifying the client code, promoting **reusability**, **flexibility**, and **code consistency**. The output confirms successful execution of payments through both PayPal and Stripe using the unified interface.

1. **Implementing the Decorator Pattern**

**Program:**

* Notifier.java:

**package** notificationManager;

**public** **interface** Notifier {

**void** send(String message);

}

* EmailNotifier.java:

**package** notificationManager;

**public** **class** EmailNotifier **implements** Notifier {

@Override

**public** **void** send(String message) {

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

}

}

* NotifierDecorator.java:

**package notificationManager;**

**public abstract class NotifierDecorator implements Notifier{**

**protected Notifier wrappedNotifier;**

**public NotifierDecorator(Notifier notifier) {**

**this.wrappedNotifier = notifier;**

**}**

**public void send(String message) {**

**wrappedNotifier.send(message);**

**}**

**}**

* SMSNotifierDecorator.java:

**package notificationManager;**

**public class SMSNotifierDecorator extends NotifierDecorator {**

**public SMSNotifierDecorator(Notifier notifier) {**

**super(notifier);**

**}**

**@Override**

**public void send(String message) {**

**super.send(message);**

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

**}**

**}**

* Main.java:

**package notificationManager;**

**public class Main {**

**public static void main(String args[]) {**

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

**notifier = new SMSNotifierDecorator(notifier);**

**notifier.send("System update at 6 PM");**

**}**

**}**

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Decorator Pattern** is used to extend a base notification (Email) with additional functionality (SMS) at runtime. This allows multiple notification types to be combined flexibly without changing the original classes. The output confirms that both Email and SMS notifications are sent using the decorator chain.

1. **Implementing the Proxy Pattern**

**Program:**

* Image.java:

**package imageManagement;**

**public interface Image {**

**void display();**

**}**

* RealImage.java:

**package imageManagement;**

**public class RealImage implements Image{**

**private String filename;**

**public RealImage(String filename) {**

**this.filename = filename;**

**loadFromDisk();**

**}**

**private void loadFromDisk() {**

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

**}**

**@Override**

**public void display() {**

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

**}**

**}**

* ProxyImage.java:

**package imageManagement;**

**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();**

**}**

**}**

* Main.java:

**package imageManagement;**

**public class Main {**

**public static void main(String args[]) {**

**Image image1 = new ProxyImage("nature.jpg");**

**Image image2 = new ProxyImage("architecture.jpg");**

**image1.display();**

**image2.display();**

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

**System.*out*.println("Second time load");**

**image1.display();**

**image2.display();**

**}**

**}**

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Proxy Pattern** is used to control access to image loading with lazy initialization and caching. The first time an image is displayed, it is loaded from the disk. On subsequent displays, the image is shown without reloading. The output confirms that each image is **loaded only once**, demonstrating efficient resource usage through the proxy.

1. **Implementing the Observer Pattern**

**Program:**

* Observer.java:

**package** stockMarket;

**public** **interface** Observer {

**void** update(String stockName, **double** newPrice);

}

* Stock.java:

**package** stockMarket;

**public** **interface** Stock {

**void** registerObserver(Observer o);

**void** removeObserver(Observer o);

**void** notifyObservers();

}

* StockMarket.java:

**package stockMarket;**

**import java.util.ArrayList;**

**import java.util.List;**

**public class StockMarket implements Stock {**

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

**private String stockName;**

**private double stockPrice;**

**public StockMarket(String stockName, double initialPrice) {**

**this.stockName = stockName;**

**this.stockPrice = initialPrice;**

**}**

**public void setPrice(double newPrice) {**

**System.*out*.println("\nUpdating price for " + stockName + " to $" + newPrice);**

**this.stockPrice = newPrice;**

**notifyObservers();**

**}**

**@Override**

**public void registerObserver(Observer o) {**

**observers.add(o);**

**}**

**@Override**

**public void removeObserver(Observer o) {**

**observers.remove(o);**

**}**

**@Override**

**public void notifyObservers() {**

**for (Observer o : observers) {**

**o.update(stockName, stockPrice);**

**}**

**}**

**}**

* MobileApp.java:

**package stockMarket;**

**public class MobileApp implements Observer {**

**private String name;**

**public MobileApp(String name) {**

**this.name = name;**

**}**

**@Override**

**public void update(String stockName, double newPrice) {**

**System.*out*.println(name + " (Mobile) received update: " + stockName + " is now $" + newPrice);**

**}**

**}**

* WebApp.java:

**package stockMarket;**

**public class WebApp implements Observer {**

**private String name;**

**public WebApp(String name) {**

**this.name = name;**

**}**

**@Override**

**public void update(String stockName, double newPrice) {**

**System.*out*.println(name + " (Web) received update: " + stockName + " is now $" + newPrice);**

**}**

**}**

* Main.java:

**package stockMarket;**

**public class Main {**

**public class ObserverDemo {**

**public static void main(String[] args) {**

**StockMarket stockMarket = new StockMarket("ACME", 100.0);**

**Observer mobileApp = new MobileApp("Sanjai's Phone");**

**Observer webApp = new WebApp("Dashboard");**

**stockMarket.registerObserver(mobileApp);**

**stockMarket.registerObserver(webApp);**

**stockMarket.setPrice(101.5);**

**stockMarket.setPrice(98.7);**

**stockMarket.removeObserver(mobileApp);**

**stockMarket.setPrice(105.2);**

**}**

**}**

**}**

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Observer Pattern** is used to notify multiple clients (e.g., MobileApp, WebApp) whenever there is a change in stock prices. The StockMarket class maintains a list of observers and informs them using the update() method when prices change. The output confirms that **all registered observers are notified** in real-time, ensuring consistent updates across different platforms.

1. **Implementing the Strategy Pattern**

**Program:**

* PaymentStratergy.java:

**package paymentPackage;**

**public interface PaymentStrategy {**

**void pay(double amount);**

**}**

* PaymentContext.java:

**package paymentPackage;**

**public class PaymentContext {**

**private PaymentStrategy strategy;**

**public void setPaymentStrategy(PaymentStrategy strategy) {**

**this.strategy = strategy;**

**}**

**public void processPayment(double amount) {**

**if (strategy == null) {**

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

**} else {**

**strategy.pay(amount);**

**}**

**}**

**}**

* PayPalPayment.java:

**package paymentPackage;**

**public class PayPalPayment implements PaymentStrategy {**

**private String email;**

**public PayPalPayment(String email) {**

**this.email = email;**

**}**

**@Override**

**public void pay(double amount) {**

**System.*out*.println("Paid ₹" + amount + " using PayPal account: " + email);**

**}**

**}**

* UPIPayment.java:

**package paymentPackage;**

**public class UPIPayment implements PaymentStrategy {**

**private String upiId;**

**public UPIPayment(String upiId) {**

**this.upiId = upiId;**

**}**

**@Override**

**public void pay(double amount) {**

**System.*out*.println("Paid ₹" + amount + " via UPI ID: " + upiId);**

**}**

**}**

* WebApp.java:

**package paymentPackage;**

**public class CreditCardPayment implements PaymentStrategy {**

**private String cardNumber;**

**public CreditCardPayment(String cardNumber) {**

**this.cardNumber = cardNumber;**

**}**

**@Override**

**public void pay(double amount) {**

**System.*out*.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);**

**}**

**}**

* Main.java:

**package paymentPackage;**

**public class Main {**

**public static void main(String[] args) {**

**PaymentContext context = new PaymentContext();**

**context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));**

**context.processPayment(2500.0);**

**context.setPaymentStrategy(new PayPalPayment("sanjai@example.com"));**

**context.processPayment(1500.0);**

**context.setPaymentStrategy(new UPIPayment("sanjai@upi"));**

**context.processPayment(800.0);**

**}**

**}**

**Output:**

A white background with a group of people

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Strategy Pattern** is used to enable dynamic selection of payment methods such as Credit Card, PayPal, and UPI at runtime. Each payment method is encapsulated as a separate strategy implementing the common PaymentStrategy interface. The PaymentContext class uses these strategies to execute the appropriate payment logic. The output confirms that **different payment methods can be used interchangeably**, demonstrating flexibility and modularity in the payment system design.

1. **Implementing the Command Pattern**

**Program:**

* Command.java:

**package automationSystem;**

**public interface Command {**

**void execute();**

**}**

* Fan.java:

**package automationSystem;**

**public class Fan {**

**public void turnOn() {**

**System.*out*.println("Fan is spinning");**

**}**

**public void turnOff() {**

**System.*out*.println("Fan is stopped");**

**}**

**}**

* Light.java:

**package** automationSystem;

**public** **class** Light {

**public** **void** turnOn() {

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

}

**public** **void** turnOff() {

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

}

}

* LightOFFCommand.java:

**package automationSystem;**

**public class LightOffCommand implements Command {**

**private Light light;**

**public LightOffCommand(Light light) {**

**this.light = light;**

**}**

**@Override**

**public void execute() {**

**light.turnOff();**

**}**

**}**

* LightOnCommand.java:

**package automationSystem;**

**public class LightOnCommand implements Command {**

**private Light light;**

**public LightOnCommand(Light light) {**

**this.light = light;**

**}**

**@Override**

**public void execute() {**

**light.turnOn();**

**}**

**}**

* RemoteControl.java:

**package automationSystem;**

**public class RemoteControl {**

**private Command slot;**

**public void setCommand(Command command) {**

**this.slot = command;**

**}**

**public void pressButton() {**

**if (slot != null) {**

**slot.execute();**

**} else {**

**System.*out*.println("No command set.");**

**}**

**}**

**}**

* Main.java:

**package automationSystem;**

**public class Main {**

**public static void main(String[] args){**

**Light livingRoomLight = new Light();**

**Fan ceilingFan = new Fan();**

**Command lightOn = new LightOnCommand(livingRoomLight);**

**Command lightOff = new LightOffCommand(livingRoomLight);**

**Command fanOn = new FanOnCommand(ceilingFan);**

**Command fanOff = new FanOffCommand(ceilingFan);**

**RemoteControl remote = new RemoteControl();**

**remote.setCommand(lightOn);**

**remote.pressButton();**

**remote.setCommand(fanOn);**

**remote.pressButton();**

**remote.setCommand(lightOff);**

**remote.pressButton();**

**remote.setCommand(fanOff);**

**remote.pressButton();**

**}**

**}**

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **Command Pattern** is used to encapsulate requests as command objects, allowing actions like turning lights and fans on or off to be issued through a centralized controller (RemoteControl). Each command (e.g., LightOnCommand, FanOffCommand) encapsulates a specific action and can be executed independently. The output confirms that **commands are issued and executed successfully**, demonstrating clear separation between the invoker, command, and receiver.

1. **Implementing the MVC Pattern**

**Program:**

* StudentView.java:

**package studentManagement;**

**public class StudentView {**

**public void printStudentDetails(String name, String rollNo) {**

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

**System.*out*.println("Name: " + name);**

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

**}**

**}**

* StudentController.java:

**package studentManagement;**

**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(String rollNo) {**

**model.setRollNo(rollNo);**

**}**

**public String getStudentName() {**

**return model.getName();**

**}**

**public String getStudentRollNo() {**

**return model.getRollNo();**

**}**

**public void updateView() {**

**view.printStudentDetails(model.getName(), model.getRollNo());**

**}**

**}**

* Student.java:

**package studentManagement;**

**public class Student {**

**private String name;**

**private String rollNo;**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getRollNo() {**

**return rollNo;**

**}**

**public void setRollNo(String rollNo) {**

**this.rollNo = rollNo;**

**}**

**}**

* Main.java:

**package studentManagement;**

**public class Main {**

**public static void main(String[] args) {**

**Student student = new Student();**

**student.setName("Sanjai Jaivardhan");**

**student.setRollNo("2201261");**

**StudentView view = new StudentView();**

**StudentController controller = new StudentController(student, view);**

**controller.updateView();**

**controller.setStudentName("Sanjai Jaivardhan");**

**controller.updateView();**

**}**

**}**

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Result:**

In this implementation, the **MVC (Model-View-Controller) Pattern** is used to separate concerns in a student management application. The Student class acts as the **Model**, holding the data; the StudentView class is the **View**, responsible for displaying student details; and the StudentController is the **Controller**, managing the interaction between the model and view. The output confirms that **student information is successfully displayed**, demonstrating a clean separation of logic, presentation, and control flow.

1. **Implementing Dependency Injection**

**Program:**

* CustomerRepository.java:

**package repositroyManagement;**

**public interface CustomerRepository {**

**Customer findCustomerById(int id);**

**}**

* CustomerRepositoryImpl.java:

**package repositroyManagement;**

**public class Customer {**

**private int id;**

**private String name;**

**public Customer(int id, String name) {**

**this.id = id;**

**this.name = name;**

**}**

**public String getName() {**

**return name;**

**}**

**public int getId() {**

**return id;**

**}**

**}**

* Customer.java:

**package repositroyManagement;**

**public class Customer {**

**private int id;**

**private String name;**

**public Customer(int id, String name) {**

**this.id = id;**

**this.name = name;**

**}**

**public String getName() {**

**return name;**

**}**

**public int getId() {**

**return id;**

**}**

**}**

* CustomerService.java:

**package repositroyManagement;**

**public class CustomerService {**

**private CustomerRepository customerRepository;**

**public CustomerService(CustomerRepository customerRepository) {**

**this.customerRepository = customerRepository;**

**}**

**public void getCustomerInfo(int id) {**

**Customer customer = customerRepository.findCustomerById(id);**

**System.*out*.println("Customer found: ID = " + customer.getId() + ", Name = " + customer.getName());**

**}**

**}**

* CustomerService.java:

**package repositroyManagement;**

**public class Main {**

**public static void main(String[] args) {**

**CustomerRepository repository = new CustomerRepositoryImpl();**

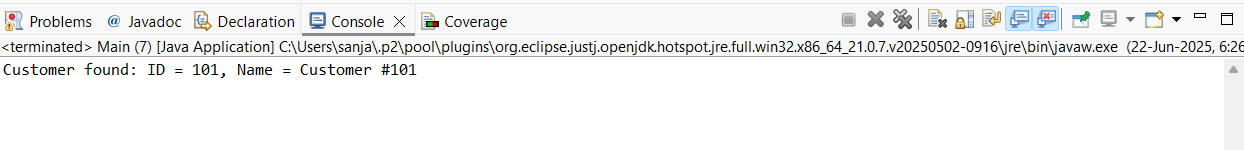
**CustomerService service = new CustomerService(repository);**

**service.getCustomerInfo(101);**

**}**

**}**

**Output:**



**Result:**

In this implementation, **Dependency Injection** is used to decouple the CustomerService from the CustomerRepositoryImpl by injecting the dependency via the constructor. This promotes better modularity, easier testing, and maintainability. The output confirms that the CustomerService successfully accessed customer data through the injected repository, demonstrating proper use of constructor-based dependency injection.