**DESIGN PATTERNS**

* **Exercise 1: Implementing the Singleton Pattern**

**Java Project Name : SingletonPatternExample**

**Logger.java code**

package defaultpackage;

public 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);

}

}

**SingletonTest.java**

package defaultpackage;

public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

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

Logger logger2 = Logger.getInstance();

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

if (logger1 == logger2) {

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

} else {

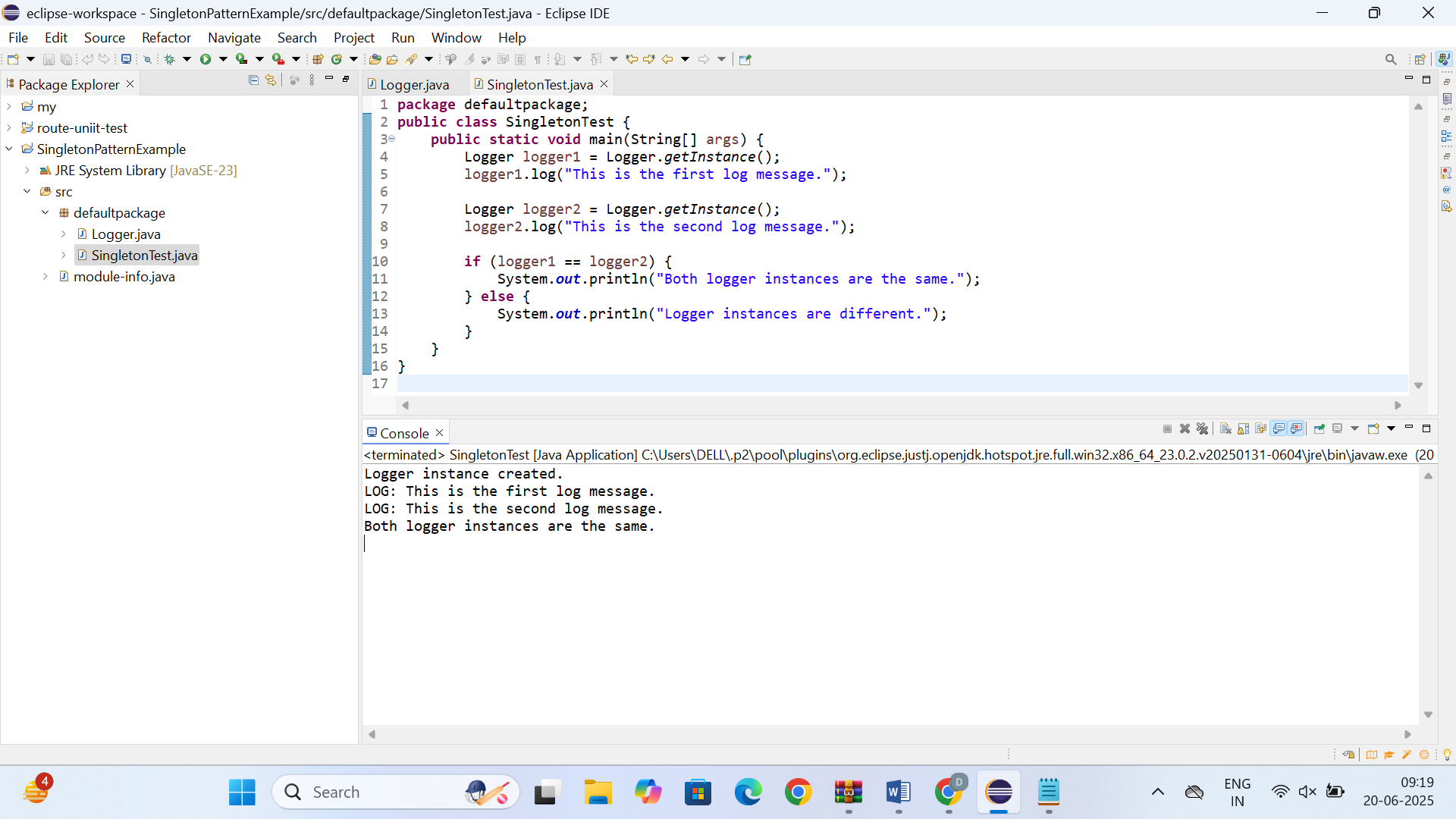
System.out.println("Logger instances are different.");

}

}

}

**OUTPUT**



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

**Java Project Name : FactoryMethodPatternExample**

**Document.java**

package documents;

public interface Document {

void open();

}

**WordDocument.java**

package documents;

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

package documents;

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java (Concrete Class)**

package documents;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**DocumentFactory.java (Abstract Factory)**

package documents;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java (Concrete Factory)**

package documents;

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java (Concrete Factory)**

package documents;

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java (Concrete Factory)**

package documents;

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**DocumentFactoryTest.java (Test Class with main Method)**

package documents;

public class DocumentFactoryTest {

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();

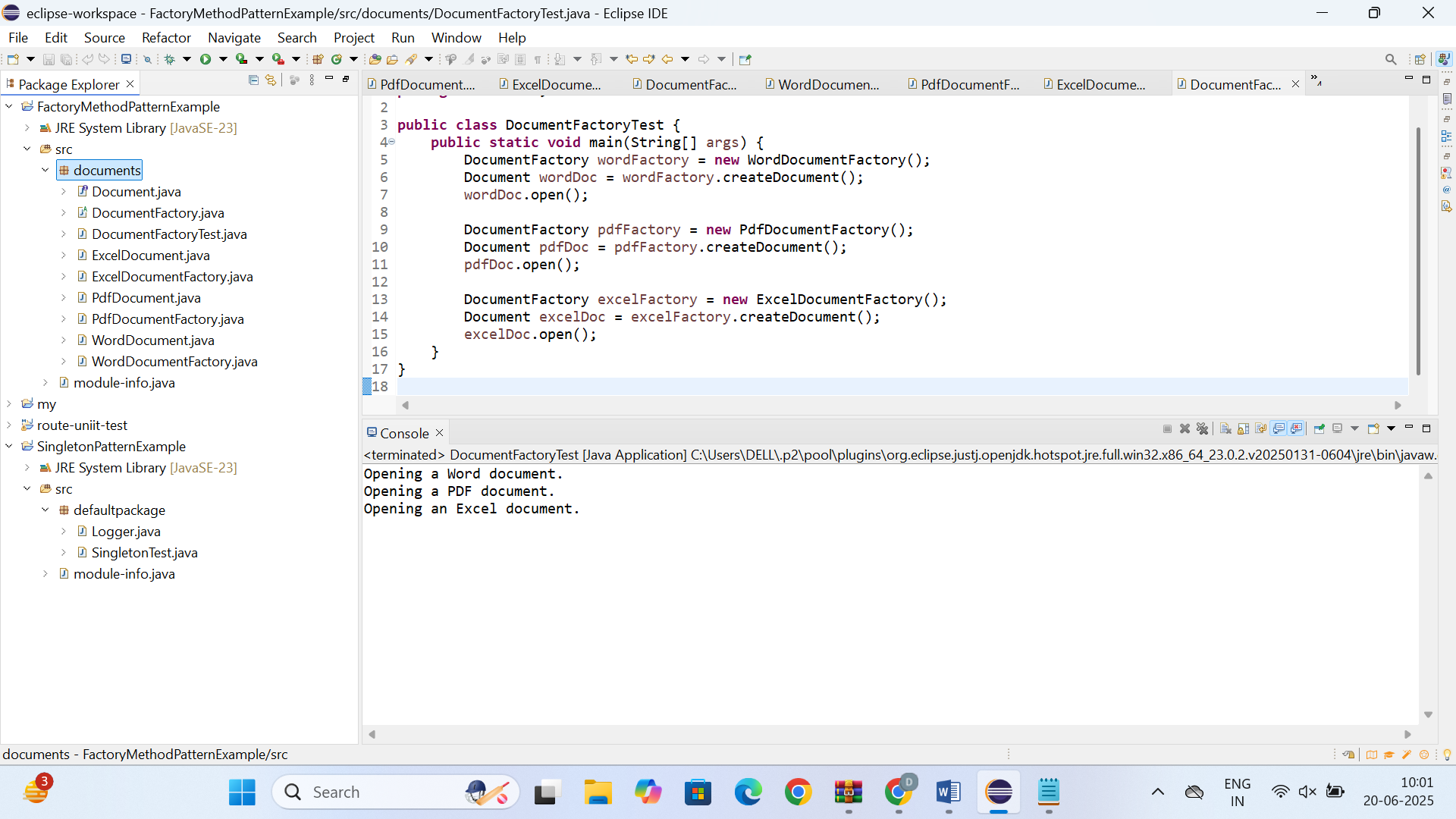
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT**



* **Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

private String cpu;

private String ram;

private String storage;

private boolean hasGraphicsCard;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.hasGraphicsCard = builder.hasGraphicsCard;

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private boolean hasGraphicsCard;

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(boolean hasGraphicsCard) {

this.hasGraphicsCard = hasGraphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void displaySpecs() {

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

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

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

System.out.println("Graphics Card: " + (hasGraphicsCard ? "Yes" : "No"));

}

}

**BuilderTest.java**

public class BuilderTest {

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGraphicsCard(true)

.build();

Computer officePC = new Computer.Builder()

.setCpu("Intel i5")

.setRam("16GB")

.setStorage("512GB SSD")

.setGraphicsCard(false)

.build();

System.out.println("Gaming PC Specs:");

gamingPC.displaySpecs();

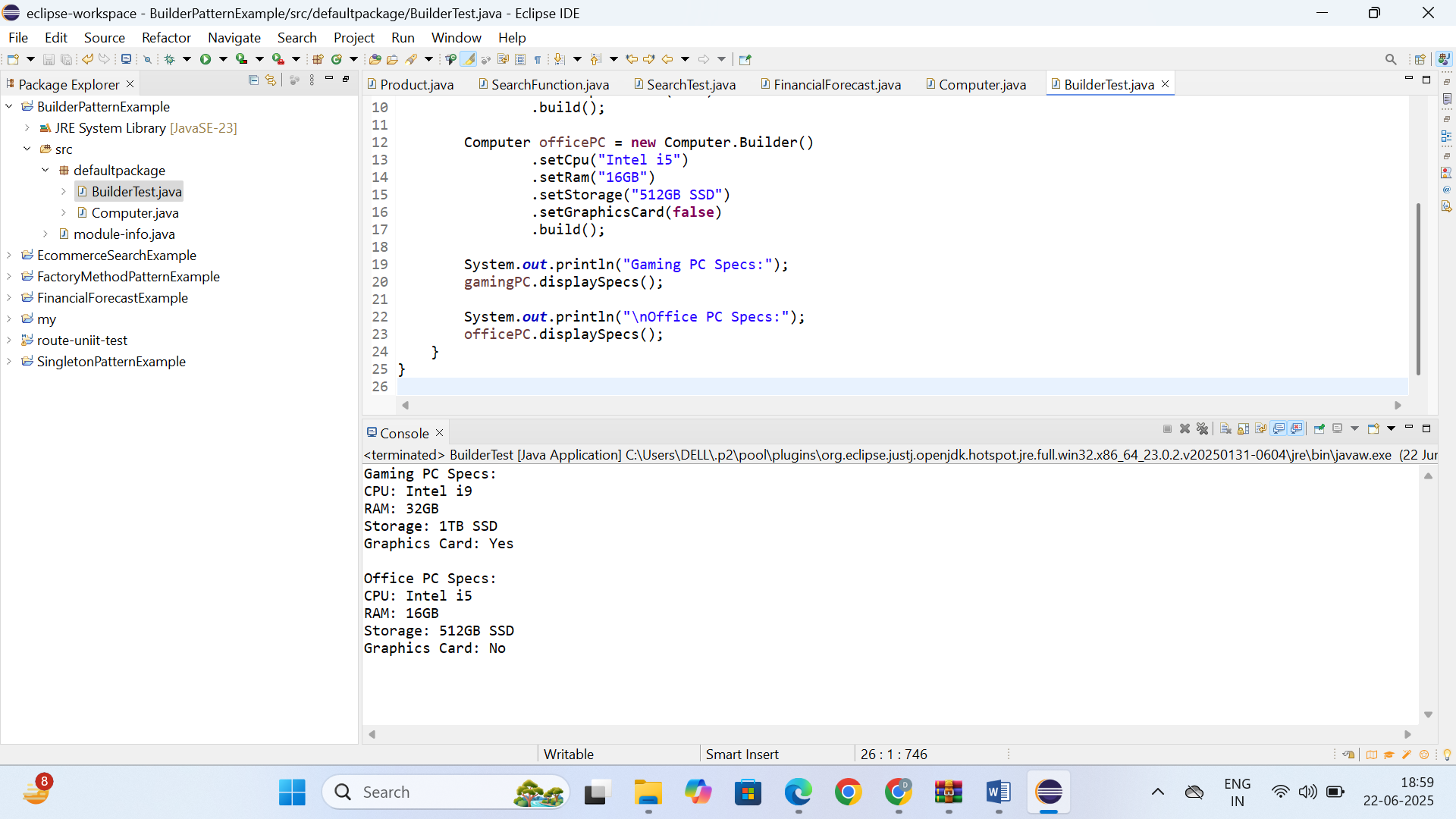
System.out.println("\nOffice PC Specs:");

officePC.displaySpecs();

}

}

**OUTPUT**



* **Exercise 4: Implementing the Adapter Pattern**

**Project: AdapterPatternExample**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Payment of $" + amount + " made using PayPal.");

}

}

**StripeGateway.java**

public class StripeGateway {

public void sendPayment(double amount) {

System.out.println("Payment of $" + amount + " sent through Stripe.");

}

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.sendPayment(amount);

}

}

**AdapterTest.java**

public class AdapterTest {

public static void main(String[] args) {

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

paypalProcessor.processPayment(150.00);

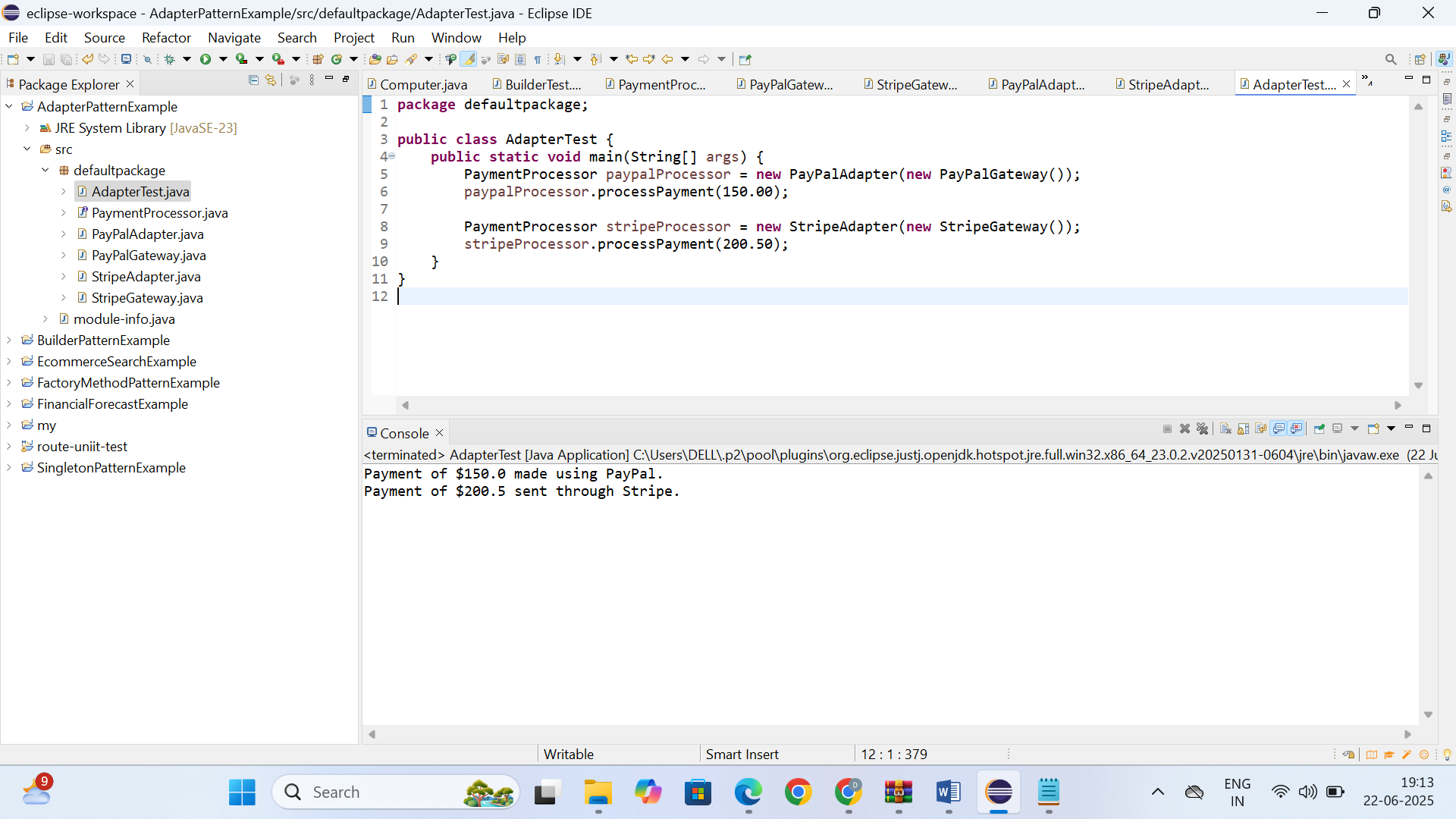
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(200.50);

}

}

**OUTPUT**



* **Exercise 5: Implementing the Decorator Pattern**

**Java Project: DecoratorPatternExample**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

System.out.println("SMS sent: " + message);

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**DecoratorTest.java**

public class DecoratorTest {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

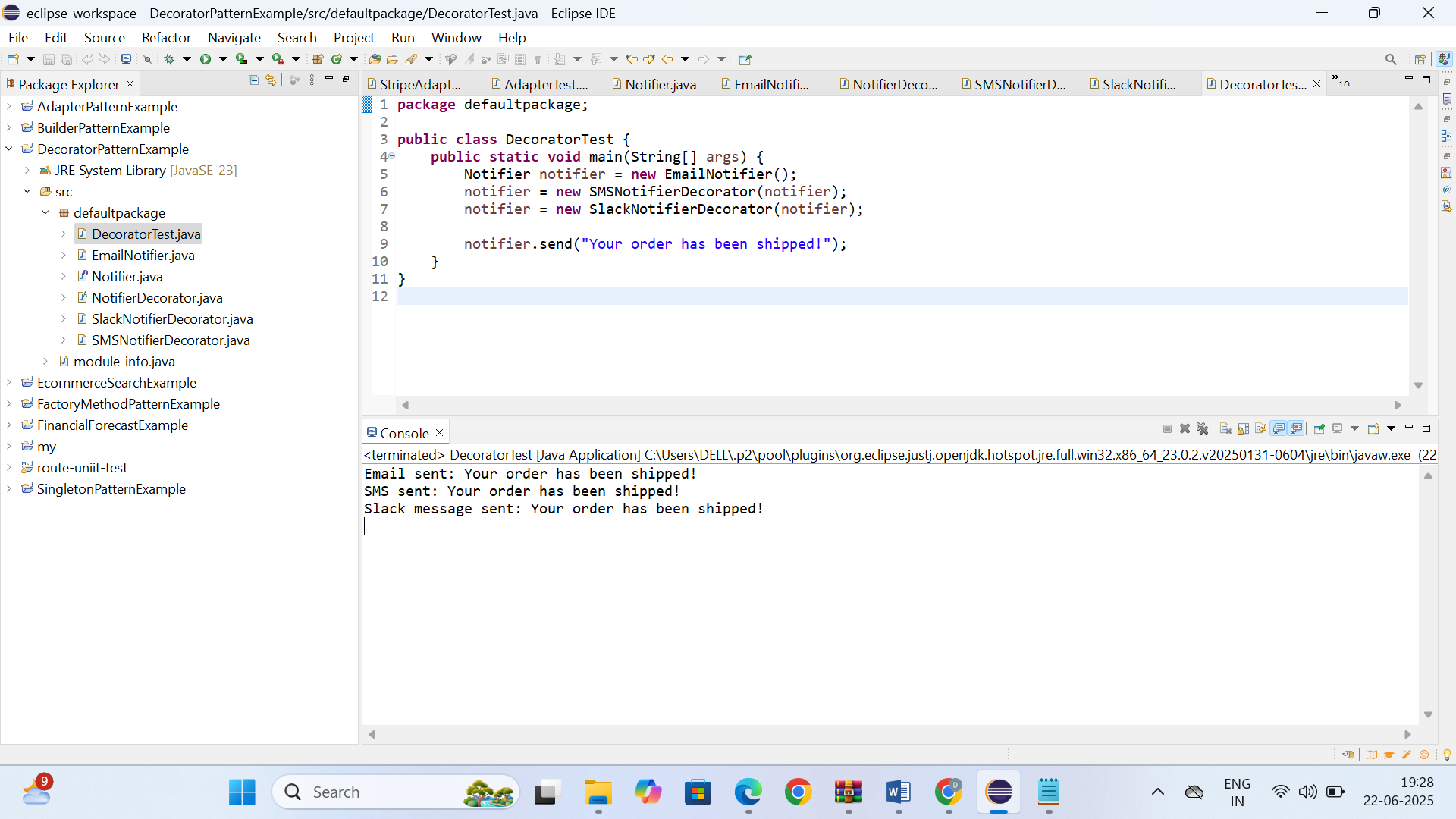
notifier = new SlackNotifierDecorator(notifier);

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

}

}

**OUTPUT**



**Java 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 image from remote server: " + filename);

}

public void display() {

System.out.println("Displaying image: " + 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 image1 = new ProxyImage("design-pattern.png");

image1.display();

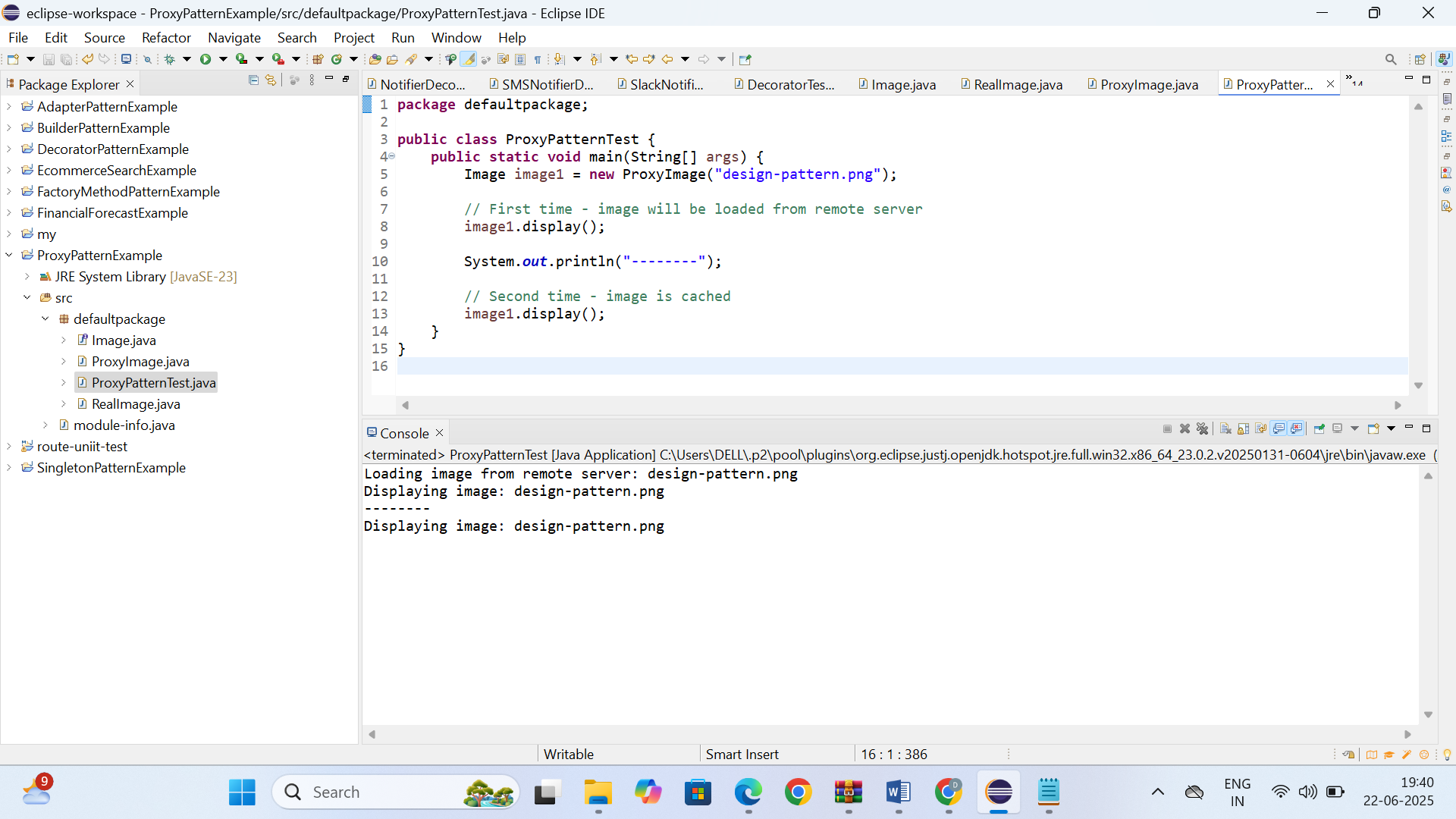
System.out.println("--------");

image1.display();

}

}

**OUTPUT**



* **Exercise 7: Implementing the Observer Pattern**

**Java Project: ObserverPatternExample**

**Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**Observer.java**

public interface Observer {

void update(float price);

}

**StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private float stockPrice;

public void setStockPrice(float price) {

this.stockPrice = price;

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

}

**MobileApp.java**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

public void update(float price) {

System.out.println(appName + " received stock price update: " + price);

}

}

**WebApp.java**

public class WebApp implements Observer {

private String siteName;

public WebApp(String siteName) {

this.siteName = siteName;

}

public void update(float price) {

System.out.println(siteName + " received stock price update: " + price);

}

}

**ObserverPatternTest.java**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp("StockTracker Mobile");

Observer web = new WebApp("FinanceWeb");

market.registerObserver(mobile);

market.registerObserver(web);

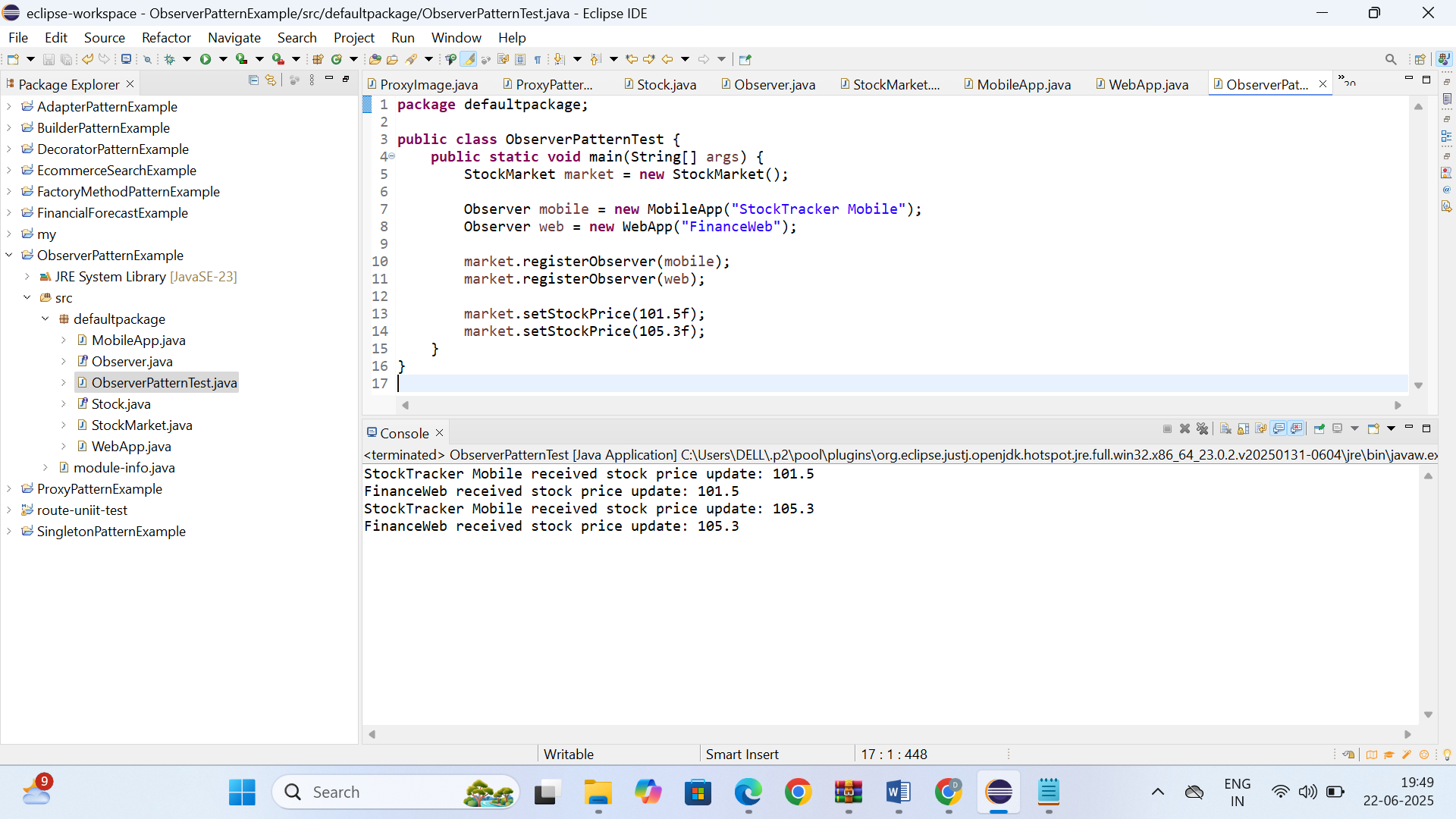
market.setStockPrice(101.5f);

market.setStockPrice(105.3f);

}

}

**OUTPUT**



* **Exercise 8: Implementing the Strategy Pattern**

**Java Project: StrategyPatternExample**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card: " + cardNumber);

}

}

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid " + amount + " using PayPal: " + email);

}

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

strategy.pay(amount);

}

}

**StrategyPatternTest.java**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.payAmount(250.0);

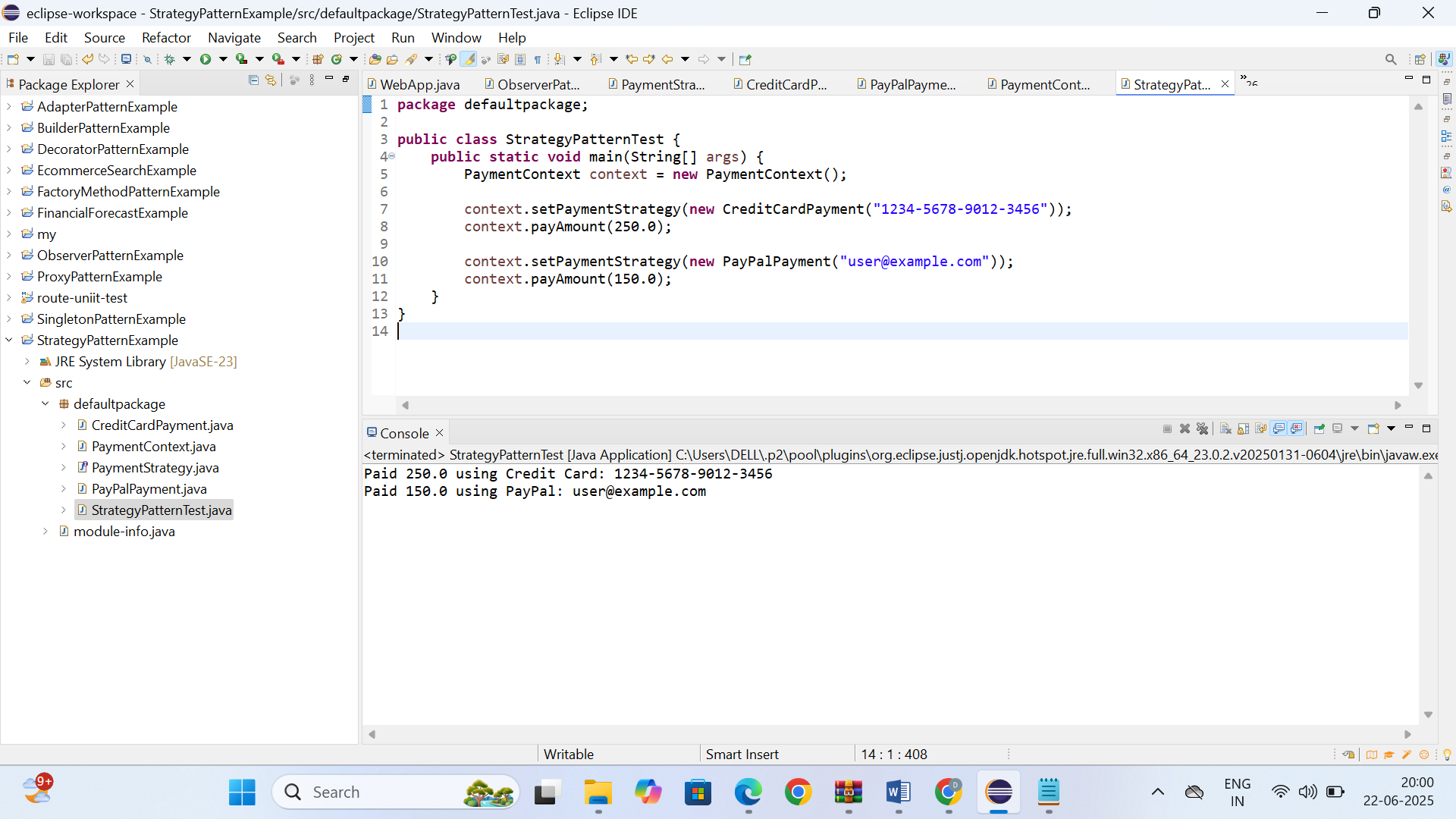
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.payAmount(150.0);

}

}

**OUTPUT**



* **Exercise 9: Implementing the Command Pattern**

**Java Project: CommandPatternExample**

**Command.java**

public interface Command {

void execute();

}

**Light.java**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**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();

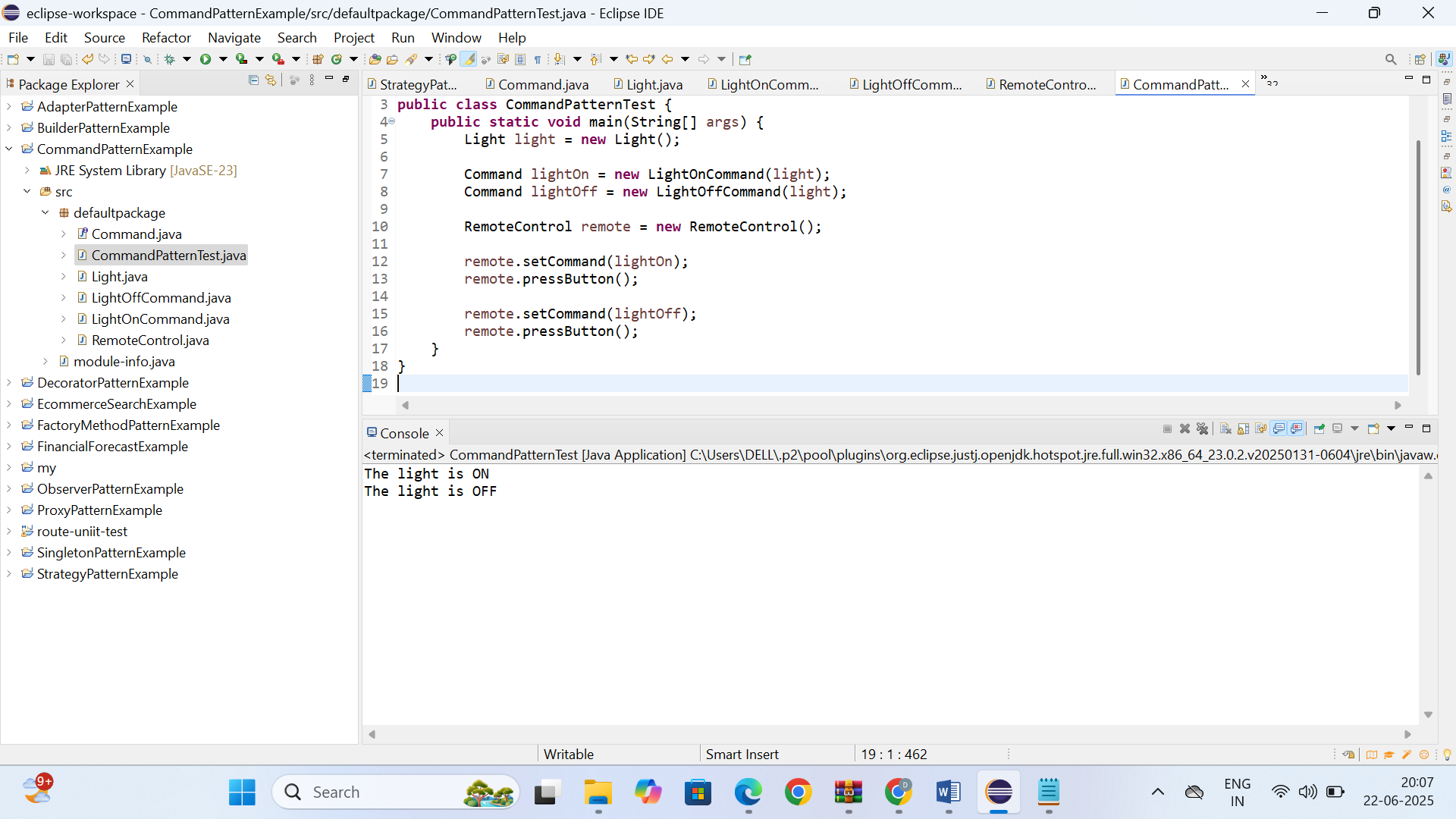
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT**



* **Exercise 10: Implementing the MVC Pattern**

**Java Project: MVCPatternExample**

**Student.java**

public class Student {

private String name;

private String id;

private String 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;

}

}

**StudentView.java**

public class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

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

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

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

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

}

}

**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 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());

}

}

**MVCPatternTest.java**

public class MVCPatternTest {

public static void main(String[] args) {

Student student = new Student();

student.setName("Alice");

student.setId("S101");

student.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Bob");

controller.setStudentGrade("B+");

controller.updateView();

}

}

**OUTPUT**

