# Design Pattern and principles

## Exercise 1: Implementing the Singleton Pattern

### Code:

#### Logger.java

package SingletonPattern; public class Logger {

// Private static instance of Logger private static Logger instance;

// Private constructor prevents instantiation from other classes

private Logger() {

System.out.println("Logger initialized.");

}

// Public static method to return the singleton instance public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

// Logging method

public void log(String message) { System.out.println("[LOG] " + message);

}

}

LoggerTest.java package SingletonPattern; public class LoggerTest {

public static void main(String[] args) {

// Get singleton instances

Logger logger1 = Logger.getInstance(); Logger logger2 = Logger.getInstance();

// Use the logger

logger1.log("This is the first log message."); logger2.log("This is the second log message.");

// Validate that both references point to the same instance

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Singleton confirmed).");

} else {

System.out.println("Different logger instances (Singleton violated).");

}

}

}

# output

## Exercise 2: Implementing the Factory Method Pattern

### Code:

#### Document.java

package FactoryMethodPattern; public interface Document {

void open(); void save(); void close();

}

#### DocumentFactory.java

package FactoryMethodPattern;

// File: DocumentFactory.java

public abstract class DocumentFactory { public abstract Document createDocument();

}

#### ExcelDocument.java

package FactoryMethodPattern;

// File: ExcelDocument.java

public class ExcelDocument implements Document { @Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving Excel document...");

}

@Override

public void close() {

System.out.println("Closing Excel document...");

}

}

#### ExcelDocumentFactory.java

package FactoryMethodPattern;

// File: ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory { @Override

public Document createDocument() {

return new ExcelDocument();

}

}

#### PdfDocument.java

package FactoryMethodPattern;

// File: PdfDocument.java

public class PdfDocument implements Document { @Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving PDF document...");

}

@Override

public void close() {

System.out.println("Closing PDF document...");

}

}

#### PdfDocumentFactory.java

package FactoryMethodPattern;

// File: PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory { @Override

public Document createDocument() {

return new PdfDocument();

}

}

#### WordDocument.java

package FactoryMethodPattern;

// File: WordDocument.java

public class WordDocument implements Document { @Override

public void open() {

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

}

@Override

public void save() {

System.out.println("Saving Word document...");

}

@Override

public void close() {

System.out.println("Closing Word document...");

}

}

#### WordDocumentFactory.java

package FactoryMethodPattern;

// File: WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory { @Override

public Document createDocument() {

return new WordDocument();

}

}

FactoryMethodTest.java package FactoryMethodPattern; public class FactoryMethodTest { public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory(); Document wordDoc = wordFactory.createDocument();

wordDoc.open(); wordDoc.save(); wordDoc.close();

System.out.println();

DocumentFactory pdfFactory = new PdfDocumentFactory(); Document pdfDoc = pdfFactory.createDocument(); pdfDoc.open();

pdfDoc.save(); pdfDoc.close(); System.out.println();

DocumentFactory excelFactory = new ExcelDocumentFactory();

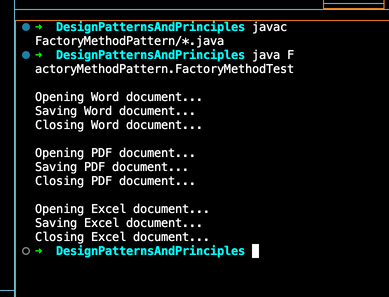
Document excelDoc = excelFactory.createDocument(); excelDoc.open();

excelDoc.save(); excelDoc.close();

}

}

# output



## Exercise 3: Implementing the Builder Pattern

### Code:

Computer.java package BuildPattern; public class Computer { private final String CPU; private final String RAM;

// Optional parameters private final String storage;

private final String graphicsCard;

private final String operatingSystem;

// Private constructor to enforce object creation via builder private Computer(Builder builder) {

this.CPU = builder.CPU; this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

// Static nested Builder class public static class Builder { private final String CPU; private final String RAM; private String storage; private String graphicsCard;

private String operatingSystem;

public Builder(String CPU, String RAM) { this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) { this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) { this.graphicsCard = graphicsCard;

return this;

}

public Builder setOperatingSystem(String operatingSystem) { this.operatingSystem = operatingSystem;

return this;

}

public Computer build() { return new Computer(this);

}

}

public void displayConfiguration() { System.out.println("Computer Configuration:"); System.out.println("CPU: " + CPU);

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

System.out.println("Storage: " + (storage != null ? storage : "Not specified"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not specified"));

System.out.println("Operating System: " + (operatingSystem != null ? operatingSystem : "Not specified"));

}

}

#### BuildPatternTest.java

package BuildPattern;

public class BuildPatternTest {

public static void main(String[] args) {

// Basic configuration

Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build(); basicComputer.displayConfiguration();

System.out.println();

// Advanced configuration

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setOperatingSystem("Windows 11")

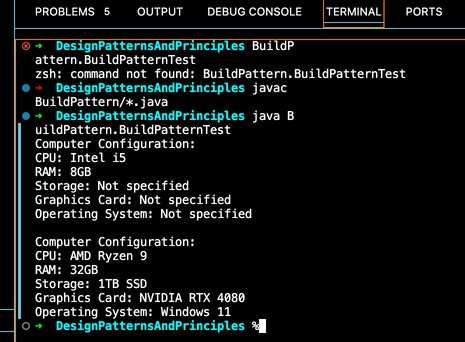
.build();

gamingComputer.displayConfiguration();

}

}

# output



## Exercise 4: Implementing the Adapter Pattern

Code: PaymentProcessor.java package AdapterPattern;

public interface PaymentProcessor { void processPayment(double amount);

}

#### PayPalAdapter.java

package AdapterPattern;

public class PayPalAdapter implements PaymentProcessor { private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) { paypal.makePayment(amount);

}

}

PayPalGateway.java package AdapterPattern; public class PayPalGateway {

public void makePayment(double amountInDollars) {

System.out.println("Processing payment via PayPal: $" + amountInDollars);

}

}

#### StripeAdapter.java

package AdapterPattern;

public class StripeAdapter implements PaymentProcessor { private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) { this.stripe = stripe;

}

@Override

public void processPayment(double amount) { stripe.executePayment(amount);

}

}

StripeGateway.java package AdapterPattern; public class StripeGateway {

public void executePayment(double totalAmount) {

System.out.println("Processing payment via Stripe: $" + totalAmount);

}

}

#### AdapterPatternTest.java

package AdapterPattern;

public class AdapterPatternTest { public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway()); paypalProcessor.processPayment(150.0);

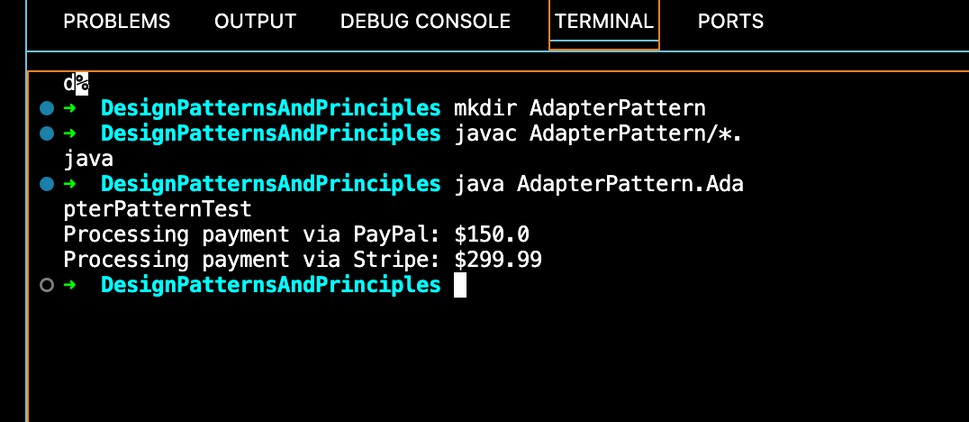
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(299.99);

}

}

# output



Exercise 5: Implementing the Decorator Pattern

Code EmailNotifler.java package DecoratorPattern;

public class EmailNotifier implements Notifier { @Override

public void send(String message) {

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

}

}

#### Notifler.java

package DecoratorPattern; public interface Notifier { void send(String message);

}

#### NotiflerDecorator.java

package DecoratorPattern;

public abstract class NotifierDecorator implements Notifier { protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) { this.wrappee = notifier;

}

@Override

public void send(String message) { wrappee.send(message);

}

}

#### SlackNotiflerDecorator.java

package DecoratorPattern;

public class SlackNotifierDecorator extends NotifierDecorator { public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) { super.send(message); sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

#### SMSNotiflerDecorator.java

package DecoratorPattern;

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);

}

}

#### DecoratorPatternTest.java

package DecoratorPattern;

public class DecoratorPatternTest { public static void main(String[] args) {

// Base notifier

Notifier notifier = new EmailNotifier();

// Add SMS functionality

notifier = new SMSNotifierDecorator(notifier);

// Add Slack functionality

notifier = new SlackNotifierDecorator(notifier);

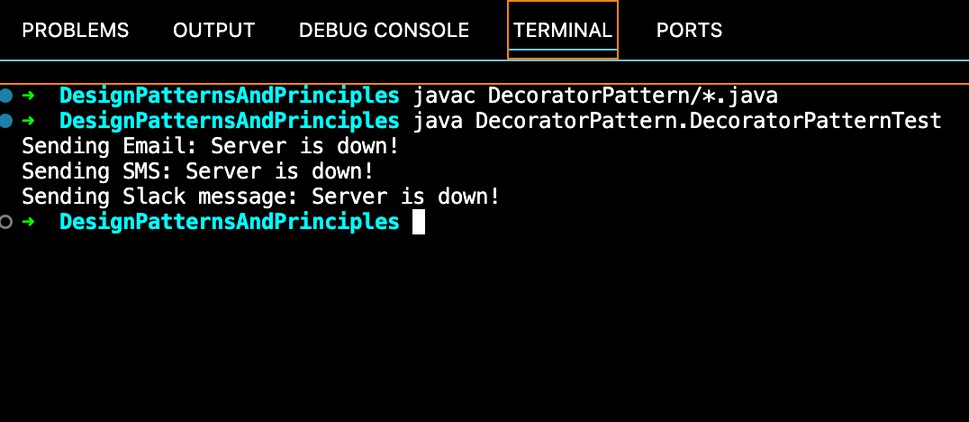
// Send notification

notifier.send("Server is down!");

}

}

# output



Exercise 6: Implementing the Proxy Pattern

### Code

#### Image.java

package ProxyPattern; public interface Image { void display();

}

#### ProxyImage.java

package ProxyPattern;

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); // Lazy loading

} else {

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

}

realImage.display();

}

}

#### RealImage.java

package ProxyPattern;

public class RealImage implements Image { private String filename;

public RealImage(String filename) { this.filename = filename;

loadFromServer();

}

private void loadFromServer() {

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

}

@Override

public void display() {

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

}

}

#### ProxyPatternTest.java

package ProxyPattern;

public class ProxyPatternTest { public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg"); Image image2 = new ProxyImage("photo2.jpg");

// Image will be loaded from remote server image1.display();

System.out.println();

// Image is cached, no reloading image1.display();

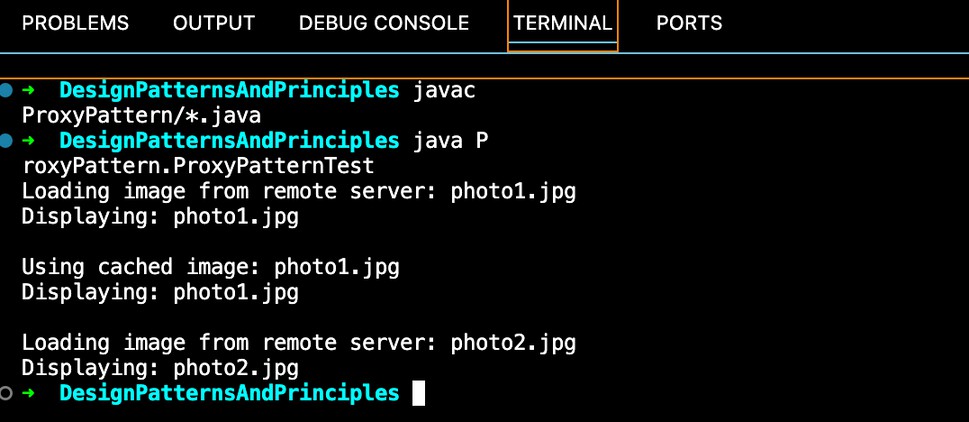
System.out.println();

// Load another image image2.display();

}

}

# output



Exercise 7: Implementing the Observer Pattern

### Code

#### Observer.java

package ObserverPattern; public interface Observer {

void update(String stockName, double price);

}

#### MobileApp.java

package ObserverPattern;

public class MobileApp implements Observer { private String name;

public MobileApp(String name) { this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println(name + " - Mobile App: " + stockName + " updated to $" + price);

}

}

#### Stock.java

package ObserverPattern; public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o); void notifyObservers();

}

StockMarket.java package ObserverPattern; 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 void setStockPrice(String stockName, double stockPrice) { this.stockName = stockName;

this.stockPrice = stockPrice; 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 observer : observers) { observer.update(stockName, stockPrice);

}

}

}

#### WebApp.java

package ObserverPattern;

public class WebApp implements Observer { private String name;

public WebApp(String name) { this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println(name + " - Web App: " + stockName + " updated to $" + price);

}

}

#### ObserverPatternTest.java

package ObserverPattern;

public class ObserverPatternTest { public static void main(String[] args) {

StockMarket stockMarket = new StockMarket(); Observer mobileObserver = new MobileApp("Alice"); Observer webObserver = new WebApp("Bob"); stockMarket.registerObserver(mobileObserver); stockMarket.registerObserver(webObserver); stockMarket.setStockPrice("AAPL", 187.50); System.out.println();

stockMarket.setStockPrice("GOOGL", 2800.00); System.out.println();

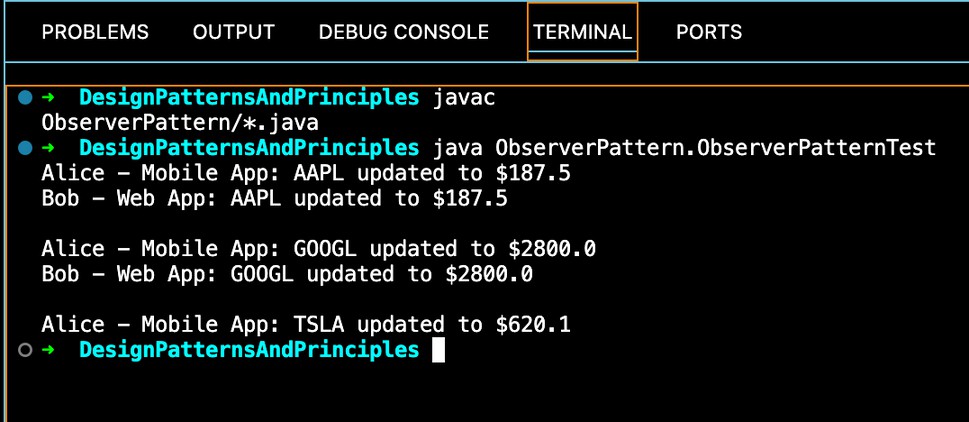
stockMarket.removeObserver(webObserver);

stockMarket.setStockPrice("TSLA", 620.10);

}

}

# output



## Exercise 8: Implementing the Strategy Pattern

Code CreditCardPayment.java package StrategyPattern;

public class CreditCardPayment implements PaymentStrategy { private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) { this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card [" + cardNumber + "] - " + cardHolder);

}

}

PaymentContext.java package StrategyPattern; public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) { this.strategy = strategy;

}

public void payAmount(double amount) { if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

#### PaymentStrategy.java

package StrategyPattern;

public interface PaymentStrategy { void pay(double amount);

}

#### PayPalPayment.java

package StrategyPattern;

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);

}

}

#### StrategyPatternTest.java

package StrategyPattern;

public class StrategyPatternTest { public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9012-3456", "Hemavathy R");

context.setPaymentStrategy(creditCard); context.payAmount(2500.00); System.out.println();

// Pay using PayPal

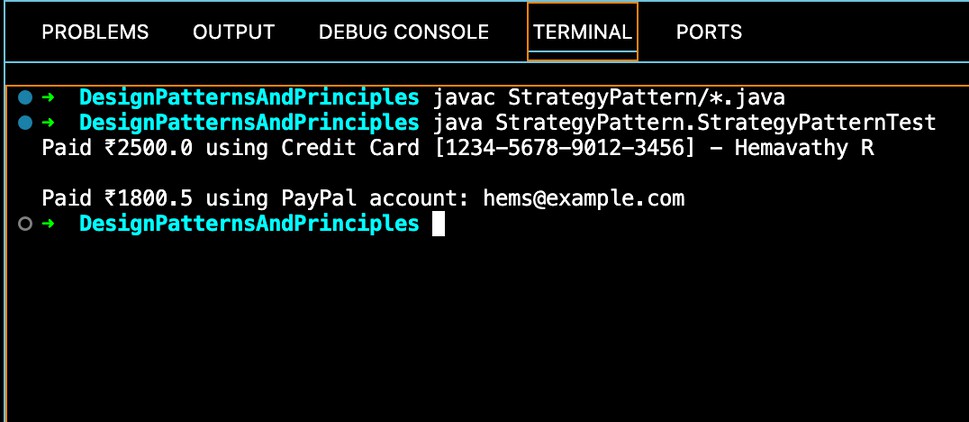
PaymentStrategy paypal = new PayPalPaymen[t("hems@example.com](mailto:hems@example.com)"); context.setPaymentStrategy(paypal);

context.payAmount(1800.50);

}

}

# output



Exercise 9: Implementing the Command Pattern

### Code

#### Command.java

package CommandPattern; public interface Command { void execute();

}

#### Light.java

package CommandPattern; public class Light {

public void turnOn() {

System.out.println("The light is ON");

}

public void turnOff() {

System.out.println("The light is OFF");

}

}

#### LightOffCommand.java

package CommandPattern;

public class LightOffCommand implements Command { private Light light;

public LightOffCommand(Light light) { this.light = light;

}

@Override

public void execute() { light.turnOff();

}

}

#### LightOnCommand.java

package CommandPattern;

public class LightOnCommand implements Command { private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() { light.turnOn();

}

}

RemoteControl.java package CommandPattern; public class RemoteControl { private Command command;

public void setCommand(Command command) { this.command = command;

}

public void pressButton() { if (command != null) { command.execute();

} else {

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

}

}

}

#### CommandPatternTest.java

package CommandPattern;

public class CommandPatternTest { public static void main(String[] args) { Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight); Command lightOff = new LightOffCommand(livingRoomLight); RemoteControl remote = new RemoteControl();

System.out.println("Pressing ON button:"); remote.setCommand(lightOn);

remote.pressButton();

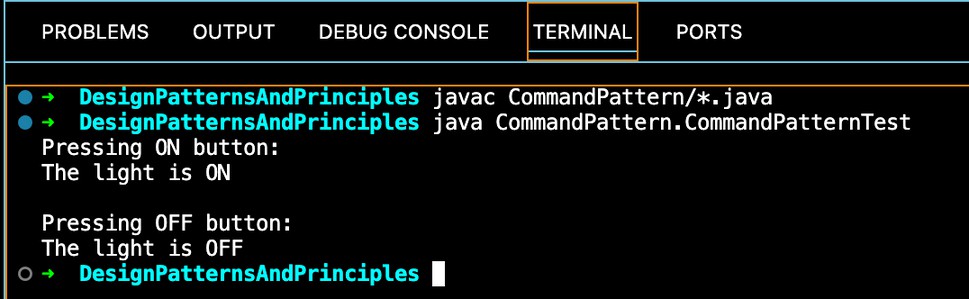
System.out.println("\nPressing OFF button:"); remote.setCommand(lightOff);

remote.pressButton();

}

}

# output



## Exercise 10: Implementing the MVC Pattern

Code Student.java package MVCPattern; public class Student { private String id;

private String name; private String grade;

// Constructor

public Student(String id, String name, String grade) { this.id = id;

this.name = name;

this.grade = grade;

}

// Getters and Setters 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;

}

}

#### StudentController.java

package MVCPattern;

public class StudentController { private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) { this.model = model;

this.view = view;

}

// Model data getters/setters public String getStudentName() { return model.getName();

}

public void setStudentName(String name) { model.setName(name);

}

public String getStudentId() { return model.getId();

}

public void setStudentId(String id) { model.setId(id);

}

public String getStudentGrade() { return model.getGrade();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Update view

public void updateView() {

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

}

}

StudentView.java package MVCPattern; public class StudentView {

public void displayStudentDetails(String id, String name, String grade) { System.out.println("Student Details:");

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

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

}

}

#### MVCPatternTest.java

package MVCPattern;

public class MVCPatternTest {

public static void main(String[] args) {

// Create Model

Student student = new Student("S001", "Hemavathy", "A");

// Create View

StudentView view = new StudentView();

// Create Controller

StudentController controller = new StudentController(student, view);

// Display initial student data controller.updateView();

System.out.println("\nUpdating student name and grade...\n");

// Update student data

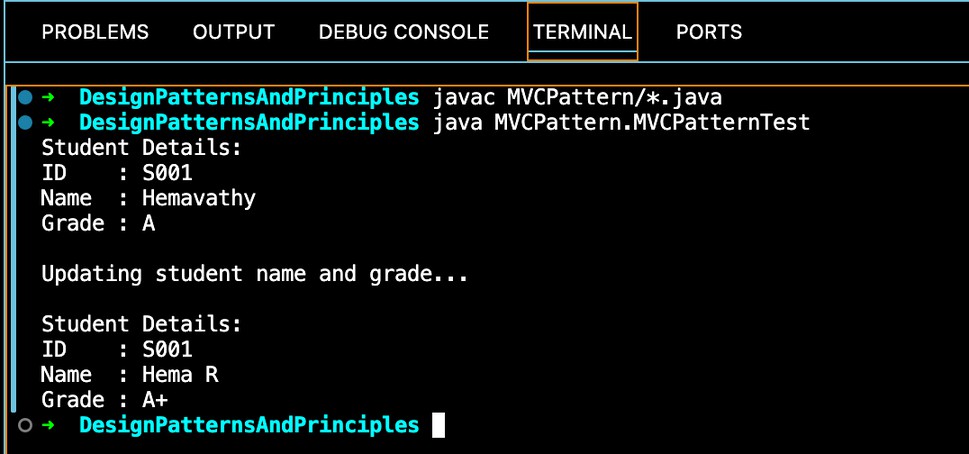
controller.setStudentName("Hema R"); controller.setStudentGrade("A+");

// Display updated data controller.updateView();

}

}

# output



## Exercise 11: Implementing Dependency Injection

### Code

#### Customer.java

package DependencyInjection; public class Customer { private int id;

private String name;

// Constructor

public Customer(int id, String name) { this.id = id;

this.name = name;

}

// Getters

public int getId() { return id;

}

public String getName() { return name;

}

}

#### CustomerRepository.java

package DependencyInjection;

public interface CustomerRepository { Customer findCustomerById(int id);

}

#### CustomerRepositoryImpl.java

package DependencyInjection;

public class CustomerRepositoryImpl implements CustomerRepository { @Override

public Customer findCustomerById(int id) {

// Dummy data

return new Customer(id, "Hemavathy");

}

}

CustomerService.java package DependencyInjection; public class CustomerService {

private final CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) { this.customerRepository = customerRepository;

}

public void showCustomerDetails(int id) {

Customer customer = customerRepository.findCustomerById(id); System.out.println("Customer ID : " + customer.getId()); System.out.println("Customer Name : " + customer.getName());

}

}

DIExampleTest.java package DependencyInjection; public class DIExampleTest {

public static void main(String[] args) {

// Manual Dependency Injection

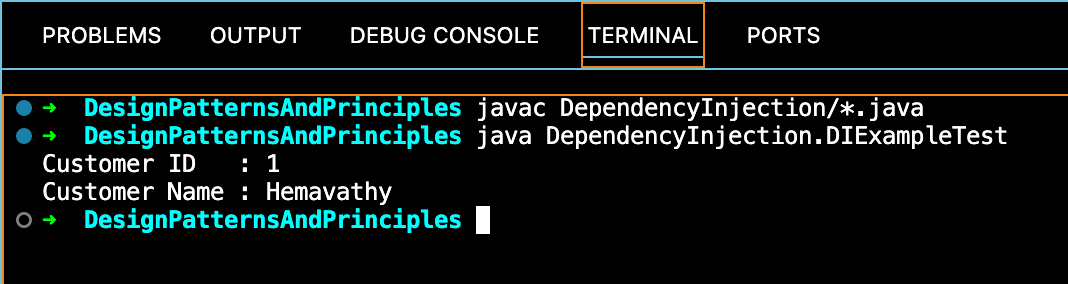
CustomerRepository repo = new CustomerRepositoryImpl(); CustomerService service = new CustomerService(repo);

// Test

service.showCustomerDetails(1);

}

}

output