**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**Logger.java:**

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: " + message);

}

}

**LoggerTest.java:**

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

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

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

if (logger1 == logger2) {

System.***out***.println("Both logger1 and logger2 are the same instance.");

} else {

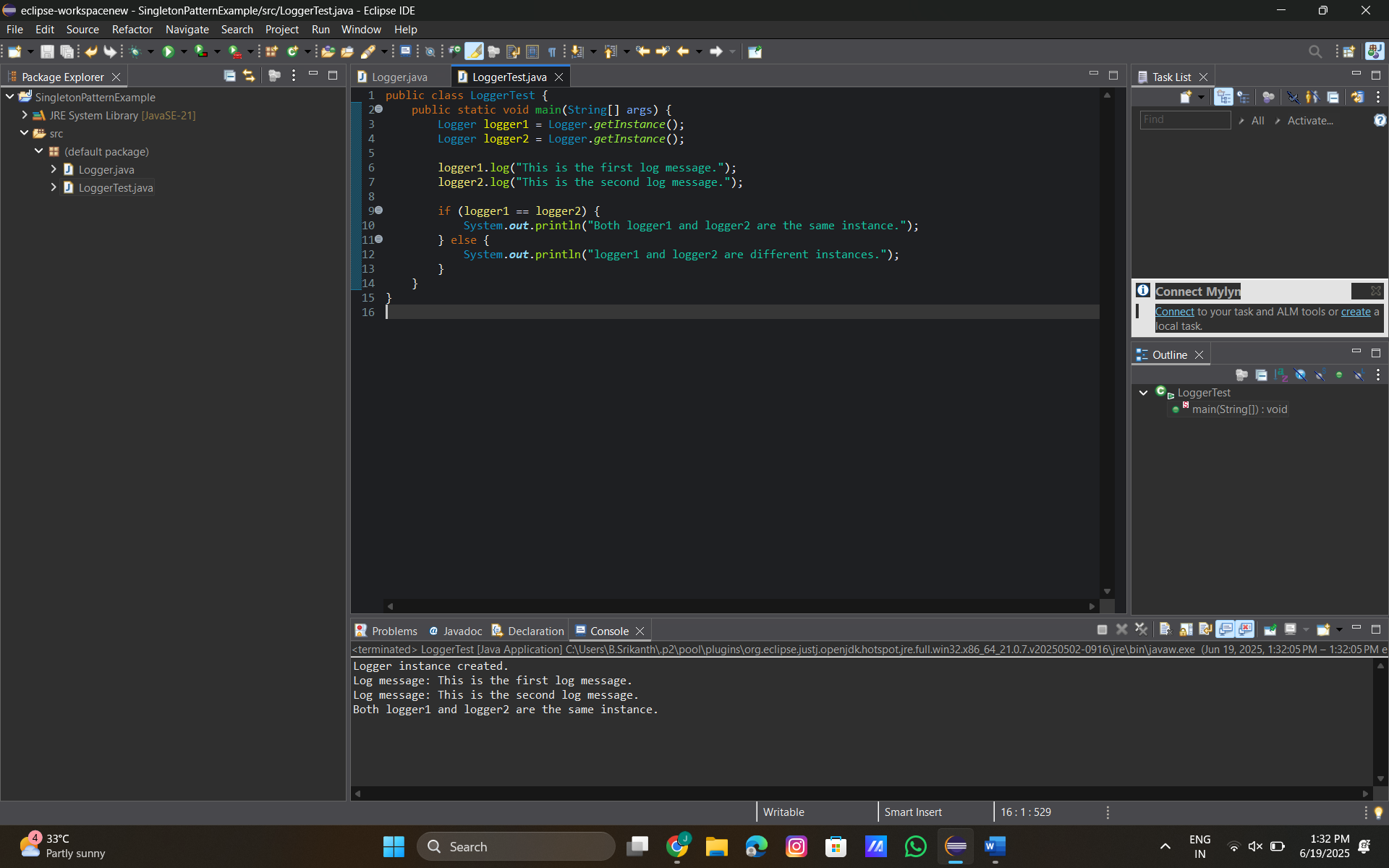
System.***out***.println("logger1 and logger2 are different instances.");

}

}

}

**Output:**



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

**Code:**

**Document.java:**

public interface Document {

void open();

}

**Document Factory.java:**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocument.java:**

public class WordDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java:**

public class ExcelDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java:**

public class PdfDocument implements Document {

public void open() {

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

}

}

**WordFactory.java:**

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**ExcelFactory.java:**

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**PdfFactory.java:**

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**TestFactoryPattern.java:**

public class TestFactoryPattern {

public static void main(String[] args) {

DocumentFactory factory;

factory = new WordFactory();

Document word = factory.createDocument();

word.open();

factory = new PdfFactory();

Document pdf = factory.createDocument();

pdf.open();

factory = new ExcelFactory();

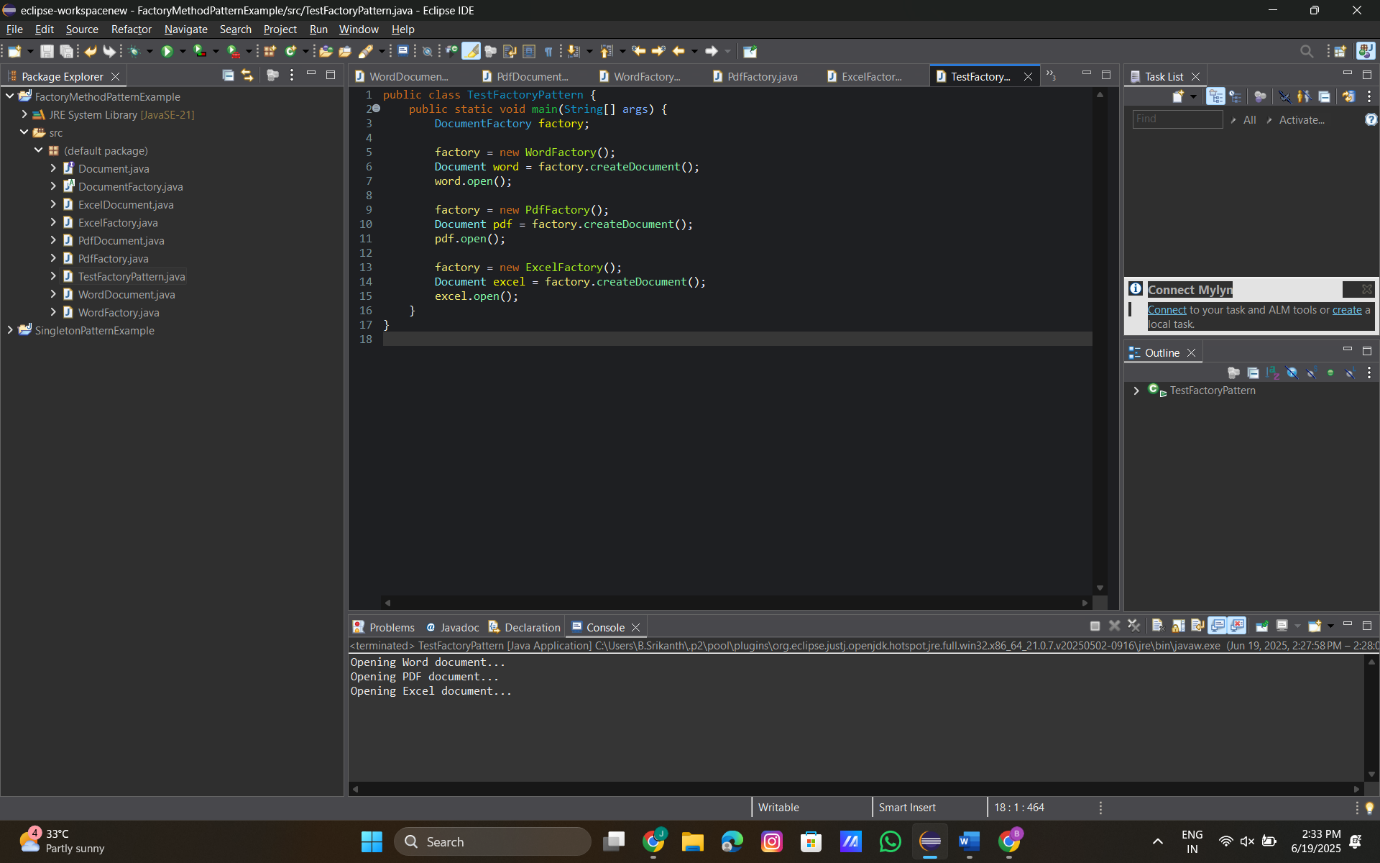
Document excel = factory.createDocument();

excel.open();

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Code:**

**Computer.java:**

public class Computer {

// Required parameters

private String CPU;

private String RAM;

// Optional parameters

private String storage;

private String graphicsCard;

// Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

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);

}

}

// Display method

public void displayConfiguration() {

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

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

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

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

}

}

**BuildPatternTest.java:**

public class BuilderPatternTest {

public static void main(String[] args) {

// Create a basic computer

Computer basicComputer = new Computer.Builder("Intel i5", "16GB")

.build();

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

basicComputer.displayConfiguration();

System.***out***.println("\n");

// Create a high-end gaming computer

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.build();

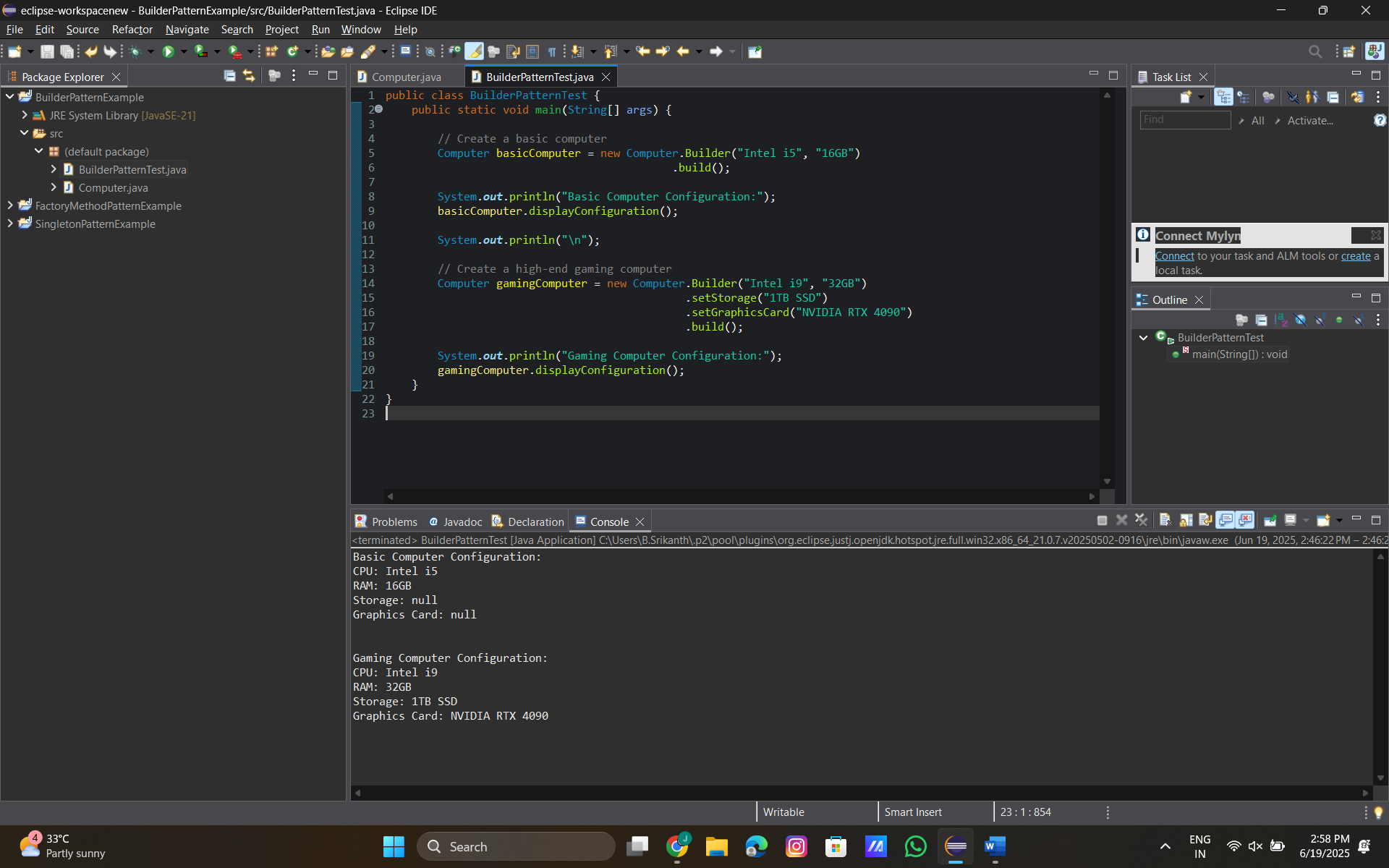
System.***out***.println("Gaming Computer Configuration:");

gamingComputer.displayConfiguration();

}

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**PaymentProcessor.java:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**NewWalletAdapter.java:**

public class NewWalletAdapter implements PaymentProcessor {

private NewWalletAPI newWalletAPI;

public NewWalletAdapter() {

this.newWalletAPI = new NewWalletAPI();

}

public void processPayment(double amount) {

newWalletAPI.pay(amount);

}

}

**NewWalletAPI.java:**

public class NewWalletAPI {

public void pay(double money) {

System.***out***.println("Paid ₹" + money + " using NewWalletAPI.");

}

}

**OldBankAdapter.java:**

public class NewWalletAPI {

public void pay(double money) {

System.***out***.println("Paid ₹" + money + " using NewWalletAPI.");

}

}

**OldBankAPI.java:**

public class OldBankAPI {

public void makePayment(String account, double amount) {

System.***out***.println("Payment of ₹" + amount + " made from account: " + account + " via OldBankAPI.");

}

}

**PaymentTest.java:**

public class PaymentTest {

public static void main(String[] args) {

PaymentProcessor oldBankPayment = new OldBankAdapter("ACC12345");

oldBankPayment.processPayment(50000.00);

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

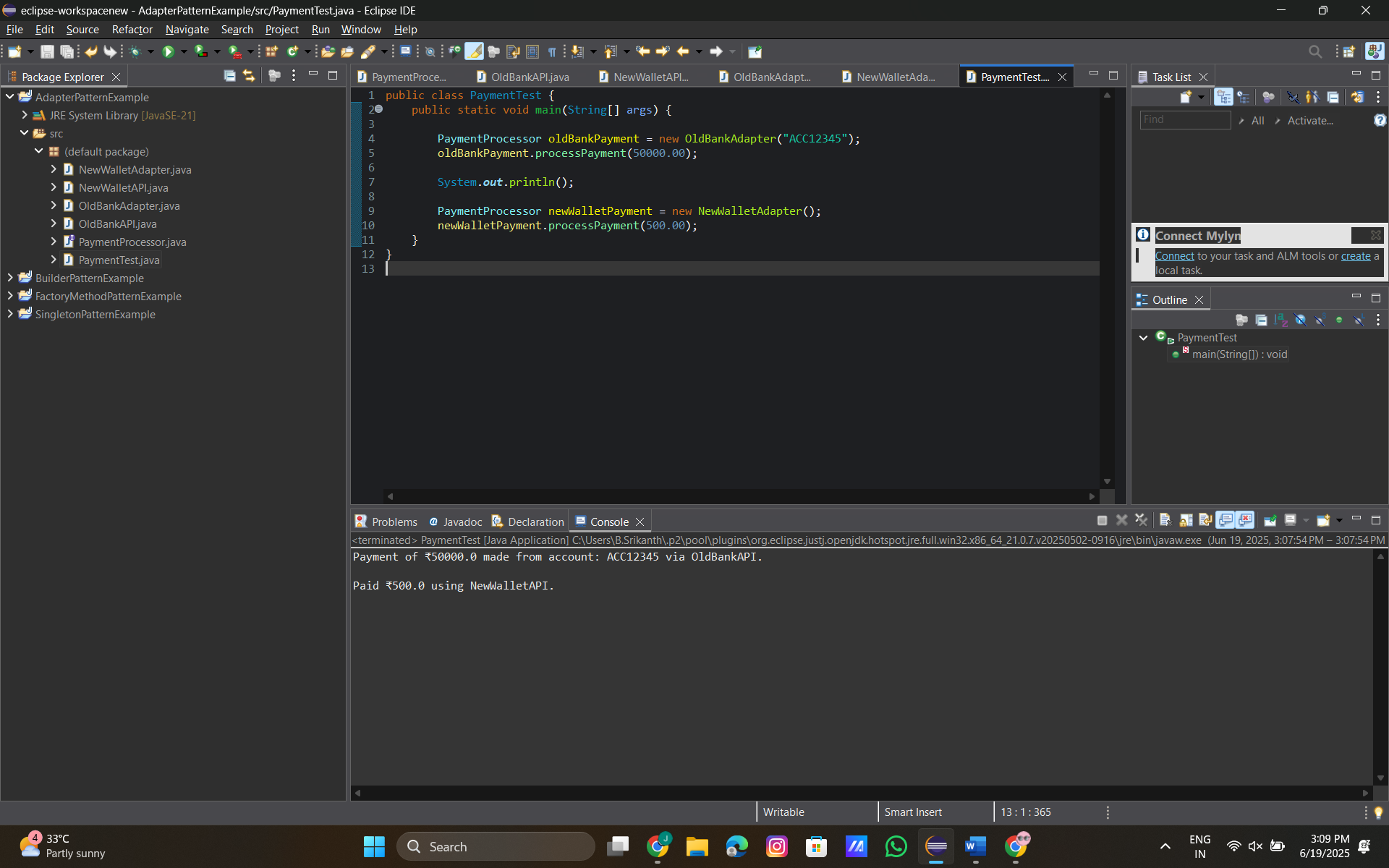
PaymentProcessor newWalletPayment = new NewWalletAdapter();

newWalletPayment.processPayment(500.00);

}

}

**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("Sending Email: " + message);

}

}

**NotifierDecorator.java:**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

**SMSNotifierDecorator.java:**

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:**

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);

}

}

**NotificationTest.java:**

public class NotificationTest {

public static void main(String[] args) {

// Base notifier - Email only

Notifier email = new EmailNotifier();

System.***out***.println("=== Email Only ===");

email.send("Hello via Email!");

// Email + SMS

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

System.***out***.println("\n=== Email + SMS ===");

smsEmail.send("Hi via Email and SMS!");

// Email + SMS + Slack

Notifier allChannels = new SlackNotifierDecorator(

new SMSNotifierDecorator(

new EmailNotifier()

)

);

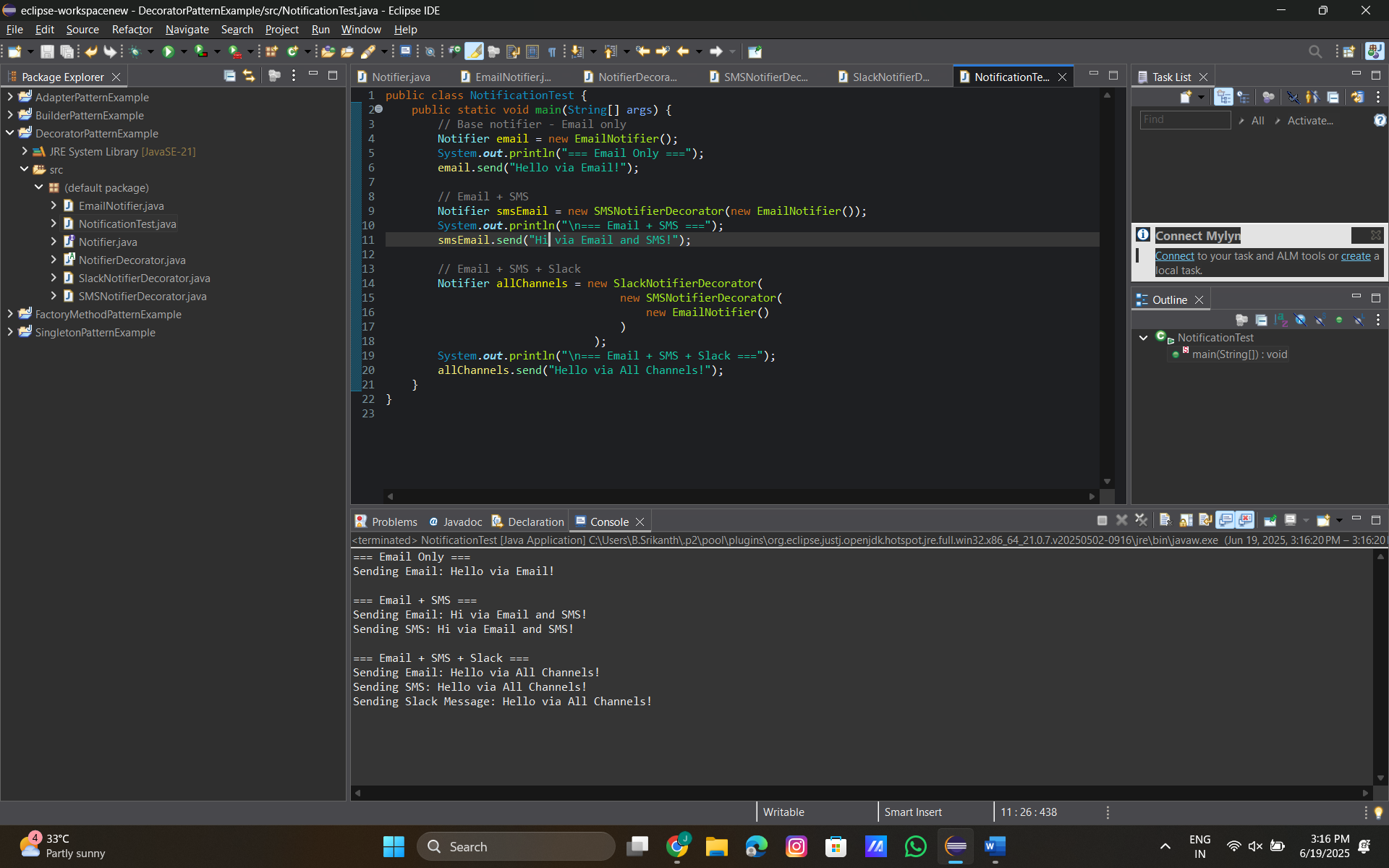
System.***out***.println("\n=== Email + SMS + Slack ===");

allChannels.send("Hello via All Channels!");

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Image.java:**

public interface Image {

void display();

}

**RealImage.java:**

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.***out***.println("Loading " + fileName + " from disk...");

}

public void display() {

System.***out***.println("Displaying " + 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); // Lazy initialization

}

realImage.display();

}

}

**ImageViewerTest.java:**

public class ImageViewerTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

// image1 is displayed for the first time → loads from disk

System.***out***.println("First display of photo1:");

image1.display();

System.***out***.println("\nSecond display of photo1 (cached):");

image1.display(); // Should not reload from disk

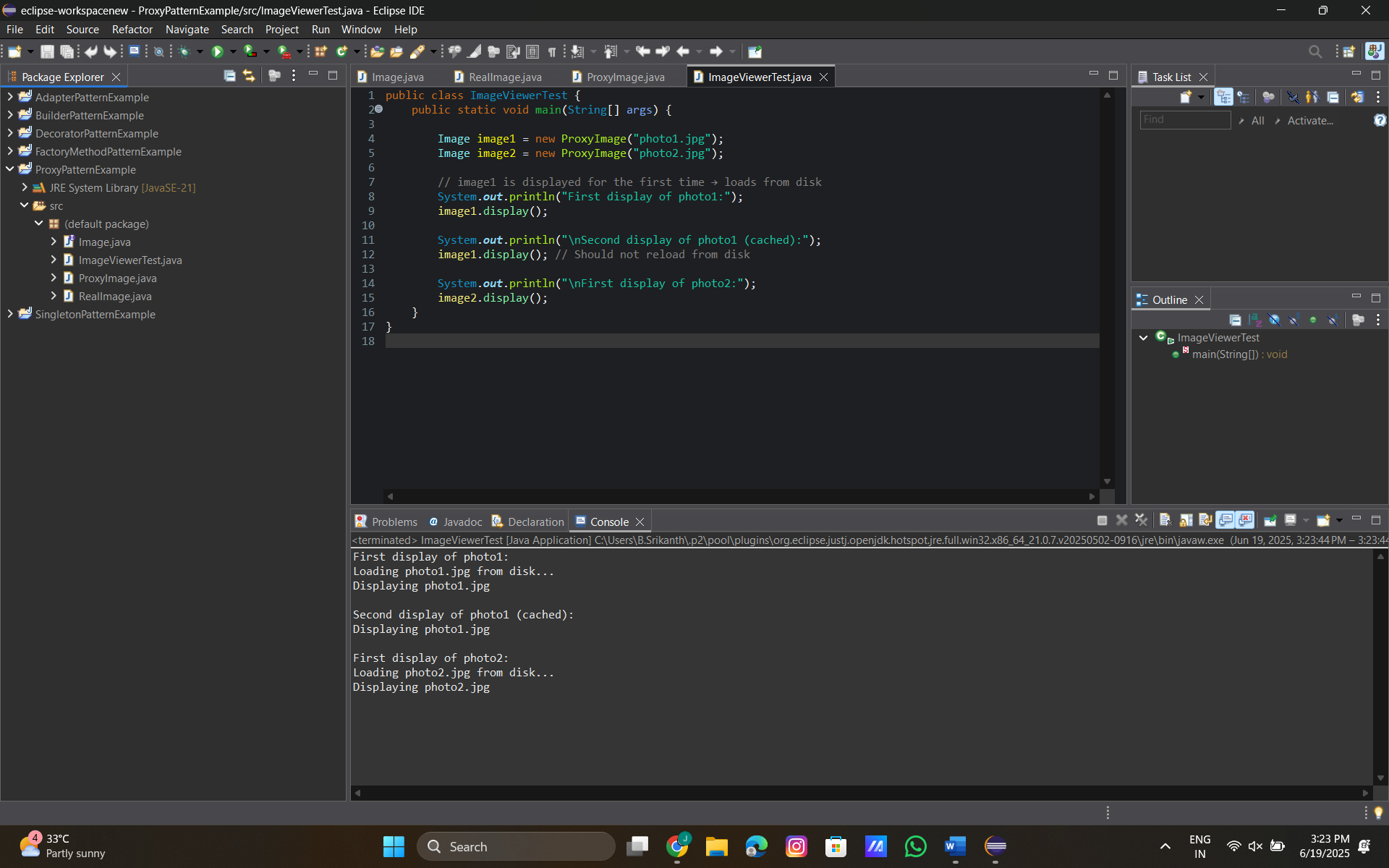
System.***out***.println("\nFirst display of photo2:");

image2.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Stock.java:**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**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 StockMarket(String stockName, double initialPrice) {

this.stockName = stockName;

this.stockPrice = initialPrice;

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

public void setStockPrice(double newPrice) {

this.stockPrice = newPrice;

notifyObservers();

}

}

**Observer.java:**

public interface Observer {

void update(String stockName, double stockPrice);

}

**MobileApp.java:**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

public void update(String stockName, double stockPrice) {

System.***out***.println(appName + " - Mobile Notification: " + stockName + " is ₹" + stockPrice);

}

}

**WebApp.java:**

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

public void update(String stockName, double stockPrice) {

System.***out***.println(appName + " - Web Dashboard: " + stockName + " is ₹" + stockPrice);

}

}

**ObserverPatternTest.java:**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stock = new StockMarket("TCS", 3500.00);

Observer mobile = new MobileApp("InvestMate");

Observer web = new WebApp("StockWatch");

stock.registerObserver(mobile);

stock.registerObserver(web);

System.***out***.println("Stock updated to ₹3550:");

stock.setStockPrice(3550.00);

System.***out***.println("\nStock updated to ₹3600:");

stock.setStockPrice(3600.00);

stock.removeObserver(web);

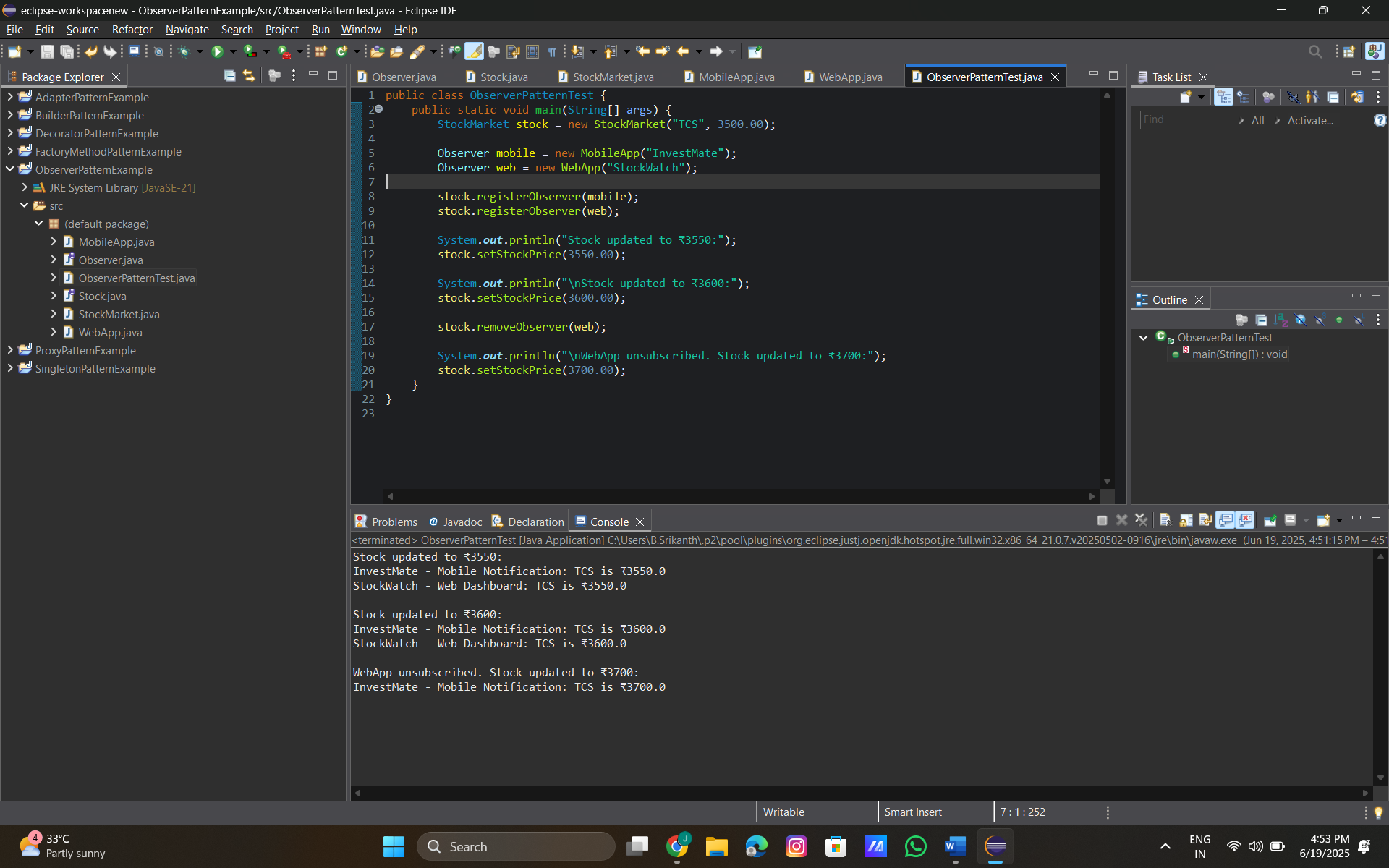
System.***out***.println("\nWebApp unsubscribed. Stock updated to ₹3700:");

stock.setStockPrice(3700.00);

}

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**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 account: " + email);

}

}

**PaymentContext.java:**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

**StrategyPatternTest.java:**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Use Credit Card Payment

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

context.processPayment(2500.00);

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

// Switch to PayPal Payment

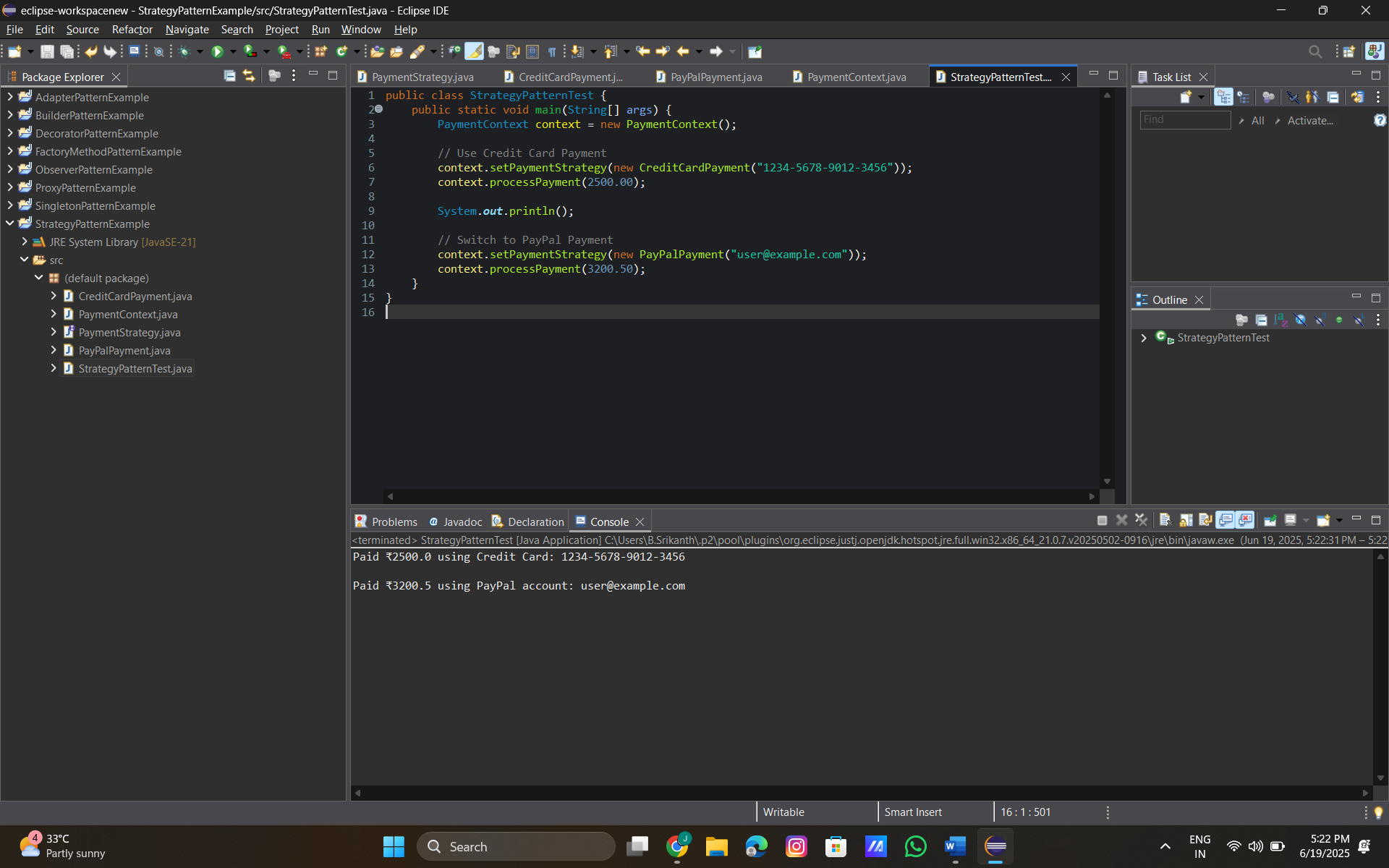
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(3200.50);

}

}

**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();

} else {

System.***out***.println("No command set.");

}

}

}

**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();

System.***out***.println("Turning light ON:");

remote.setCommand(lightOn);

remote.pressButton();

System.***out***.println("\nTurning light OFF:");

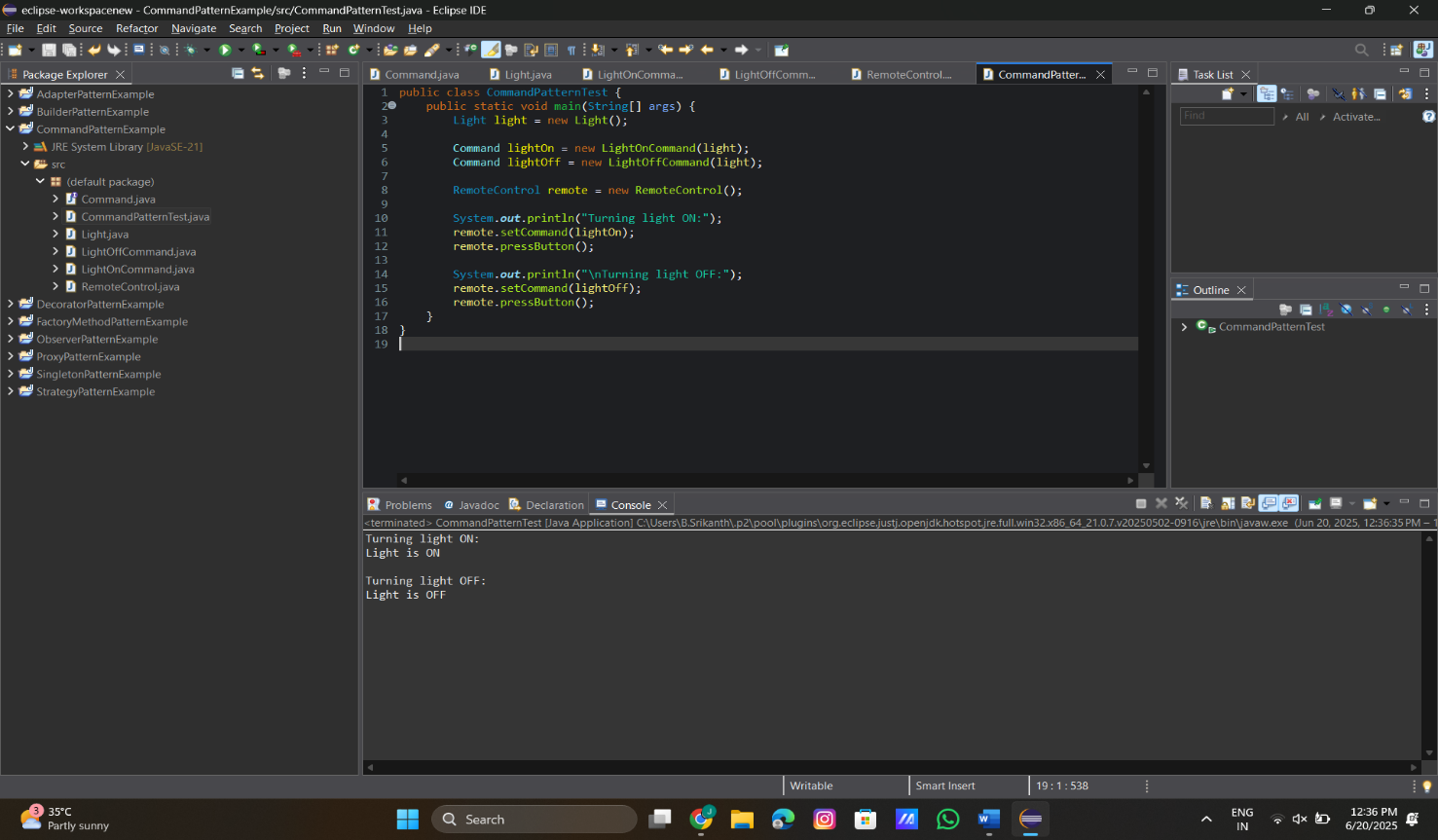
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Code:**

**Student.java:**

public class Student {

private String name;

private String id;

private String grade;

// Constructor

public Student(String name, String id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and setters

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) {

// Create Model

Student student = new Student("Alice", "S123", "A");

// Create View

StudentView view = new StudentView();

// Create Controller

StudentController controller = new StudentController(student, view);

// Display initial data

System.***out***.println("Initial Student Record:");

controller.updateView();

// Update model data via controller

controller.setStudentName("Alice Johnson");

controller.setStudentGrade("A+");

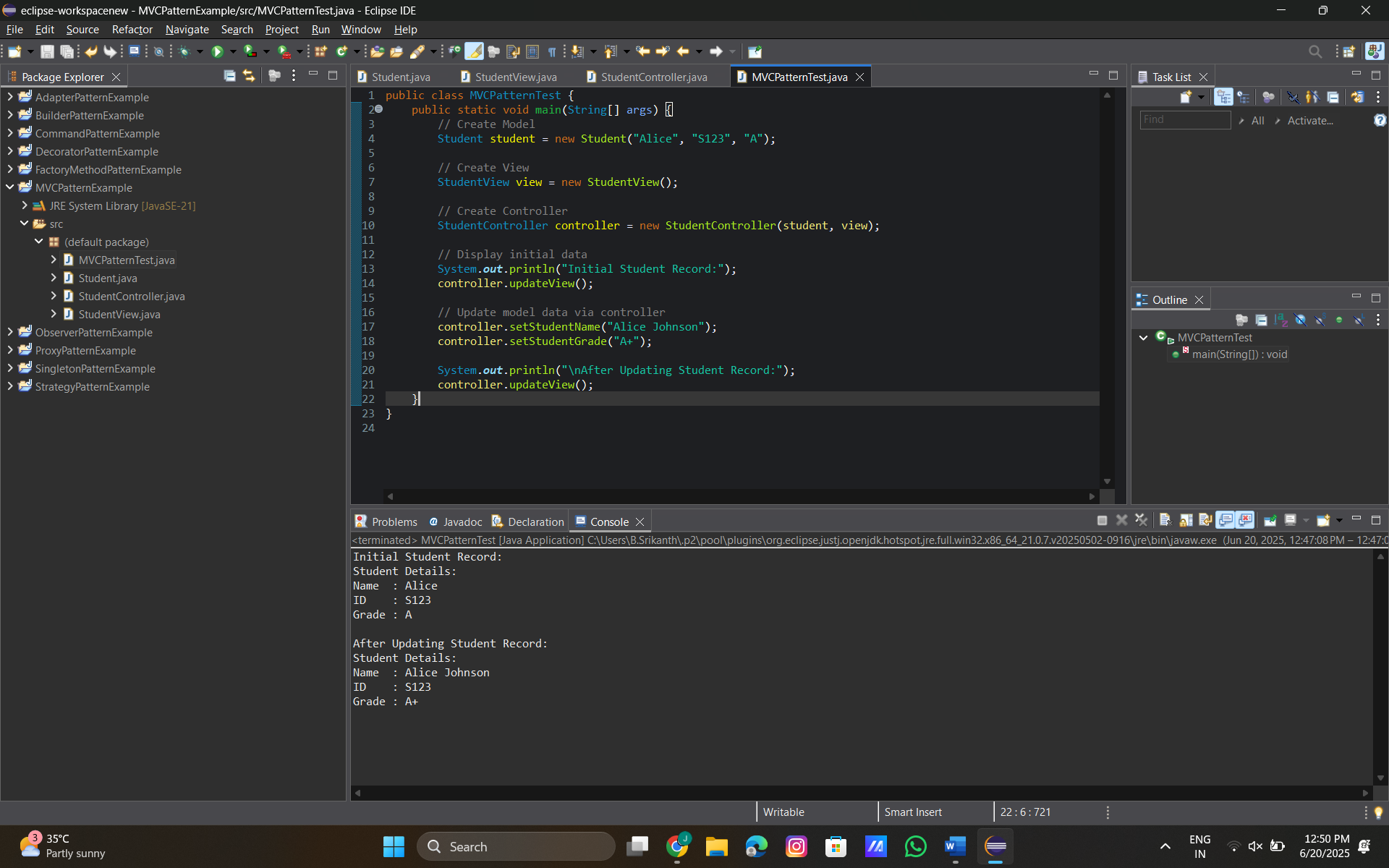
System.***out***.println("\nAfter Updating Student Record:");

controller.updateView();

}

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**Code:**

**CustomerRepositry.java:**

public interface CustomerRepository {

String findCustomerById(String id);

}

**CustomerRepositoryImpl.java:**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

// Simulated data source

return "Customer[id=" + id + ", name=Manideep, email=mani@example.com]";

}

}

**CustomerService.java:**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(String id) {

String customerData = customerRepository.findCustomerById(id);

System.***out***.println("Customer Data: " + customerData);

}

}

**DependencyInjectionTest.java:**

public class DependencyInjectionTest {

public static void main(String[] args) {

// Manually inject dependency

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject into service

CustomerService service = new CustomerService(repository);

// Use the service

service.displayCustomer("C101");

}

}

**Output:**

