**WEEK - 1**

**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

package com.example.singleton;

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger Initialized");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

**Main.java**

package com.example.singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

if (logger1 == logger2) {

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

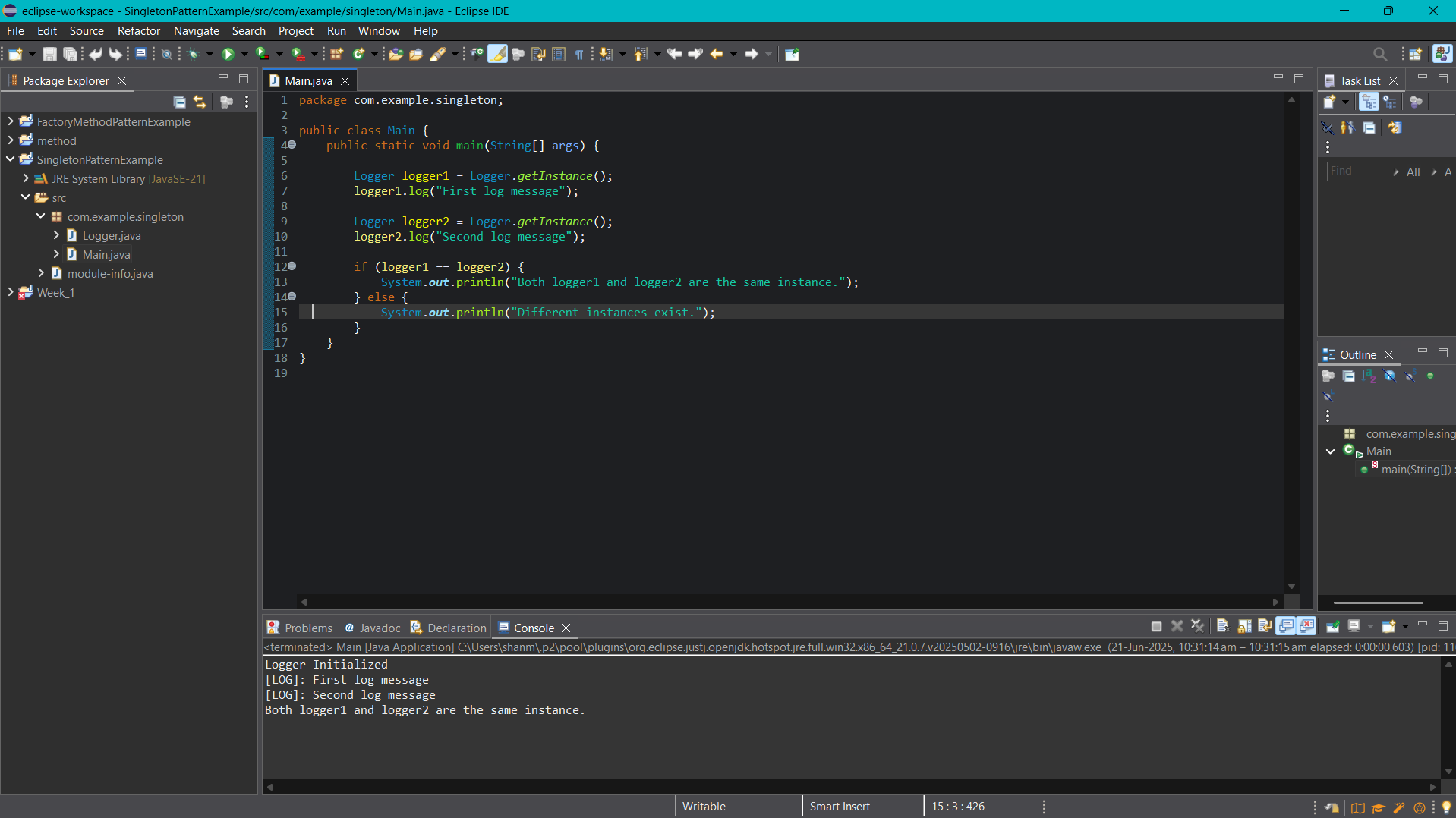
} else {

System.out.println("Different instances exist.");

}

}

}



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

**Document.java**

package com.example.factory;

public interface Document {

void open();

}

**WordDocument.java**

package com.example.factory;

public class WordDocument implements Document{

@Override

public void open() {

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

}

}

**PdfDocument.java**

package com.example.factory;

public class PdfDocument implements Document{

@Override

public void open() {

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

}

}

**ExcelDocument.java**

package com.example.factory;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**DocumentFactory.java**

package com.example.factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

package com.example.factory;

public class WordDocumentFactory extends DocumentFactory{

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

package com.example.factory;

public class PdfDocumentFactory extends DocumentFactory{

@Override

public Document createDocument() {

return new WordDocument();

}

}

**ExcelDocumentFactory.java**

package com.example.factory;

public class ExcelDocumentFactory extends DocumentFactory{

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**FactoryMethodTest.java**

package com.example.factory;

public class FactoryMethodTest {

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();

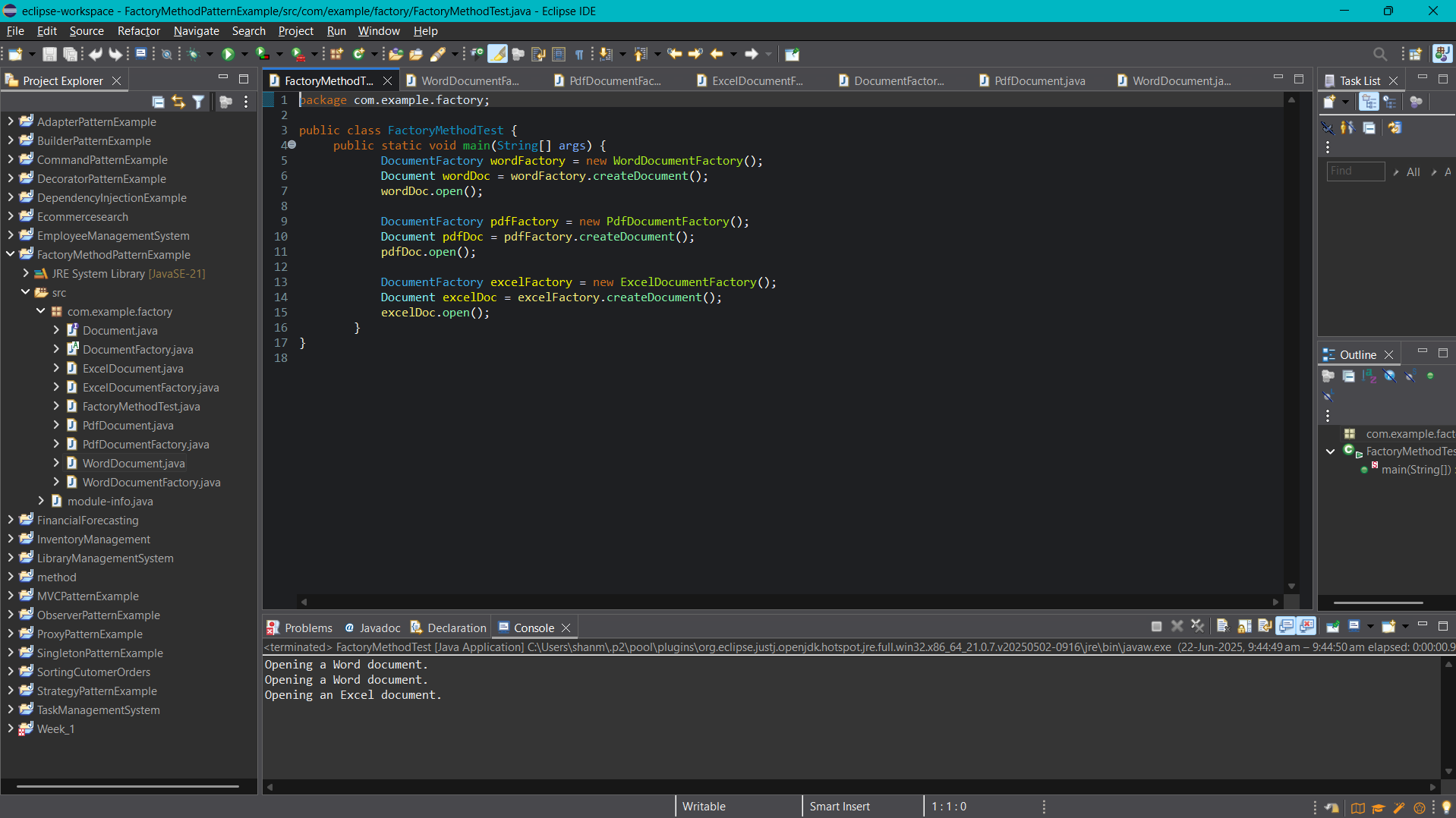
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}



**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

package com.example.builder;

public class Computer {

private final String CPU;

private final String RAM;

private final String storage;

private final String GPU;

private final String keyboard;

private final String display;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

this.keyboard = builder.keyboard;

this.display = builder.display;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String keyboard;

private String display;

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 setGPU(String GPU) {

this.GPU = GPU;

return this;

}

public Builder setKeyboard(String keyboard) {

this.keyboard = keyboard;

return this;

}

public Builder setDisplay(String display) {

this.display = display;

return this;

}

// Builds the final Computer object

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer{" +

"CPU='" + CPU + '\'' +

", RAM='" + RAM + '\'' +

", storage='" + storage + '\'' +

", GPU='" + GPU + '\'' +

", keyboard='" + keyboard + '\'' +

", display='" + display + '\'' +

'}';

}

}

**BuilderPatternTest.java**

package com.example.builder;

public class BuilderPatternTest {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4080")

.setKeyboard("Mechanical RGB")

.setDisplay("4K UHD")

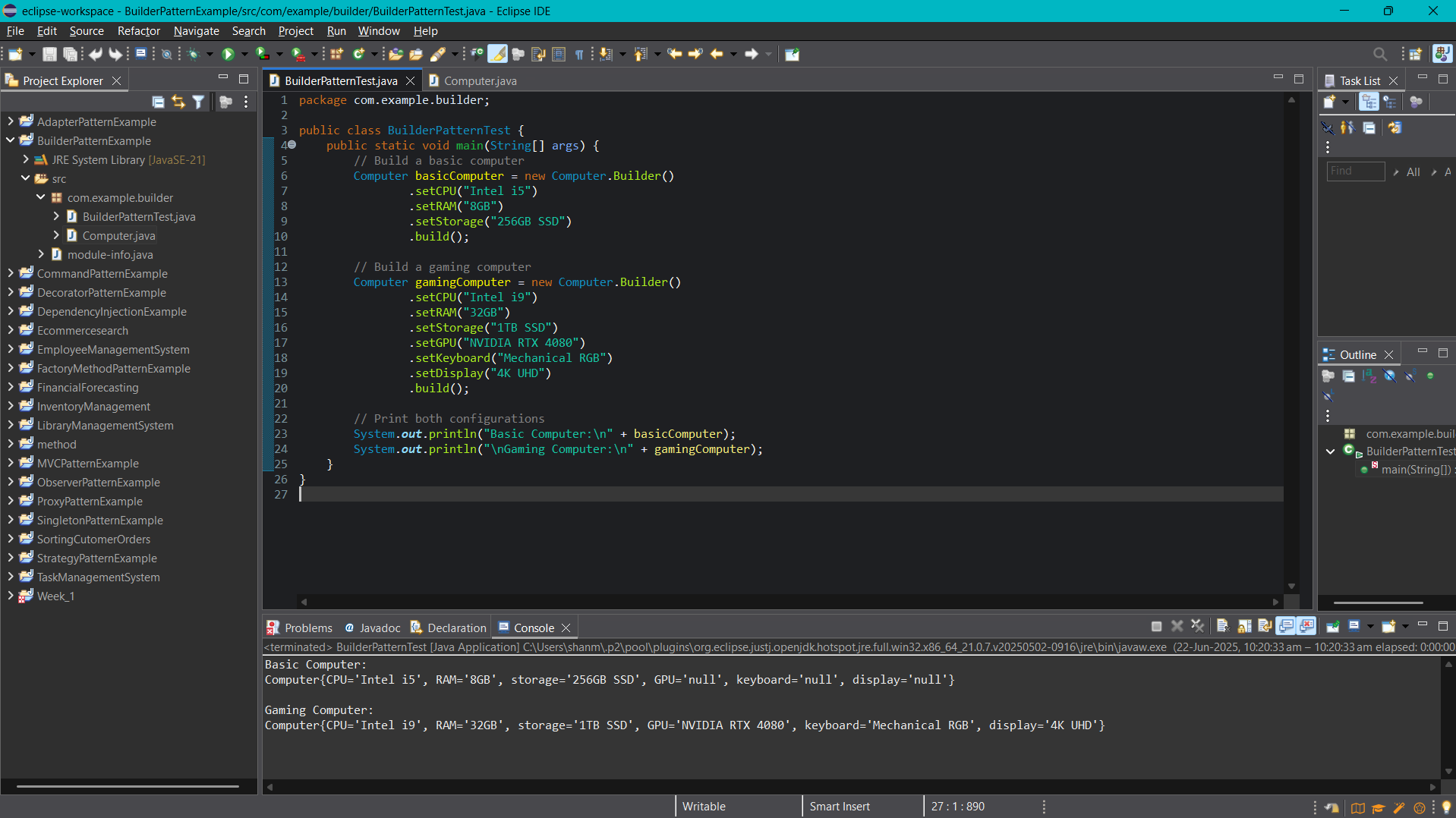
.build();

System.out.println("Basic Computer:\n" + basicComputer);

System.out.println("\nGaming Computer:\n" + gamingComputer);

}

}



**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

package com.example.adapter;

public interface PaymentProcessor {

void processPayment(double amount);

}

**GPay.java**

package com.example.adapter;

public class GPay {

public void payUsingGPay(double amount) {

System.out.println("GPay: Paid ₹" + amount);

}

}

**Phonepe.java**

package com.example.adapter;

public class PhonePe {

public void makePhonePePayment(double amount) {

System.out.println("PhonePe: Paid ₹" + amount);

}

}

**Paytm.java**

package com.example.adapter;

public class Paytm {

public void doTransaction(double amount) {

System.out.println("Paytm: Paid ₹" + amount);

}

}

**GPayAdapter.java**

package com.example.adapter;

public class GPayAdapter implements PaymentProcessor {

private GPay gpay;

public GPayAdapter(GPay gpay) {

this.gpay = gpay;

}

@Override

public void processPayment(double amount) {

gpay.payUsingGPay(amount);

}

}

**PhonepeAdapter.java**

package com.example.adapter;

public class PhonePeAdapter implements PaymentProcessor {

private PhonePe phonePe;

public PhonePeAdapter(PhonePe phonePe) {

this.phonePe = phonePe;

}

@Override

public void processPayment(double amount) {

phonePe.makePhonePePayment(amount);

}

}

**PaytmAdapter.java**

package com.example.adapter;

public class PaytmAdapter implements PaymentProcessor {

private Paytm paytm;

public PaytmAdapter(Paytm paytm) {

this.paytm = paytm;

}

@Override

public void processPayment(double amount) {

paytm.doTransaction(amount);

}

}

**Main.java**

package com.example.adapter;

public class Main {

public static void main(String[] args) {

PaymentProcessor gpayProcessor = new GPayAdapter(new GPay());

gpayProcessor.processPayment(500.0);

PaymentProcessor phonePeProcessor = new PhonePeAdapter(new PhonePe());

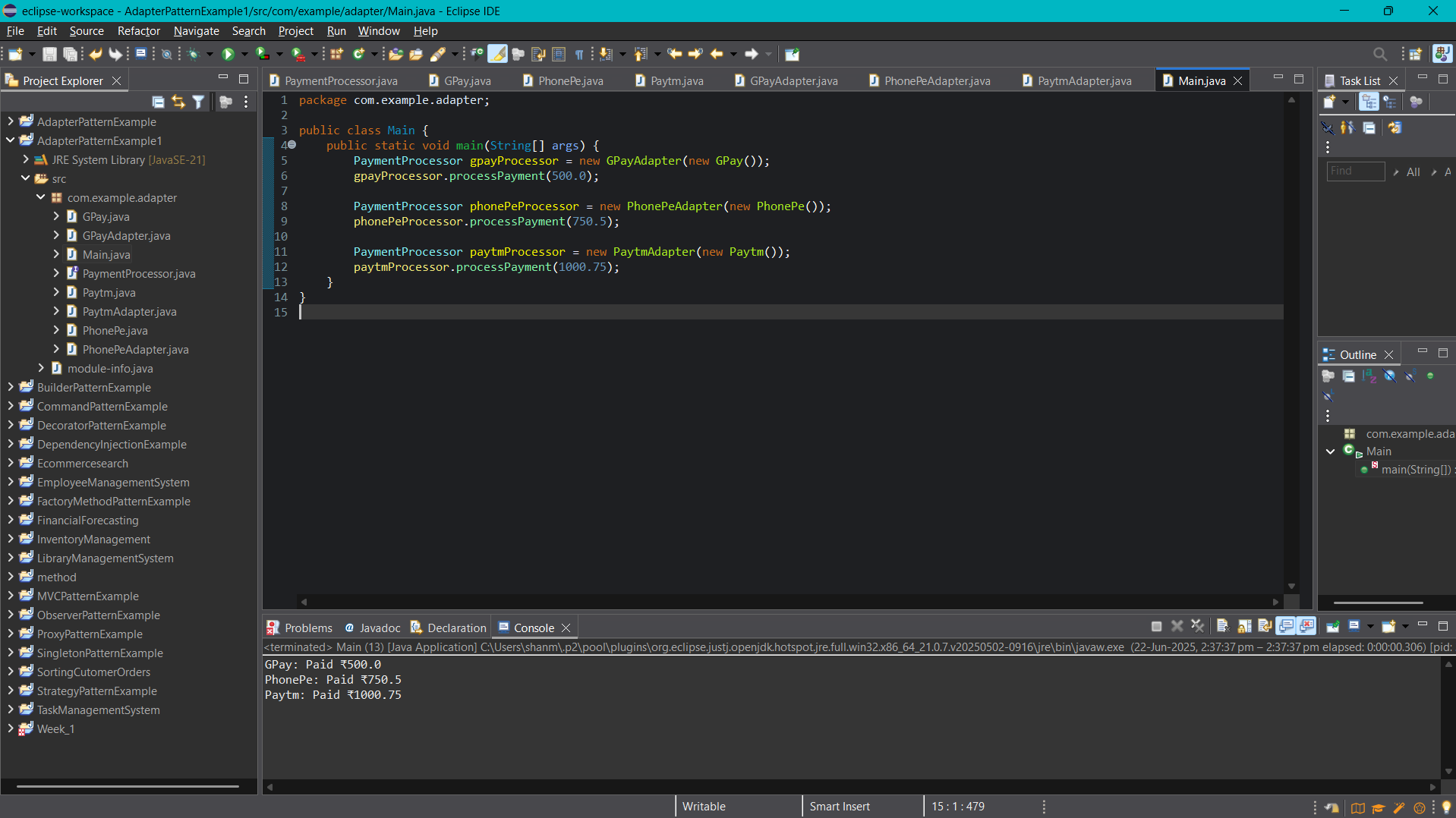
phonePeProcessor.processPayment(750.5);

PaymentProcessor paytmProcessor = new PaytmAdapter(new Paytm());

paytmProcessor.processPayment(1000.75);

}

}



**Exercise 5: Implementing the Decorator Pattern**

**Notifier.java**

package com.example.decorator;

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

package com.example.decorator;

public class EmailNotifier implements Notifier{

@Override

public void send(String message) {

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

}

}

**NotifierDecorator.java**

package com.example.decorator;

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message);

}

}

**SMSNotifierDecorator.java**

package com.example.decorator;

public 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: " + message);

}

}

**SlackNotifierDecorator.java**

package com.example.decorator;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**Main.java**

package com.example.decorator;

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

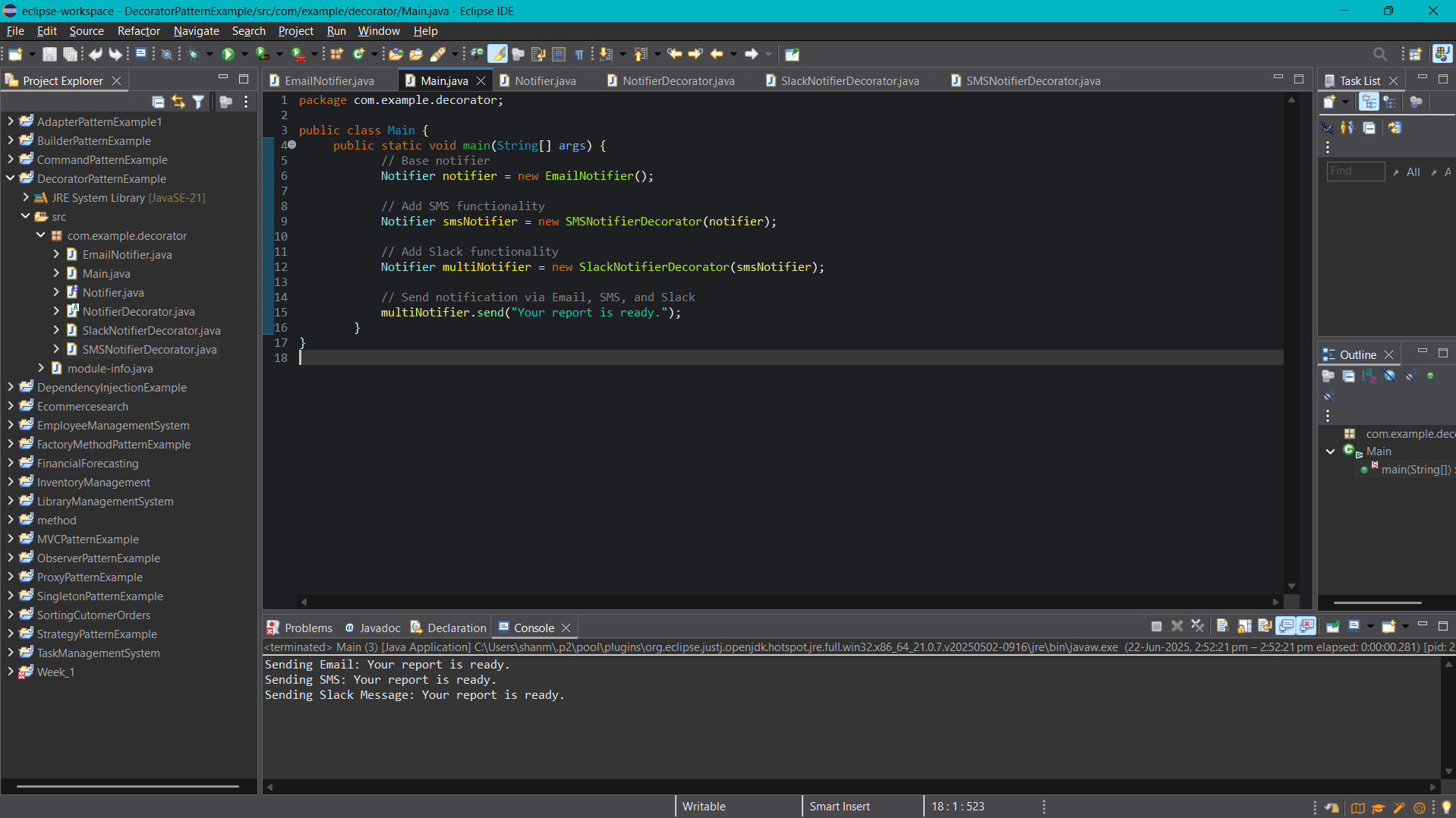
Notifier smsNotifier = new SMSNotifierDecorator(notifier);

Notifier multiNotifier = new SlackNotifierDecorator(smsNotifier);

multiNotifier.send("Your report is ready.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

package com.example.proxy;

public interface Image {

void display();

}

**RealImage.java**

package com.example.proxy;

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);

}

@Override

public void display() {

System.out.println("Displaying image: " + fileName);

}

}

**ProxyImage.java**

package com.example.proxy;

import java.util.HashMap;

import java.util.Map;

public class ProxyImage implements Image {

private String fileName;

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

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

RealImage realImage = cache.get(fileName);

if (realImage == null) {

System.out.println("Image not in cache. Creating RealImage...");

realImage = new RealImage(fileName);

cache.put(fileName, realImage);

} else {

System.out.println("Image found in cache.");

}

realImage.display();

}

}

**ProxyPatternDemo.java**

package com.example.proxy;

public class ProxyPatternDemo {

public static void main(String[] args) {

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

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

Image image3 = new ProxyImage("nature.jpg");

System.out.println("\n--- First call to nature.jpg ---");

image1.display();

System.out.println("\n--- First call to portrait.jpg ---");

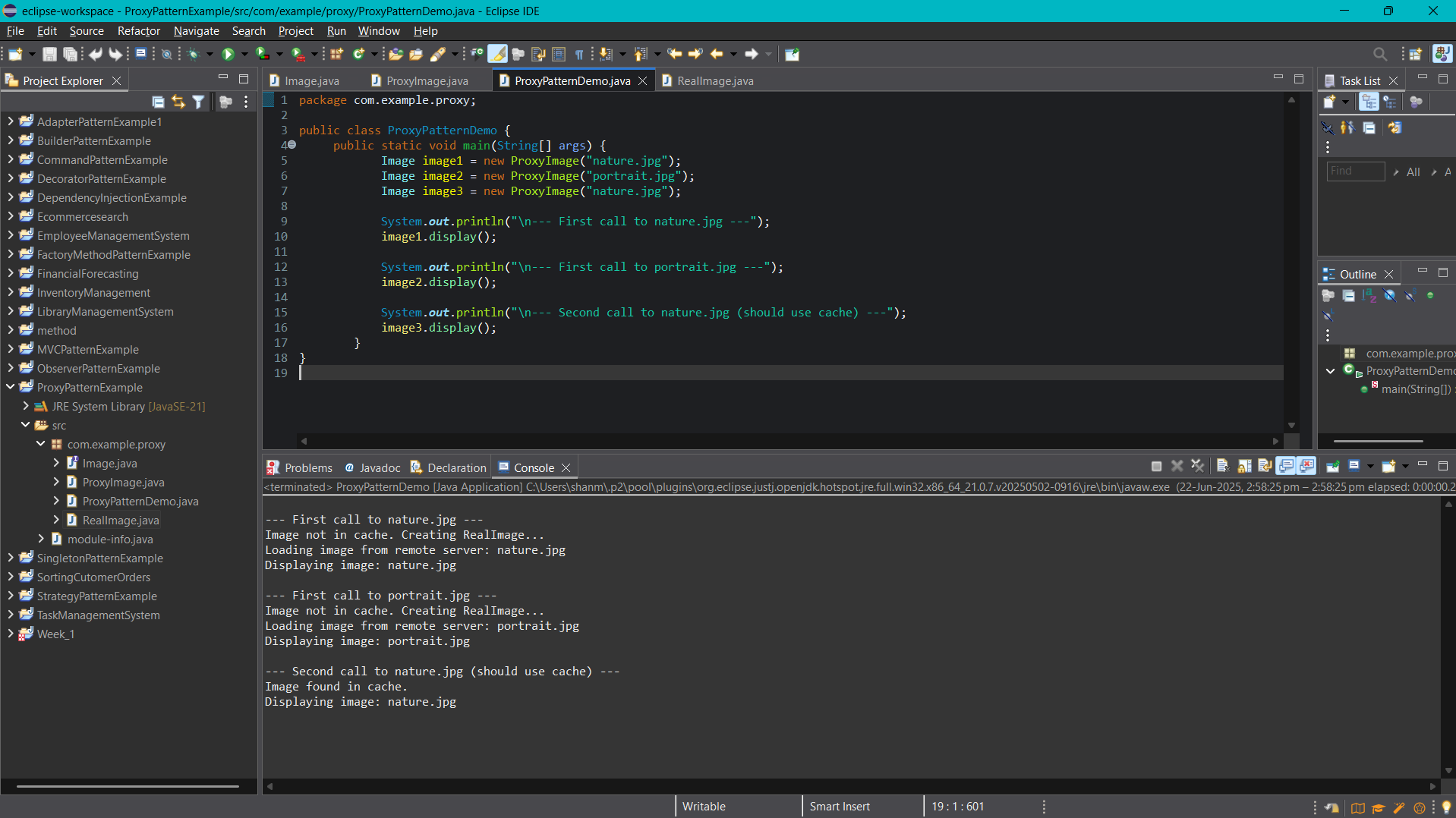
image2.display();

System.out.println("\n--- Second call to nature.jpg (should use cache) ---");

image3.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

**Stock.java**

package com.example.observer;

public interface Stock {

void registerObserver(Observer o);

void deregisterObserver(Observer o);

void notifyObservers();

}

**StockMarket.java**

package com.example.observer;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void deregisterObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

public void setStockPrice(double newPrice) {

System.out.println("\nStock price updated to: Rs" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

Observer.java

package com.example.observer;

public interface Observer {

void update(double stockPrice);

}

**MobileApp.java**

package com.example.observer;

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("MobileApp [" + name + "] received update: Stock price is Rs" + stockPrice);

}

}

**WebApp.java**

package com.example.observer;

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("WebApp [" + name + "] received update: Stock price is Rs" + stockPrice);

}

}

**Main.java**

package com.example.observer;

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile1 = new MobileApp("Client A");

Observer mobile2 = new MobileApp("Client B");

Observer web1 = new WebApp("Dashboard 1");

stockMarket.registerObserver(mobile1);

stockMarket.registerObserver(mobile2);

stockMarket.registerObserver(web1);

stockMarket.setStockPrice(100.25);

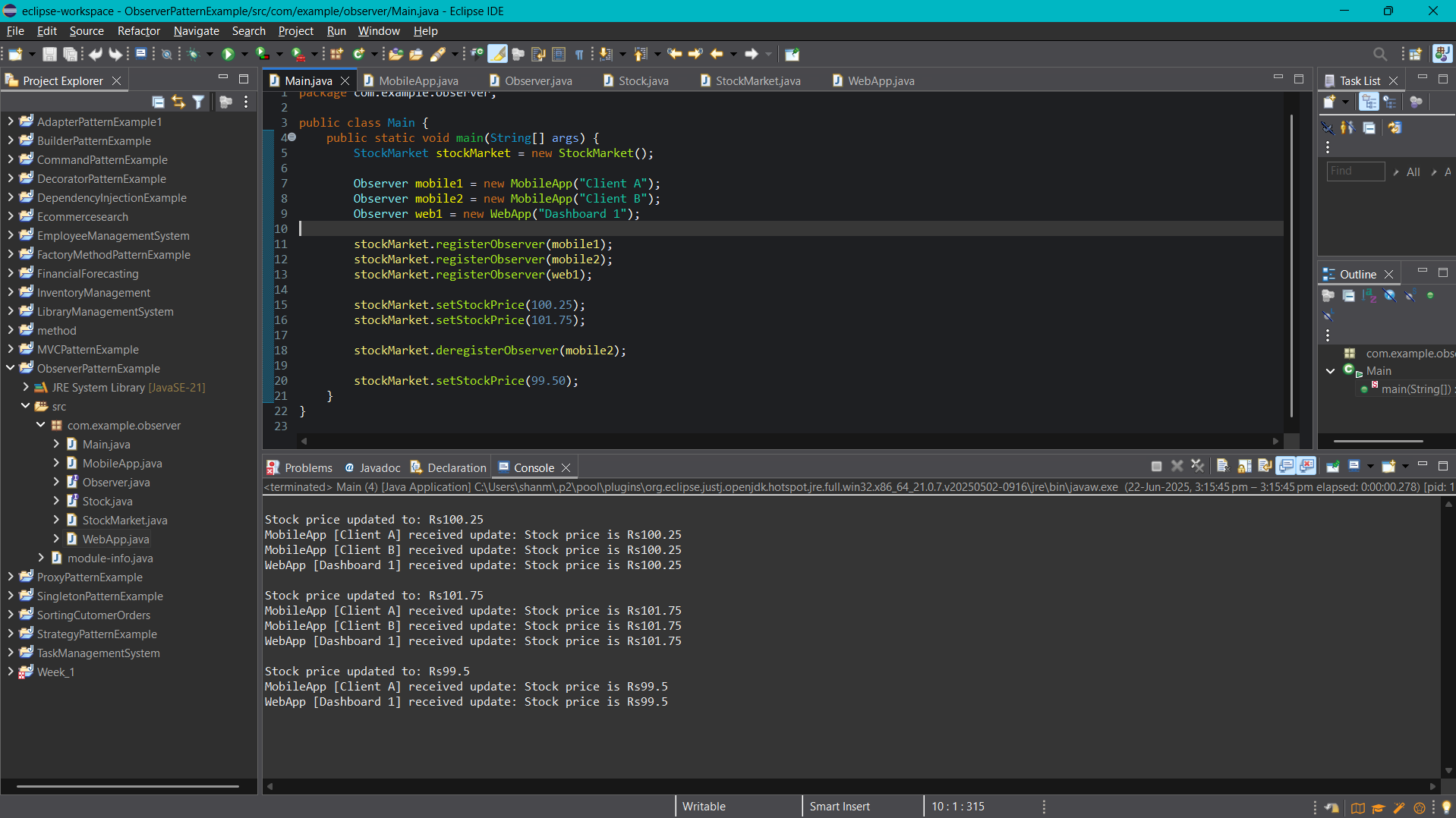
stockMarket.setStockPrice(101.75);

stockMarket.deregisterObserver(mobile2);

stockMarket.setStockPrice(99.50);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

package com.example.strategy;

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

package com.example.strategy;

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

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

System.out.println("Card Holder: " + cardHolder);

System.out.println("Card Number: " + cardNumber);

}

}

**PhonePePayment.java**

package com.example.strategy;

public class PhonePePayment implements PaymentStrategy {

private String phoneNumber;

public PhonePePayment(String phoneNumber) {

this.phoneNumber = phoneNumber;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PhonePe");

System.out.println("PhonePe Account: " + phoneNumber);

}

}

**PaymentContext.java**

**package com.example.strategy;**

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void makePayment(double amount) {

if (paymentStrategy == null) {

System.out.println("No payment strategy selected.");

} else {

paymentStrategy.pay(amount);

}

}

}

**Main.java**

package com.example.strategy;

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

PaymentStrategy creditCardPayment = new CreditCardPayment("9876-5432-1234-0000", "Riya Sharma");

context.setPaymentStrategy(creditCardPayment);

context.makePayment(1200.00);

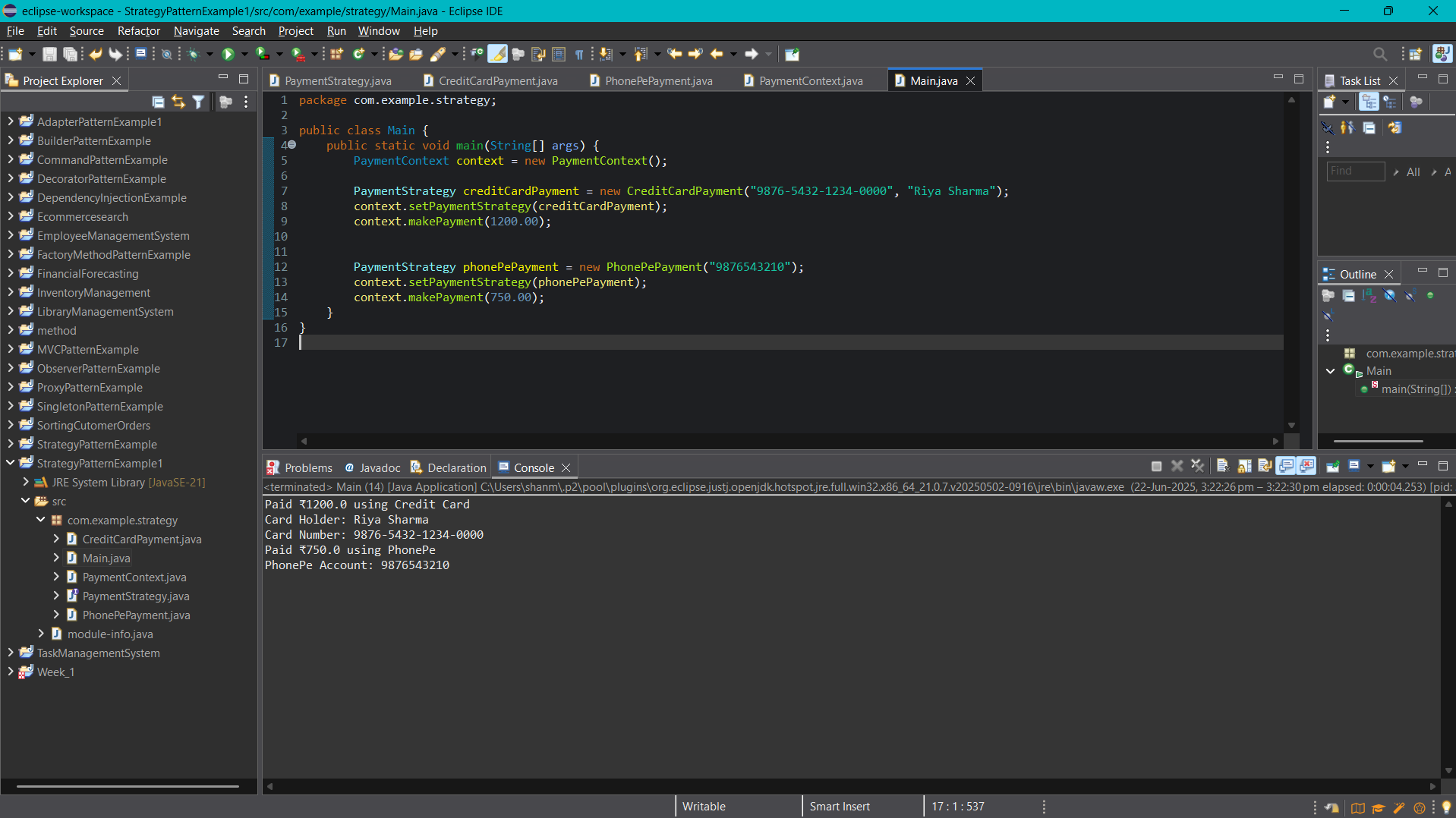
PaymentStrategy phonePePayment = new PhonePePayment("9876543210");

context.setPaymentStrategy(phonePePayment);

context.makePayment(750.00);

}

}



**Exercise 9: Implementing the Command Pattern**

**Command.java**

package com.example.command;

public interface Command {

void execute();

}

**LightOnCommand.java**

package com.example.command;

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

package com.example.command;

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**RemoteControl.java**

package com.example.command;

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**Light.java**

package com.example.command;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**Main.java**

package com.example.command;

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);

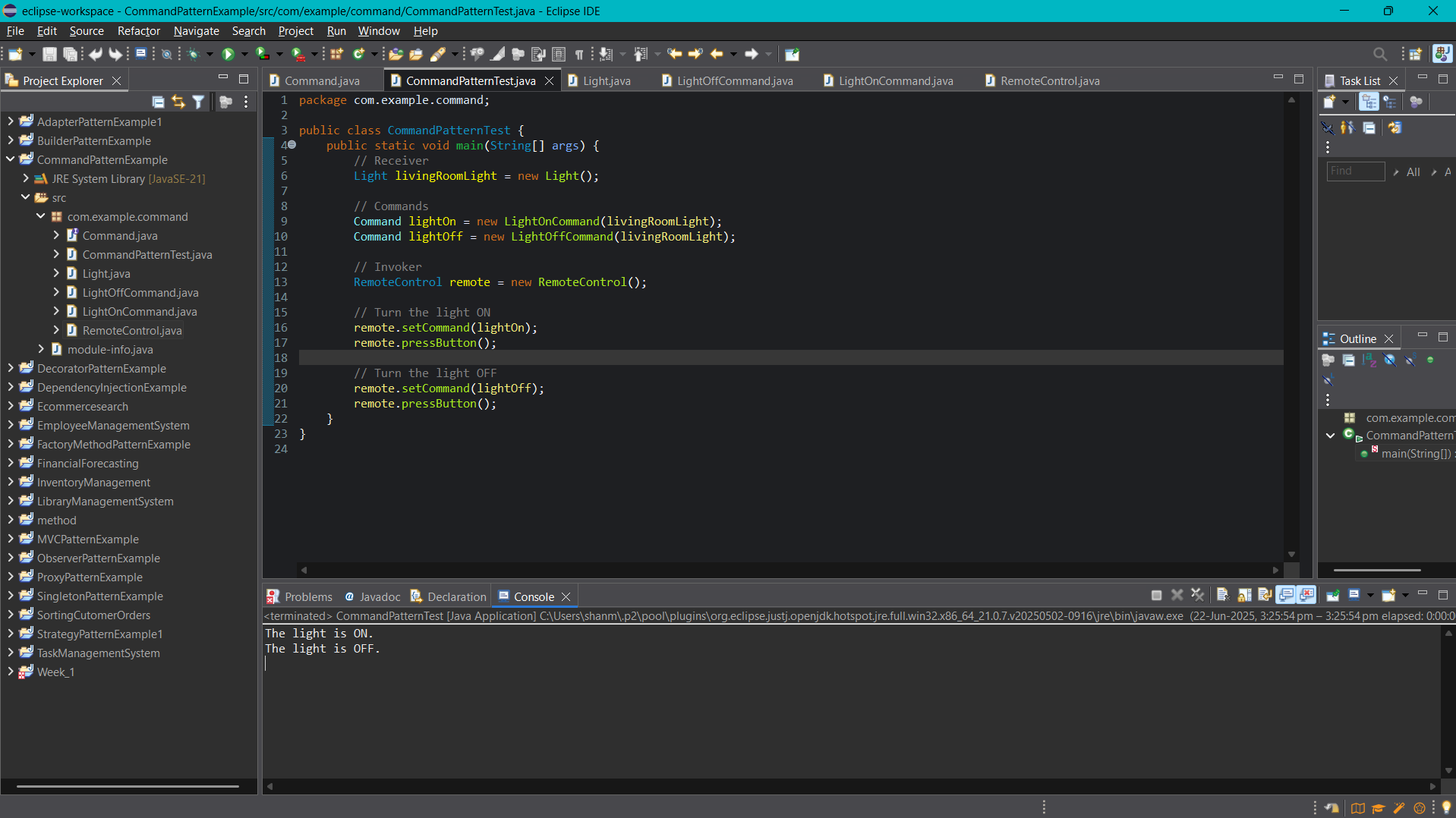
remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Student.java**

package com.example.mvc;

public class Student {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

**StudentView.java**

package com.example.mvc;

public class StudentView {

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

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

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

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

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

}

}

**StudentController.java**

package com.example.mvc;

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

}

}

**Main.java**

package com.example.mvc;

public class Main {

public static void main(String[] args) {

Student student = new Student("123", "Pari", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

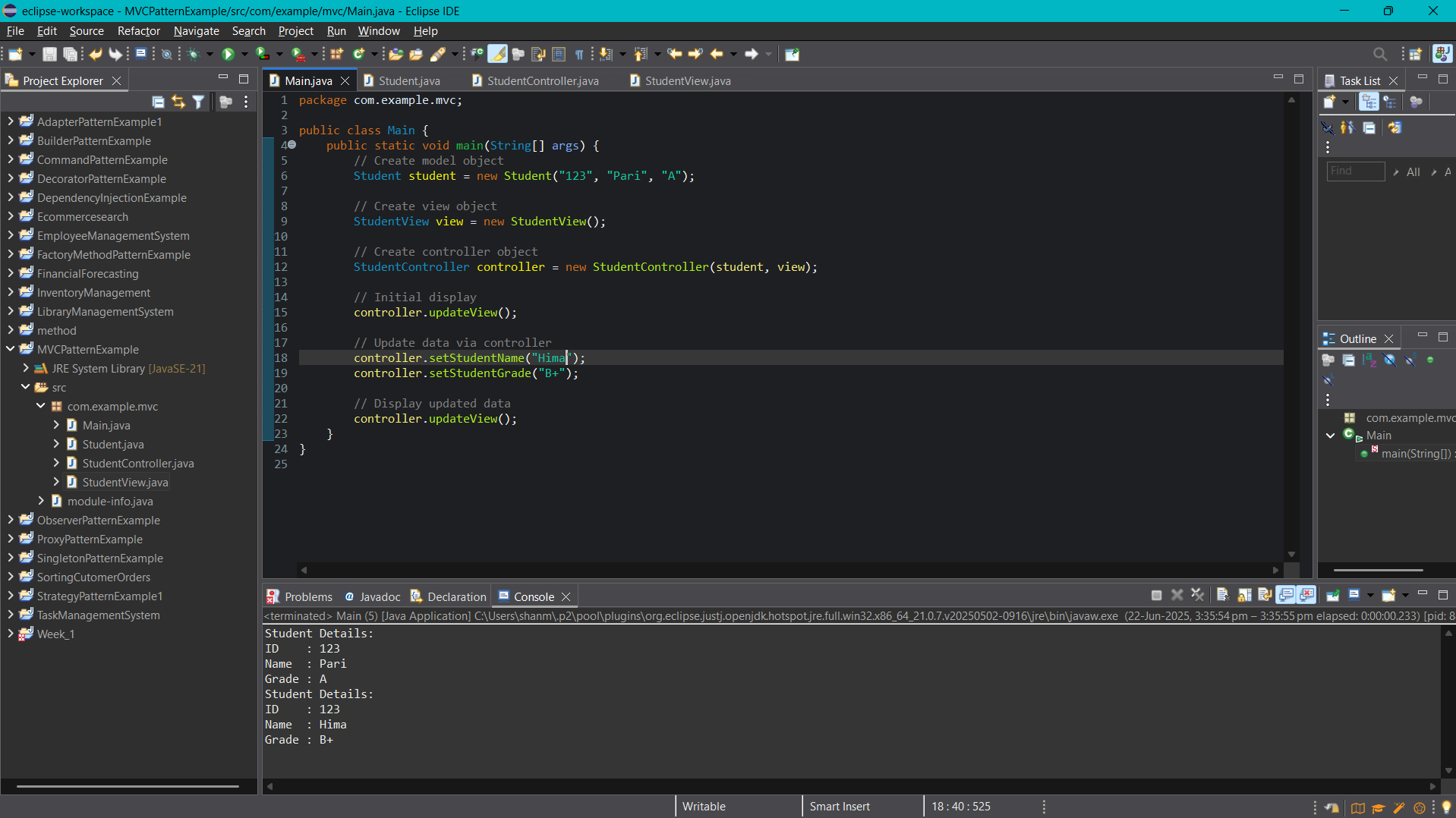
controller.setStudentName("Hima");

controller.setStudentGrade("B+");

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

package com.example.dependencyinjection;

public interface CustomerRepository {

Customer findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

package com.example.dependencyinjection;

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(int id) {

// Dummy data for demonstration

return new Customer(id, "krishna", "krishna123@gmail.com");

}

}

**Customer.java**

package com.example.dependencyinjection;

public class Customer {

private int id;

private String name;

private String email;

public Customer(int id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

public void displayInfo() {

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

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

System.out.println("Email : " + email);

}

}

**CustomerService.java**

package com.example.dependencyinjection;

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(int id) {

Customer customer = customerRepository.findCustomerById(id);

customer.displayInfo();

}

}

**Main.java**

package com.example.dependencyinjection;

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomerDetails(1);

}

}

