**WEEK 1**

**Design Patterns and Principles**

**NAME : THILAGAR S**

**SUPERSET ID : 6420519**

**Exercise 1 : Implementing the Singleton Pattern**

**CODE:**

package SingletonPatternExample;  
public class SingletonPattern {  
 public static void main(String[] args) {  
 Logger loggerSample1 = Logger.*getInstance*();  
 loggerSample1.log("First message");  
 Logger loggerSample2 = Logger.*getInstance*();  
 loggerSample2.log("Second message");  
 if (loggerSample1 == loggerSample2) {  
 System.*out*.println("Only one Logger instance is used.");  
 } else {  
 System.*out*.println("Different Logger instances exist.");  
 }  
 }  
}  
class Logger {  
 private static Logger *staticInstance*;  
 private Logger() {  
 System.*out*.println("Logger initialized.");  
 }  
 public static Logger getInstance() {  
 if (*staticInstance* == null)   
 *staticInstance* = new Logger();  
 return *staticInstance*;  
 }  
 public void log(String message) {  
 System.*out*.println("Log: " + message);  
 }  
}

**Output:**

A screen shot of a computer

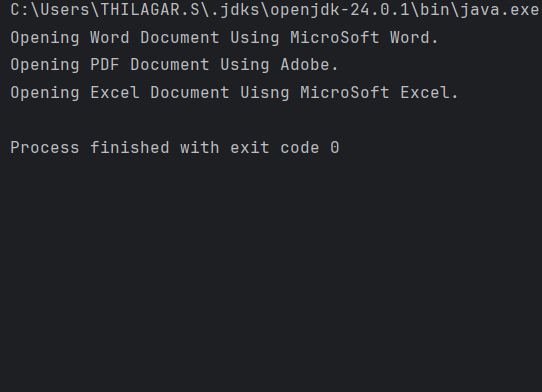
AI-generated content may be incorrect.

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

**CODE:**

package FactoryMethodPatternExample;  
interface Document {  
 void open();  
}  
class WordDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening Word Document Using MicroSoft Word.");  
 }  
}  
class PdfDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening PDF Document Using Adobe.");  
 }  
}  
class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening Excel Document Uisng MicroSoft Excel.");  
 }  
}  
abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}  
public class FactoryMethod {  
 public static void main(String[] args) {  
 DocumentFactory wordFac = new WordDocumentFactory();  
 Document wordDoc = wordFac.createDocument();  
 wordDoc.open();  
 DocumentFactory pdfFac = new PdfDocumentFactory();  
 Document pdfDoc = pdfFac.createDocument();  
 pdfDoc.open();  
 DocumentFactory excelFac = new ExcelDocumentFactory();  
 Document excelDoc = excelFac.createDocument();  
 excelDoc.open();  
 }  
}

**Output:**

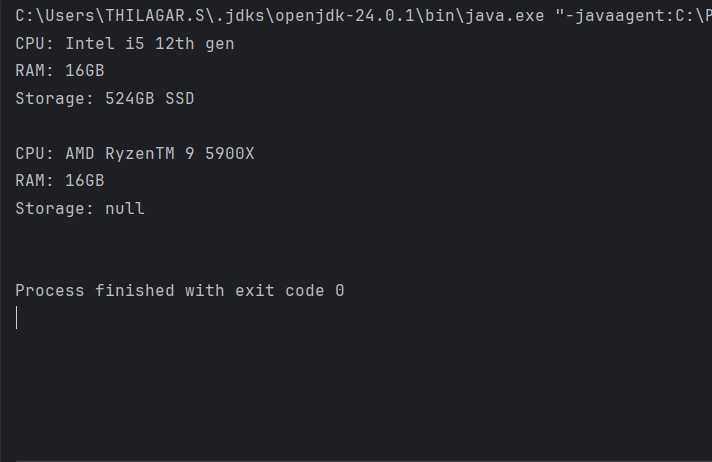


**Exercise 3: Implementing the Builder Pattern**

**CODE:**

package BuilderPatternExample;  
public class BuilderPattern {  
 public static void main(String[] args) {  
 Computer computerFirst = new Computer.Builder()  
 .setCPU("Intel i5 12th gen")  
 .setRAM("16GB")  
 .setStorage("524GB SSD")  
 .build();  
 Computer computerSecond = new Computer.Builder()  
 .setCPU("AMD RyzenTM 9 5900X")  
 .setRAM("16GB")  
 .build();  
 computerFirst.displayConfiguration();  
 computerSecond.displayConfiguration();  
 }  
}  
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 void displayConfiguration() {  
 System.*out*.println("CPU: " + CPU);  
 System.*out*.println("RAM: " + RAM);  
 System.*out*.println("Storage: " + Storage);  
 System.*out*.println();  
 }  
 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);  
 }  
 }  
}

**Output:**

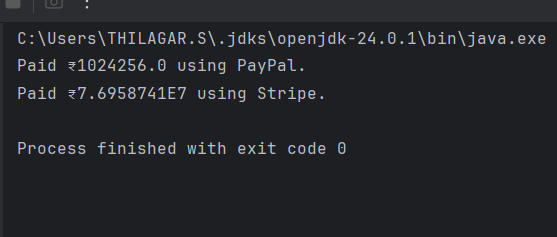


**Exercise 4: Implementing the Adapter Pattern**

**CODE:**

package AdapterPatternExample;  
  
public class AdapterPattern {  
 public static void main(String[] args) {  
 PaymentProcessor paypalPro = new PayPalAdapter();  
 paypalPro.processPayment(1024256);  
 PaymentProcessor stripePro = new StripeAdapter();  
 stripePro.processPayment(76958741);  
 }  
}  
interface PaymentProcessor {  
 void processPayment(double amount);  
}  
class PayPalGateway {  
 public void makePayment(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using PayPal.");  
 }  
}  
class StripeGateway {  
 public void initiateTransaction(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using Stripe.");  
 }  
}  
class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway payPalGateway = new PayPalGateway();  
  
 public void processPayment(double amount) {  
 payPalGateway.makePayment(amount);  
 }  
}  
class StripeAdapter implements PaymentProcessor {  
 private StripeGateway stripeGateway = new StripeGateway();  
  
 public void processPayment(double amount) {  
 stripeGateway.initiateTransaction(amount);  
 }  
}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**CODE:**

package DecoratorPatternExample;  
public class DecoratorPattern {  
 public static void main(String[] args) {  
 Notifier email = new EmailNotifier();  
 Notifier sms = new SMSNotifierDecorator(email);  
 Notifier slack = new SlackNotifierDecorator(sms);  
 slack.send();  
 }  
}  
interface Notifier {  
 void send();  
}  
class EmailNotifier implements Notifier {  
 public void send() {  
 System.*out*.println("Sending Email Notification");  
 }  
}  
abstract class NotifierDecorator implements Notifier {  
 protected Notifier notifier;  
 public NotifierDecorator(Notifier notifier) {  
 this.notifier = notifier;  
 }  
 public void send() {  
 notifier.send();  
 }  
}  
class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send() {  
 super.send();  
 System.*out*.println("Sending SMS Notification");  
 }  
}  
class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send() {  
 super.send();  
 System.*out*.println("Sending Slack Notification");  
 }  
}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

package ProxyPatternExample;  
public class ProxyPattern {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("familyPhoto1.jpg");  
 image1.display();  
 image1.display();  
  
 Image image2 = new ProxyImage("groupPhoto2.jpg");  
 image2.display();  
 }  
}  
interface Image {  
 void display();  
}  
class RealImage implements Image {  
 private String fileName;  
 public RealImage(String fileName) {  
 this.fileName = fileName;  
 loadFromRemoteServer();  
 }  
 private void loadFromRemoteServer() {  
 System.*out*.println("Loading " + fileName + " from remote server...");  
 }  
 public void display() {  
 System.*out*.println("Displaying " + fileName);  
 }  
}  
class ProxyImage implements Image {  
 private RealImage realImage;  
 private String fileName;  
 public ProxyImage(String fileName) {  
 this.fileName = fileName;  
 }  
 public void display() {  
 if (realImage == null) {  
 realImage = new RealImage(fileName);  
 }  
 realImage.display();  
 }  
}

**Output:**

A computer screen shot of a program

AI-generated content may be incorrect.

**Exercise 7: Implementing the Observer Pattern**

**CODE:**

package ObserverPatternExample;  
import java.util.\*;  
public class ObserverPattern {  
 public static void main(String[] args) {  
 StockMarket stockMarket = new StockMarket();  
 Observer mobileApp = new MobileApp("MobileApp");  
 Observer webApp = new WebApp("WebApp");  
 stockMarket.register(mobileApp);  
 stockMarket.register(webApp);  
 stockMarket.setStockPrice(10078.5);  
 stockMarket.setStockPrice(112459.0);  
 stockMarket.deregister(mobileApp);  
 stockMarket.setStockPrice(12119.25);  
 }  
}  
interface Stock {  
 void register(Observer o);  
 void deregister(Observer o);  
 void notifyObservers();  
}  
class StockMarket implements Stock {  
 private List<Observer> observers = new ArrayList<>();  
 private double stockPrice;  
 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(stockPrice);  
 }  
 }  
 public void setStockPrice(double price) {  
 this.stockPrice = price;  
 notifyObservers();  
 }  
}  
interface Observer {  
 void update(double price);  
}  
class MobileApp implements Observer {  
 private String name;  
 public MobileApp(String name) {  
 this.name = name;  
 }  
 public void update(double price) {  
 System.*out*.println(name + " received stock price update: ₹" + price);  
 }  
}  
class WebApp implements Observer {  
 private String name;  
 public WebApp(String name) {  
 this.name = name;  
 }  
 public void update(double price) {  
 System.*out*.println(name + " received stock price update: ₹" + price);  
 }  
}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 8: Implementing the Strategy Pattern**

**CODE:**

package StrategyPatternExample;  
public class StrategyPattern {  
 public static void main(String[] args) {  
 PaymentContext context = new PaymentContext();  
 context.setPaymentStrategy(new CreditCardPayment());  
 context.pay(825467);  
 context.setPaymentStrategy(new PayPalPayment());  
 context.pay(8795468);  
 }  
}  
interface PaymentStrategy {  
 void pay(double amount);  
}  
class CreditCardPayment implements PaymentStrategy {  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using Credit Card.");  
 }  
}  
class PayPalPayment implements PaymentStrategy {  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using PayPal.");  
 }  
}  
class PaymentContext {  
 private PaymentStrategy paymentStrategy;  
 public void setPaymentStrategy(PaymentStrategy paymentStrategy) {  
 this.paymentStrategy = paymentStrategy;  
 }  
 public void pay(double amount) {  
 paymentStrategy.pay(amount);  
 }  
}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 9: Implementing the Command Pattern**

**CODE:**

package CommandPatternExample;  
public class CommandPattern{  
 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();  
 }  
}  
interface Command {  
 void execute();  
}  
class Light {  
 public void turnOn() {  
 System.*out*.println("Light is ON");  
 }  
 public void turnOff() {  
 System.*out*.println("Light is OFF");  
 }  
}  
class LightOnCommand implements Command {  
 private Light lightSwitch;  
 public LightOnCommand(Light light) {  
 this.lightSwitch = light;  
 }  
 public void execute() {  
 lightSwitch.turnOn();  
 }  
}  
class LightOffCommand implements Command {  
 private Light lightSwitch;  
 public LightOffCommand(Light light) {  
 this.lightSwitch = light;  
 }  
 public void execute() {  
 lightSwitch.turnOff();  
 }  
}  
class RemoteControl {  
 private Command command;  
 public void setCommand(Command command) {  
 this.command = command;  
 }  
 public void pressButton() {  
 command.execute();  
 }  
}

**Output:**

A screen shot of a computer

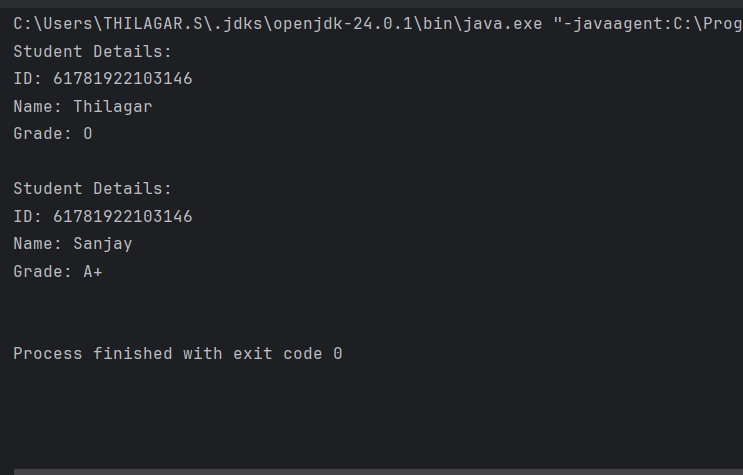
AI-generated content may be incorrect.

**Exercise 10: Implementing the MVC Pattern**

**CODE:**

package MVCPatternExample;  
public class MVCPattern {  
 public static void main(String[] args) {  
 Student modelDetails = new Student();  
 modelDetails.setId("61781922103146");  
 modelDetails.setName("Thilagar");  
 modelDetails.setGrade("O");  
 StudentView viewDetails = new StudentView();  
 StudentController controller = new StudentController(modelDetails, viewDetails);  
 controller.updateView();  
 controller.setStudentName("Sanjay");  
 controller.setStudentGrade("A+");  
 controller.updateView();  
 }  
}  
class Student {  
 private String id;  
 private String name;  
 private String 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;  
 }  
}  
class StudentView {  
 public void displayStudentDetails(String name, String id, String grade) {  
 System.*out*.println("Student Details:");  
 System.*out*.println("ID: " + id);  
 System.*out*.println("Name: " + name);  
 System.*out*.println("Grade: " + grade);  
 System.*out*.println();  
 }  
}  
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 setStudentId(String id) {  
 model.setId(id);  
 }  
 public void setStudentGrade(String grade) {  
 model.setGrade(grade);  
 }  
 public void updateView() {  
 view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());  
 }  
}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**CODE:**

package DependencyInjectionExample;  
public class DependencyInjection {  
 public static void main(String[] args) {  
 CustomerRepository repo = new CustomerRepositoryImpl();  
 CustomerService service = new CustomerService(repo);  
 service.findCustomer("22ECEBE133");  
 }  
}  
interface CustomerRepository {  
 void findCustomerById(String id);  
}  
class CustomerRepositoryImpl implements CustomerRepository {  
 public void findCustomerById(String id) {  
 System.*out*.println("Customer found with ID: " + id);  
 }  
}  
class CustomerService {  
 private CustomerRepository customerRepository;  
 public CustomerService(CustomerRepository customerRepository) {  
 this.customerRepository = customerRepository;  
 }  
 public void findCustomer(String id) {  
 customerRepository.findCustomerById(id);  
 }  
}

**Output:**

