**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.

**Implementation:**

*Logger.java*

**package** Singleton;

**public** **class** Logger {

**private** **static** Logger *instance*;

**private** Logger() {

System.***out***.println("Singleton instance created");

}

**public** **static** Logger getInstance() {

**if**(*instance*==**null**) {

*instance*=**new** Logger();

}

**return** *instance*;

}

**public** **void** log(String message) {

System.***out***.println("[LOG] " + message);

}

}

*Test.java*

**package** Singleton;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Logger obj1=Logger.*getInstance*();

obj1.log("First Log");

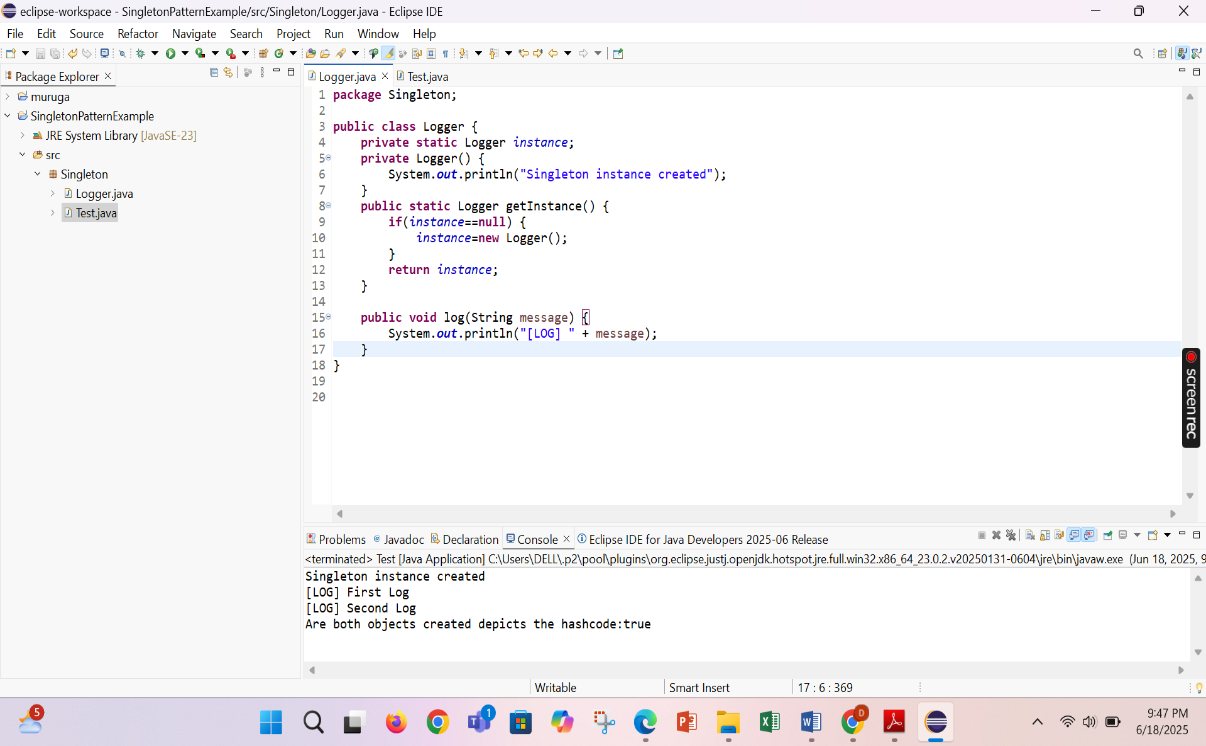
Logger obj2=Logger.*getInstance*();

obj2.log("Second Log");

System.***out***.println("Are both objects created depicts the hashcode:"+ (obj1==obj2));

}

}



**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.

*Document.java*

**package** FactoryMethodPattern;

**public** **interface** Document {

**void** open();

}

*WordDocument.java*

**package** FactoryMethodPattern;

**public** **class** WordDocument **implements** Document{

@Override

**public** **void** open() {

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

}

}

*PdfDocument.java*

**package** FactoryMethodPattern;

**public** **class** PdfDocument **implements** Document{

@Override

**public** **void** open() {

System.***out***.println("Opening Pdf Document");

}

}

*ExcelDocument.java*

**package** FactoryMethodPattern;

**public** **class** ExcelDocument **implements** Document{

@Override

**public** **void** open() {

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

}

}

*DocumentFactory.java*

**package** FactoryMethodPattern;

**public** **abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument();

}

*WordDocumentFactory.java*

**package** FactoryMethodPattern;

**public** **class** WordDocumentFactory **extends** DocumentFactory{

@Override

**public** Document createDocument() {

**return** **new** WordDocument();

}

}

*PdfDocumentFactory.java*

**package** FactoryMethodPattern;

**public** **class** PdfDocumentFactory **extends** DocumentFactory{

@Override

**public** Document createDocument() {

**return** **new** PdfDocument();

}

}

*ExcelDocumentFactory.java*

**package** FactoryMethodPattern;

**public** **class** PdfDocumentFactory **extends** DocumentFactory{

@Override

**public** Document createDocument() {

**return** **new** PdfDocument();

}

}

*FactoryPatternTest.java*

**package** FactoryMethodPattern;

**public** **class** FactoryPatternTest {

**public** **static** **void** main(String []args) {

DocumentFactory wfactory=**new** WordDocumentFactory();

Document wordDoc=wfactory.createDocument();

wordDoc.open();

DocumentFactory pfactory=**new** PdfDocumentFactory();

Document pdfDoc=pfactory.createDocument();

pdfDoc.open();

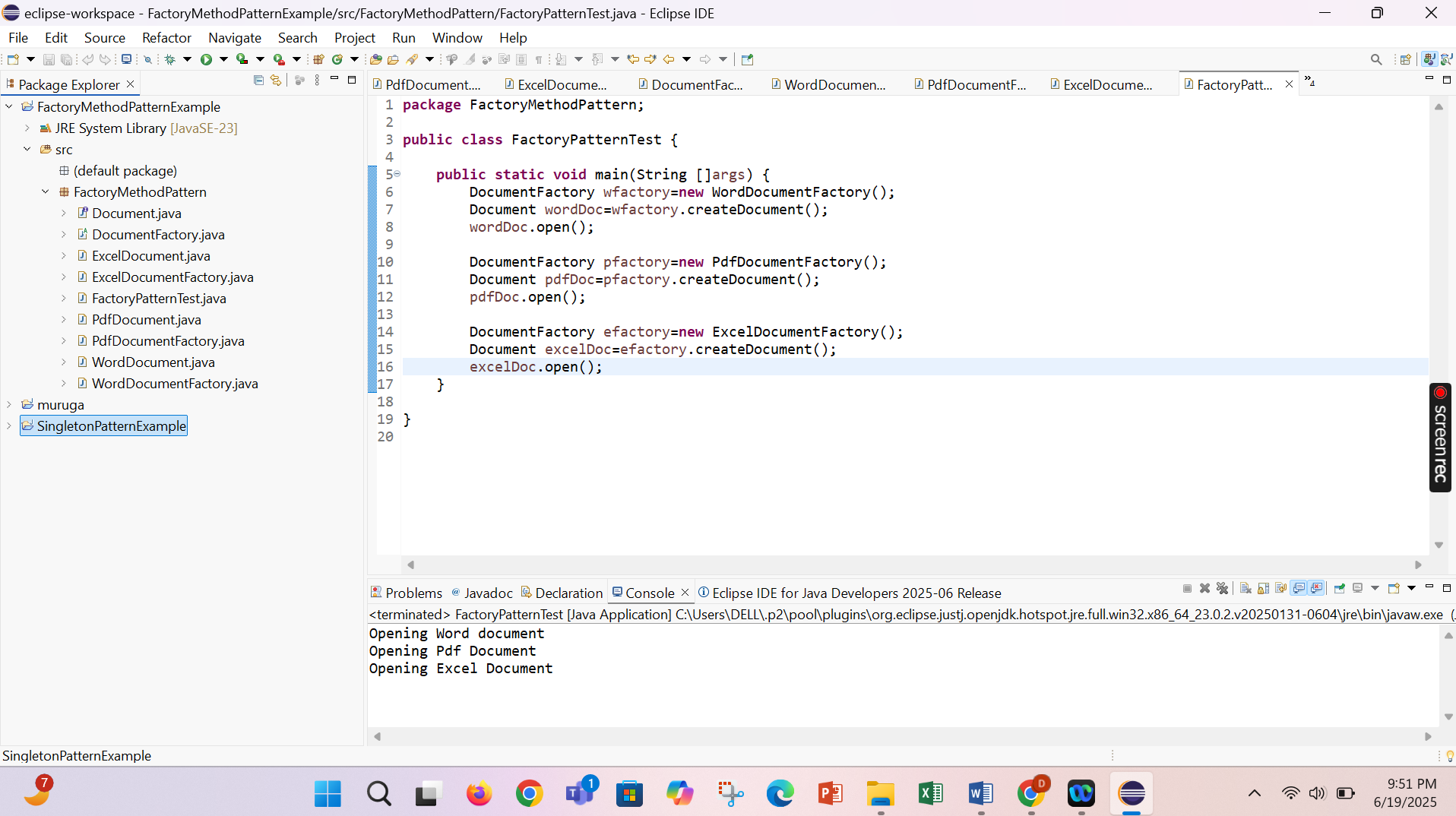
DocumentFactory efactory=**new** ExcelDocumentFactory();

Document excelDoc=efactory.createDocument();

excelDoc.open();

}

}



**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.

*Computer.java*

**package** BuilderPattern;

**import** java.security.KeyStore.Builder;

**public** **class** Computer {

**private** **final** String CPU;

**private** **final** String RAM;

**private** **final** String Storage;

**private** **boolean** hasGraphicsCard;

**private** **boolean** hasBluetooth;

**private** Computer(Builder builder){

**this**.CPU=builder.CPU;

**this**.RAM=builder.RAM;

**this**.Storage=builder.Storage;

**this**.hasGraphicsCard=builder.hasGraphicsCard;

**this**.hasBluetooth=builder.hasBluetooth;

}

**public** **void** showSpecs() {

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: "+(hasGraphicsCard?"Yes":"No"));

System.***out***.println("Bluetooth: "+(hasBluetooth?"Yes":"No"));

System.***out***.println();

}

**public** **static** **class** Builder{

**private** String CPU;

**private** String RAM;

**private** String Storage;

**private** **boolean** hasGraphicsCard;

**private** **boolean** hasBluetooth;

**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** Builder setBluetooth(**boolean** hasBluetooth) {

**this**.hasBluetooth = hasBluetooth;

**return** **this**;

}

**public** Computer build() {

**return** **new** Computer(**this**);

}

}

}

*BuilderPatternTest.java*

**package** BuilderPattern;

**public** **class** BuilderPatternTest {

**public** **static** **void** main(String[] args) {

Computer basicComputer = **new** Computer.Builder()

.setCPU("Intel i3")

.setRAM("4GB")

.setStorage("256GB SSD")

.build();

basicComputer.showSpecs();

Computer gamingComputer = **new** Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard(**true**)

.setBluetooth(**true**)

.build();

gamingComputer.showSpecs();

Computer officeComputer = **new** Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

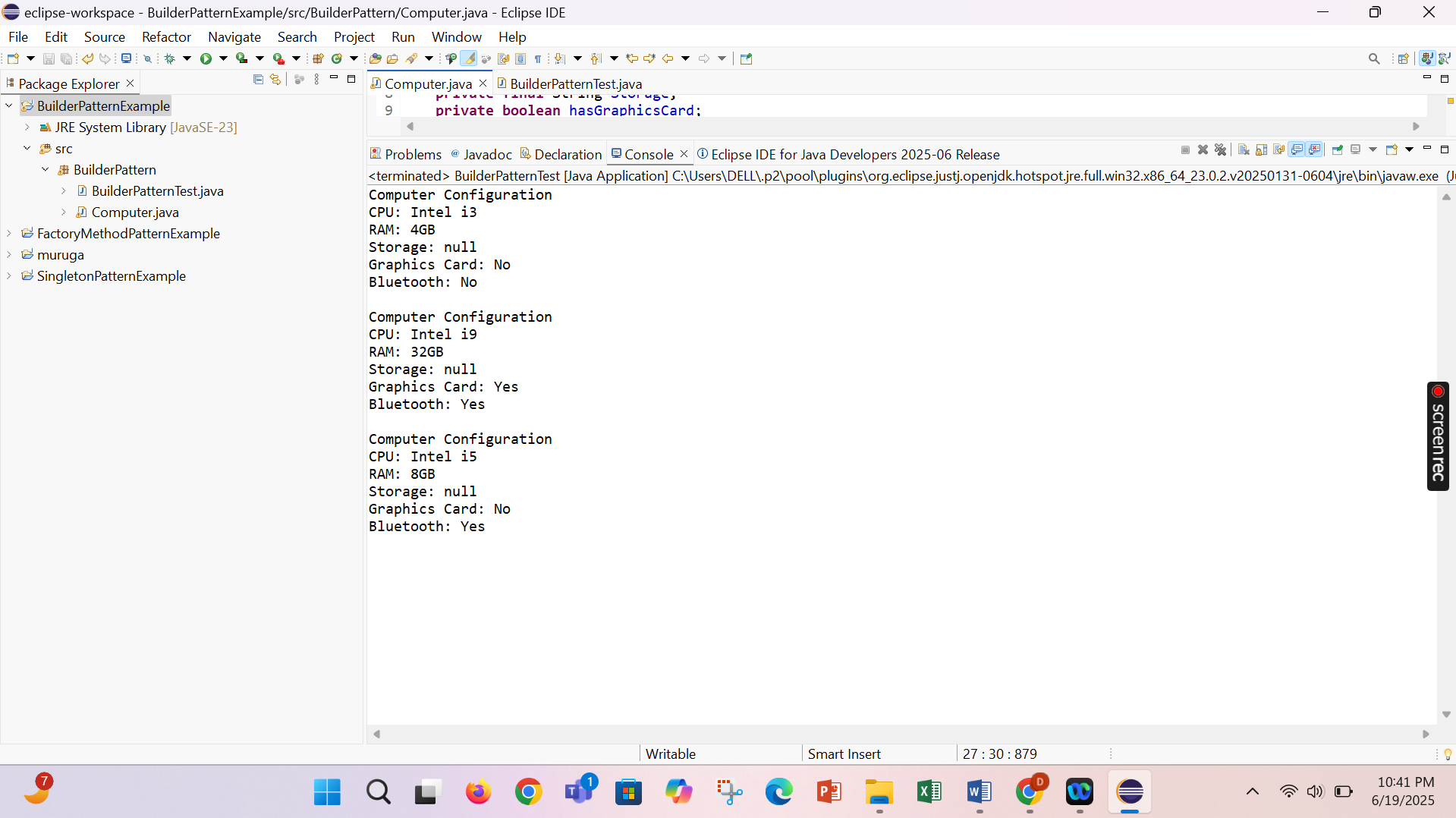
.setBluetooth(**true**)

.build();

officeComputer.showSpecs();

}

}



**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.

PaymentProcessor.java

**package** AdaperPattern;

**public** **interface** PaymentProcessor {

**public** **void** processPayment(**double** amount);

}

PaypalAdapter.java

**package** AdaperPattern;

**public** **class** PaypalAdapter **implements** PaymentProcessor{

**private** PaypalGateway pay;

**public** PaypalAdapter(PaypalGateway pay) {

**this**.pay=pay;

}

@Override

**public** **void** processPayment(**double** amount) {

pay.sendPayment(amount);

}

}

StripeAdapter.java

**package** AdaperPattern;

**public** **class** StripeAdapter **implements** PaymentProcessor{

**private** StripeGateway stripe;

**public** StripeAdapter(StripeGateway stripe) {

**this**.stripe=stripe;

}

@Override

**public** **void** processPayment(**double** amount) {

stripe.makePayment(amount);

}

}

GpayAdapter.java

**package** AdaperPattern;

**public** **class** GpayAdapter **implements** PaymentProcessor{

**private** Gpay gpay;

**public** GpayAdapter(Gpay gpay) {

**this**.gpay=gpay;

}

@Override

**public** **void** processPayment(**double** amount) {

gpay.payPayment(amount);

}

}

PaypalGateway.java

**package** AdaperPattern;

**public** **class** PaypalGateway {

**public** **void** sendPayment(**double** amount) {

System.***out***.println("Paid "+ amount+" using paypal");

}

}

Gpay.java

**package** AdaperPattern;

**public** **class** Gpay {

**public** **void** payPayment(**double** amount) {

System.***out***.println("Paid "+ amount+" using Gpay");

}

}

StripeGateway.java

**package** AdaperPattern;

**public** **class** StripeGateway {

**public** **void** makePayment(**double** amount) {

System.***out***.println("Paid "+ amount+" using Stripe Payment");

}

}

Test.java

**package** AdaperPattern;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

PaymentProcessor dhar=**new** PaypalAdapter(**new** PaypalGateway());

dhar.processPayment(45.00);

PaymentProcessor dhar1=**new** StripeAdapter(**new** StripeGateway());

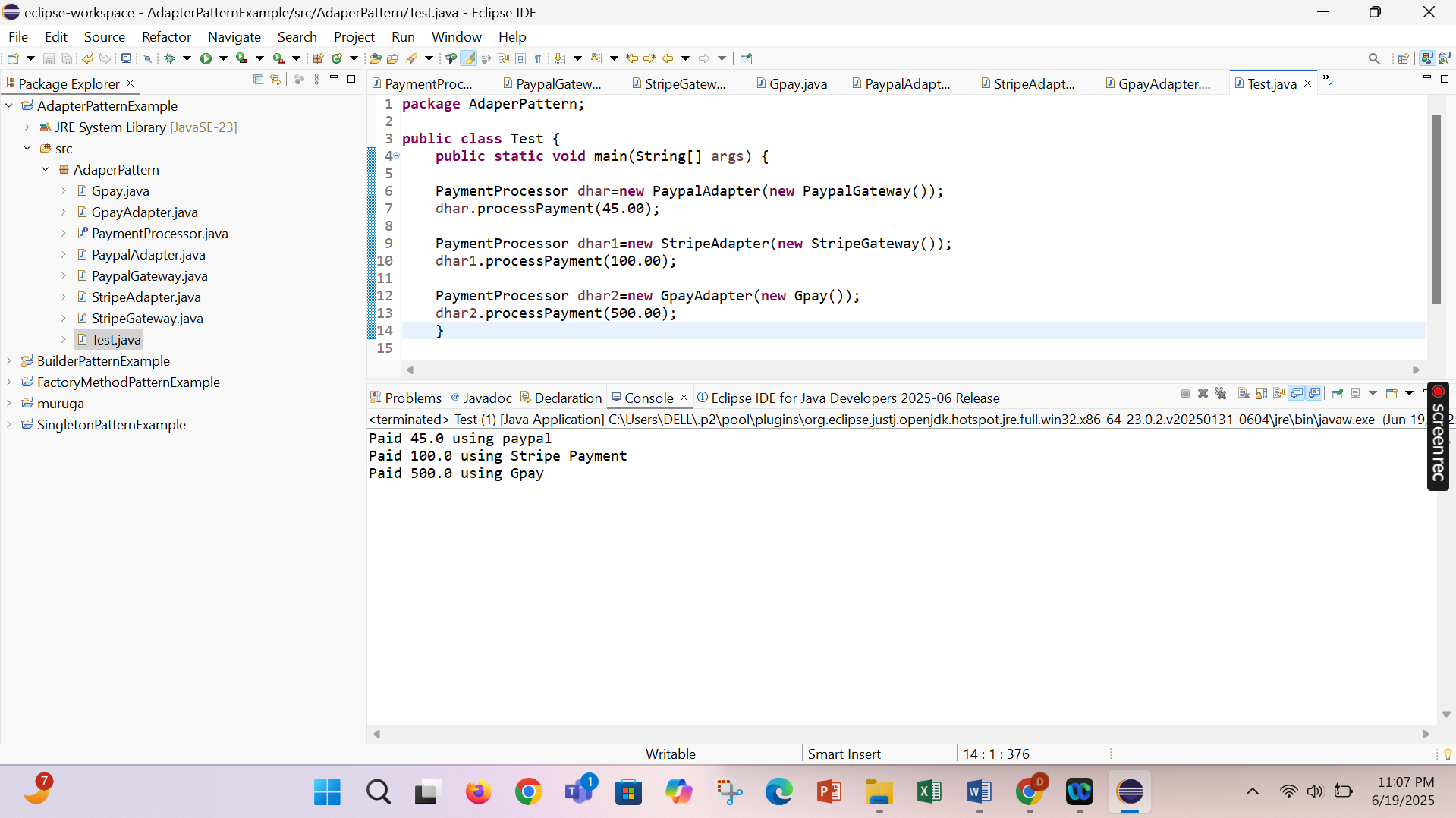
dhar1.processPayment(100.00);

PaymentProcessor dhar2=**new** GpayAdapter(**new** Gpay());

dhar2.processPayment(500.00);

}

}



**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.

Notifier.java

**package** DecoratorPattern;

**public** **interface** Notifier {

**public** **void** send(String msg);

}

EmailNotifier.java

**package** DecoratorPattern;

**public** **class** EmailNotifier **implements** Notifier{

@Override

**public** **void** send(String msg) {

System.***out***.println("Sending Email: "+msg);

}

}

NotifierDecorator.java

**package** DecoratorPattern;

**public** **abstract** **class** NotifierDecorator **implements** Notifier{

**protected** Notifier wrapped;

**public** NotifierDecorator(Notifier notifier) {

**this**.wrapped=notifier;

}

@Override

**public** **void** send(String msg) {

wrapped.send(msg);

}

}

SMSNotifierDecorator.java

**package** DecoratorPattern;

**public** **class** SMSNotifierDecorator **extends** NotifierDecorator{

**public** SMSNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

**public** **void** send(String message) {

**super**.send(message);

sendSMS(message);

}

**private** **void** sendSMS(String message) {

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

}

}

SlackNotifierDecorator.java

**package** DecoratorPattern;

**public** **class** SlackNotifierDecorator **extends** NotifierDecorator{

**public** SlackNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

**public** **void** send(String message) {

**super**.send(message);

sendSlack(message);

}

**private** **void** sendSlack(String message) {

System.***out***.println("Sending Slack Message: " + message);

}

}

DecoratorPattern.java

**package** DecoratorPattern;

**public** **class** DecoratorPatternTest {

**public** **static** **void** main(String[] args) {

Notifier notifier = **new** EmailNotifier();

Notifier multiChannelNotifier = **new** SlackNotifierDecorator(

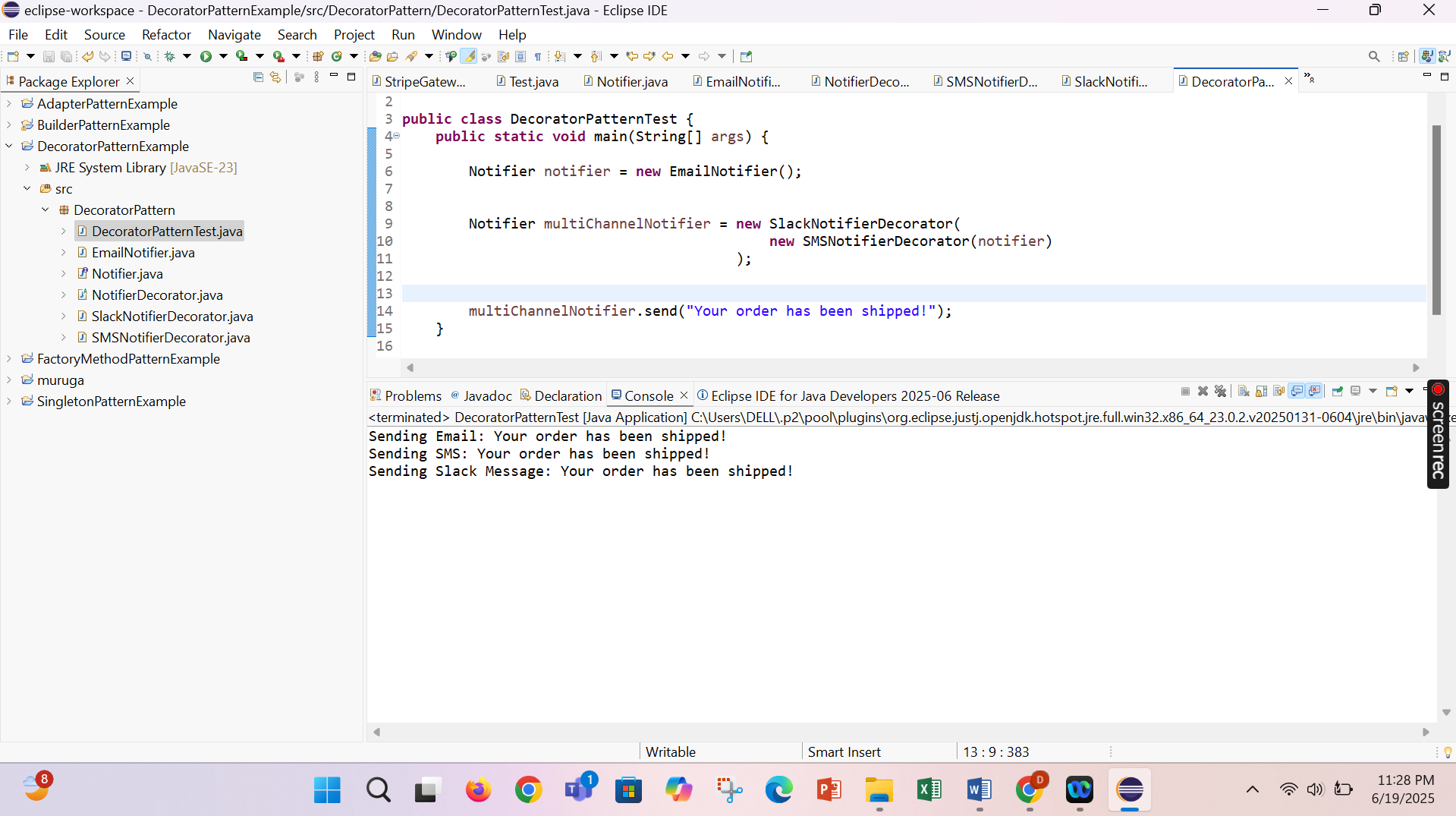
**new** SMSNotifierDecorator(notifier)

);

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

}

}



**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.

Image.java

**package** ProxyPattern;

**public** **interface** Image {

**public** **void** display();

}

RealImage.java

**package** ProxyPattern;

**public** **class** RealImage **implements** Image{

**private** String filename;

**public** RealImage(String filename) {

**this**.filename=filename;

loadfromserver();

}

**private** **void** loadfromserver() {

System.***out***.println("Loading "+ filename + " from server");

}

@Override

**public** **void** display() {

System.***out***.println("Displaying Image: "+filename);

}

}

ProxyImage.java

**package** ProxyPattern;

**public** **class** ProxyImage **implements** Image{

**private** RealImage realimage;

**private** String filename;

**public** ProxyImage(String filename) {

**this**.filename=filename;

}

@Override

**public** **void** display() {

**if**(realimage==**null**) {

realimage=**new** RealImage(filename);

}

realimage.display();

}

}

Test.java

**package** ProxyPattern;

**public** **class** Test {

**public** **static** **void** main(String []args) {

Image image=**new** ProxyImage("XYZ");

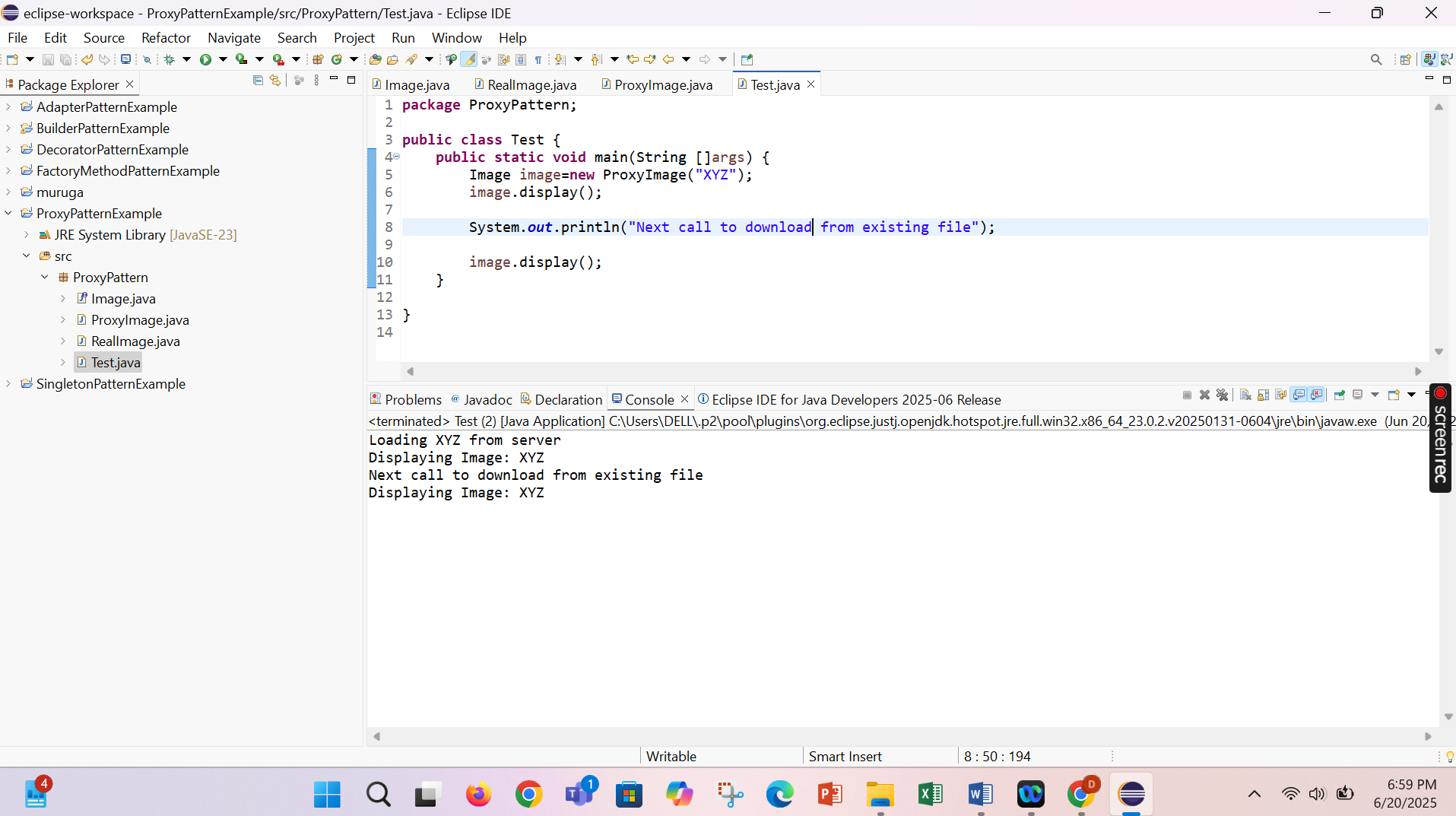
image.display();

System.***out***.println("Next call to download from existing file");

image.display();

}

}



**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.

Stock.java

**package** ObserverPattern;

**public** **interface** Stock {

**void** register(Observer obs);

**void** deregister(Observer obs);

**void** notifyobs();

}

Observer.java

**package** ObserverPattern;

**public** **interface** Observer {

**void** update(String stockName,**double** price);

}

StockMarket.java

**package** ObserverPattern;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StockMarket **implements** Stock{

**private** List<Observer> observer=**new** ArrayList<>();

**private** String stockName;

**private** **double** price;

**public** **void** setStockData(String stockname,**double** price) {

**this**.stockName=stockname;

**this**.price=price;

notifyobs();

}

**public** **void** register(Observer obs) {

observer.add(obs);

}

**public** **void** deregister(Observer obs) {

observer.remove(obs);

}

**public** **void** notifyobs() {

**for**(Observer obs:observer) {

obs.update(stockName,price);

}

}

}

WebApp.java

**package** ObserverPattern;

**public** **class** WebApp **implements** Observer{

**private** String siteName;

**public** WebApp(String siteName) {

**this**.siteName=siteName;

}

@Override

**public** **void** update(String stockName, **double** price) {

// **TODO** Auto-generated method stub

System.***out***.println(siteName+" receives update: "+stockName+" is now at Rs: "+price);

}

}

MobileApp.java

**package** ObserverPattern;

**public** **class** MobileApp **implements** Observer{

**private** String appName;

**public** MobileApp(String appName) {

**this**.appName=appName;

}

@Override

**public** **void** update(String stockName, **double** price) {

// **TODO** Auto-generated method stub

System.***out***.println(appName+" receives update: "+stockName+" is now at Rs: "+price);

}

}

TestObserver.java

**package** ObserverPattern;

**public** **class** TestObserverPattern {

**public** **static** **void** main(String[] args) {

StockMarket st=**new** StockMarket();

Observer mobile=**new** MobileApp("My Mobile");

Observer web=**new** WebApp("My Web");

st.register(mobile);

st.register(web);

System.***out***.println("Updating stock:AAPL");

st.setStockData("AAPL ",150.25);

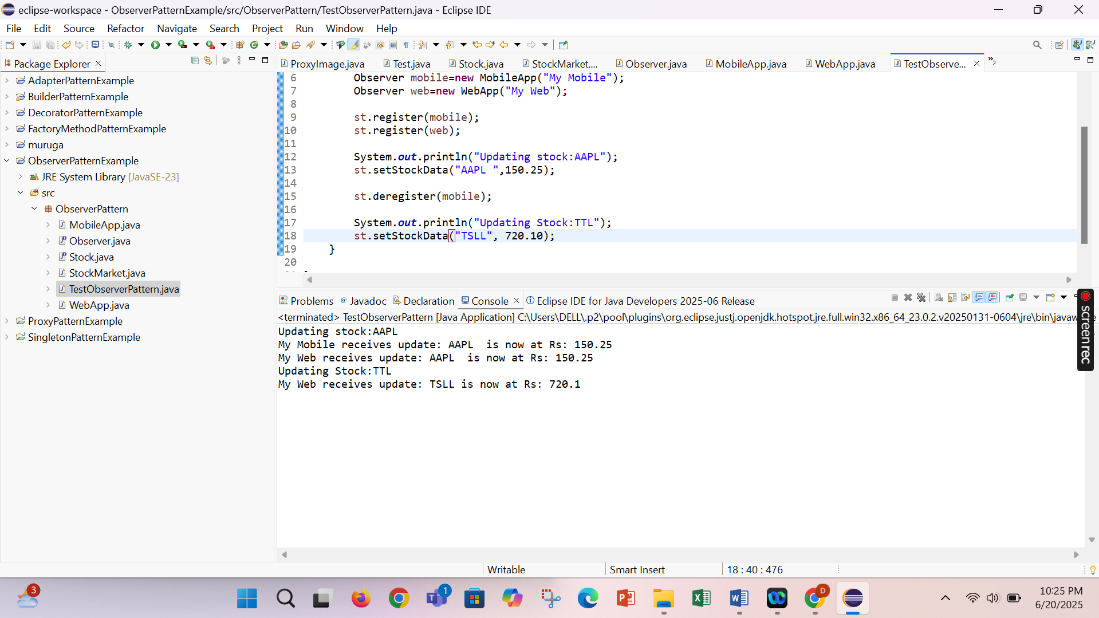
st.deregister(mobile);

System.***out***.println("Updating Stock:TTL");

st.setStockData("TSLL", 720.10);

}

}



**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.

PaymentStrategy.java

**package** StrategyPattern;

**public** **interface** PaymentStrategy {

**void** pay(**double** amount);

}

CredutcardPayment.java

**package** StrategyPattern;

**public** **class** Creditcardpayment **implements** PaymentStrategy{

@Override

**public** **void** pay(**double** amount) {

System.***out***.println("Making Payment "+amount+" by Credit");

}

}

PaypalPayment.java

**package** StrategyPattern;

**public** **class** PayPalPayment **implements** PaymentStrategy{

@Override

**public** **void** pay(**double** amount) {

System.***out***.println("Making payment "+amount+" by PayPal");

}

}

PaymentContext.java

**package** StrategyPattern;

**public** **class** PaymentContext {

**private** PaymentStrategy payment;

**public** PaymentContext(PaymentStrategy payment) {

**this**.payment=payment;

}

**public** **void** setPayment(PaymentStrategy payment) {

**this**.payment=payment;

}

**public** **void** pay(**double** amount) {

payment.pay(amount);

}

}

Test.java

**package** StrategyPattern;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

PaymentContext paym=**new** PaymentContext(**new** Creditcardpayment());

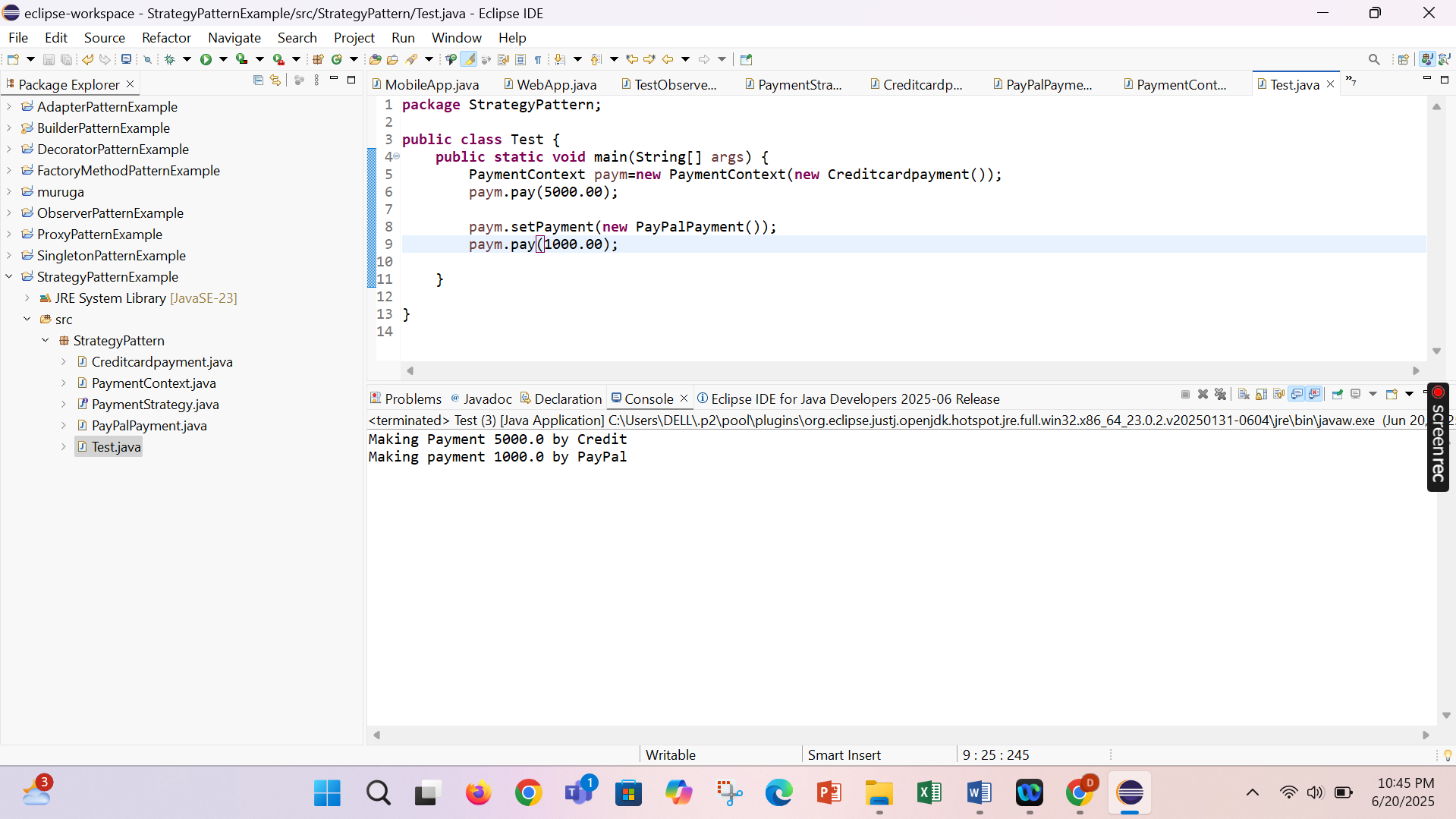
paym.pay(5000.00);

paym.setPayment(**new** PayPalPayment());

paym.pay(1000.00);

}

}



**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**.

Command.java

**package** CommandPattern;

**public** **interface** Command {

**void** execute();

}

Light.java

**package** CommandPattern;

**public** **class** Light {

**public** **void** turnon() {

System.***out***.println("Light On");

}

**public** **void** turnOff() {

System.***out***.println("Light Off");

}

}

LightOnCommand.java

**package** CommandPattern;

**public** **class** LightOnCommand **implements** Command{

**private** Light light;

**public** LightOnCommand(Light light) {

**this**.light=light;

}

@Override

**public** **void** execute() {

light.turnon();

}

}

LightOffCommand.java

**package** CommandPattern;

**public** **class** LightOffCommand **implements** Command{

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light=light;

}

@Override

**public** **void** execute() {

light.turnOff();

}

}

RemoteControl.java

**package** CommandPattern;

**public** **class** LightOffCommand **implements** Command{

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light=light;

}

@Override

**public** **void** execute() {

light.turnOff();

}

}

Test.java

**package** CommandPattern;

**public** **class** Test {

**public** **static** **void** main(String[ ] args) {

Light dhar=**new** Light();

Command lighton=**new** LightOnCommand(dhar);

Command lightoff=**new** LightOffCommand(dhar);

RemoteControl rc=**new** RemoteControl();

rc.SetCommand(lighton);

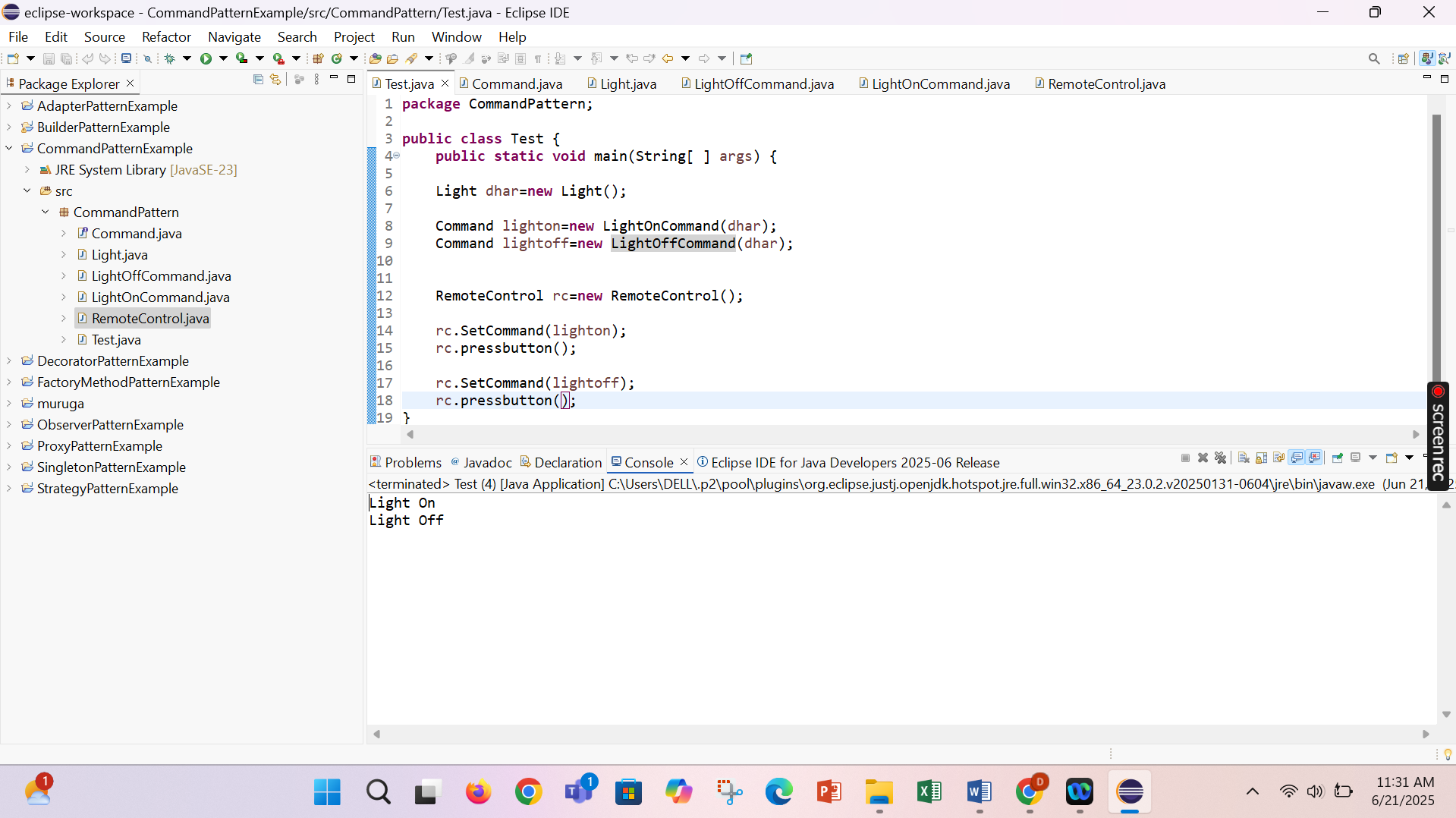
rc.pressbutton();

rc.SetCommand(lightoff);

rc.pressbutton();

}

}



**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**.

Student.java

**package** MVCPattern;

**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** **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;

}

}

StudentController.java

**package** MVCPattern;

**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** getStudentName() {

model.getName();

}

**public** **void** setStudentid(String id) {

model.setID(id);

}

**public** **void** getStudentId() {

model.getId();

}

**public** **void** setStudentGrade(String grade) {

model.setGrade(grade);

}

**public** **void** getStudentGrade() {

model.getGrade();

}

**public** **void** updateDetails() {

view.display(model.getName(),model.getId(),model.getGrade());

}

}

StudentView.java

**package** MVCPattern;

**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** getStudentName() {

model.getName();

}

**public** **void** setStudentid(String id) {

model.setID(id);

}

**public** **void** getStudentId() {

model.getId();

}

**public** **void** setStudentGrade(String grade) {

model.setGrade(grade);

}

**public** **void** getStudentGrade() {

model.getGrade();

}

**public** **void** updateDetails() {

view.display(model.getName(),model.getId(),model.getGrade());

}

}

Test.java

**package** MVCPattern;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Student stu=**new** Student("Dharshini","D40","A");

StudentView view=**new** StudentView();

StudentController control=**new** StudentController(stu,view);

control.updateDetails();

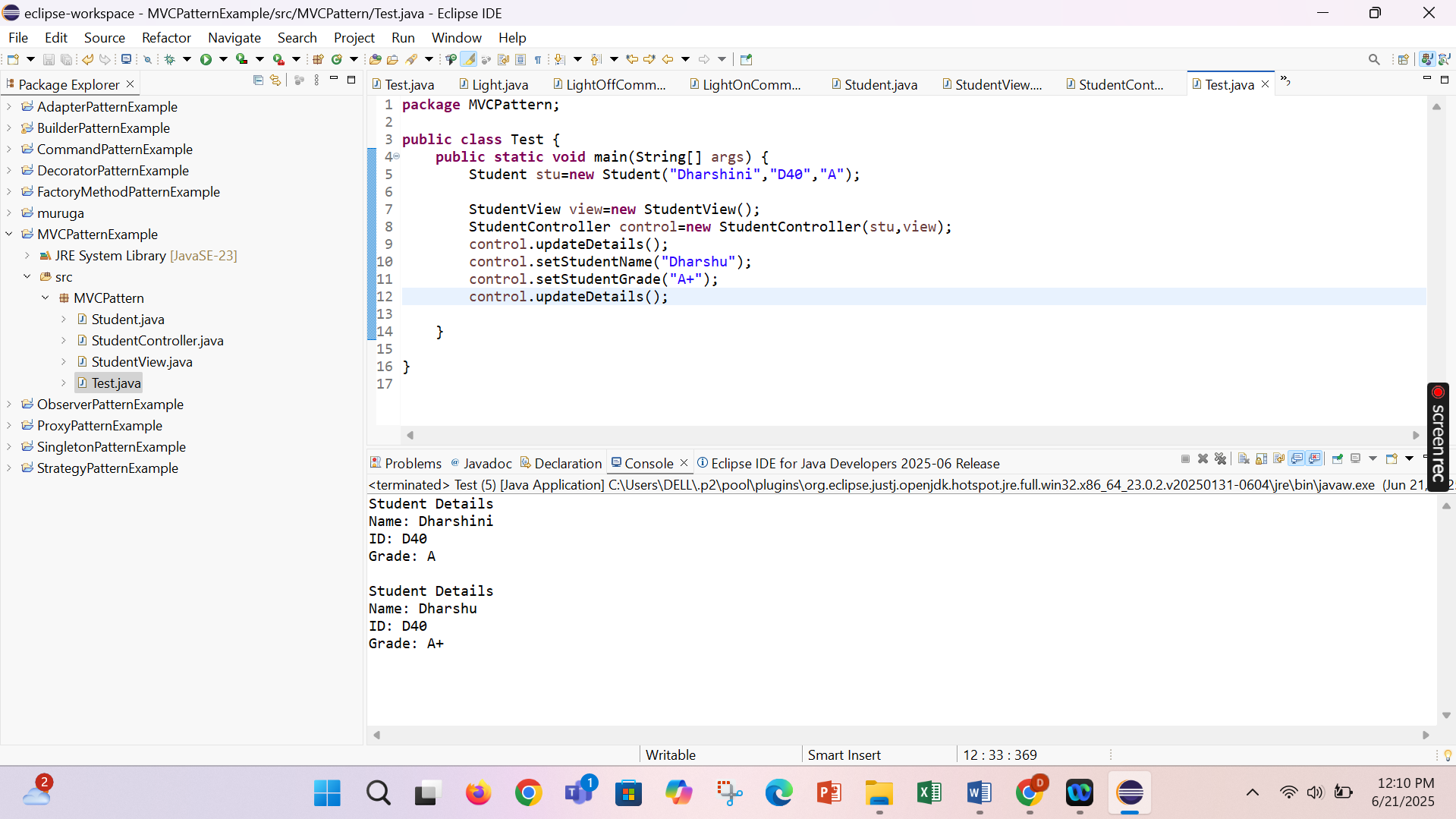
control.setStudentName("Dharshu");

control.setStudentGrade("A+");

control.updateDetails();

}

}



**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.

CustomerRepository.java

**package** DependancyInjection;

**public** **interface** CustomerRepository {

String findCustomerById(String id);

}

CustomerRepositoryImpl.java

**package** DependancyInjection;

**public** **interface** CustomerRepository {

String findCustomerById(String id);

}

CustomerService.java

**package** DependancyInjection;

**public** **class** CustomerService{

**private** CustomerRepository customerRepository;

**public** CustomerService(CustomerRepository customerRepository) {

**this**.customerRepository=customerRepository;

}

**public** **void** getCustomerDetails(String id) {

String customer=customerRepository.findCustomerById(id);

System.***out***.println("Retrieved: "+customer);

}

}

Test.java

**package** DependancyInjection;

**public** **class** Test {

**public** **static** **void** main(String []args) {

CustomerRepository dhar=**new** CustomerRepositoryImpl();

CustomerService service=**new** CustomerService(dhar);

service.getCustomerDetails("CUST101");

}

}

