**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:**

package com.example.singleton;

public class SingletonTest {

public static class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger instance created.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public static void main(String[] args) {

System.out.println("Attempting to get first Logger instance...");

Logger logger1 = Logger.getInstance();

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

System.out.println("\nAttempting to get second Logger instance...");

Logger logger2 = Logger.getInstance();

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

System.out.println("\nComparing instances:");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 refer to the same instance. Singleton pattern successful!");

} else {

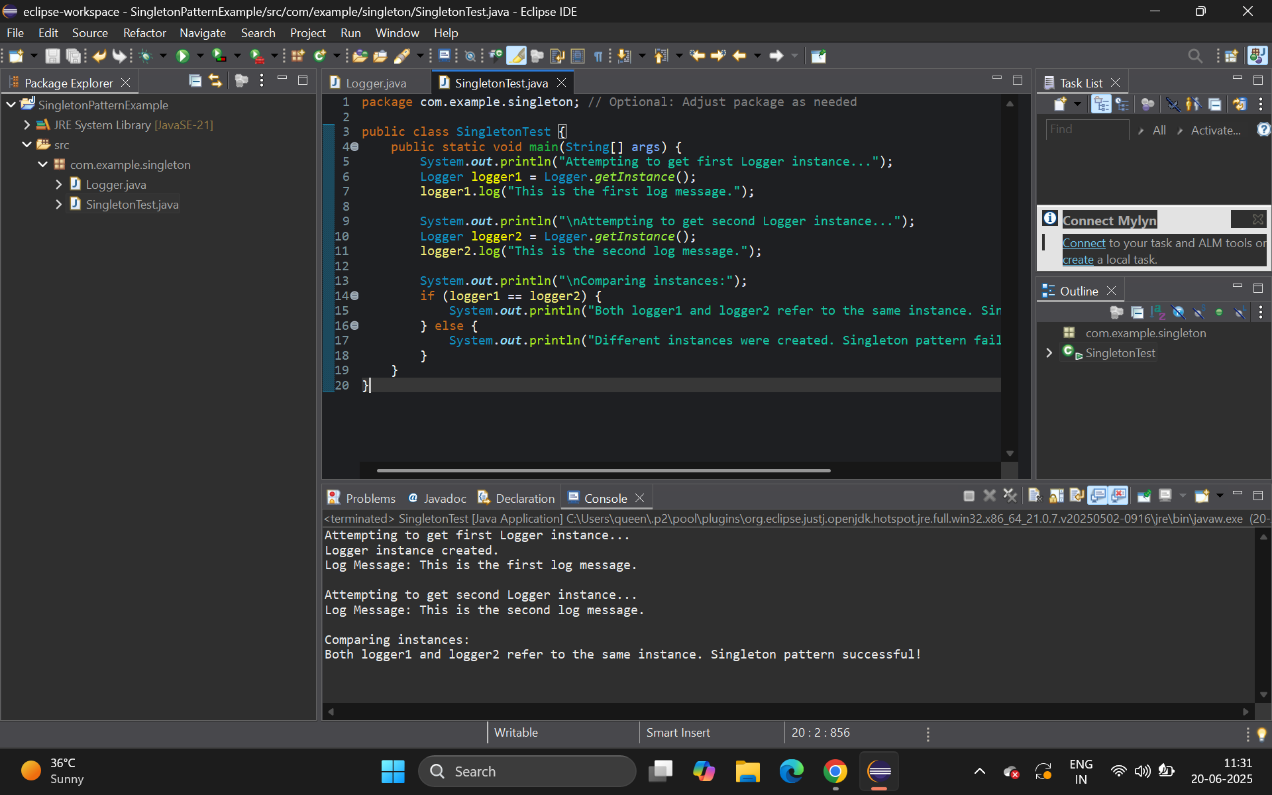
System.out.println("Different instances were created. Singleton pattern failed!");

}

}

}

**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:**

package com.example.factory;

public class FactoryMethodTest {

public interface Document {

void open();

void save();

void close();

}

public static 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."); }

}

public static 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."); }

}

public static 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."); }

}

public static abstract class DocumentFactory {

public abstract Document createDocument();

}

public static class WordDocumentFactory extends DocumentFactory {

@Override public Document createDocument() { return new WordDocument(); }

}

public static class PdfDocumentFactory extends DocumentFactory {

@Override public Document createDocument() { return new PdfDocument(); }

}

public static class ExcelDocumentFactory extends DocumentFactory {

@Override public Document createDocument() { return new ExcelDocument(); }

}

public static void main(String[] args) {

System.out.println("Creating Word Document:");

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

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

System.out.println("\nCreating PDF Document:");

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open(); pdfDoc.save(); pdfDoc.close();

System.out.println("\nCreating Excel Document:");

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

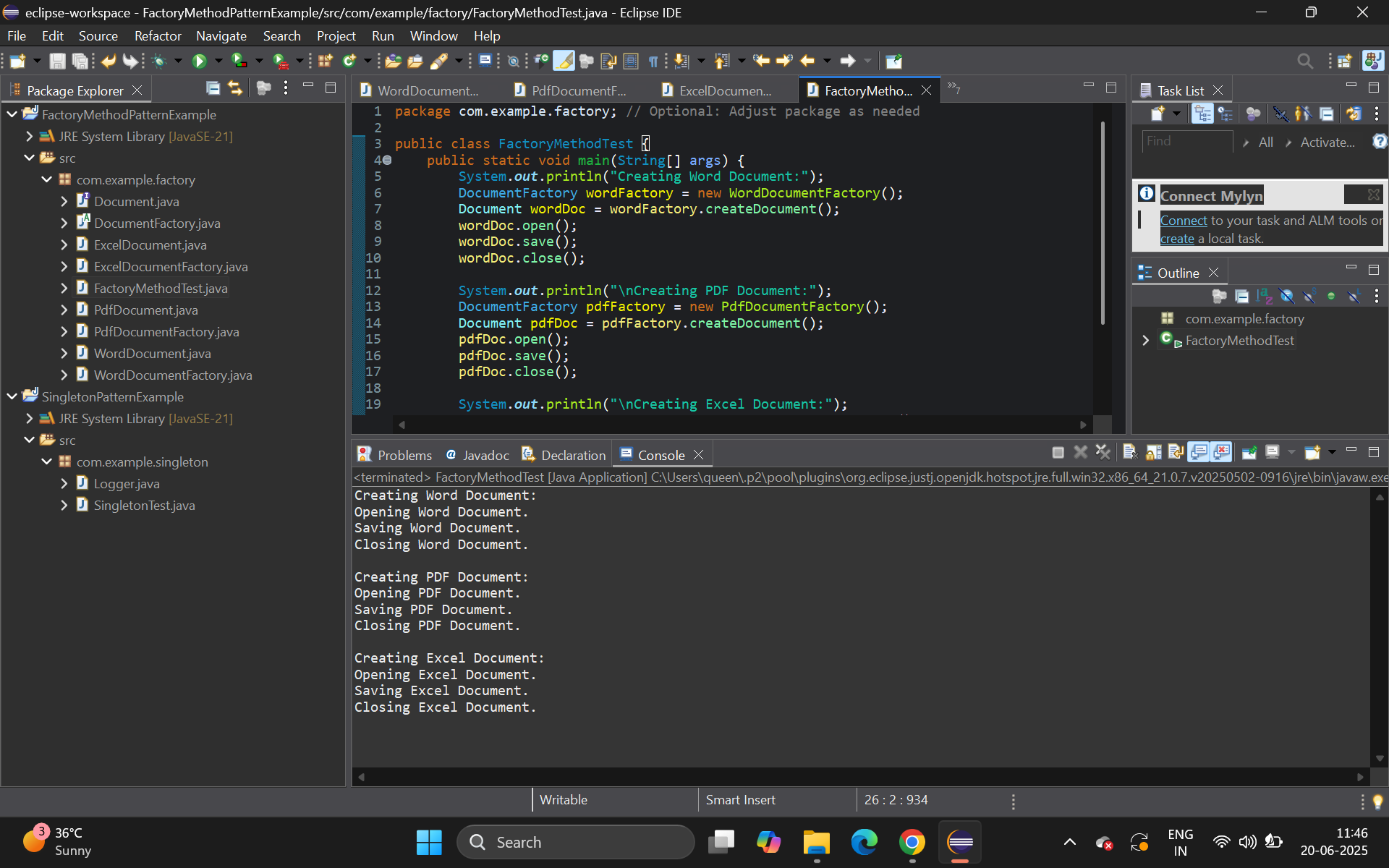
excelDoc.save();

excelDoc.close();

}

}

**OUTPUT:**



**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:**

package com.example.builder;

public class BuilderTest {

public static class Computer {

private String CPU;

private String RAM;

private String Storage;

private String graphicsCard;

private String operatingSystem;

private String powerSupply;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.Storage = builder.Storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

this.powerSupply = builder.powerSupply;

}

@Override

public String toString() {

return "Computer Configuration:\n" +

" CPU: " + CPU + "\n" +

" RAM: " + RAM + "\n" +

" Storage: " + Storage + "\n" +

" Graphics Card: " + (graphicsCard != null ? graphicsCard : "N/A") + "\n" +

" Operating System: " + (operatingSystem != null ? operatingSystem : "N/A") + "\n" +

" Power Supply: " + (powerSupply != null ? powerSupply : "N/A");

}

public static class Builder {

private String CPU;

private String RAM;

private String Storage;

private String graphicsCard;

private String operatingSystem;

private String powerSupply;

public Builder(String CPU, String RAM, String Storage) {

this.CPU = CPU;

this.RAM = RAM;

this.Storage = Storage;

}

public Builder withGraphicsCard(String graphicsCard) { this.graphicsCard = graphicsCard; return this; }

public Builder withOperatingSystem(String operatingSystem) { this.operatingSystem = operatingSystem; return this; }

public Builder withPowerSupply(String powerSupply) { this.powerSupply = powerSupply; return this; }

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

}

}

public static void main(String[] args) {

System.out.println("Building a Gaming PC:");

Computer gamingPC = new Computer.Builder("Intel i9", "32GB DDR5", "1TB NVMe SSD")

.withGraphicsCard("NVIDIA RTX 4080")

.withOperatingSystem("Windows 11 Pro")

.withPowerSupply("850W Gold")

.build();

System.out.println(gamingPC);

System.out.println("\nBuilding an Office PC:");

Computer officePC = new Computer.Builder("Intel i5", "8GB DDR4", "256GB SSD")

.withOperatingSystem("Windows 10 Home")

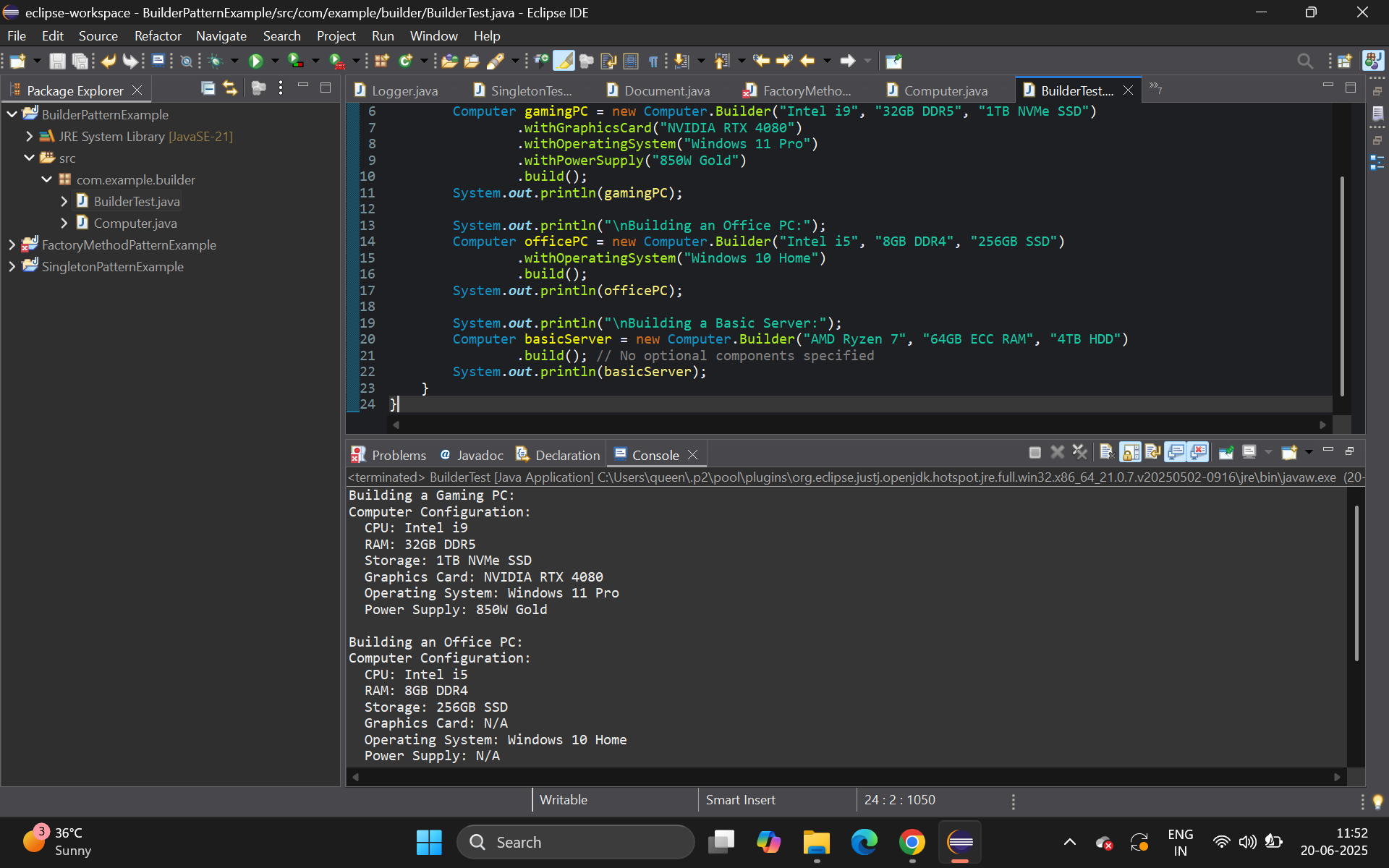
.build();

System.out.println(officePC);

}

}

**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.

**CODE:**

package com.example.adapter;

public class AdapterTest {

public interface PaymentProcessor {

void processPayment(double amount);

}

public static class StripeGateway {

public void makePayment(double amount) { System.out.println("Processing payment of $" + amount + " through Stripe."); }

}

public static class PayPalGateway {

public void sendMoney(double amount) { System.out.println("Sending $" + amount + " through PayPal."); }

}

public static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) { this.stripeGateway = stripeGateway; }

@Override public void processPayment(double amount) { stripeGateway.makePayment(amount); }

}

public static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) { this.payPalGateway = payPalGateway; }

@Override public void processPayment(double amount) { payPalGateway.sendMoney(amount); }

}

public static void main(String[] args) {

System.out.println("Using Stripe Gateway via Adapter:");

StripeGateway stripe = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

stripeProcessor.processPayment(100.50);

System.out.println("\nUsing PayPal Gateway via Adapter:");

PayPalGateway payPal = new PayPalGateway();

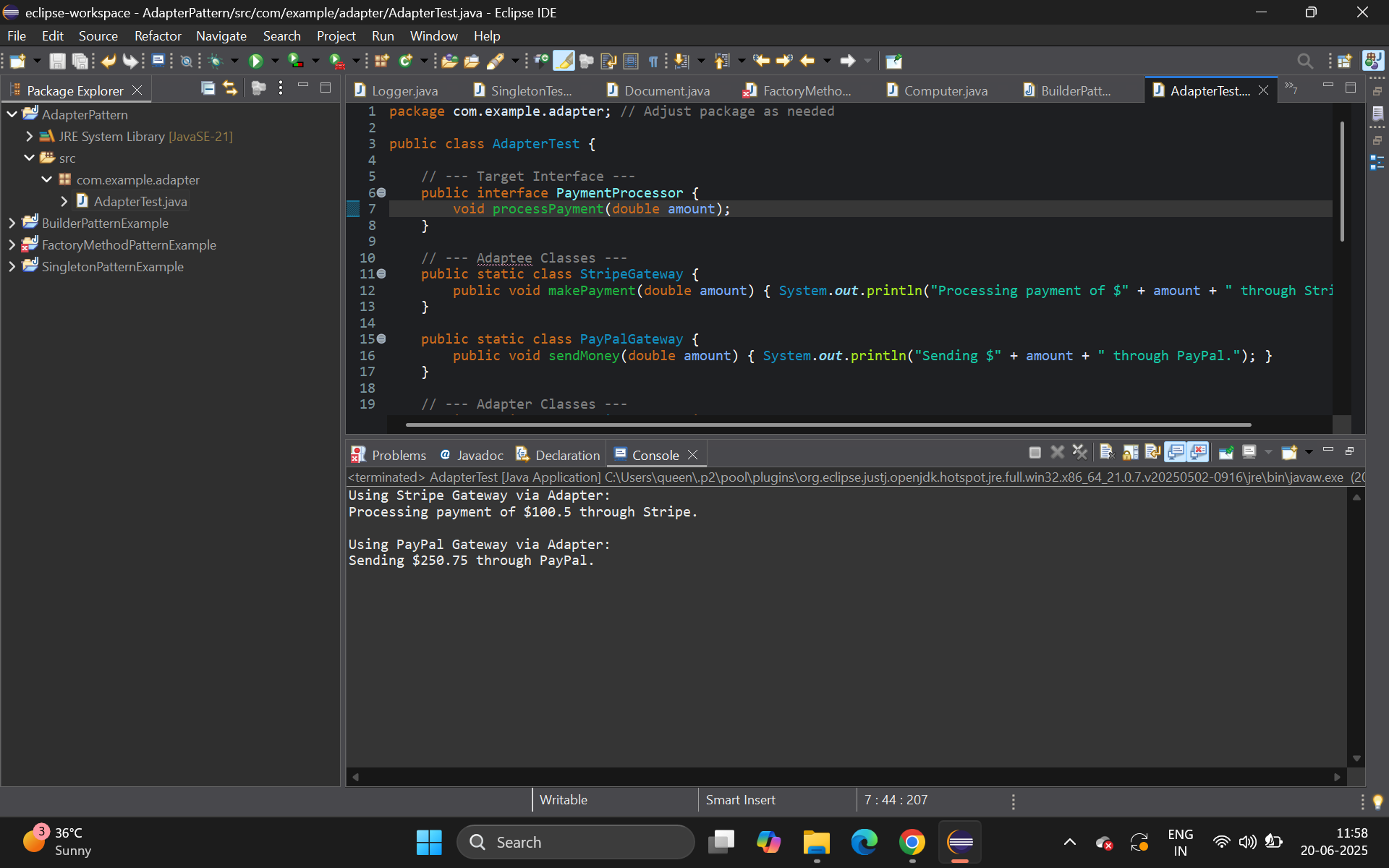
PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);

payPalProcessor.processPayment(250.75);

}

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**CODE:**

package com.example.decorator;

public class DecoratorTest{

public interface Notifier {

void send(String message);

}

public static class EmailNotifier implements Notifier {

@Override public void send(String message) { System.out.println("Sending Email with message: " + message); }

}

public static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) { this.wrappedNotifier = notifier; }

@Override public void send(String message) { wrappedNotifier.send(message); }

}

public static 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 with message: " + message); }

}

public static 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); }

}

public static void main(String[] args) {

System.out.println("--- Basic Email Notification ---");

Notifier emailOnlyNotifier = new EmailNotifier();

emailOnlyNotifier.send("Your order has been shipped!");

System.out.println("\n--- Email + SMS Notification ---");

Notifier emailAndSMSNotifier = new SMSNotifierDecorator(new EmailNotifier());

emailAndSMSNotifier.send("Your payment was successful!");

System.out.println("\n--- Email + SMS + Slack Notification ---");

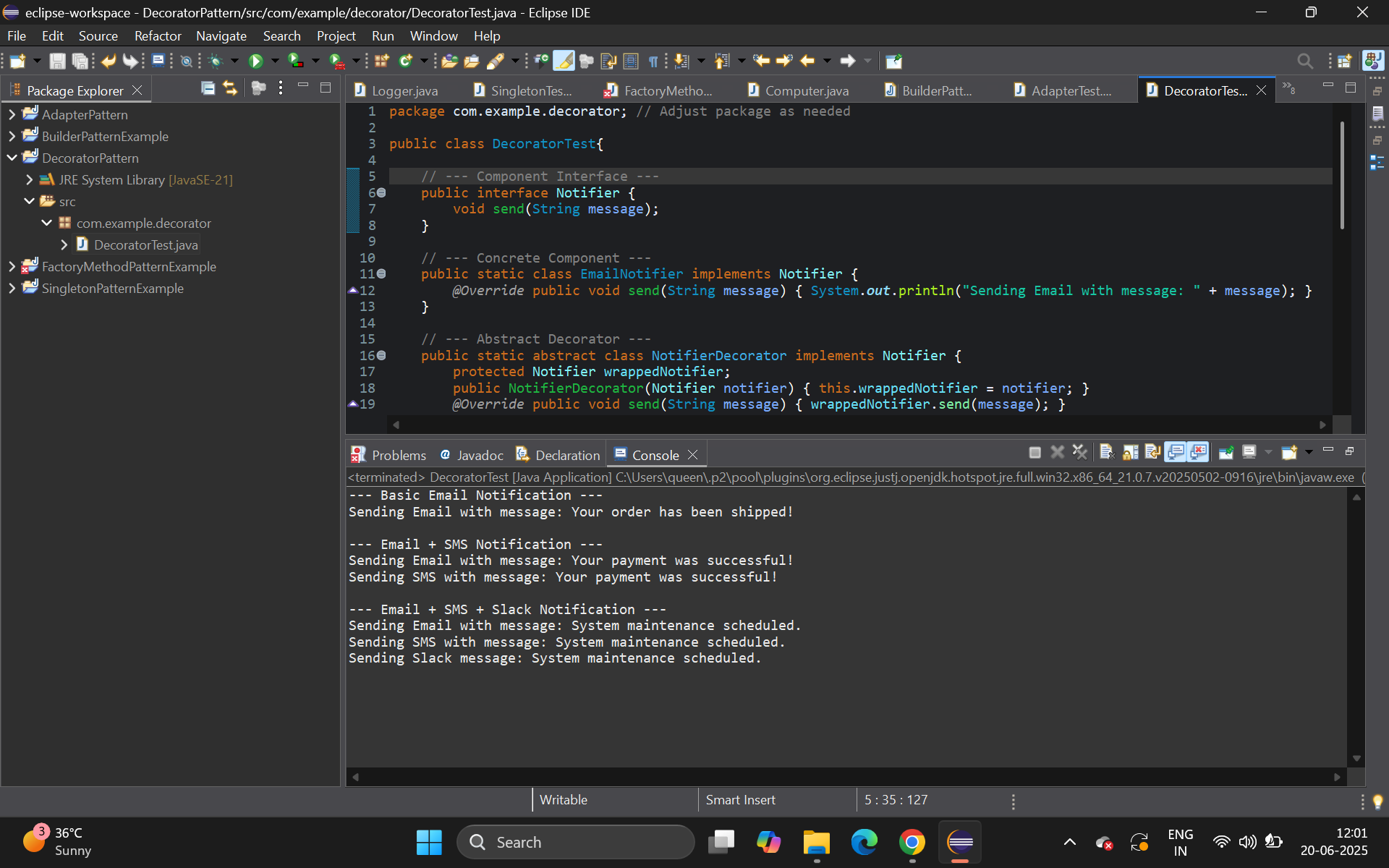
Notifier allChannelsNotifier = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

allChannelsNotifier.send("System maintenance scheduled.");

}

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**CODE:**

package com.example.proxy;

import java.util.HashMap;

import java.util.Map;

public class ProxyTest {

public interface Image {

void display();

}

public static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

try { Thread.sleep(1000); } catch (InterruptedException e) { e.printStackTrace(); }

System.out.println("Image " + filename + " loaded.");

}

@Override public void display() { System.out.println("Displaying image: " + filename); }

}

public static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

private static Map<String, RealImage> imageCache = new HashMap<>();

public ProxyImage(String filename) { this.filename = filename; }

@Override

public void display() {

if (imageCache.containsKey(filename)) {

System.out.println("Fetching image " + filename + " from cache.");

realImage = imageCache.get(filename);

} else {

if (realImage == null) {

System.out.println("Real image for " + filename + " not yet loaded. Loading now...");

realImage = new RealImage(filename);

imageCache.put(filename, realImage);

}

}

realImage.display();

}

}

public static void main(String[] args) {

System.out.println("--- First attempt to display image1.jpg (should load) ---");

Image image1 = new ProxyImage("image1.jpg");

image1.display();

System.out.println("\n--- Second attempt to display image1.jpg (should be cached) ---");

image1.display();

System.out.println("\n--- First attempt to display image2.png (should load) ---");

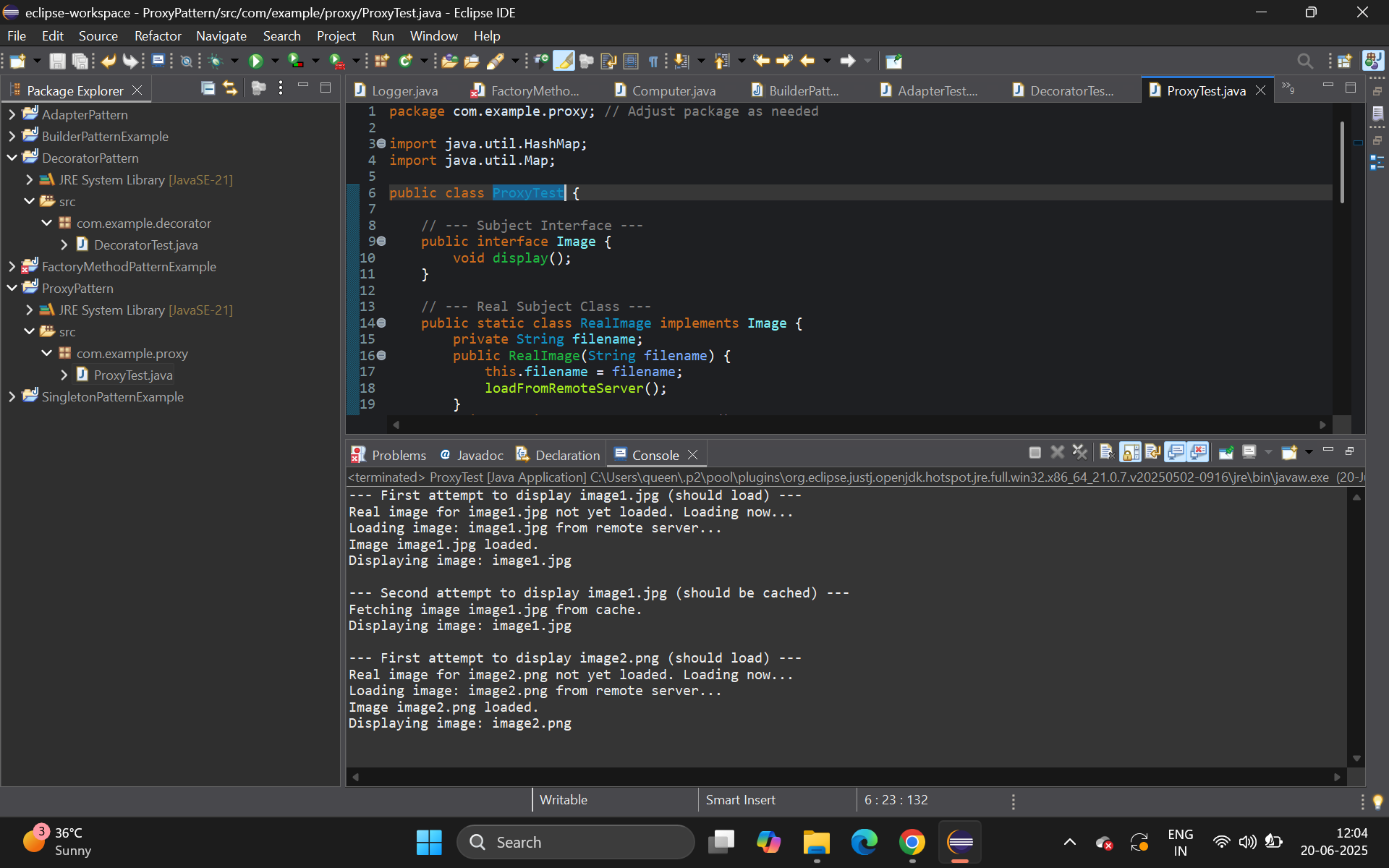
Image image2 = new ProxyImage("image2.png");

image2.display();

}

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**CODE:**

package com.example.observer;

import java.util.ArrayList;

import java.util.List;

public class ObserverTest{

public interface Observer {

void update(double stockPrice);

}

public static class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) { this.appName = appName; }

@Override public void update(double stockPrice) { System.out.println(appName + " received update: New stock price is $" + stockPrice); }

}

public static class WebApp implements Observer {

private String appName;

public WebApp(String appName) { this.appName = appName; }

@Override public void update(double stockPrice) { System.out.println(appName + " received update: Stock price updated to $" + stockPrice); }

}

public interface Stock {

void registerObserver(Observer o);

void deregisterObserver(Observer o);

void notifyObservers();

}

public static class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() { this.observers = new ArrayList<>(); }

@Override public void registerObserver(Observer o) { observers.add(o); System.out.println("Observer registered: " + o.getClass().getSimpleName()); }

@Override public void deregisterObserver(Observer o) { observers.remove(o); System.out.println("Observer deregistered: " + o.getClass().getSimpleName()); }

@Override public void notifyObservers() {

System.out.println("Notifying all observers about stock price change...");

for (Observer observer : observers) { observer.update(stockPrice); }

}

public void setStockPrice(double newPrice) {

System.out.println("\nStock price changed from " + this.stockPrice + " to " + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp1 = new MobileApp("MyStock Mobile App");

Observer webApp1 = new WebApp("StockDashboard Web App");

stockMarket.registerObserver(mobileApp1);

stockMarket.registerObserver(webApp1);

stockMarket.setStockPrice(150.75);

stockMarket.setStockPrice(151.20);

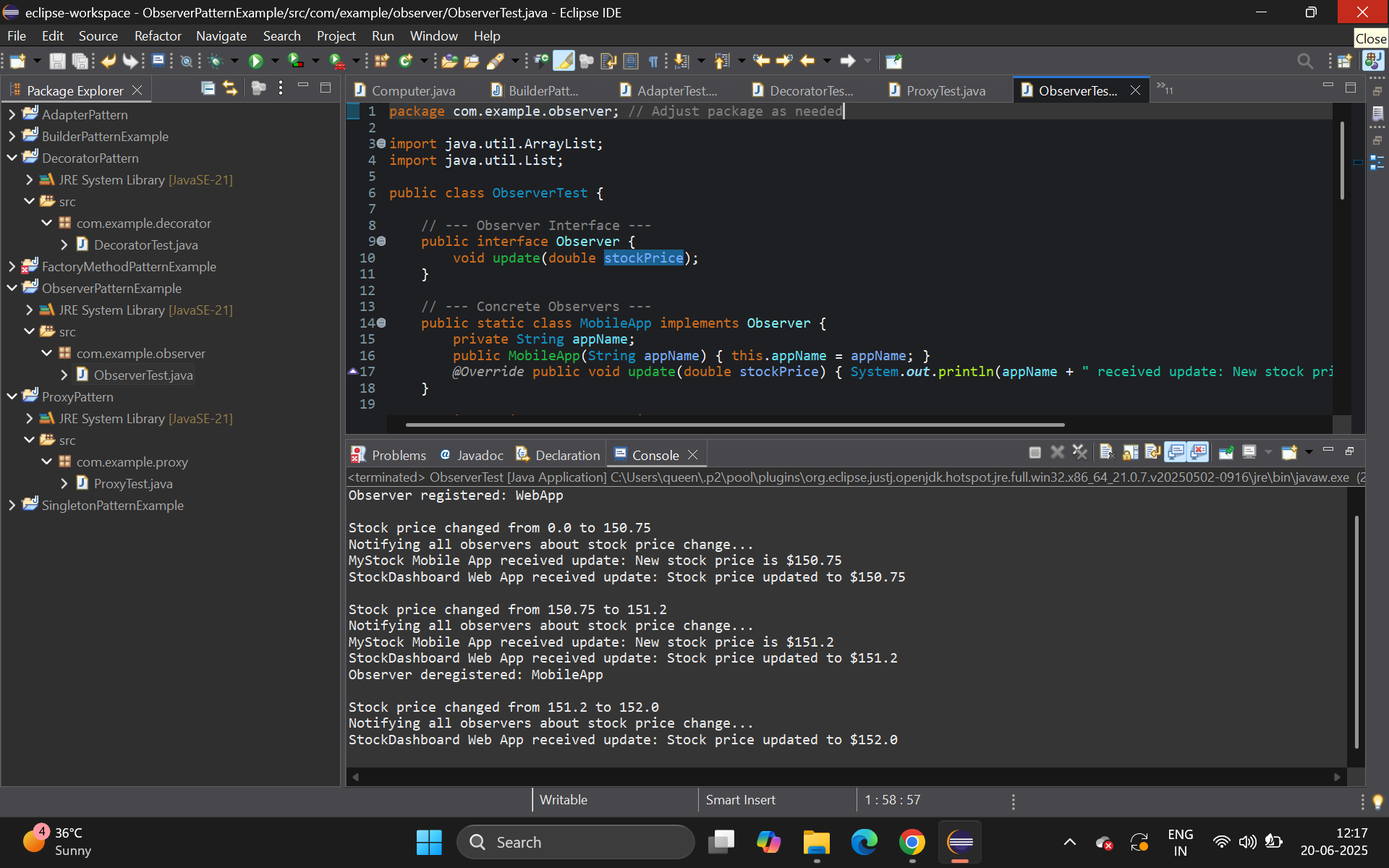
stockMarket.deregisterObserver(mobileApp1);

stockMarket.setStockPrice(152.00);

}

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

**CODE:**

package com.example.strategy;

public class StrategyTest{

public interface PaymentStrategy {

void pay(double amount);

}

public static class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

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

@Override public void pay(double amount) { System.out.println("Paying $" + amount + " using Credit Card (Card No: XXXX-XXXX-XXXX-" + cardNumber.substring(12) + ")"); }

}

public static class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) { this.email = email; }

@Override public void pay(double amount) { System.out.println("Paying $" + amount + " using PayPal (Email: " + email + ")"); }

}

public static class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) { this.paymentStrategy = paymentStrategy; }

public void executePayment(double amount) {

if (paymentStrategy == null) { System.out.println("No payment strategy selected."); return; }

paymentStrategy.pay(amount);

}

}

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

System.out.println("--- Paying with Credit Card ---");

paymentContext.setPaymentStrategy(new CreditCardPayment("1234567890123456", "John Doe"));

paymentContext.executePayment(250.75);

System.out.println("\n--- Paying with PayPal ---");

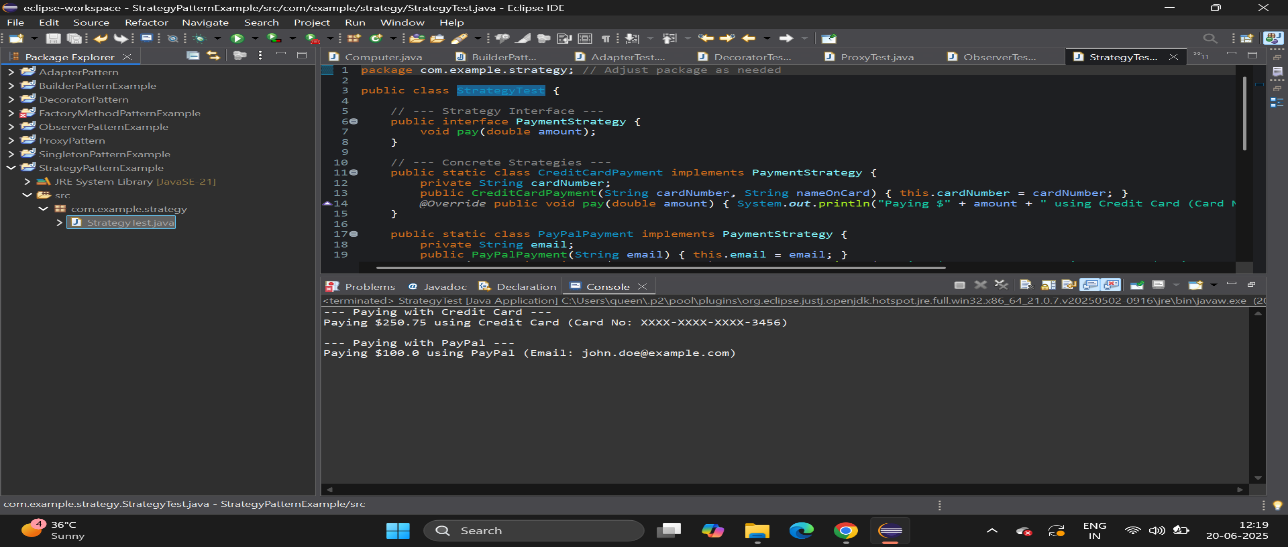
paymentContext.setPaymentStrategy(new PayPalPayment("john.doe@example.com"));

paymentContext.executePayment(100.00);

}

}

**OUTPUT:**

****

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**CODE:**

package com.example.command;

public class CommandTest {

public interface Command {

void execute();

}

public static class Light {

private String location;

public Light(String location) { this.location = location; }

public void turnOn() { System.out.println(location + " Light is ON."); }

public void turnOff() { System.out.println(location + " Light is OFF."); }

}

public static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) { this.light = light; }

@Override public void execute() { light.turnOn(); }

}

public static class LightOffCommand implements Command {

private Light light;

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

@Override public void execute() { light.turnOff(); }

}

public static 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."); }

}

}

public static void main(String[] args) {

Light livingRoomLight = new Light("Living Room");

Light kitchenLight = new Light("Kitchen");

Command livingRoomLightOn = new LightOnCommand(livingRoomLight);

Command livingRoomLightOff = new LightOffCommand(livingRoomLight);

Command kitchenLightOn = new LightOnCommand(kitchenLight);

RemoteControl remote = new RemoteControl();

System.out.println("--- Pressing button for Living Room Light ON ---");

remote.setCommand(livingRoomLightOn);

remote.pressButton();

System.out.println("\n--- Pressing button for Living Room Light OFF ---");

remote.setCommand(livingRoomLightOff);

remote.pressButton();

System.out.println("\n--- Pressing button for Kitchen Light ON ---");

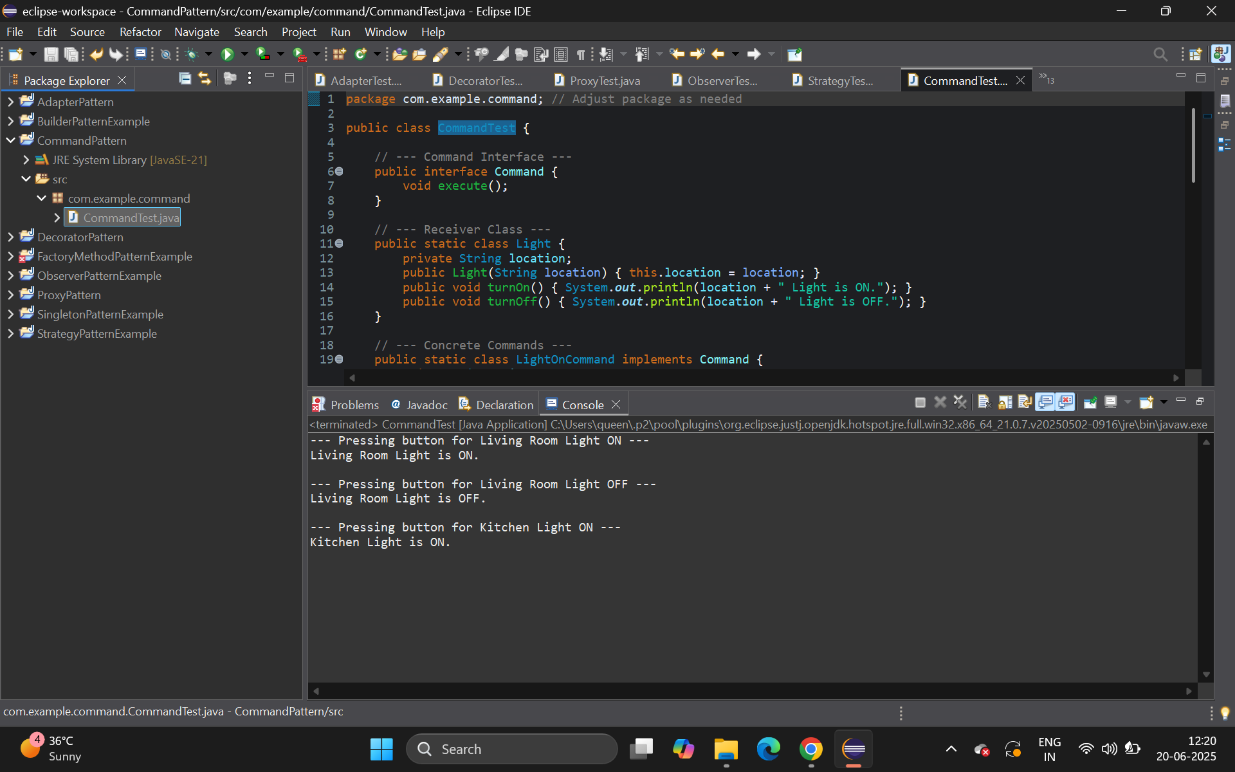
remote.setCommand(kitchenLightOn);

remote.pressButton();

}

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**CODE:**

package com.example.mvc;

public class MVCTest{

public static 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 void setName(String name) { this.name = name; }

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

public static class StudentView {

public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

System.out.println("--- Student Details ---");

System.out.println("Name: " + studentName);

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

System.out.println("Grade: " + studentGrade);

}

}

public static 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());

}

}

public static void main(String[] args) {

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

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

System.out.println("--- Initial Student Details ---");

controller.updateView();

System.out.println("\n--- Updating Student Grade to B ---");

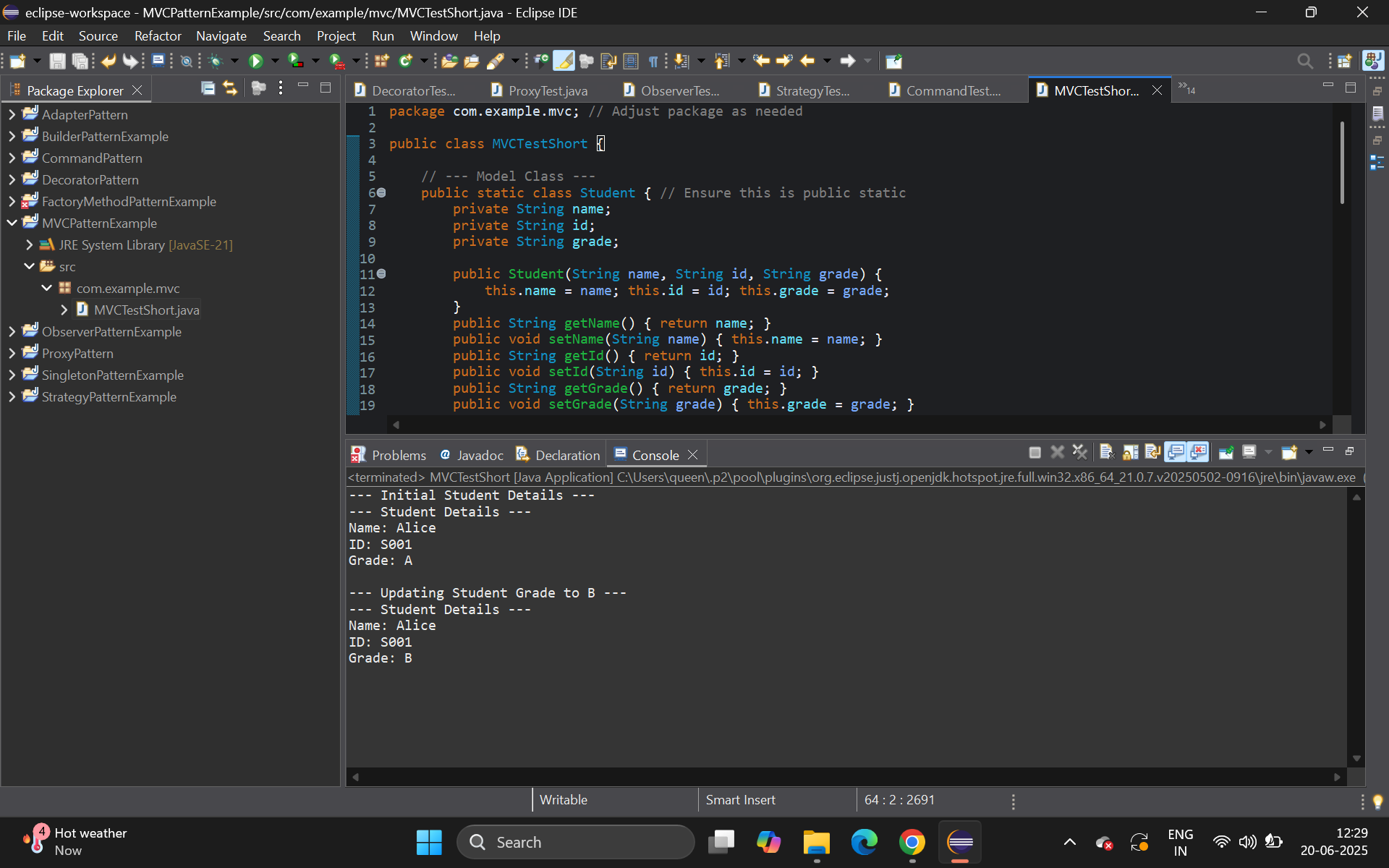
controller.setStudentGrade("B");

controller.updateView();

}

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**CODE:**

package com.example.di;

public class DITest {

public interface CustomerRepository {

String findCustomerById(String id);

}

public static class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

if ("123".equals(id)) { return "Customer with ID " + id + ": John Doe"; }

return "Customer with ID " + id + ": Not Found";

}

}

public static class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

System.out.println("CustomerService initialized with " + customerRepository.getClass().getSimpleName());

}

public String getCustomerDetails(String id) {

System.out.println("CustomerService fetching details for ID: " + id);

return customerRepository.findCustomerById(id);

}

}

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

System.out.println("\n--- Using CustomerService ---");

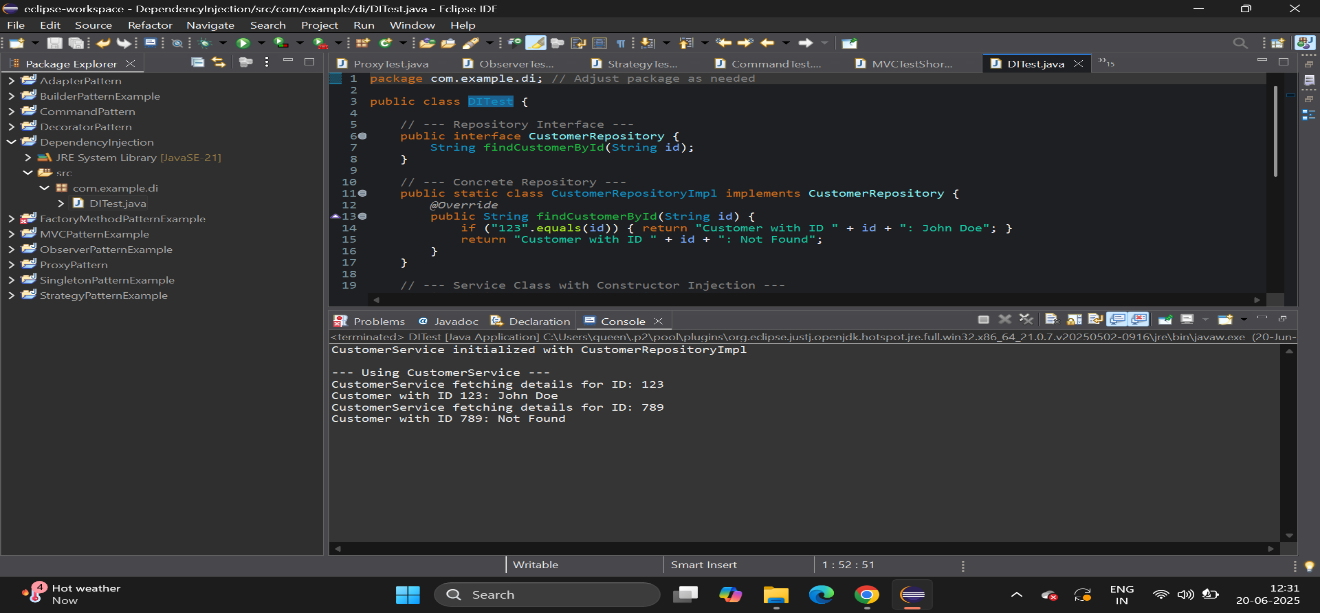
System.out.println(service.getCustomerDetails("123"));

System.out.println(service.getCustomerDetails("789"));

}

}

**OUTPUT:**

****