**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Project: SingletonPatternExample**

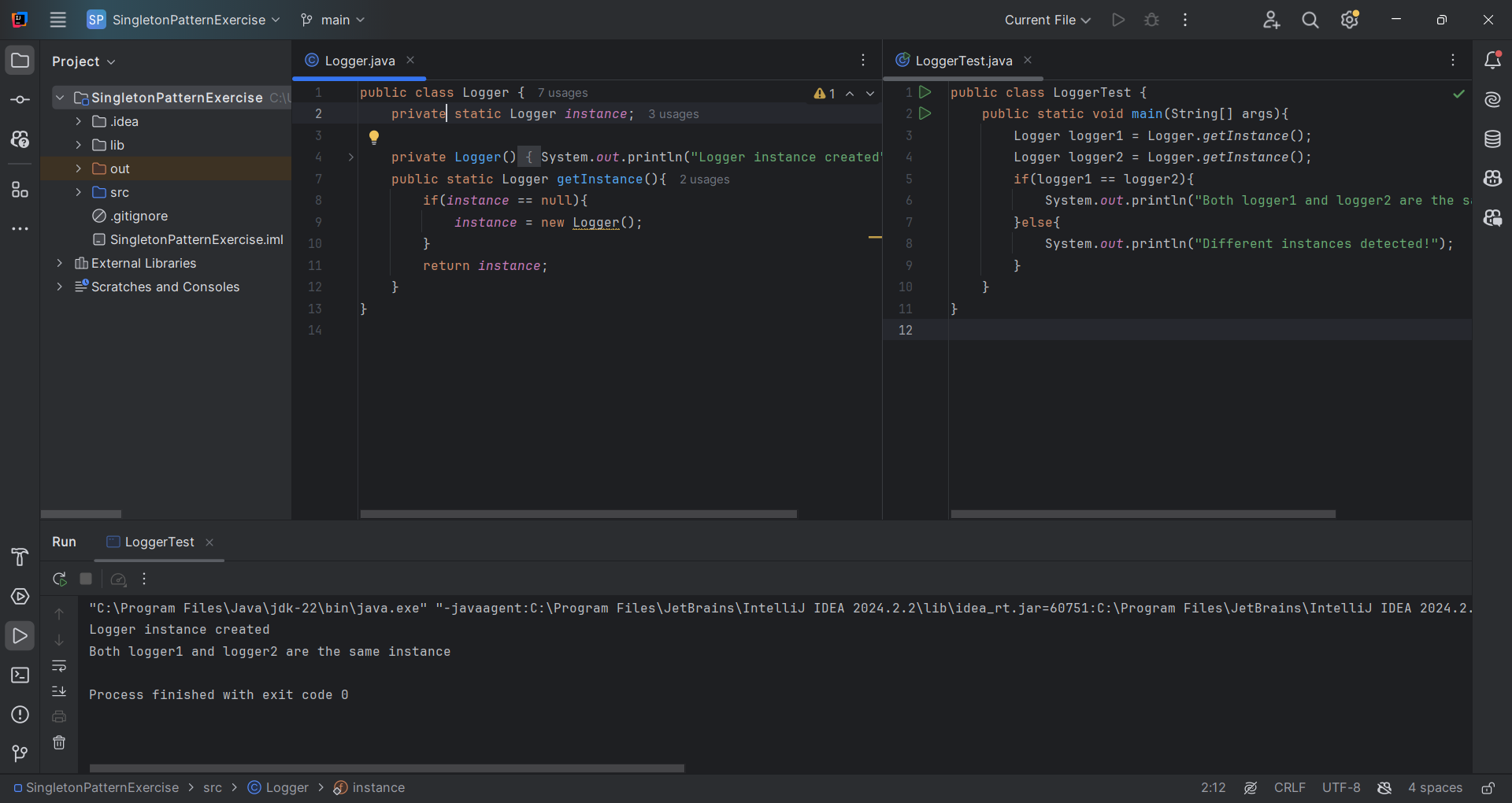
**Logger.java**

public class Logger {  
 private static Logger *instance*;  
  
 private Logger(){  
 System.*out*.println("Logger instance created");  
 }  
 public static Logger getInstance(){  
 if(*instance* == null){  
 synchronized(Logger.class){  
 if(*instance* == null){  
 *instance* = new Logger();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

**LoggerTest.java**

public class LoggerTest {  
 public static void main(String[] args){  
 Logger logger1 = Logger.*getInstance*();  
 Logger logger2 = Logger.*getInstance*();  
   
 if(logger1 == logger2){  
 System.*out*.println("Both logger1 and logger2 are the same instance");  
 }else{  
 System.*out*.println("Different instances detected!");  
 }  
 }  
}

**Output:**



**Explanation:**

A new Project named SingletonPatternExample is created.

The Logger.java class ensures that only one instance is ever created during the application’s lifetime.

This is achieved by making the constructor private, so it can only be accessed within the class itself. This prevents objects from being created using the new keyword from outside the class.

A public static method called getInstance() is defined to return the single instance of the class. This method creates an object only if the instance is null (if the object hasn’t already been created).

The LoggerTest.java class is used to test whether multiple objects are created. Two references are obtained by calling the getInstance() method from the Logger class. Both references should point to the same instance. Otherwise, the Singleton pattern has not been implemented correctly.

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Project: FactoryMethodPatternExample:**

**Document.java:**

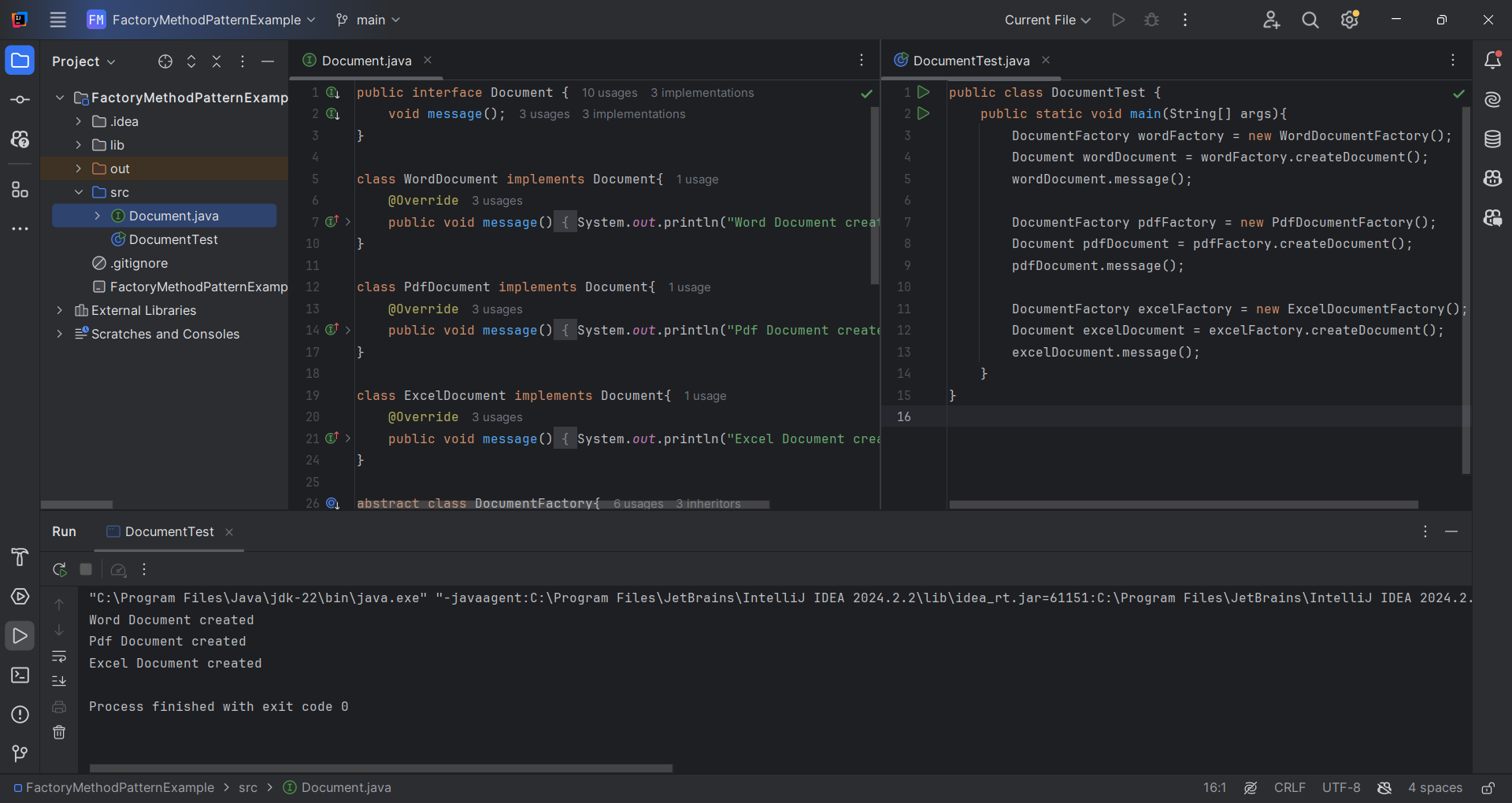
public interface Document {  
 void message();  
}  
  
class WordDocument implements Document{  
 @Override  
 public void message(){  
 System.*out*.println("Word Document created");  
 }  
}  
  
class PdfDocument implements Document{  
 @Override  
 public void message(){  
 System.*out*.println("Pdf Document created");  
 }  
}  
  
class ExcelDocument implements Document{  
 @Override  
 public void message(){  
 System.*out*.println("Excel Document created");  
 }  
}  
  
abstract class DocumentFactory{  
 public abstract Document createDocument();  
}  
  
class WordDocumentFactory extends DocumentFactory{  
 @Override  
 public Document createDocument(){  
 return new WordDocument();  
 }  
}

class PdfDocumentFactory extends DocumentFactory{  
 @Override  
 public Document createDocument(){  
 return new PdfDocument();  
 }  
}  
  
class ExcelDocumentFactory extends DocumentFactory{  
 @Override  
 public Document createDocument(){  
 return new ExcelDocument();  
 }  
}

**DocumentTest.java**

public class DocumentTest {  
 public static void main(String[] args){  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 Document wordDocument = wordFactory.createDocument();  
 wordDocument.message();  
  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 Document pdfDocument = pdfFactory.createDocument();  
 pdfDocument.message();  
  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
 Document excelDocument = excelFactory.createDocument();  
 excelDocument.message();  
 }  
}

**Output:**



**Explanation:**

A new project named FactoryMethodPatternExample is created.

The Document.java interface declares a method message(). This interface is implemented by the classes WordDocument, PdfDocument, and ExcelDocument, each of which overrides the message() method to print a message indicating the type of document created.

An abstract class named DocumentFactory is defined, containing an abstract method createDocument(). This class serves as the base for the concrete factories.

The classes WordDocumentFactory, PdfDocumentFactory, and ExcelDocumentFactory extend the DocumentFactory class and override the createDocument() method. Each of these factory classes returns an instance of its corresponding document type.

A new class called DocumentTest.java is created, which contains the main() method. In this method, instances of the three factory classes are created and used to generate different document types. The message() method is then called on each created document to display a message confirming the type of document created.

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Project: BuilderPatternExample**

**Scenario:**

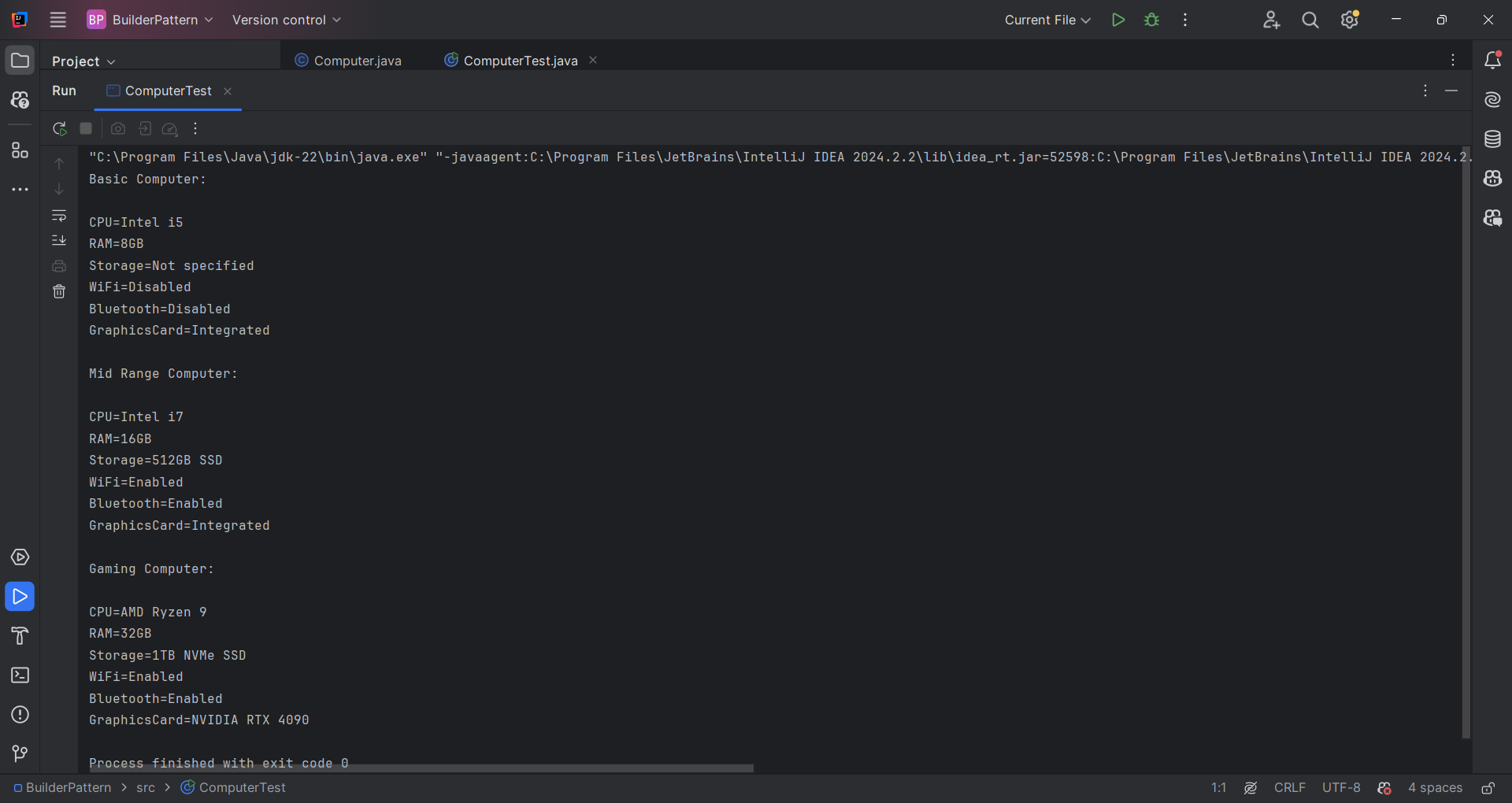
You are developing a system to create complex objects such as a computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Computer.java**

// File: Computer.java  
public class Computer {  
 private final String CPU;  
 private final String RAM;  
 private final String storage;  
 private final boolean wifiEnabled;  
 private final boolean bluetoothEnabled;  
 private final String graphicsCard;  
  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.wifiEnabled = builder.wifiEnabled;  
 this.bluetoothEnabled = builder.bluetoothEnabled;  
 this.graphicsCard = builder.graphicsCard;  
 }  
  
 public static class Builder {  
 private final String CPU;  
 private final String RAM;  
  
 private String storage = "Not specified";  
 private boolean wifiEnabled = false;  
 private boolean bluetoothEnabled = false;  
 private String graphicsCard = "Integrated";  
  
 public Builder(String CPU, String RAM) {  
 this.CPU = CPU;  
 this.RAM = RAM;  
 }  
  
 public Builder setStorage(String storage) {  
 this.storage = storage;  
 return this;  
 }  
  
 public Builder setWiFiEnabled(boolean wifiEnabled) {  
 this.wifiEnabled = wifiEnabled;  
 return this;  
 }  
  
 public Builder setBluetoothEnabled(boolean bluetoothEnabled) {  
 this.bluetoothEnabled = bluetoothEnabled;  
 return this;  
 }  
  
 public Builder setGraphicsCard(String graphicsCard) {  
 this.graphicsCard = graphicsCard;  
 return this;  
 }  
  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
  
 public void printComputer() {  
 System.*out*.println("CPU=" + CPU + "\nRAM=" + RAM + "\nStorage=" + storage +  
 "\nWiFi=" + (wifiEnabled ? "Enabled" : "Disabled") +  
 "\nBluetooth=" + (bluetoothEnabled ? "Enabled" : "Disabled") +  
 "\nGraphicsCard=" + graphicsCard);  
 }  
}

**ComputerTest.java**   
public class ComputerTest {  
 public static void main(String[] args) {  
 Computer basicComputer = new Computer.Builder("Intel i5", "8GB")  
 .build();  
  
 Computer midRangeComputer = new Computer.Builder("Intel i7", "16GB")  
 .setStorage("512GB SSD")  
 .setWiFiEnabled(true)  
 .setBluetoothEnabled(true)  
 .build();  
  
 Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")  
 .setStorage("1TB NVMe SSD")  
 .setWiFiEnabled(true)  
 .setBluetoothEnabled(true)  
 .setGraphicsCard("NVIDIA RTX 4090")  
 .build();  
  
 System.*out*.println("Basic Computer:\n");  
 basicComputer.printComputer();  
 System.*out*.println("\nMid Range Computer:\n");  
 midRangeComputer.printComputer();  
 System.*out*.println("\nGaming Computer:\n");  
 gamingComputer.printComputer();  
 }  
}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Project: AdapterPatternExample**

**PaymentProcessor.java**

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

**GooglePay.java**

public class GooglePay {  
 public void makePayment(double amount) {  
 System.*out*.println("Paid " + amount + " using Google Pay.");  
 }  
}

**PhonePe.java**

public class PhonePe {  
 public void sendPayment(double amount) {  
 System.*out*.println("Paid " + amount + " using PhonePe.");  
 }  
}

**RazorPay.java**

public class PhonePe {  
 public void sendPayment(double amount) {  
 System.*out*.println("Paid " + amount + " using PhonePe.");  
 }  
}

**GooglePayAdapter.java**

public class GooglePayAdapter implements PaymentProcessor {  
 private GooglePay paypal;  
  
 public GooglePayAdapter(GooglePay paypal) {  
 this.paypal = paypal;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 paypal.makePayment(amount);  
 }  
}

**PhonePeAdapter.java**

public class PhonePeAdapter implements PaymentProcessor{  
 private PhonePe phonePe;  
 public PhonePeAdapter(PhonePe phonePe){  
 this.phonePe = phonePe;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 phonePe.sendPayment(amount);  
 }  
}

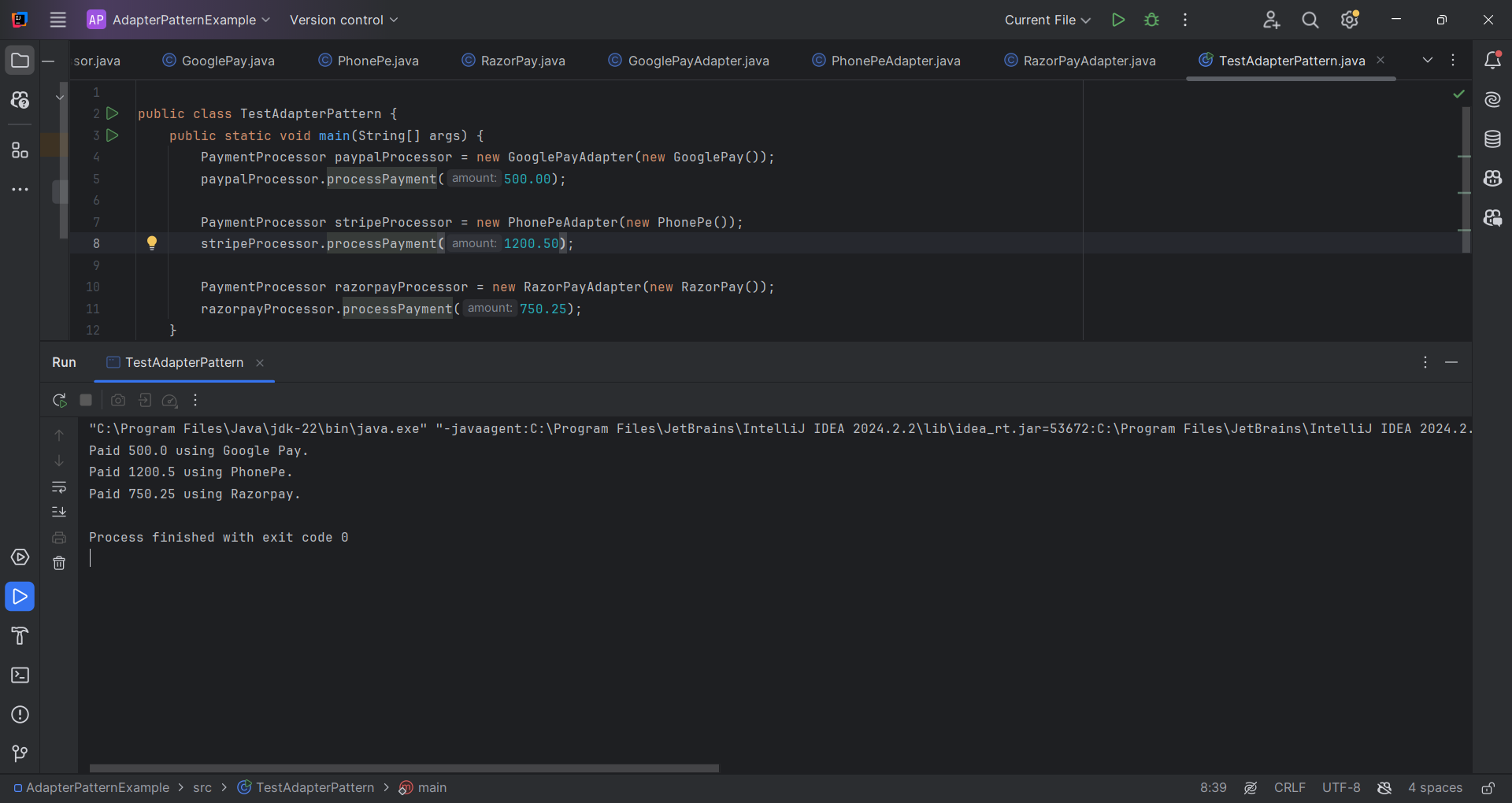
**RazorPayAdapter.java**

public class RazorPayAdapter implements PaymentProcessor{  
 private RazorPay razorPay;  
 public RazorPayAdapter(RazorPay razorPay){  
 this.razorPay = razorPay;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 razorPay.pay(amount);  
 }  
}

**TestAdapterPattern.java**

public class TestAdapterPattern {  
 public static void main(String[] args) {  
 PaymentProcessor paypalProcessor = new GooglePayAdapter(new GooglePay());  
 paypalProcessor.processPayment(500.00);  
  
 PaymentProcessor stripeProcessor = new PhonePeAdapter(new PhonePe());  
 stripeProcessor.processPayment(1200.50);  
  
 PaymentProcessor razorpayProcessor = new RazorPayAdapter(new RazorPay());  
 razorpayProcessor.processPayment(750.25);  
 }  
}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

Project: DecoratorPatternExample

**Notifier.java**

EmailNotifier.java

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

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

**Notifierdecorator.java**

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

**SMSNotifierDecorator.java**

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);  
 }  
}

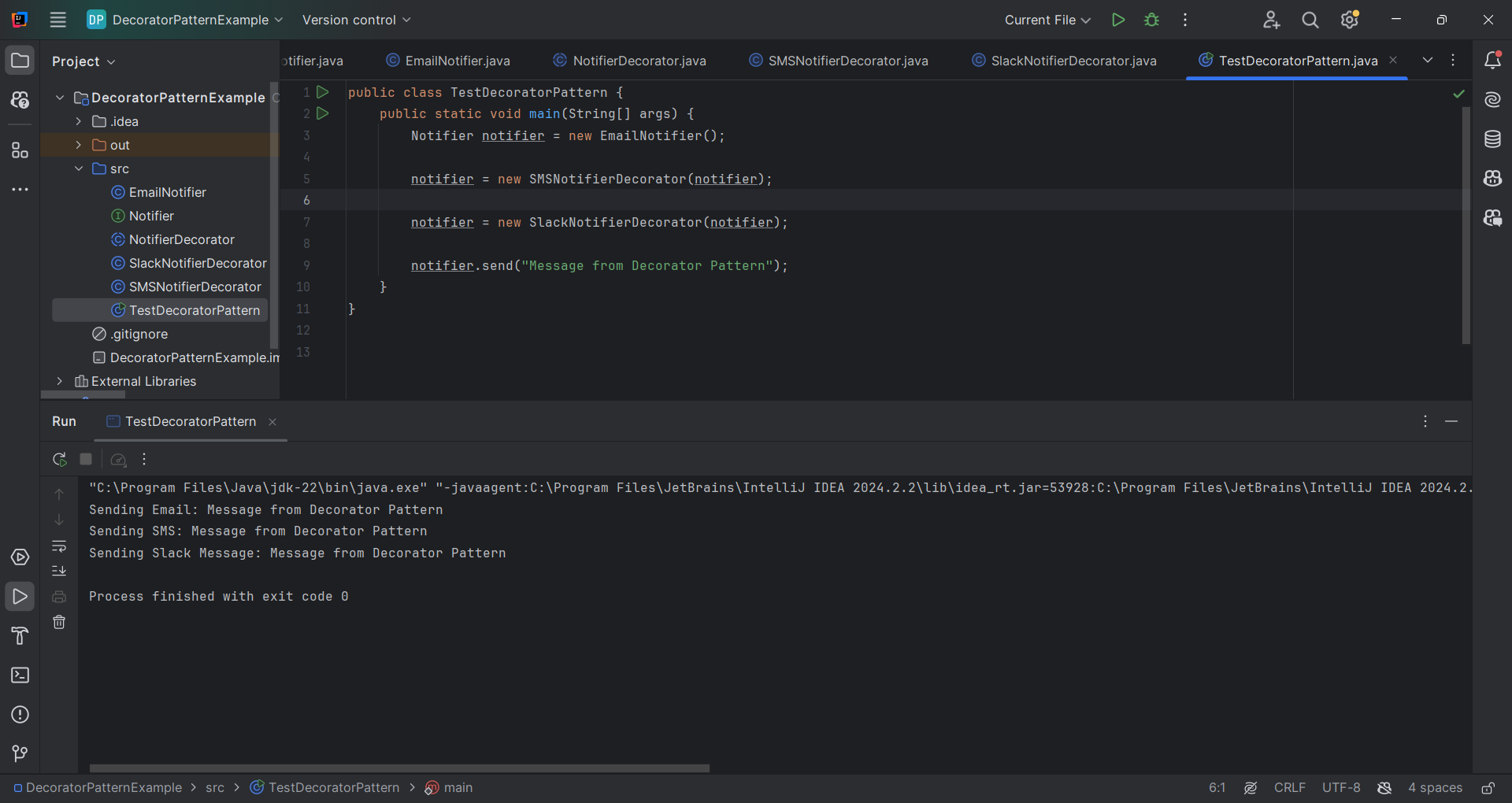
**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
  
 @Override  
 public void send(String message) {  
 super.send(message);  
 sendSlackMessage(message);  
 }  
  
 private void sendSlackMessage(String message) {  
 System.*out*.println("Sending Slack Message: " + message);  
 }  
}

**TestDecoratorPattern.java**

public class TestDecoratorPattern {  
 public static void main(String[] args) {  
 Notifier notifier = new EmailNotifier();  
  
 notifier = new SMSNotifierDecorator(notifier);  
  
 notifier = new SlackNotifierDecorator(notifier);  
  
 notifier.send("Message from Decorator Pattern");  
 }  
}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

Project: ProxyPatternExample

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");

}

@Override

public void display() {

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

}

}

ProxyImage.java

public class ProxyImage implements Image {  
 private String filename;  
 private RealImage realImage;  
  
 public ProxyImage(String filename) {  
 this.filename = filename;  
 }  
  
 @Override  
 public void display() {  
 if (realImage == null) {  
 realImage = new RealImage(filename); // Lazy initialization  
 } else {  
 System.*out*.println("Using cached image for: " + filename);  
 }  
 realImage.display();  
 }  
}

TestProxyPattern.java

public class TestProxyPattern {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("img1.jpg");  
 Image image2 = new ProxyImage("img2.jpg");  
  
 image1.display();  
 System.*out*.println();  
  
 image1.display();  
 System.*out*.println();  
  
 image2.display();  
 }  
}

Output:

