**Cognizant DN 4.0 Deep Skilling – Java FSE**

**Week 1: Design Patterns and Principles Hands On**

**Exercise 1: Implementing Singleton Pattern**

**Solution:**

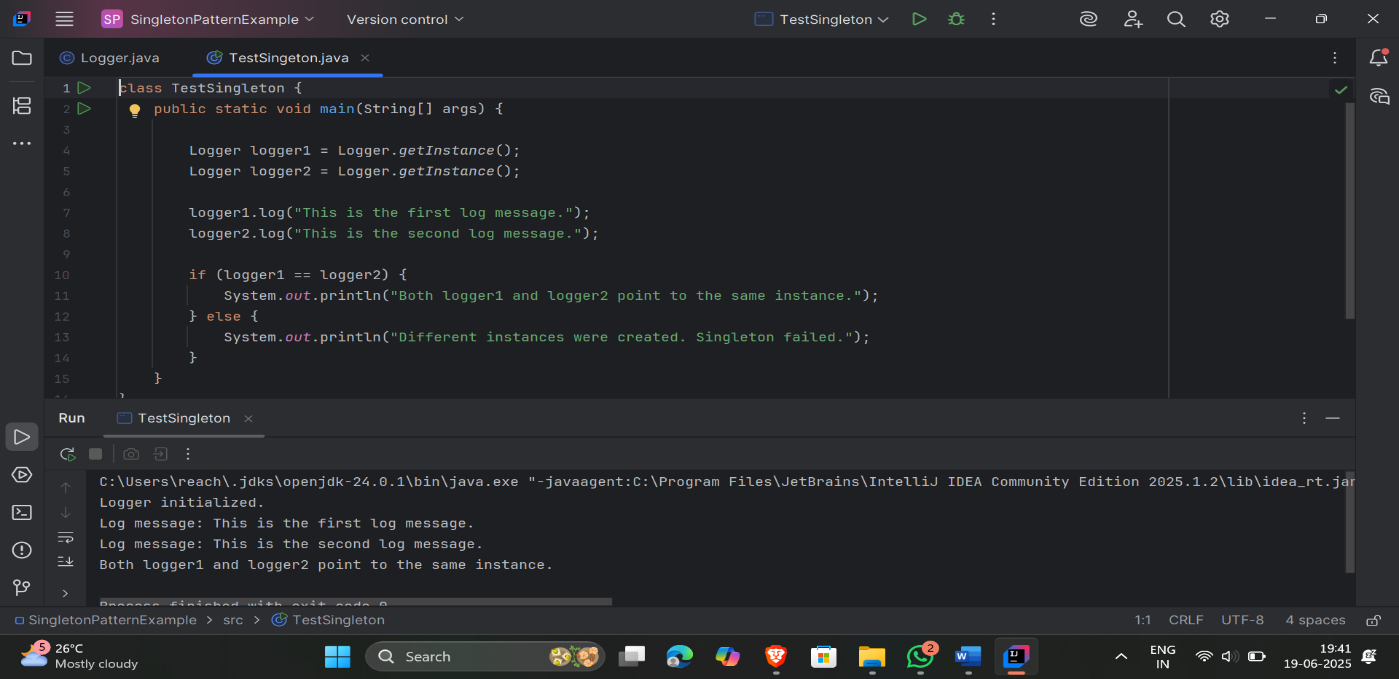
**Logger.java**

|  |
| --- |
| public class Logger {  private static Logger *instance*;  private Logger() {  System.*out*.println("Logger initialized.");  }  public static Logger getInstance() {  if (*instance* == null) {  *instance* = new Logger();  }  return *instance*;  }  public void log(String message) {  System.*out*.println("Log message: " + message);  } } |

**TestSingleton.java**

|  |
| --- |
| class TestSingleton {  public static void main(String[] args) {  Logger logger1 = Logger.*getInstance*();  Logger logger2 = Logger.*getInstance*();  logger1.log("This is the first log message.");  logger2.log("This is the second log message.");  if (logger1 == logger2) {  System.*out*.println("Both logger1 and logger2 point to the same instance.");  } else {  System.*out*.println("Different instances were created. Singleton failed.");  }  } } |

**Output**

****

**Exercise 2: Implementing Factory Method Pattern**

**Solution:**

1. **Document.java**

public interface Document {  
 void open();  
}

1. **WordDocument.java**

public class WordDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a Word document.");  
 }  
}

1. **PdfDocument.java**

public class PdfDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a PDF document.");  
 }  
}

1. **ExcelDocument.java**

public class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening an Excel document.");  
 }  
}

1. **DocumentFactory.java**

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

1. **WordDocumentFactory.java**

class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}

1. **PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}

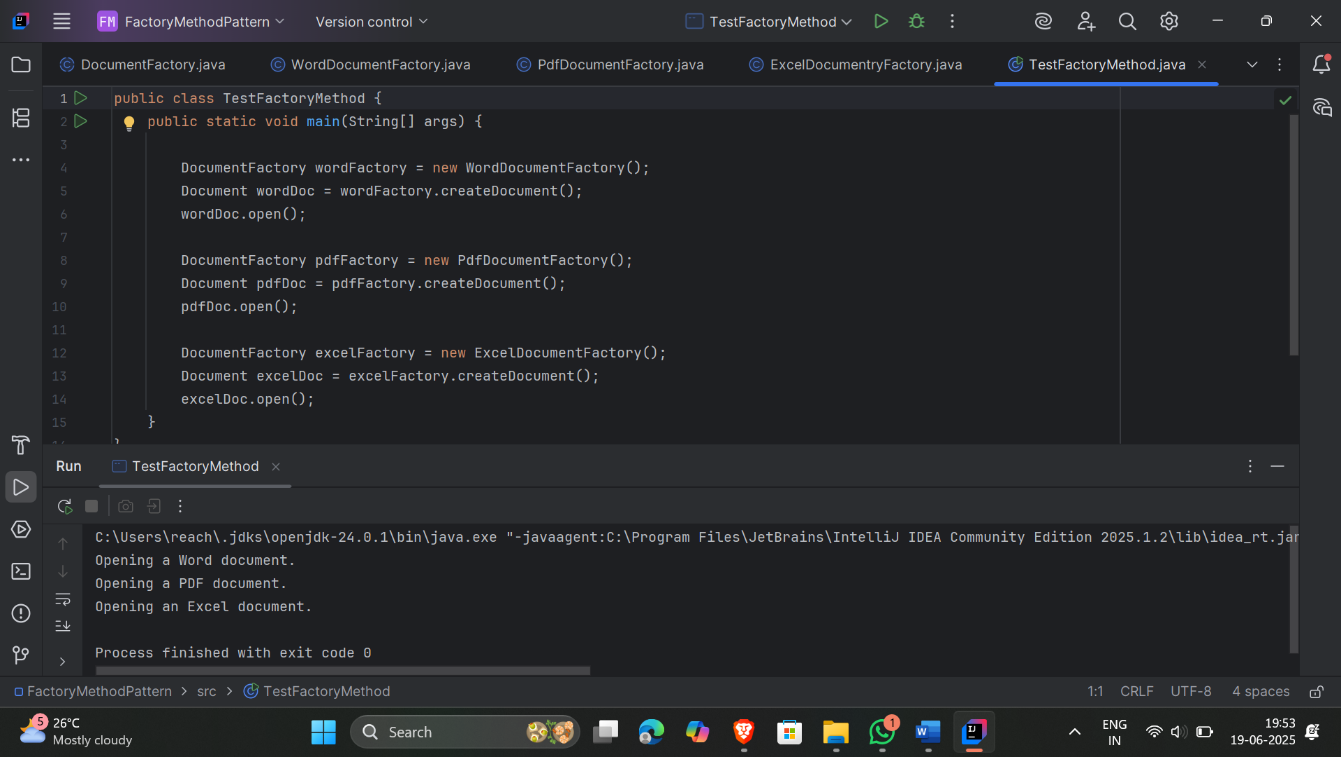
1. **ExcelDocumentFactory.java**

class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

1. **TestFactoryMethod.java**

public class TestFactoryMethod {  
 public static void main(String[] args) {  
  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 Document wordDoc = wordFactory.createDocument();  
 wordDoc.open();  
  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 Document pdfDoc = pdfFactory.createDocument();  
 pdfDoc.open();  
  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
 Document excelDoc = excelFactory.createDocument();  
 excelDoc.open();  
 }  
}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

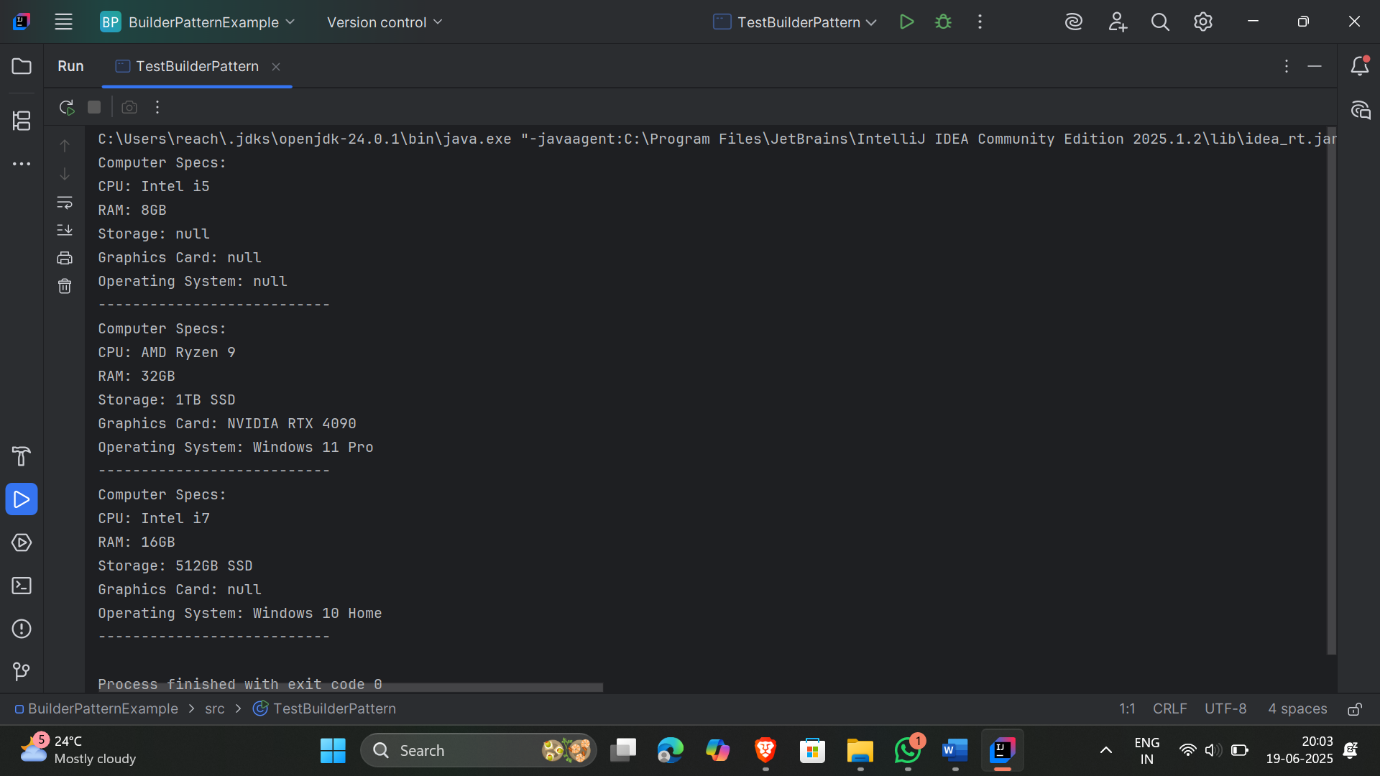
**Solution: Computer.java**

|  |
| --- |
| public class Computer {  private String CPU;  private String RAM;  private String storage;  private String graphicsCard;  private String operatingSystem;  private Computer(Builder builder) {  this.CPU = builder.CPU;  this.RAM = builder.RAM;  this.storage = builder.storage;  this.graphicsCard = builder.graphicsCard;  this.operatingSystem = builder.operatingSystem;  }  public static class Builder {  private String CPU;  private 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 showSpecs() {  System.out.println("Computer Specs:");  System.out.println("CPU: " + CPU);  System.out.println("RAM: " + RAM);  System.out.println("Storage: " + storage);  System.out.println("Graphics Card: " + graphicsCard);  System.out.println("Operating System: " + operatingSystem);  System.out.println("---------------------------");  }  } |

**TestBuilderPattern.java**

|  |
| --- |
| public class TestBuilderPattern {  public static void main(String[] args) {  Computer basicComputer = new Computer.Builder("Intel i5", "8GB")  .build();  Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")  .setStorage("1TB SSD")  .setGraphicsCard("NVIDIA RTX 4090")  .setOperatingSystem("Windows 11 Pro")  .build();  Computer ultrabook = new Computer.Builder("Intel i7", "16GB")  .setStorage("512GB SSD")  .setOperatingSystem("Windows 10 Home")  .build();  basicComputer.showSpecs();  gamingComputer.showSpecs();  ultrabook.showSpecs();  } } |

**Output**

****

**Exercise 4: Implementing the Adapter Pattern**

**Solution:**

**PaymentProcessor.java**

|  |
| --- |
| public interface PaymentProcessor {  void processPayment(double amount); } |

**RazorpayAdapter.java**

|  |
| --- |
| public class RazorpayAdapter implements PaymentProcessor {  private RazorpayGateway razorpayGateway;   public RazorpayAdapter(RazorpayGateway gateway) {  this.razorpayGateway = gateway;  }   @Override  public void processPayment(double amount) {  razorpayGateway.executeTransaction(amount);  } } |

**StripeAdapter.java**

|  |
| --- |
| public class RazorpayAdapter implements PaymentProcessor {  private RazorpayGateway razorpayGateway;   public RazorpayAdapter(RazorpayGateway gateway) {  this.razorpayGateway = gateway;  }   @Override  public void processPayment(double amount) {  razorpayGateway.executeTransaction(amount);  } } |

**PayPalAdapter.java**

|  |
| --- |
| public class PayPalAdapter implements PaymentProcessor {  private PayPalGateway payPalGateway;   public PayPalAdapter(PayPalGateway gateway) {  this.payPalGateway = gateway;  }   @Override  public void processPayment(double amount) {  payPalGateway.makePayment(amount);  } } |

**RazorpayGateway.java**

|  |
| --- |
| public class RazorpayGateway {  public void executeTransaction(double rupees) {  System.*out*.println("₹" + rupees + " paid using Razorpay.");  } } |

**StripeGateway.java**

|  |
| --- |
| public class StripeGateway {  public void pay(double amountInDollars) {  System.*out*.println("Stripe processed payment of $" + amountInDollars + ".");  } } |

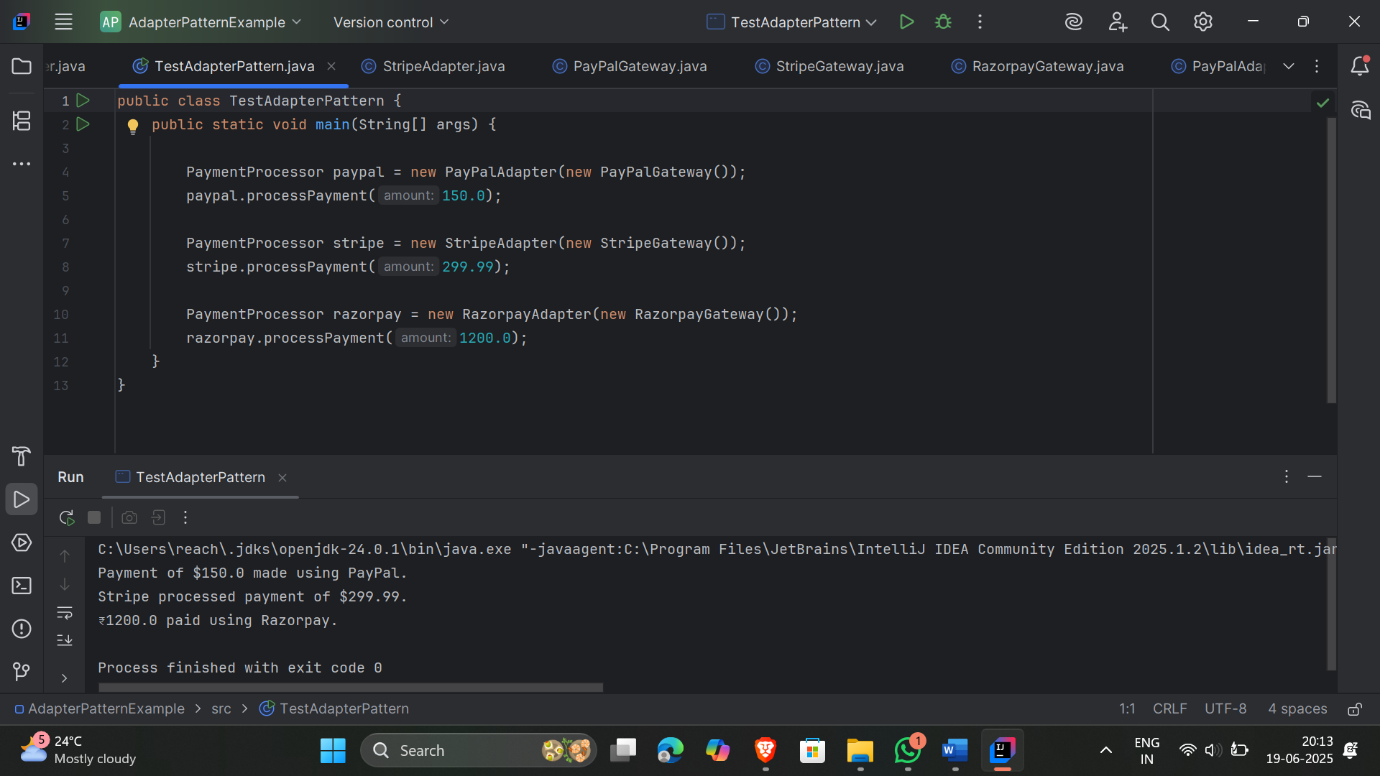
**PayPalGateway.java**

|  |
| --- |
| public class PayPalGateway {  public void makePayment(double amountInUSD) {  System.*out*.println("Payment of $" + amountInUSD + " made using PayPal.");  } } |

**TestAdapterPattern.java**

|  |
| --- |
| public class TestAdapterPattern {  public static void main(String[] args) {   PaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());  paypal.processPayment(150.0);   PaymentProcessor stripe = new StripeAdapter(new StripeGateway());  stripe.processPayment(299.99);   PaymentProcessor razorpay = new RazorpayAdapter(new RazorpayGateway());  razorpay.processPayment(1200.0);  } } |

**Output**

****

**Exercise 5: Implementing the Decorator Pattern**

**Solution:**

**Notifier.java**

|  |
| --- |
| public interface Notifier {  void send(String message); } |

**EmailNotifier.java**

|  |
| --- |
| public class EmailNotifier implements Notifier {  @Override  public void send(String message) {  System.*out*.println("Sending Email: " + message);  } } |

**NotifierDecortor.java**

|  |
| --- |
| public abstract class NotifierDecorator implements Notifier {  protected Notifier wrappedNotifier;  public NotifierDecorator(Notifier notifier) {  this.wrappedNotifier = notifier;  }  @Override  public void send(String message) {  wrappedNotifier.send(message);  } } |

**SMSNotifierDecorator.java**

|  |
| --- |
| public class SMSNotifierDecorator extends NotifierDecorator {  public SMSNotifierDecorator(Notifier notifier) {  super(notifier);  }  @Override  public void send(String message) {  super.send(message); // Send base notification  sendSMS(message); // Additional functionality  }  private void sendSMS(String message) {  System.*out*.println("Sending SMS: " + message);  } } |

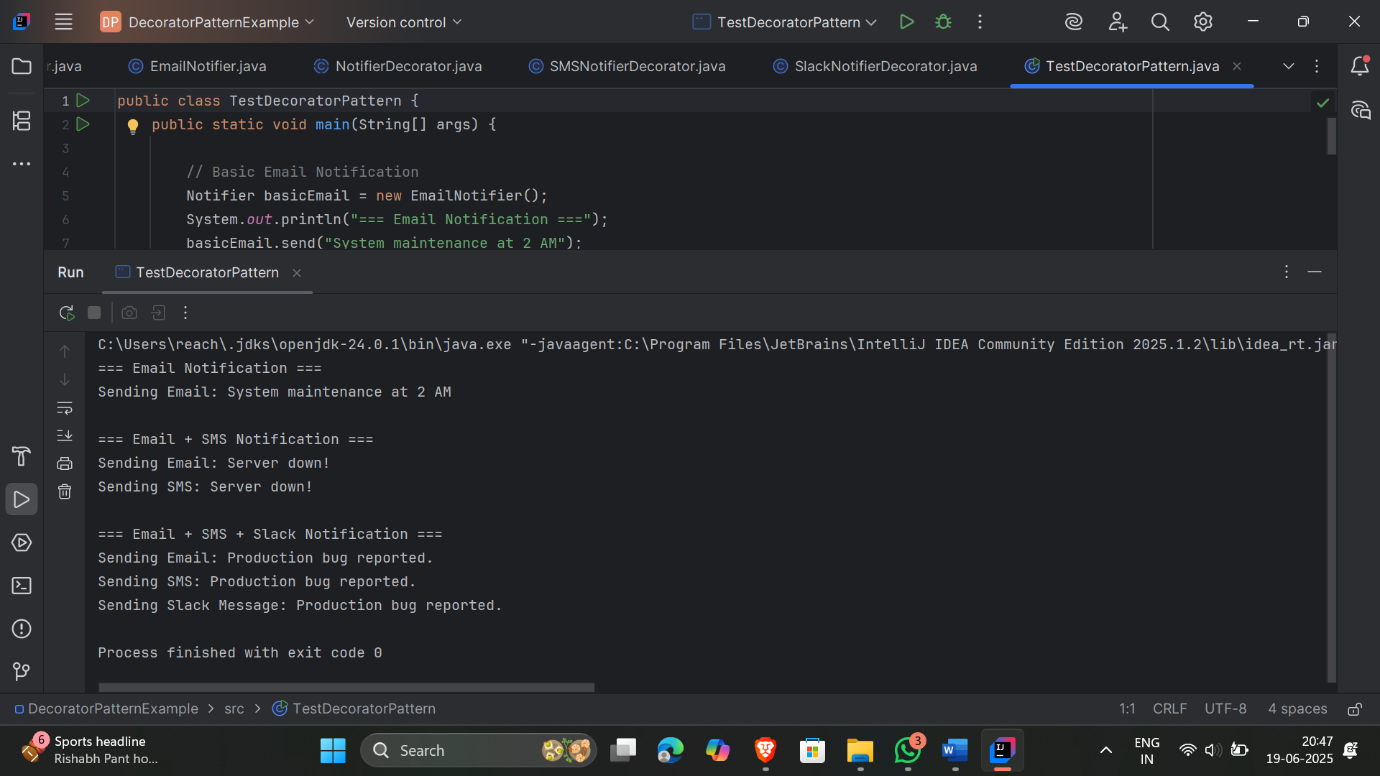
**SlackNotifierDecorator.java**

|  |
| --- |
| 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);  } } |

**TestDecoratorPattern.java**

|  |
| --- |
| public class TestDecoratorPattern {  public static void main(String[] args) {  Notifier basicEmail = new EmailNotifier();  System.*out*.println("=== Email Notification ===");  basicEmail.send("System maintenance at 2 AM");  Notifier emailAndSMS = new SMSNotifierDecorator(new EmailNotifier());  System.*out*.println("\n=== Email + SMS Notification ===");  emailAndSMS.send("Server down!");  Notifier fullNotification = new SlackNotifierDecorator(  new SMSNotifierDecorator(  new EmailNotifier()));  System.*out*.println("\n=== Email + SMS + Slack Notification ===");  fullNotification.send("Production bug reported.");  } } |

**Output**

****

**Exercise 6: Proxy Pattern**

**Solution:**

**Image.java**

|  |
| --- |
| public interface Image {  void display(); } |

**RealImage.java**

|  |
| --- |
| public 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);  } } |

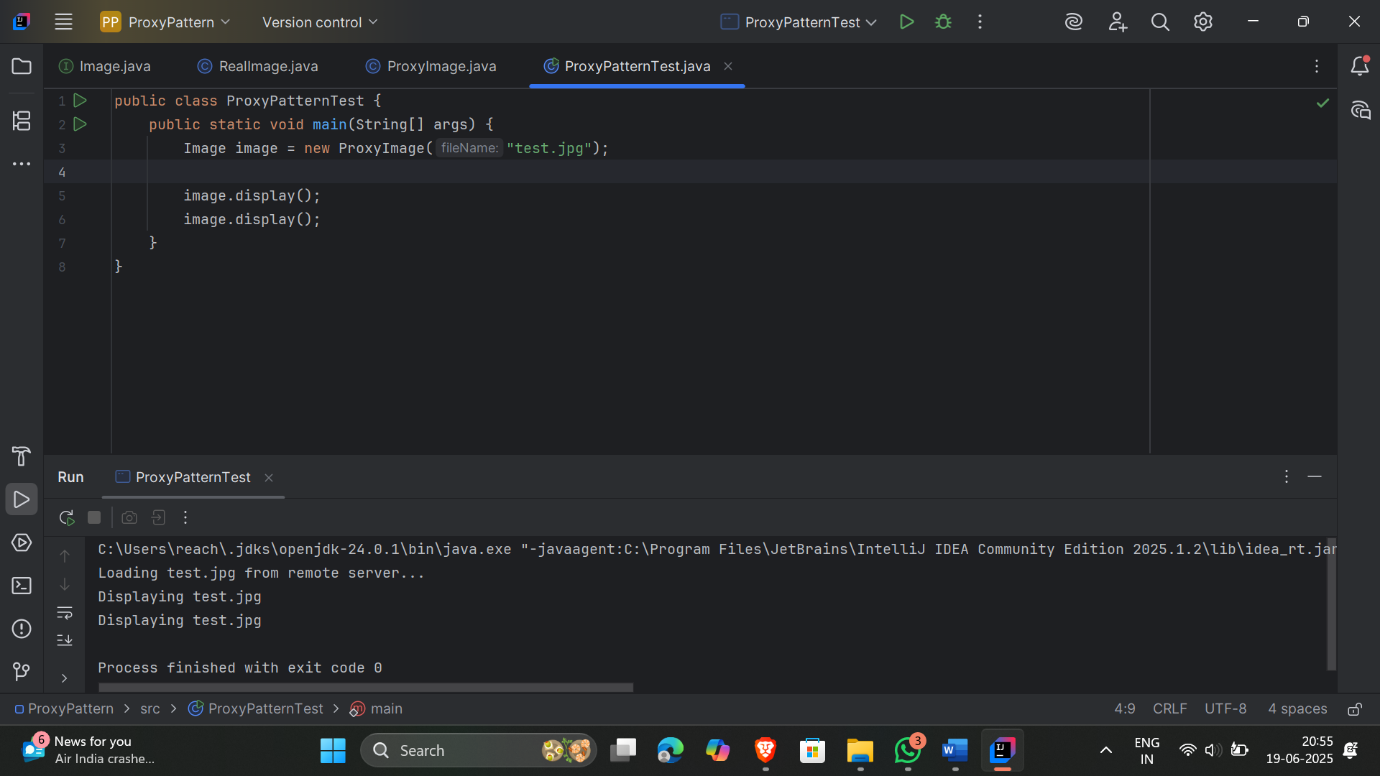
**ProxyImage.java**

|  |
| --- |
| public 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();  } } |

**ProxyPatternTest.java**

|  |
| --- |
| public class ProxyPatternTest {  public static void main(String[] args) {  Image image = new ProxyImage("test.jpg");  image.display();  image.display();  } } |

**Output**

****

**Exercise 7: Observer Pattern Example**

**Solution:**

**Stock.java**

|  |
| --- |
| public interface Stock extends Observer {  void register(Observer o);  void deregister(Observer o);  void notifyObservers(); } |

**Observer.java**

|  |
| --- |
| public interface Observer {  void update(double price); } |

**StockMarket.java**

|  |
| --- |
| import java.util.\*; public abstract class StockMarket implements Stock {  private List<Observer> observers = new ArrayList<>();  private double price;  public void setPrice(double price) {  this.price = price;  notifyObservers();  }  public double getPrice() {  return price;  }  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);  }  } } |

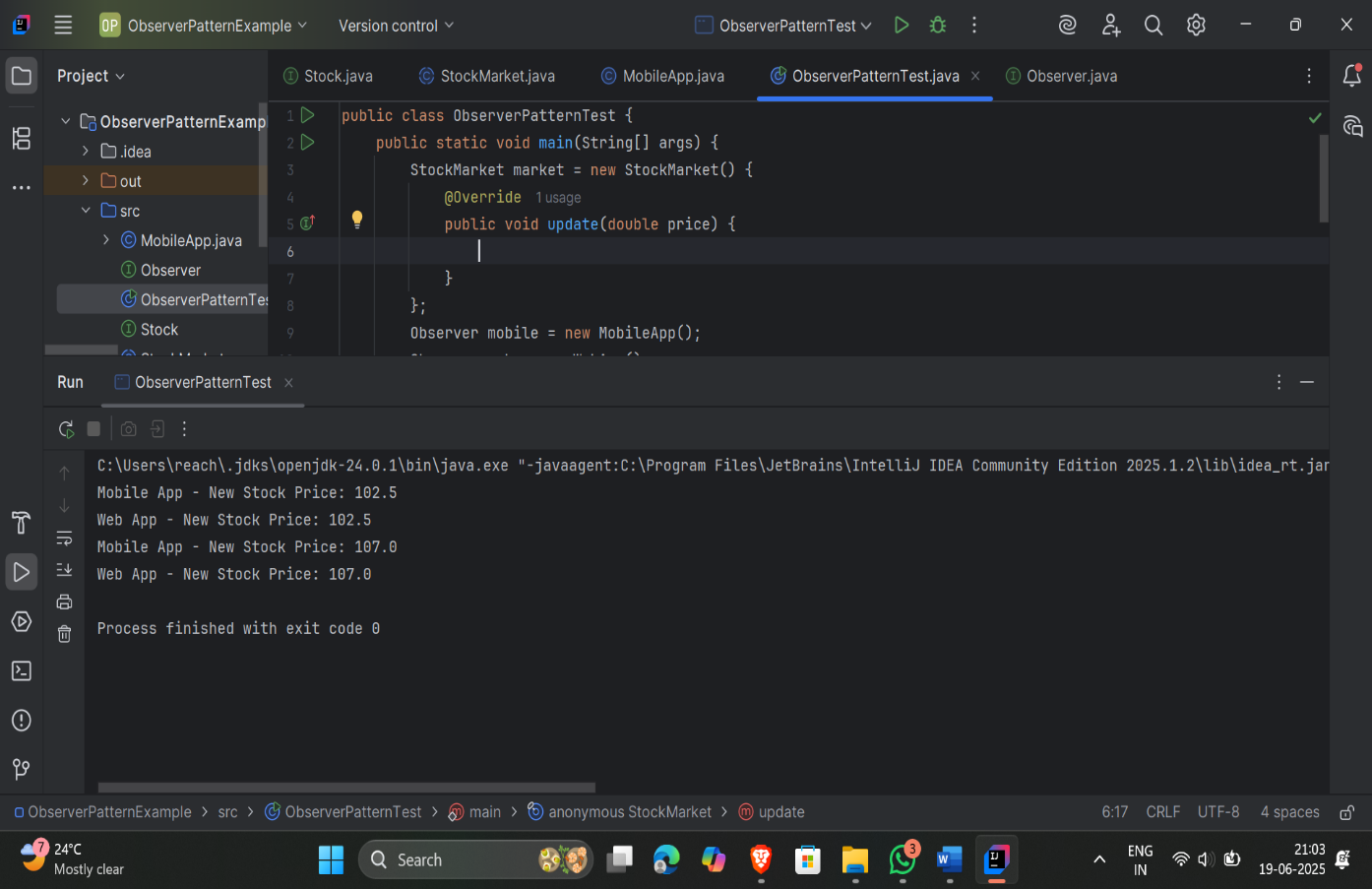
**MobileApp.java**

|  |
| --- |
| public class MobileApp implements Observer {  public void update(double price) {  System.*out*.println("Mobile App - New Stock Price: " + price);  } } class WebApp implements Observer {  public void update(double price) {  System.*out*.println("Web App - New Stock Price: " + price);  } } |

**ObserverPatternTest.java**

|  |
| --- |
| public class ObserverPatternTest {  public static void main(String[] args) {  StockMarket market = new StockMarket() {  @Override  public void update(double price) {  }  };  Observer mobile = new MobileApp();  Observer web = new WebApp();  market.register(mobile);  market.register(web);  market.setPrice(102.5);  market.setPrice(107.0);  } } |

**Output**

****

**Exercise 8: Strategy Pattern Example**

**Solution:**

**PaymentStrategy.java**

|  |
| --- |
| public interface PaymentStrategy {  void pay(double amount); } |

**CreditCardPayment.java**

|  |
| --- |
| public 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.");  } } |

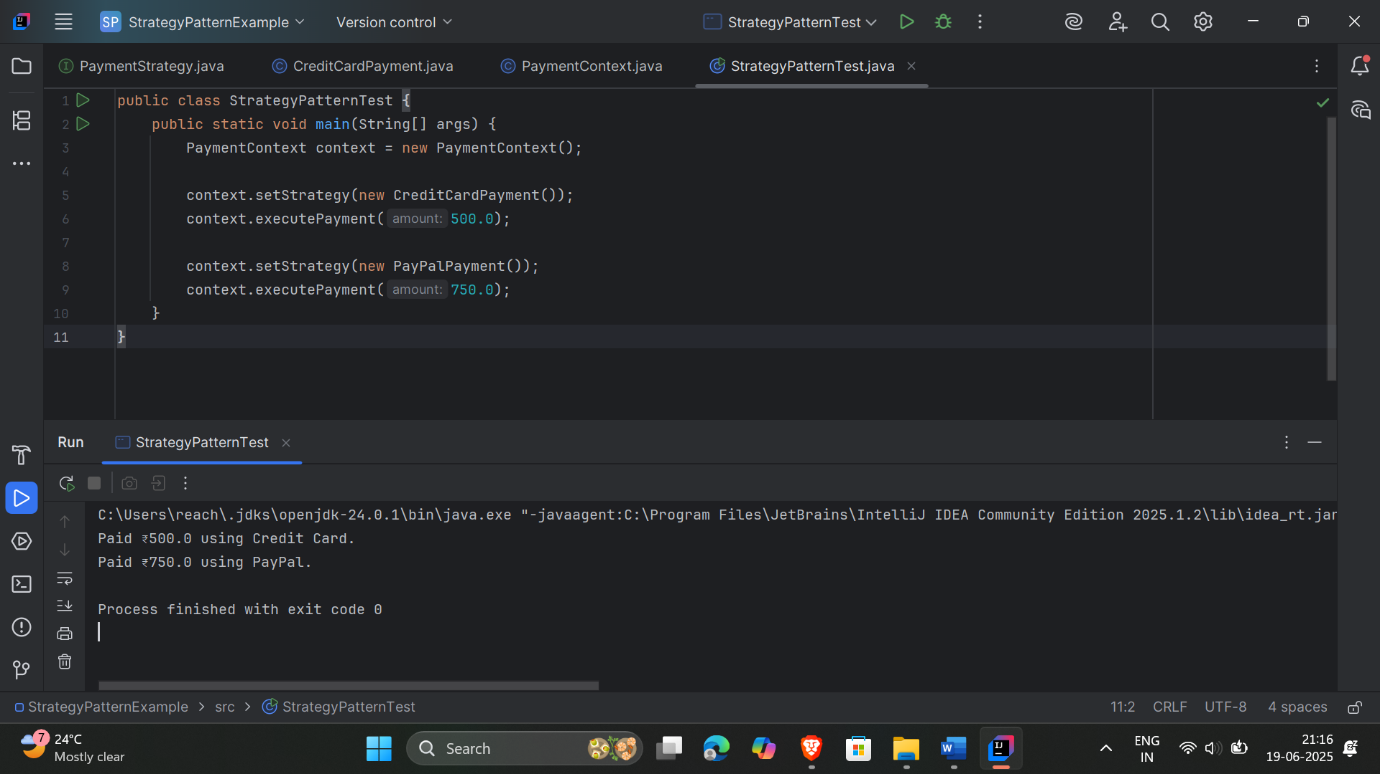
**PaymentContext.java**

|  |
| --- |
| public class PaymentContext {  private PaymentStrategy strategy;  public void setStrategy(PaymentStrategy strategy) {  this.strategy = strategy;  }  public void executePayment(double amount) {  strategy.pay(amount);  } } |

**StrategyPatternTest.java**

|  |
| --- |
| public class StrategyPatternTest {  public static void main(String[] args) {  PaymentContext context = new PaymentContext();  context.setStrategy(new CreditCardPayment());  context.executePayment(500.0);  context.setStrategy(new PayPalPayment());  context.executePayment(750.0);  } } |

**Output**

****

**Exercise 9: Command Pattern Example**

**Solution:**

**Command.java**

|  |
| --- |
| public interface Command {  void execute(); } |

**Light.java**

|  |
| --- |
| public class Light {  public void on() {  System.*out*.println("Light is ON");  }  public void off() {  System.*out*.println("Light is OFF");  } } |

**LightOnCommand.java**

|  |
| --- |
| public class LightOnCommand implements Command {  private Light light;  public LightOnCommand(Light light) {  this.light = light;  }  public void execute() {  light.on();  } } class LightOffCommand implements Command {  private Light light;  public LightOffCommand(Light light) {  this.light = light;  }  public void execute() {  light.off();  } } |

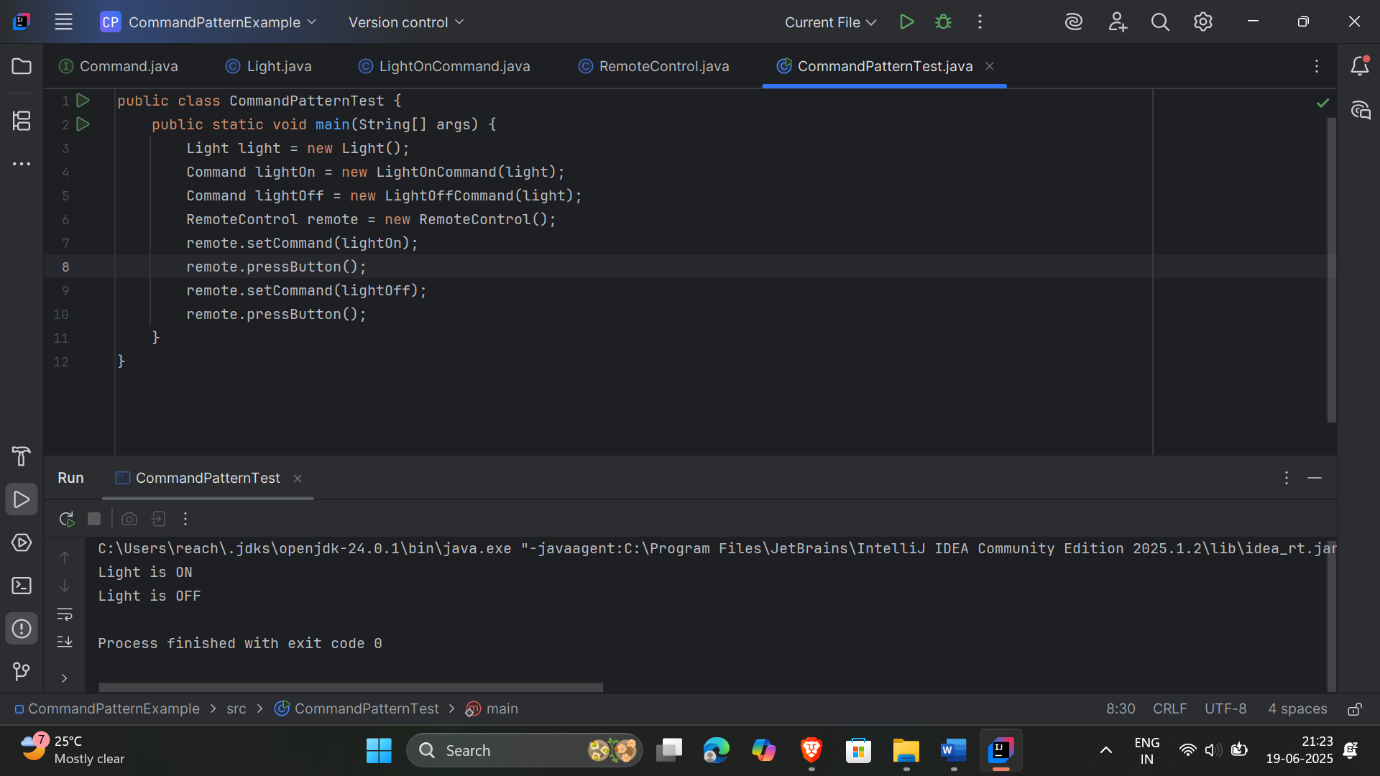
**RemoteControl.java**

|  |
| --- |
| public class RemoteControl {  private Command command;  public void setCommand(Command command) {  this.command = command;  }  public void pressButton() {  command.execute();  } } |

**CommandPatternTest.java**

|  |
| --- |
| public class CommandPatternTest {  public static void main(String[] args) {  Light light = new Light();  Command lightOn = new LightOnCommand(light);  Command lightOff = new LightOffCommand(light);  RemoteControl remote = new RemoteControl();  remote.setCommand(lightOn);  remote.pressButton();  remote.setCommand(lightOff);  remote.pressButton();  } } |

**Output**

****

**Exercise 10: MVC Pattern Example**

**Solution:**

**Student.java**

|  |
| --- |
| public class Student {  private String name;  private String id;  private String grade;  public Student(String name, String id, String grade) {  this.name = name;  this.id = id;  this.grade = grade;  }  public String getName() {  return name;  }  public String getId() {  return id;  }  public String getGrade() {  return grade;  }  public void setName(String name) {  this.name = name;  }  public void setId(String id) {  this.id = id;  }  public void setGrade(String grade) {  this.grade = grade;  } } |

**StudentView.java**

|  |
| --- |
| public class StudentView {  public void displayStudentDetails(Student student) {  System.*out*.println("=== Student Details ===");  System.*out*.println("Name: " + student.getName());  System.*out*.println("ID: " + student.getId());  System.*out*.println("Grade: " + student.getGrade());  System.*out*.println("=======================");  } } |

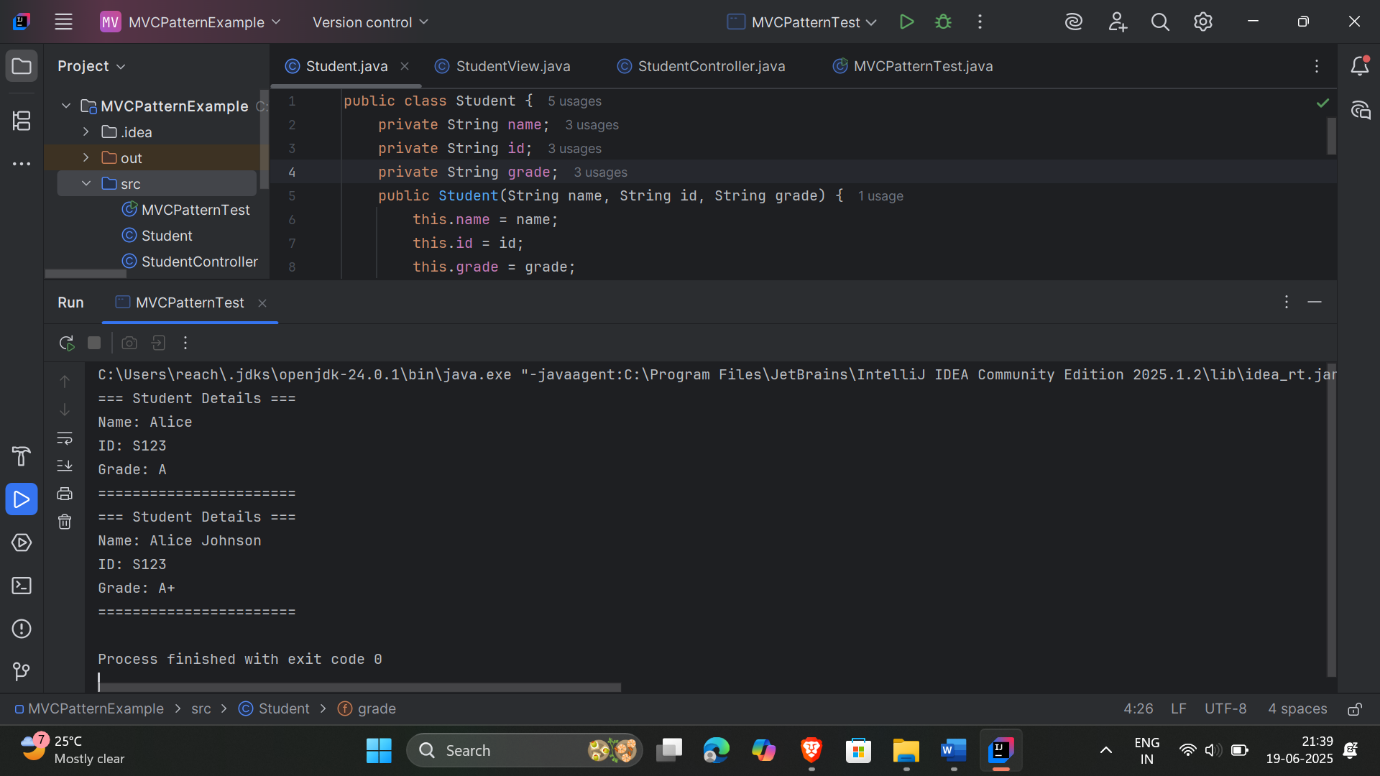
**StudentController.java**

|  |
| --- |
| public class StudentController {  private Student model;  private StudentView view;  public StudentController(Student model, StudentView view) {  this.model = model;  this.view = view;  }  public void updateView() {  view.displayStudentDetails(model);  }  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 String getStudentName() {  return model.getName();  }  public String getStudentId() {  return model.getId();  }  public String getStudentGrade() {  return model.getGrade();  } } |

**MVCPatternTest.java**

|  |
| --- |
| public class MVCPatternTest {  public static void main(String[] args) {  Student student = new Student("Alice", "S123", "A");  StudentView view = new StudentView();  StudentController controller = new StudentController(student, view);  controller.updateView();  controller.setStudentGrade("A+");  controller.setStudentName("Alice Johnson");  controller.updateView();  } } |

**Output**

****

**Exercise 11: Dependency Injection Example**

**Solution:**

**CustomerRepository.java**

|  |
| --- |
| public interface CustomerRepository {  String findCustomerById(String id); } |

**CustomerRepositoryImpl.java**

|  |
| --- |
| public class CustomerRepositoryImpl implements CustomerRepository {  public String findCustomerById(String id) {  return "Customer[id=" + id + ", name=John Doe]";  } } |

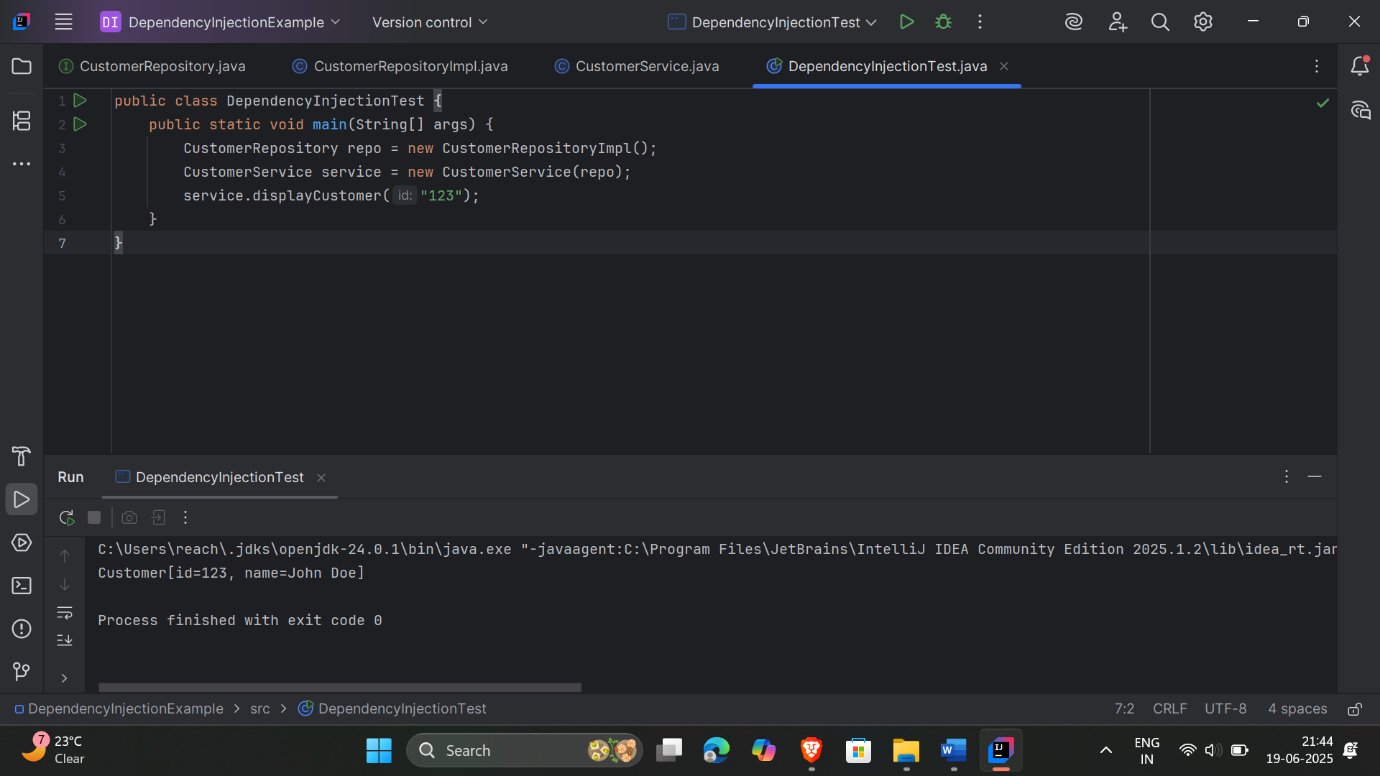
**CustomerService.java**

|  |
| --- |
| public class CustomerService {  private CustomerRepository repository;  public CustomerService(CustomerRepository repository) {  this.repository = repository;  }  public void displayCustomer(String id) {  System.*out*.println(repository.findCustomerById(id));  } } |

**DependencyInjectionTest.java**

|  |
| --- |
| public class DependencyInjectionTest {  public static void main(String[] args) {  CustomerRepository repo = new CustomerRepositoryImpl();  CustomerService service = new CustomerService(repo);  service.displayCustomer("123");  } } |

**Output**

****