**Exercise 3: Implementing the Builder Pattern**

**Code:**

**Computer.java:**

public class Computer {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

public void displayConfiguration() {

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

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

System.out.println("RAM: " + (ram != null ? ram : "Not Included"));

System.out.println("Storage: " + (storage != null ? storage : "Not Included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not Included"));

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

public Builder(String cpu) {

this.cpu = cpu;

}

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);

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i3").build();

Computer gamingPC = new Computer.Builder("AMD Ryzen 7")

.setRam("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4060")

.build();

Computer workLaptop = new Computer.Builder("Intel i5")

.setRam("16GB")

.setStorage("512GB SSD")

.build();

System.out.println("Basic Computer:");

basicComputer.displayConfiguration();

System.out.println("\nGaming PC:");

gamingPC.displayConfiguration();

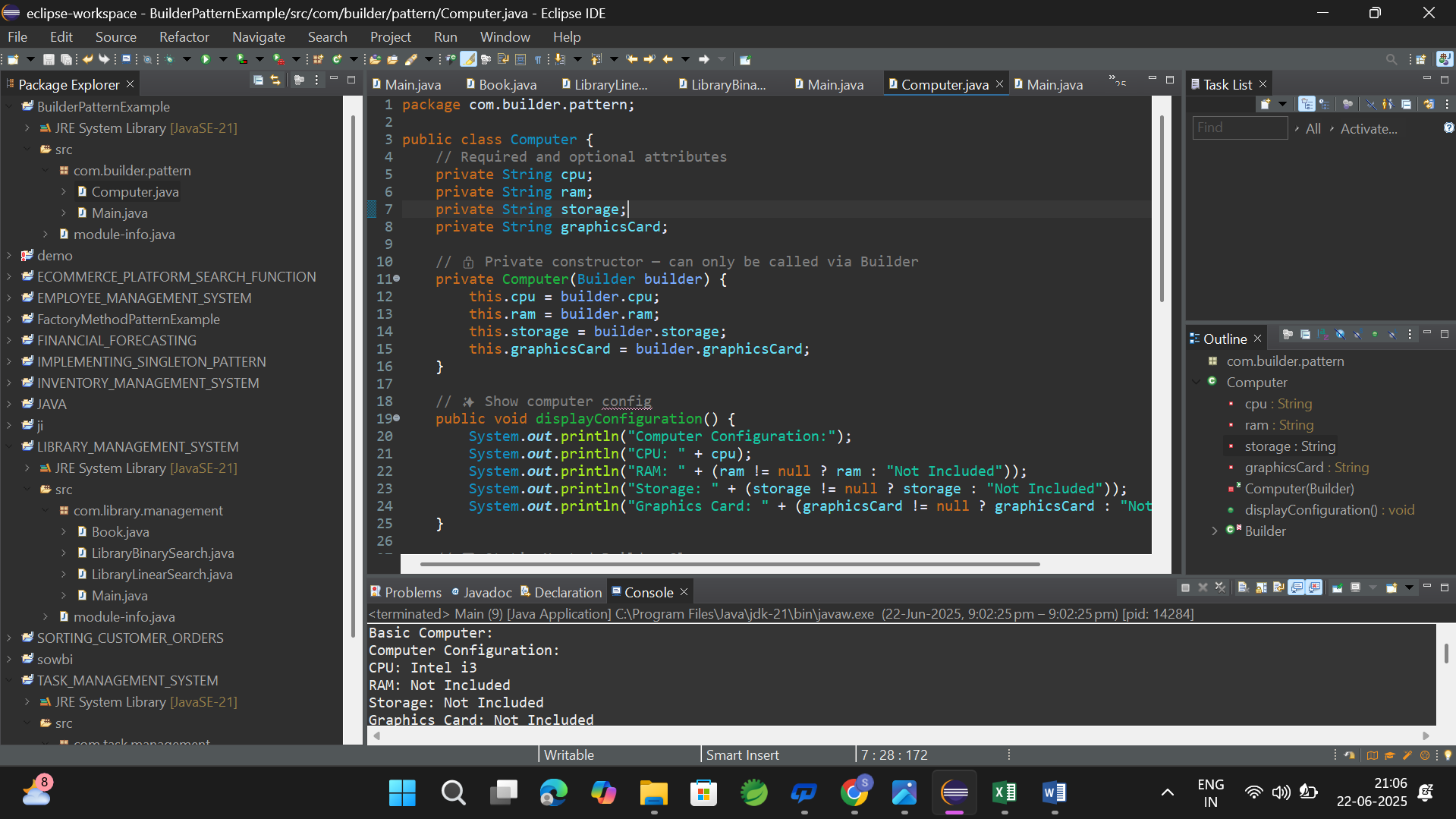
System.out.println("\nWork Laptop:");

workLaptop.displayConfiguration();

}

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**PaymentProcess.java:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**StripeGateway.java:**

public class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Stripe payment of $" + amount + " processed.");

}

}

**PayPalGateway.java:**

public class PayPalGateway {

public void sendPayPalAmount(double amount) {

System.out.println("PayPal payment of $" + amount + " processed.");

}

}

**StripeAdapter.java:**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

**PayPalAdapter.java:**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendPayPalAmount(amount);

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

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

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

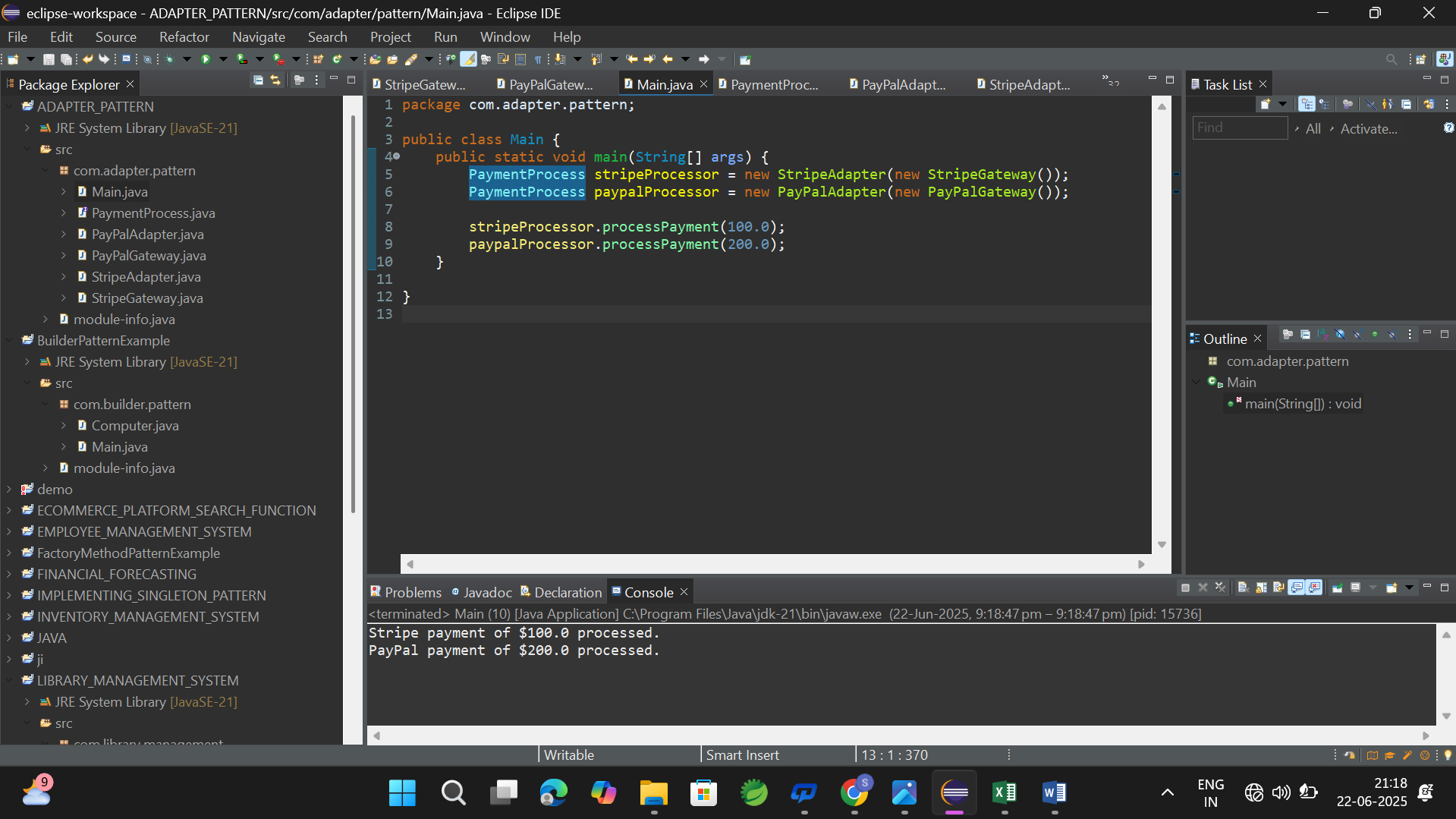
stripeProcessor.processPayment(100.0);

paypalProcessor.processPayment(200.0);

}

}

**Output:**



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

**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);

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Notifier basicNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(basicNotifier);

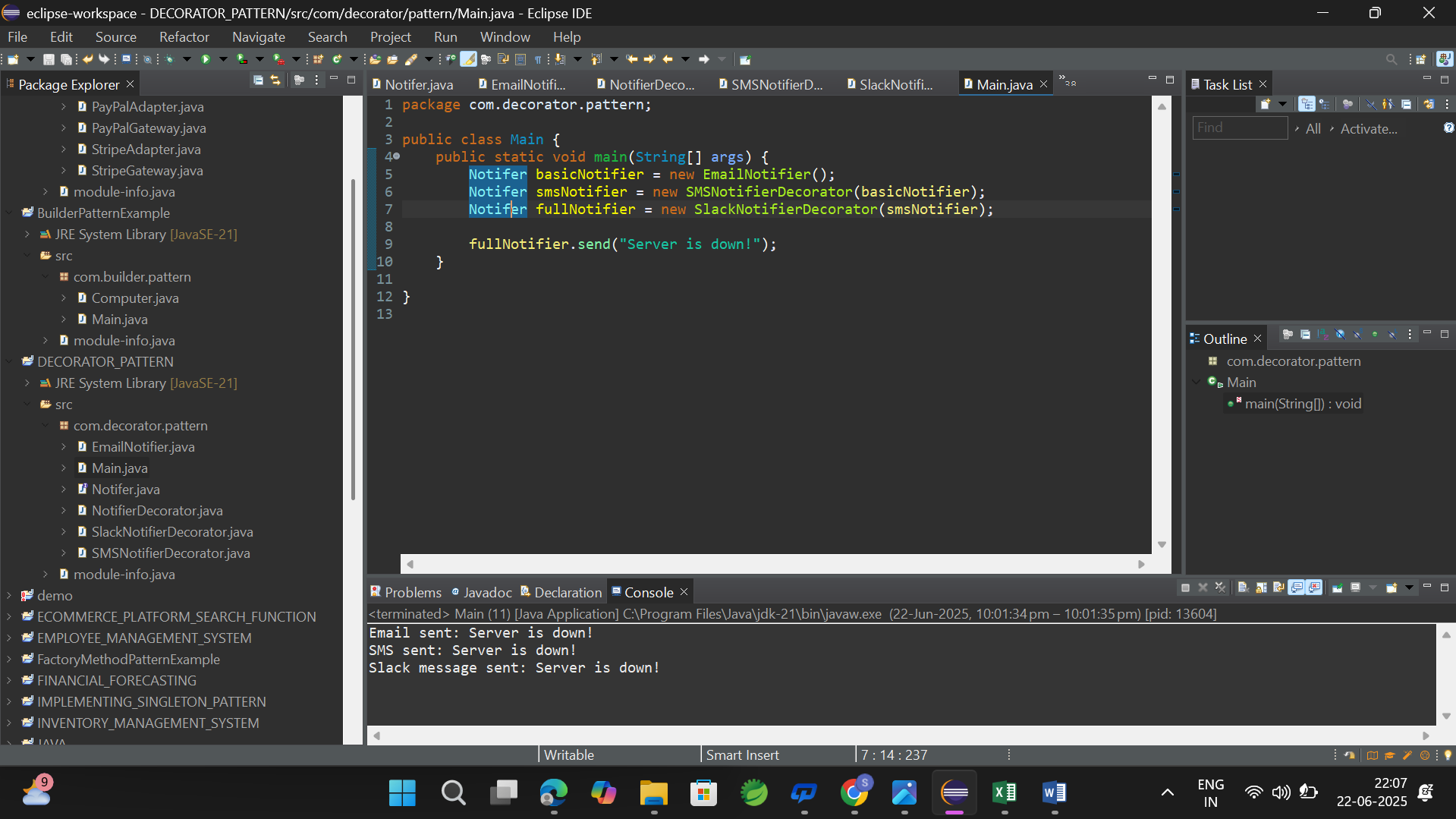
Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

fullNotifier.send("Server is down!");

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern  
  
  
Code:**

**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();

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("photo1.jpg");

Image img2 = new ProxyImage("photo2.jpg");

img1.display();

System.out.println();

img1.display();

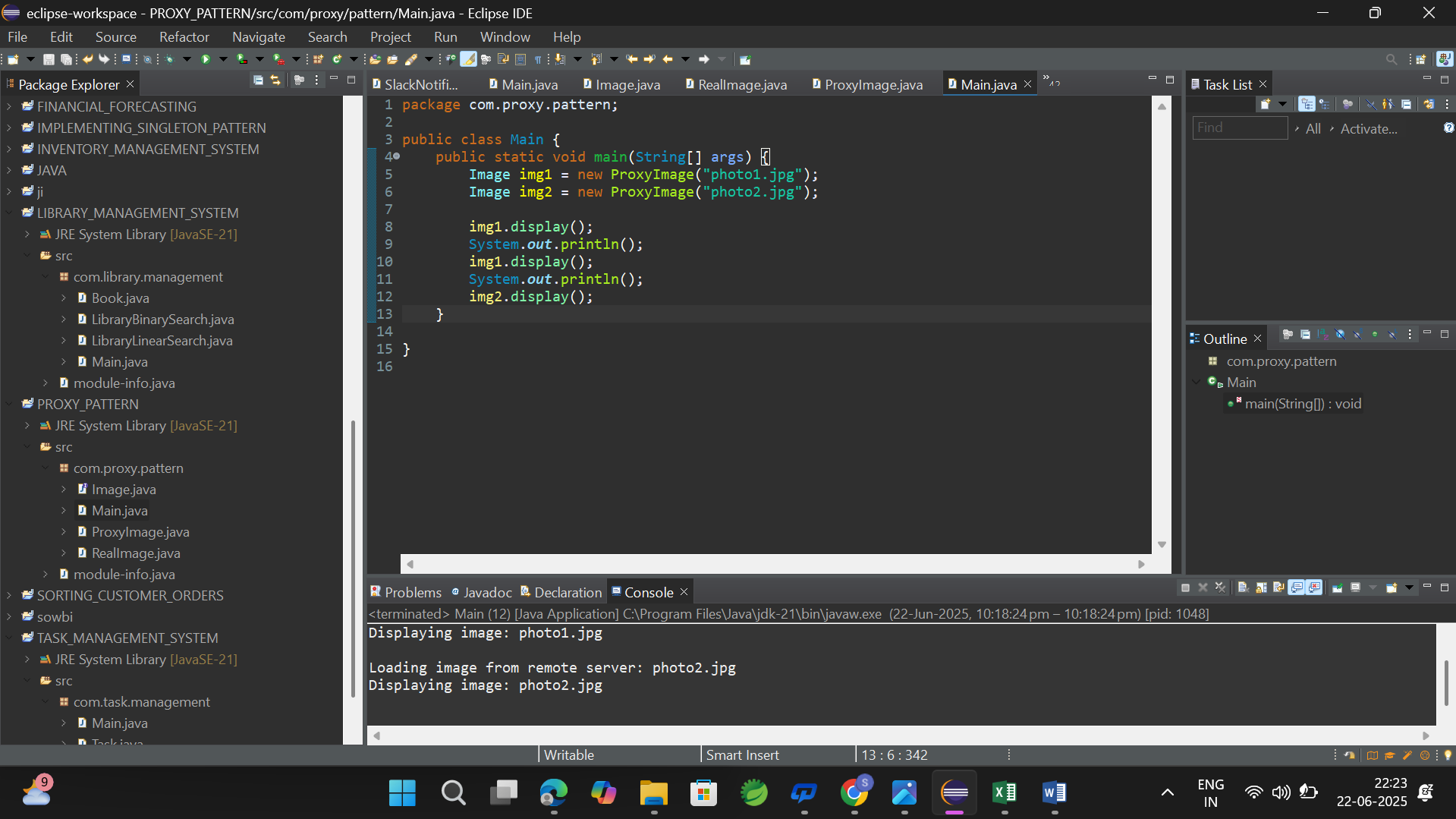
System.out.println();

img2.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Stock.java:**

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

**Observer.java:**

public interface Observer {

void update(String stockName, double price);

}

**StockMarket.java:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

**MobileApp.java:**

public class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp: " + stockName + " updated to $" + price);

}

}

**WebApp.java:**

public class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp: " + stockName + " updated to $" + price);

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

stockMarket.register(mobileApp);

stockMarket.register(webApp);

stockMarket.setStock("Apple", 180.25);

stockMarket.setStock("Google", 2750.10);

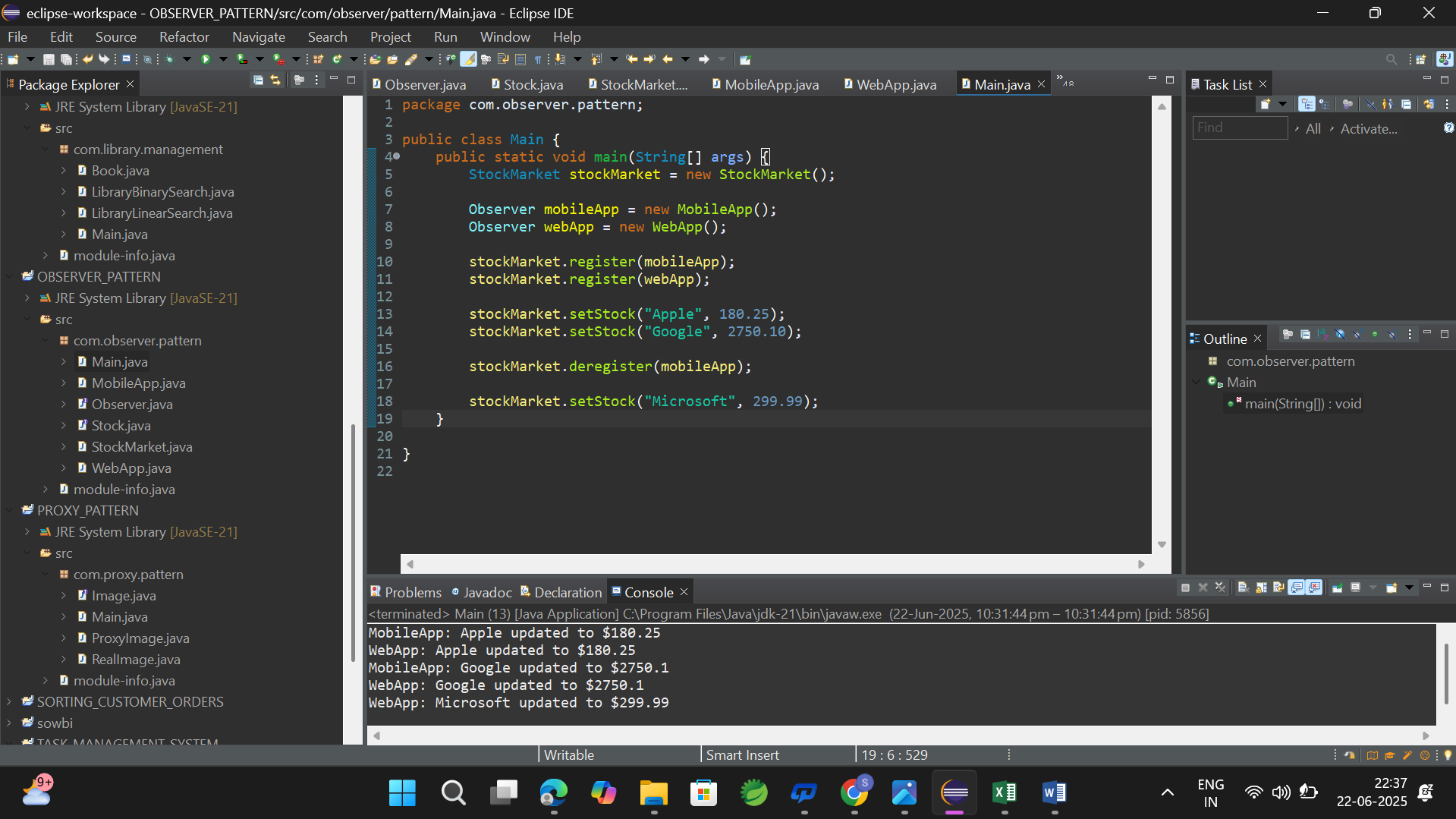
stockMarket.deregister(mobileApp);

stockMarket.setStock("Microsoft", 299.99);

}

}

**Output:**



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

**PaymentStrategy.java:**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java:**

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

**PayPalPayment.java:**

public class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

**PaymentContext.java:**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

System.out.println("Payment strategy not set.");

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.makePayment(250.00);

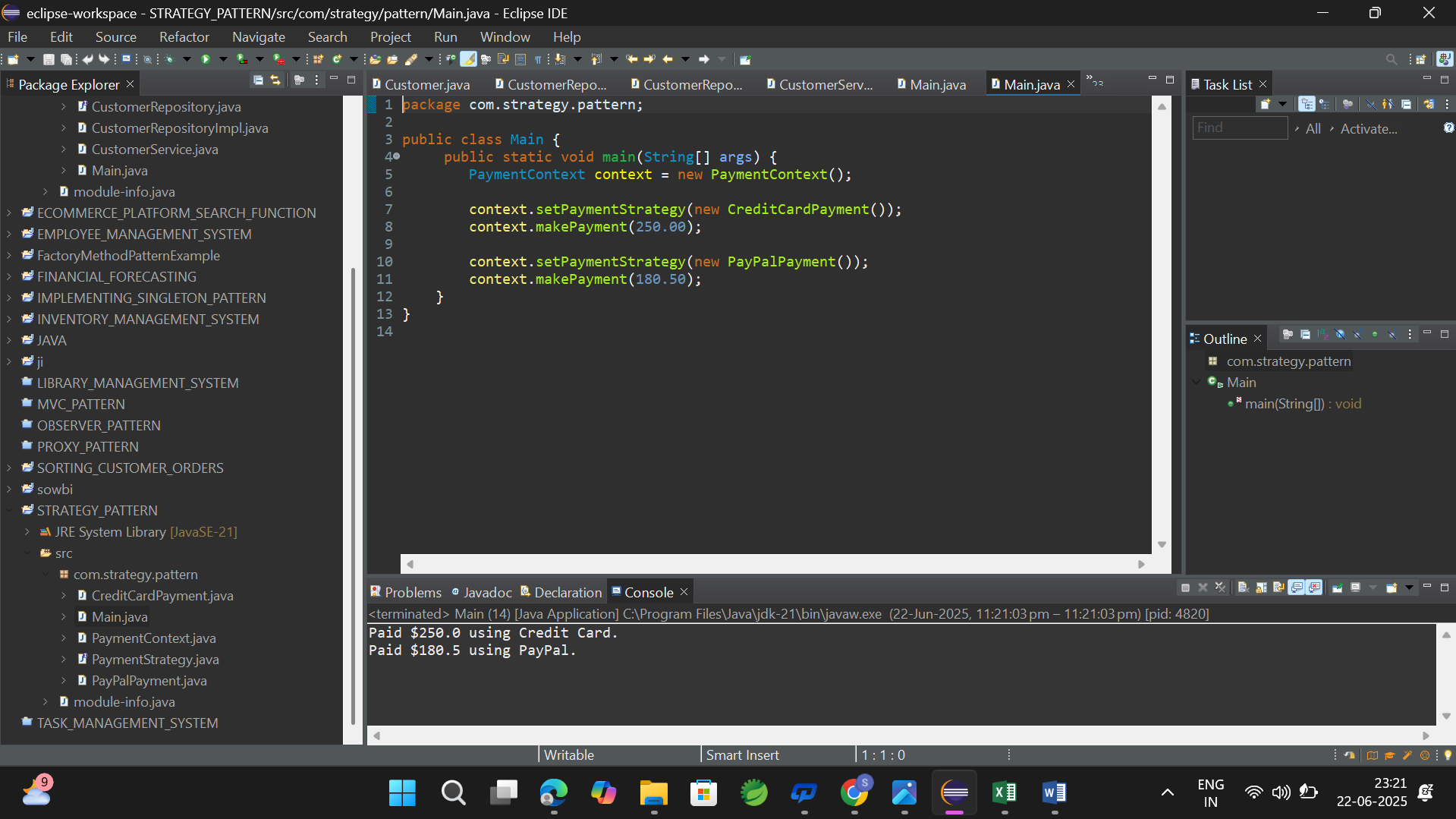
context.setPaymentStrategy(new PayPalPayment());

context.makePayment(180.50);

}

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Command.java:**

public interface Command {

void execute();

}

**Light.java:**

public class Light {

public void turnOn() {

System.out.println("Light is ON");

}

public void turnOff() {

System.out.println("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() {

if (command != null) {

command.execute();

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

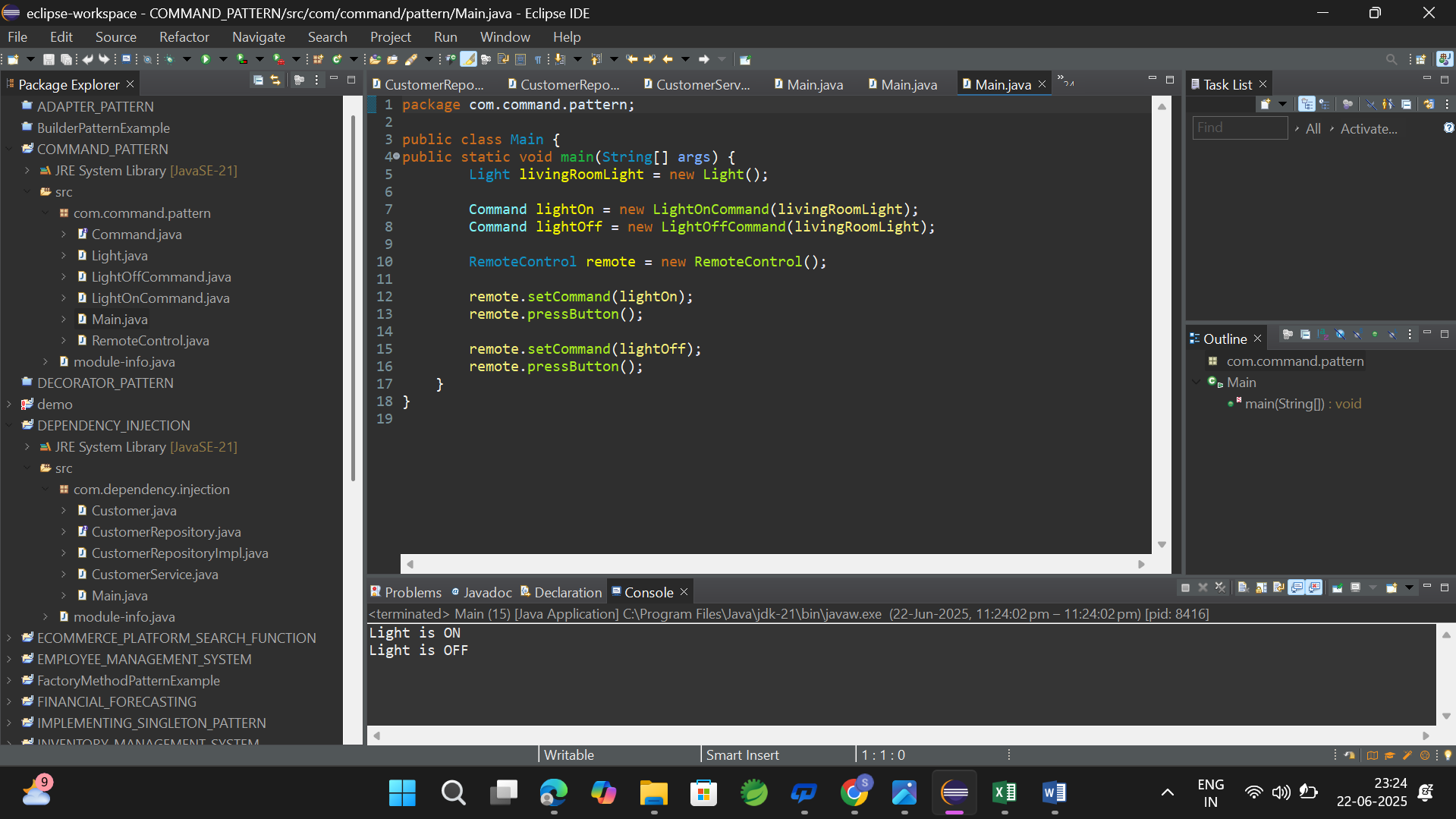
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**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 String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Student student = new Student();

student.setName("Arun");

student.setId("S101");

student.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Arnika");

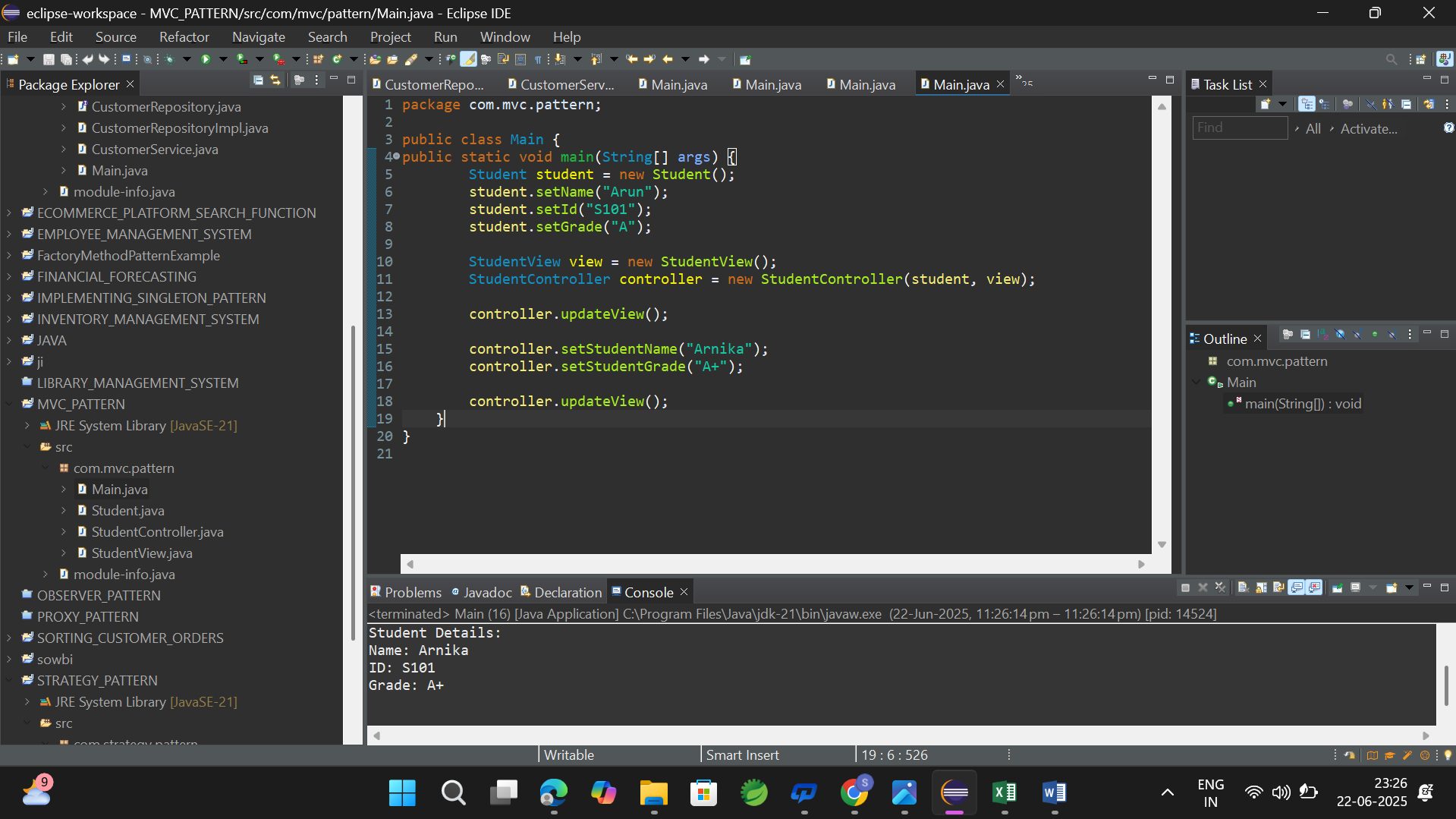
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

**Customer.java:**

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

}

**CustomerRepository.java:**

public interface CustomerRepository {

Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java:**

public class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String id) {

return new Customer(id, "Customer\_" + id);

}

}

**CustomerService.java:**

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

Customer customer = repository.findCustomerById(id);

System.out.println("Customer ID: " + customer.getId());

System.out.println("Customer Name: " + customer.getName());

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

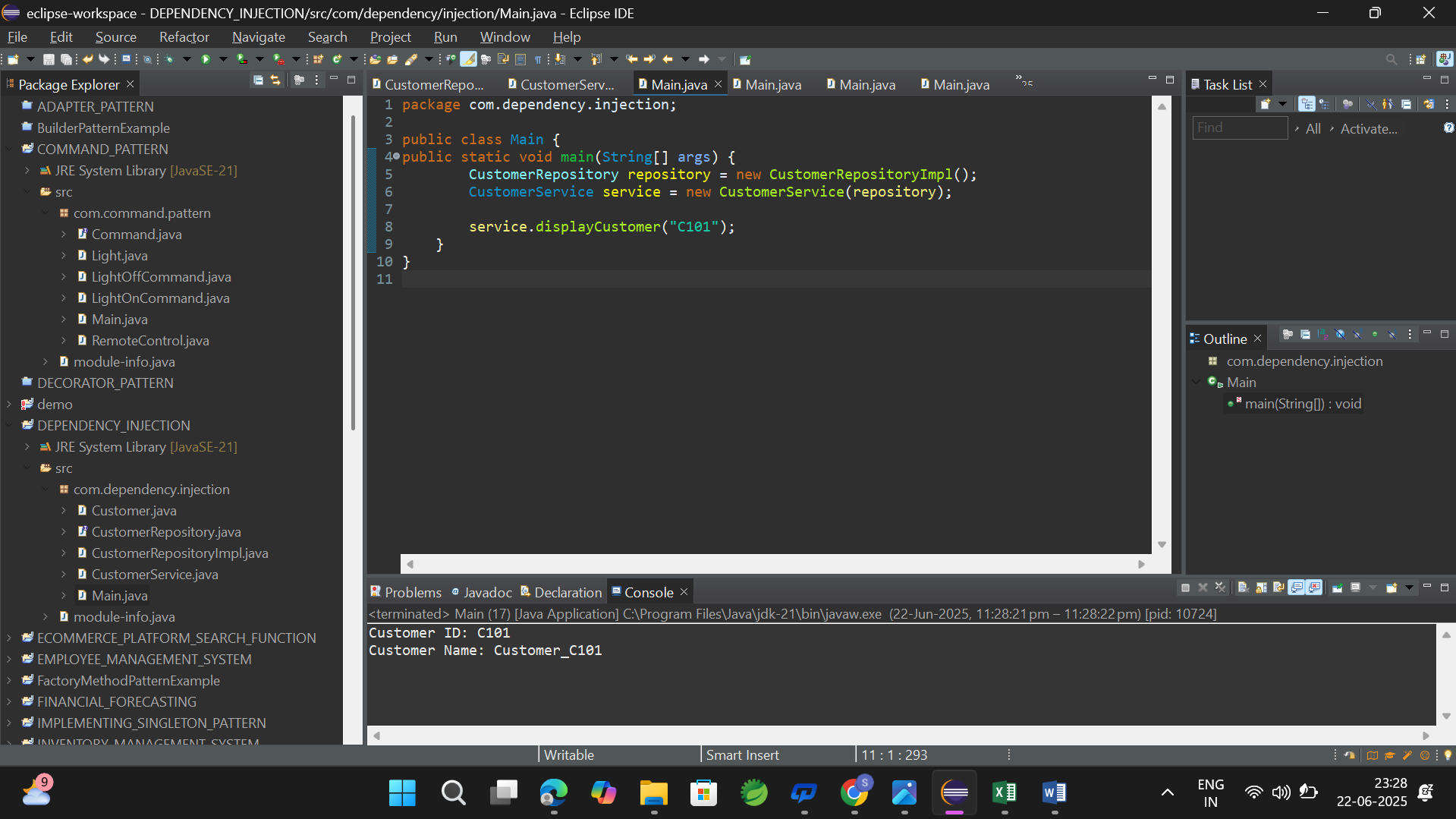
CustomerService service = new CustomerService(repository);

service.displayCustomer("C101");

}

}

**Output:**

****