Superset ID: 6373431

COGNIZANT DIGITAL NURTURE 4.0 JAVA FSE

WEEK-1

DESIGN PATTERNS AND PRINCIPLES

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

CODE

SingletonPatternExample

Logger.java

package com.example.singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("Logger Created");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

LoggerTest.java

package com.example.singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("Logger Created");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

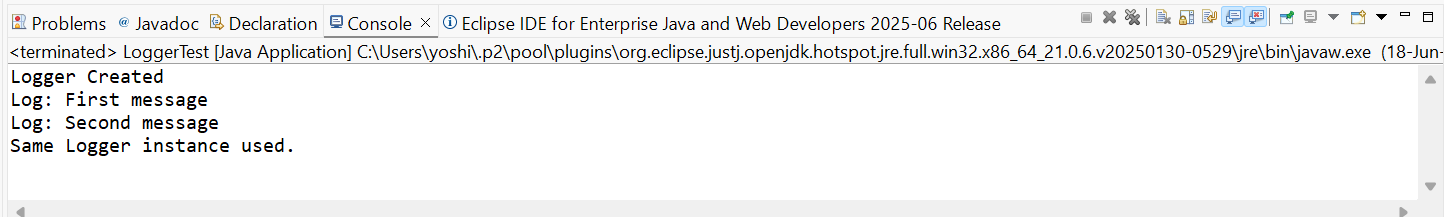
SingletonPatternExample

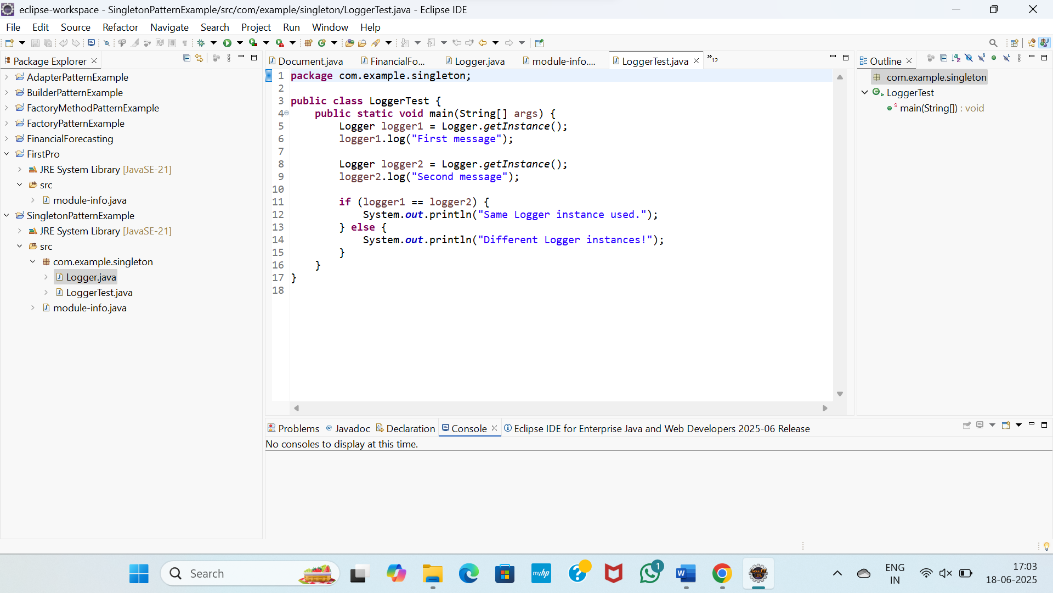
This experiment demonstrates the Singleton Design Pattern in Java.

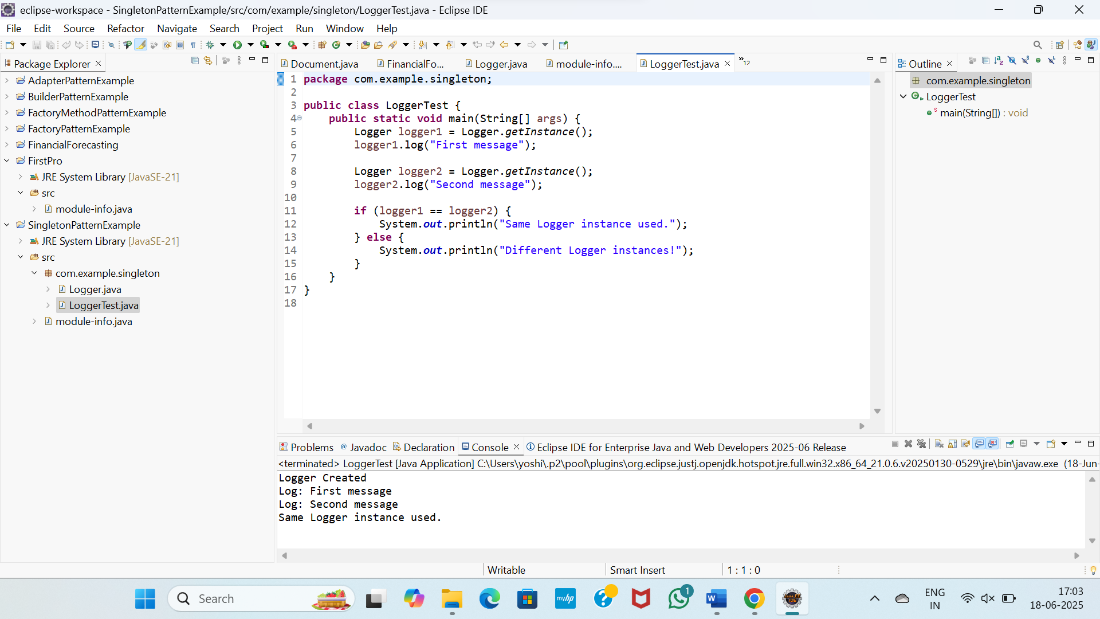
- Logger.java: Singleton Logger class.

- LoggerTest.java: Main class to test the Singleton behavior.

Output:







**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

CODE

FactoryMethodPatternExample

FactoryPatternSimple.java

package com.example.factorysimple;

public class FactoryPatternSimple {

// Document interface

interface Document {

void open();

}

// Concrete document classes

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

// Abstract factory

static abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete factories

static class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Test method

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document doc1 = wordFactory.createDocument();

doc1.open();

DocumentFactory pdfFactory = new PdfFactory();

Document doc2 = pdfFactory.createDocument();

doc2.open();

DocumentFactory excelFactory = new ExcelFactory();

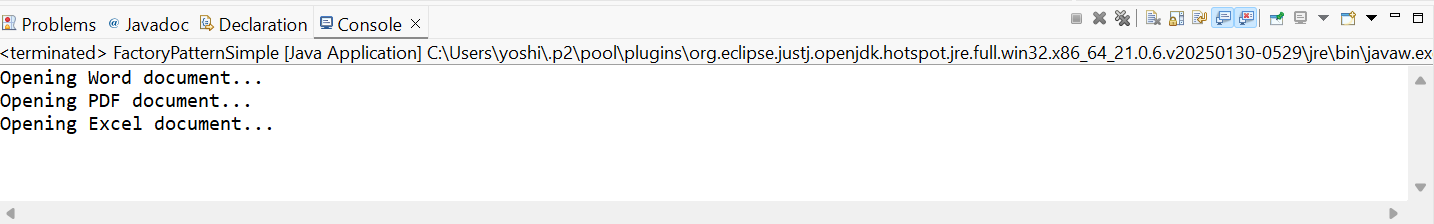
Document doc3 = excelFactory.createDocument();

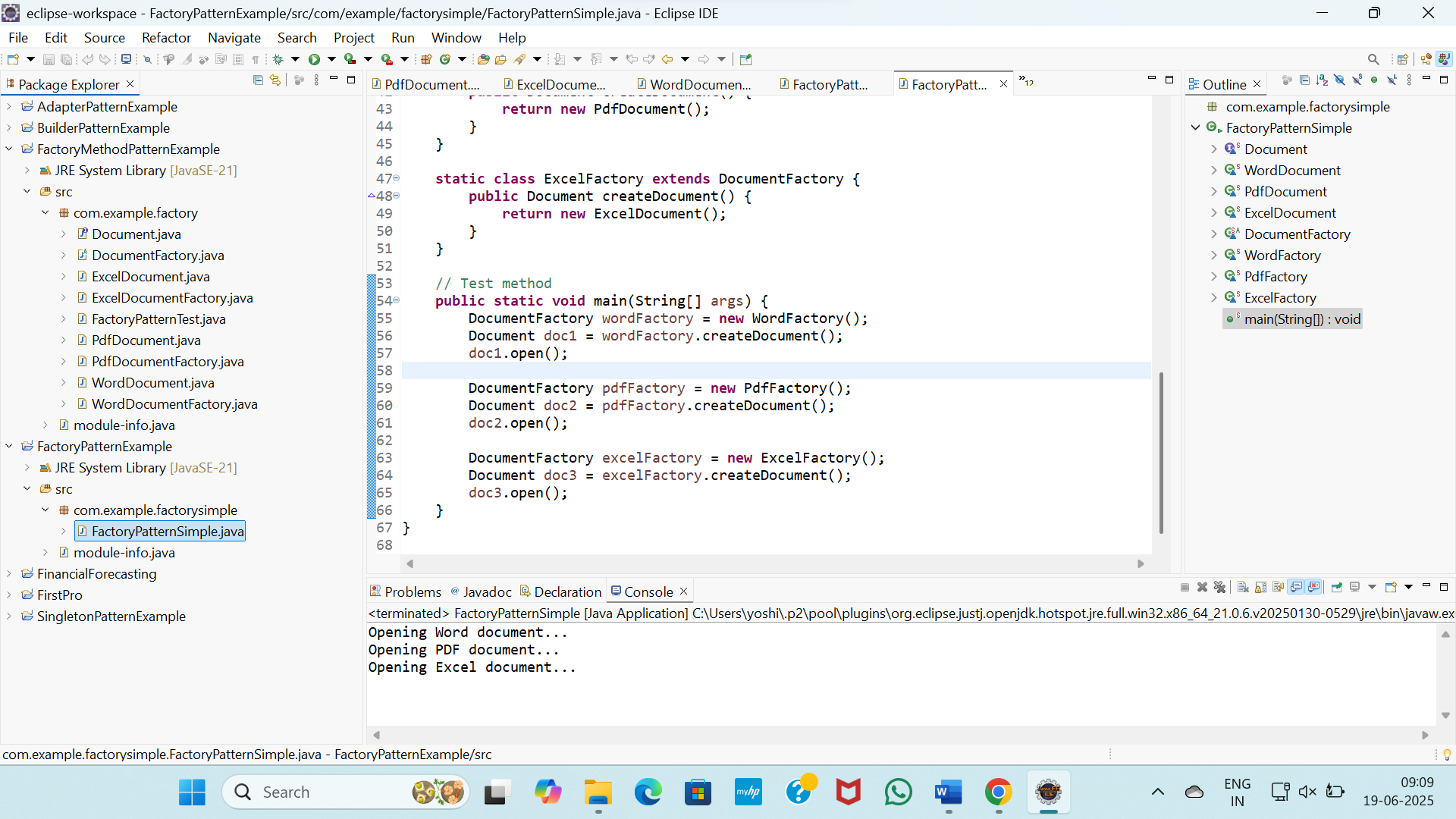
doc3.open();

}

}

OUTPUT





This cleanly demonstrates the Factory Method Pattern, where:

* Object creation logic is abstracted into factories.
* The main class does not directly use new, but relies on createDocument().

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

CODE

BuilderPatternExample

Computer.java

package com.example.builder;

public class Computer {

// Required and optional parts

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

// Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

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 graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void showConfig() {

System.*out*.println("Computer Configuration:");

System.*out*.println("CPU: " + CPU);

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

System.*out*.println("Storage: " + storage);

System.*out*.println("Graphics Card: " + graphicsCard);

System.*out*.println();

}

}

BuilderPatternTest.java

package com.example.builder;

public class BuilderPatternTest {

public static void main(String[] args) {

// Basic configuration

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("256GB SSD")

.build();

// Gaming configuration

Computer gamingComputer = new Computer.Builder()

.setCPU("AMD Ryzen 9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.build();

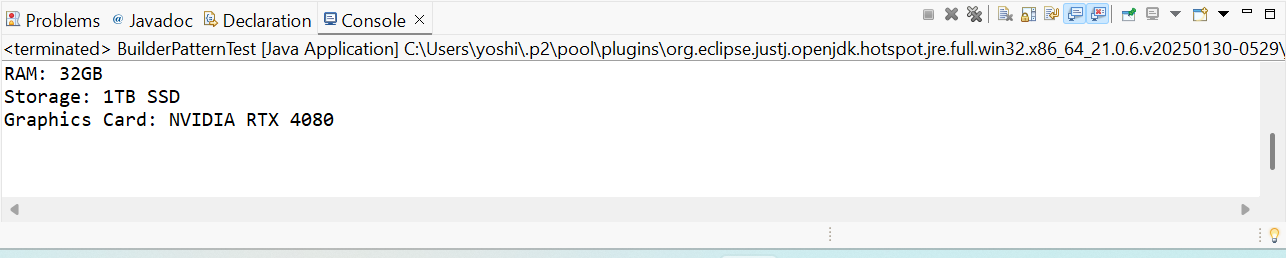
basicComputer.showConfig();

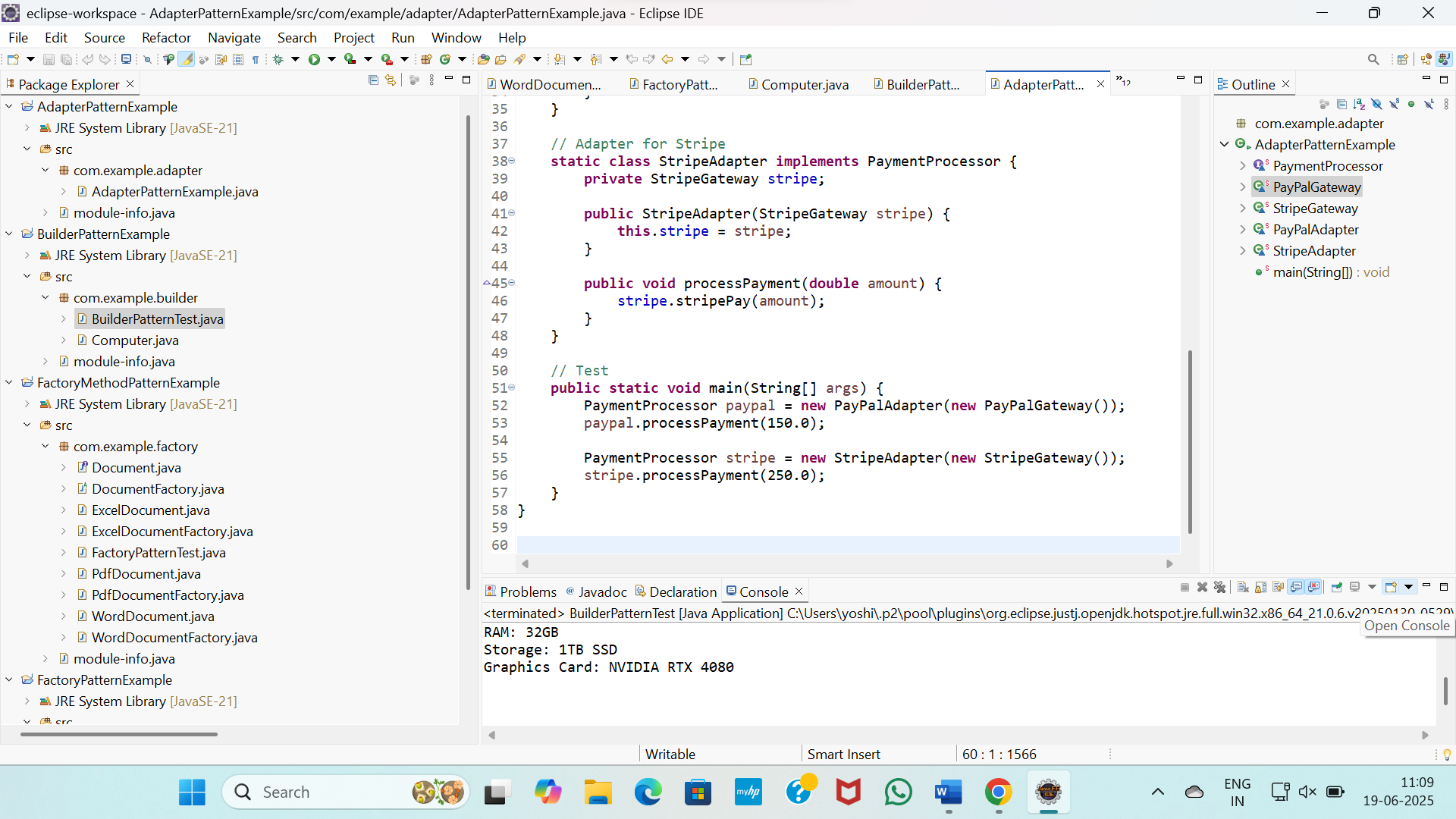
gamingComputer.showConfig();

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

AdapterPatternExample

CODE

package com.example.adapter;

public class AdapterPatternExample {

// Target interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Adaptee 1

static class PayPalGateway {

public void makePayment(double amount) {

System.*out*.println("PayPal: Processing payment of $" + amount);

}

}

// Adaptee 2

static class StripeGateway {

public void stripePay(double amount) {

System.*out*.println("Stripe: Charging amount $" + amount);

}

}

// Adapter for PayPal

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

// Adapter for Stripe

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.stripePay(amount);

}

}

// Test

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());

paypal.processPayment(150.0);

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

stripe.processPayment(250.0);

}

}

OUTPUT

