Design Patterns and Principles - Java Implementations

# Exercise 1: Singleton Pattern - Logger

## Code

public class Logger {

private static Logger instance = new Logger();

private Logger() {

System.out.println("Logger instance created");

}

public static Logger getInstance() {

return instance;

}

public void log(String message) {

System.out.println("Log: " + message);

}

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("Singleton confirmed: Both logger instances are the same.");

} else {

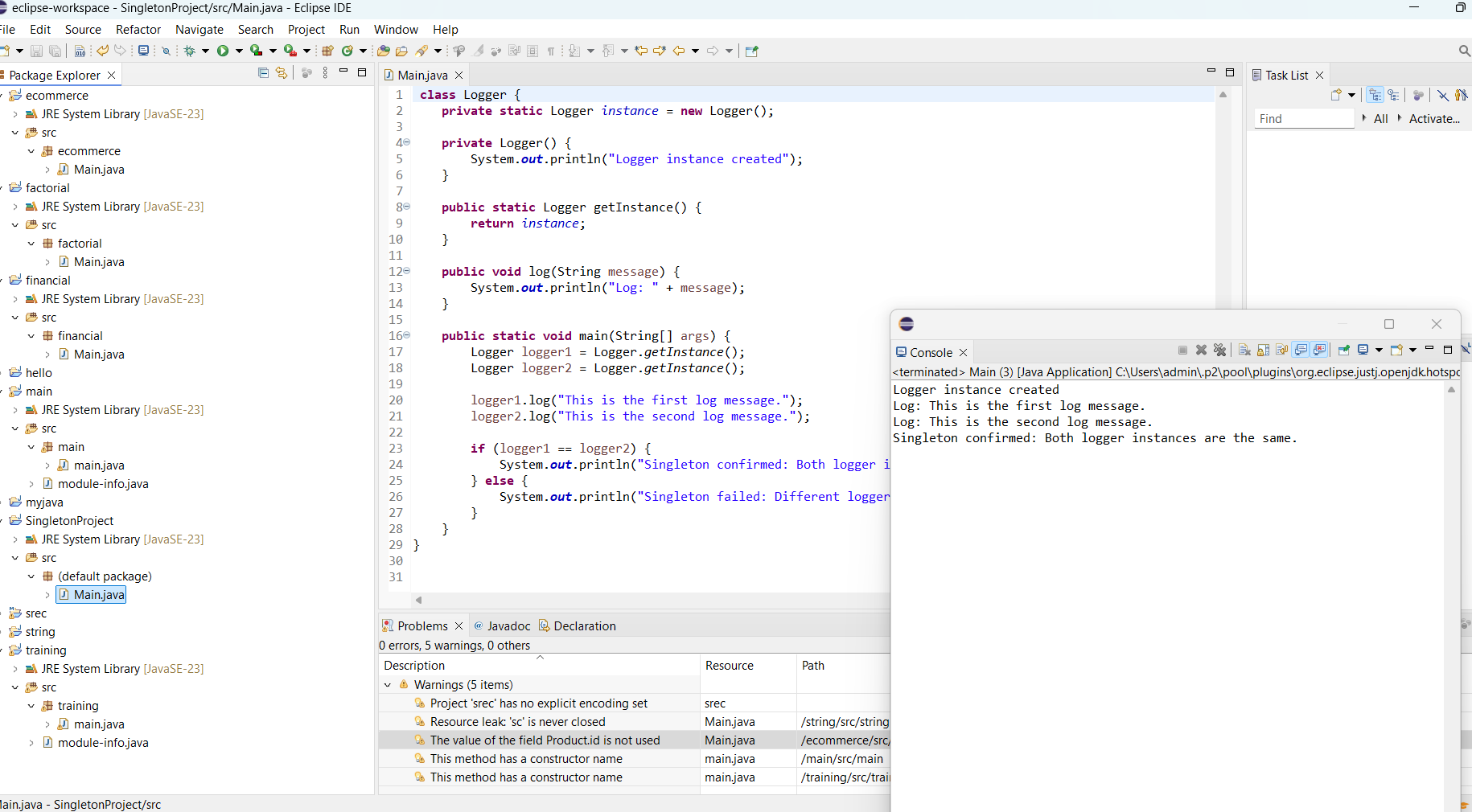
System.out.println("Singleton failed: Different logger instances exist.");

}

}

}

**OUTPUT:**



# Exercise 2: Implementing the Factory method Pattern

**CODE:**

// Document.java - Interface for all document types

interface Document {

void open();

}

// WordDocument.java - Concrete implementation of Document

class WordDocument implements Document {

@Override

public void open() {

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

}

}

// PdfDocument.java - Concrete implementation of Document

class PdfDocument implements Document {

@Override

public void open() {

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

}

}

// ExcelDocument.java - Concrete implementation of Document

class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

// DocumentFactory.java - Abstract Factory

abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordDocumentFactory.java - Concrete Factory for Word documents

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java - Concrete Factory for PDF documents

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java - Concrete Factory for Excel documents

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

// TestFactoryMethod.java - Test class to demonstrate Factory Method

public class TestFactoryMethod {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

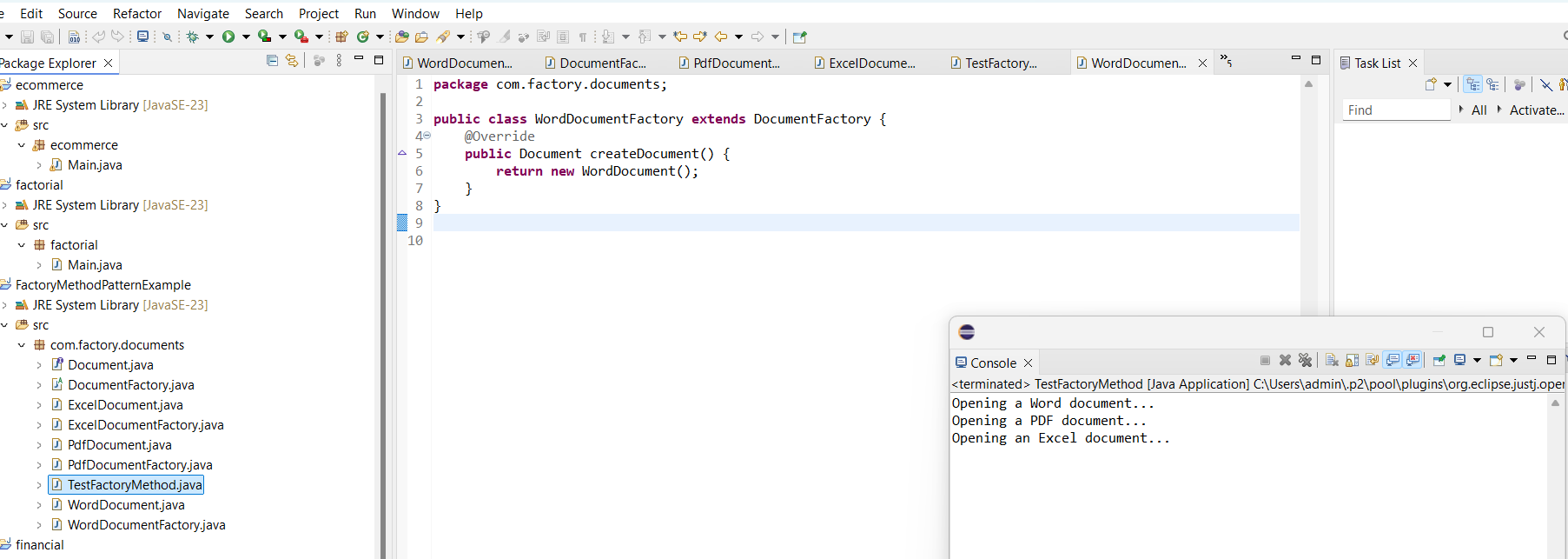
Document excel = excelFactory.createDocument();

excel.open();

}

}

**OUTPUT:**



**EXERCISE 3: IMPLEMENTING BUILDER PATTERN**

**CODE:**

**1.COMPUTER.JAVA**

package builder;

public class Computer {

// Required attributes

private String cpu;

private String ram;

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;

}

public void displayConfig() {

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: " + graphicsCard);

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

}

// Static nested Builder class

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

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(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

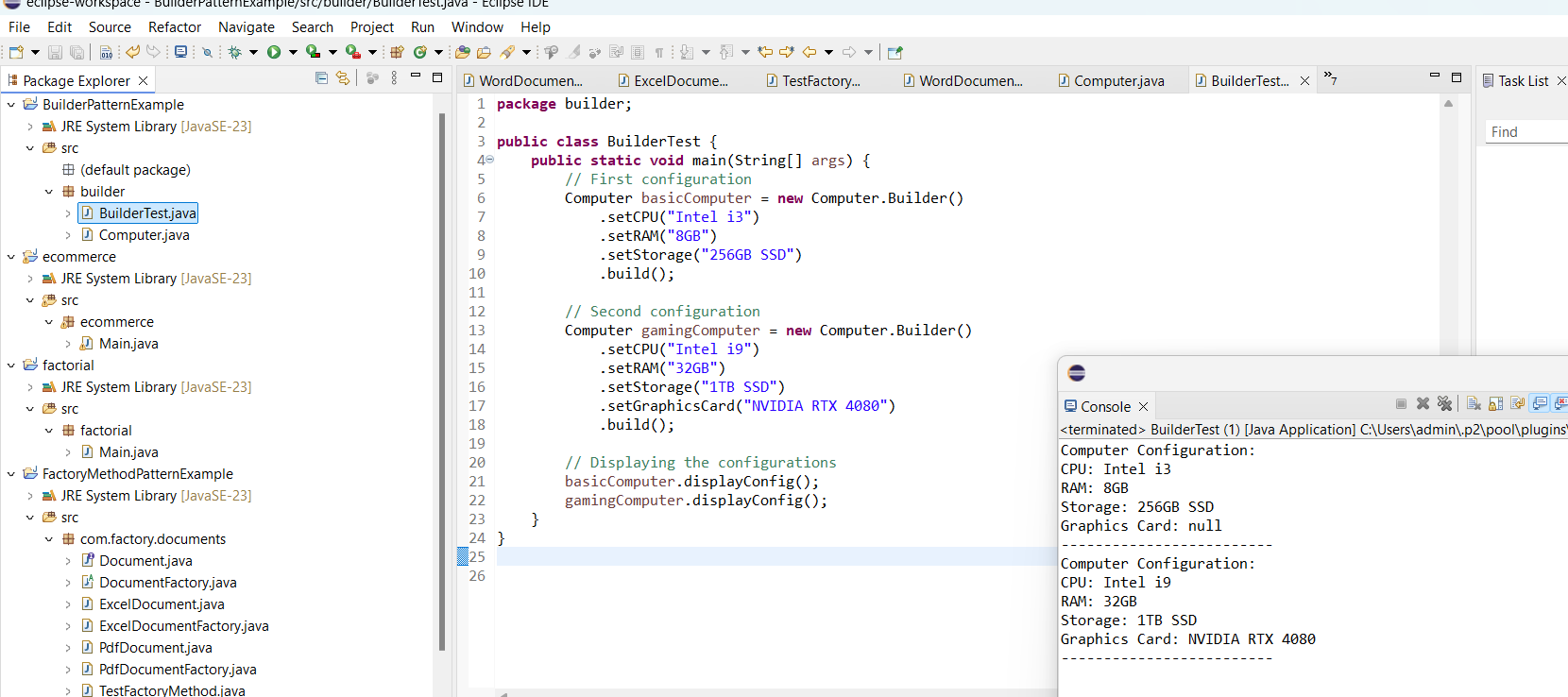
return new Computer(this);

}

}

}

**OUTPUT:**

****

**Exercise 4: Implementing the Adapter Pattern**

**CODE:**

// File: PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

// File: PayPalGateway.java

public class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

// File: StripeGateway.java

public class StripeGateway {

public void makeStripePayment(double amount) {

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

}

}

// File: PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway gateway) {

this.payPalGateway = gateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.sendPayment(amount);

}

}

// File: StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway gateway) {

this.stripeGateway = gateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.makeStripePayment(amount);

}

}

// File: AdapterPatternTest.java

public class AdapterPatternTest {

public static void main(String[] args) {

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

paypalProcessor.processPayment(500.0);

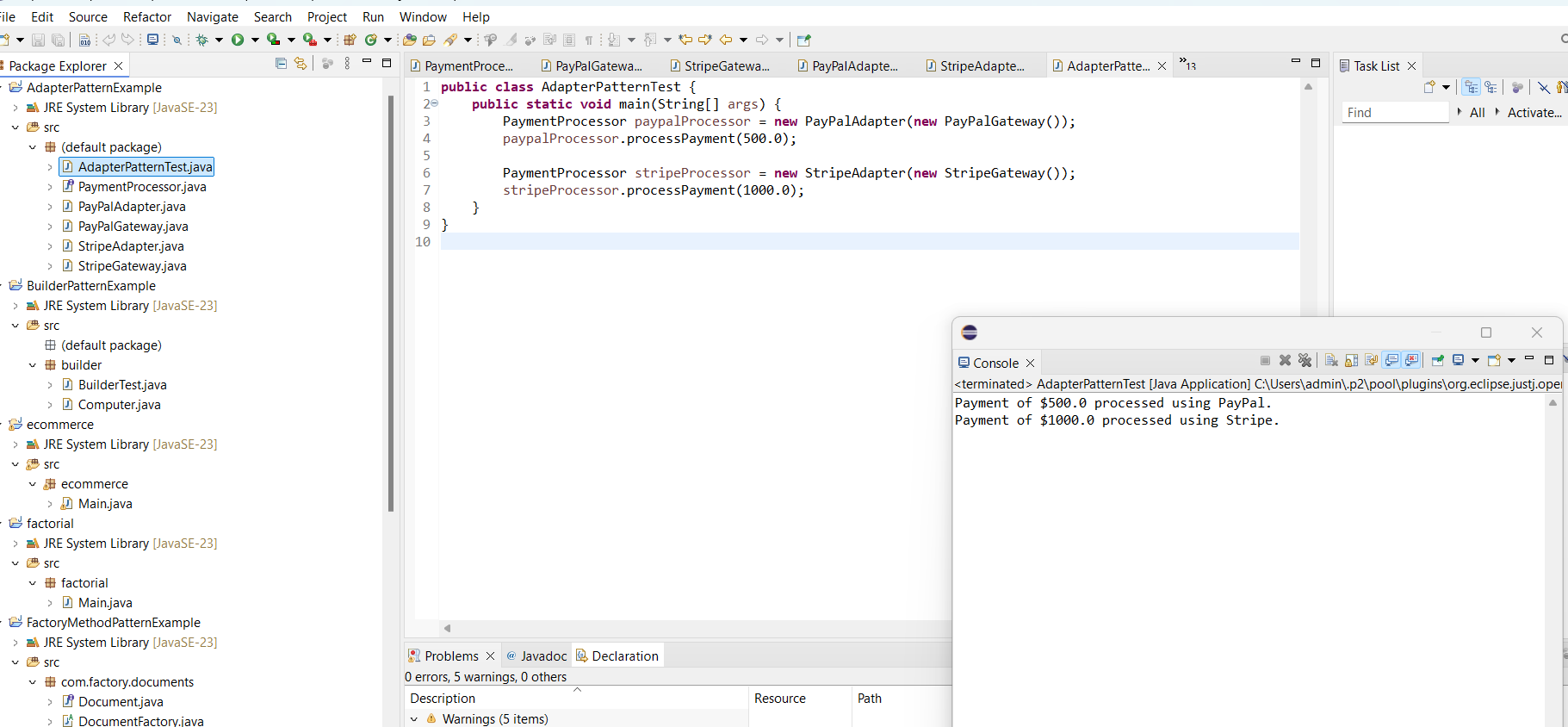
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(1000.0);

}

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**CODE:**

// Package Declaration

package com.decorator.notification;

// Component Interface

public interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

@Override

public void send(String message) {

wrappee.send(message);

}

}

// Concrete Decorator: SMS

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);

}

}

// Concrete Decorator: Slack

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);

}

}

// Test Class

public class DecoratorPatternTest {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier(); // Base notification

notifier = new SMSNotifierDecorator(notifier); // Add SMS

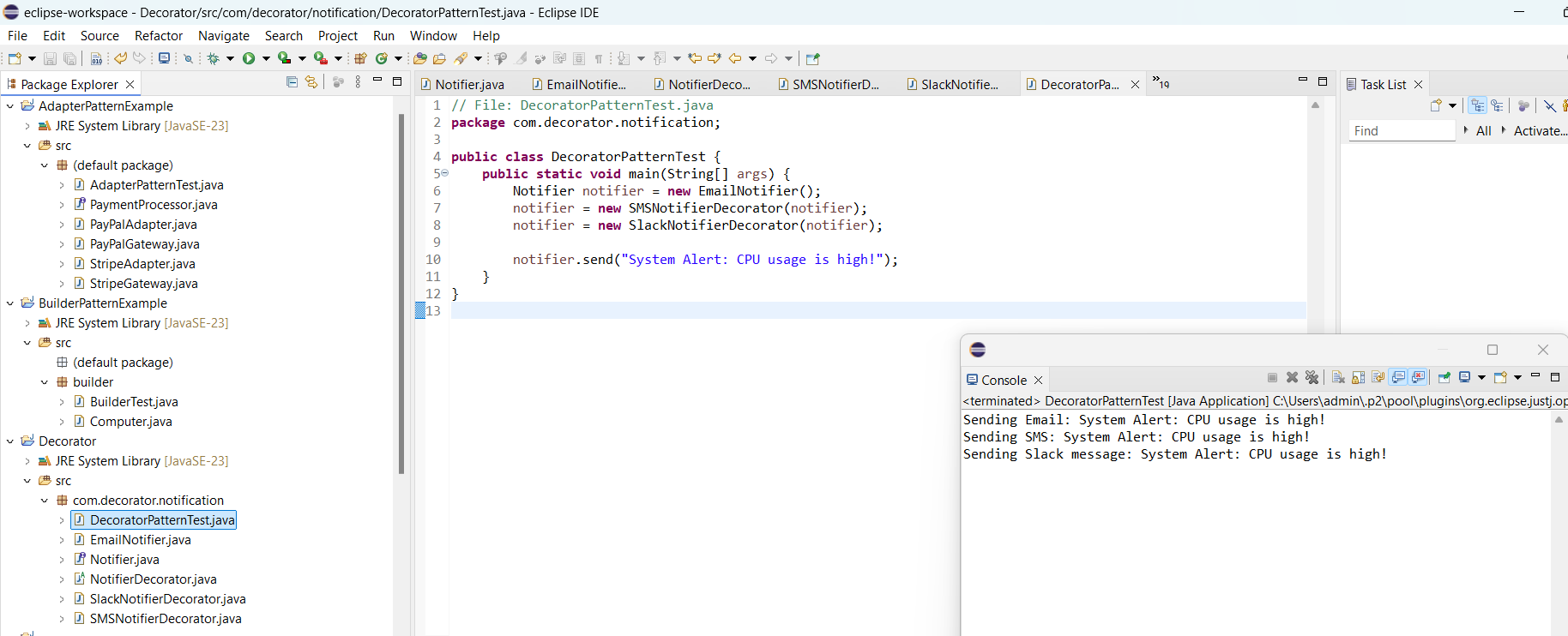
notifier = new SlackNotifierDecorator(notifier); // Add Slack

notifier.send("System Alert: CPU usage is high!");

}

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

// Package Declaration

package com.proxy.imageviewer;

// Subject Interface

public interface Image {

void display();

}

// Real Subject: Loads image from a remote server

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);

}

}

// Proxy Class: Controls access to RealImage

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy initialization

} else {

System.out.println("Image already loaded: " + filename);

}

realImage.display();

}

}

// Test Class

public class ProxyPatternTest {

public static void main(String[] args) {

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

Image image2 = new ProxyImage("city.png");

// First time loading (from remote)

image1.display(); // Loads and displays

System.out.println();

// Cached display (no loading again)

image1.display(); // Just displays

System.out.println();

// Another image loading

image2.display(); // Loads and displays

System.out.println();

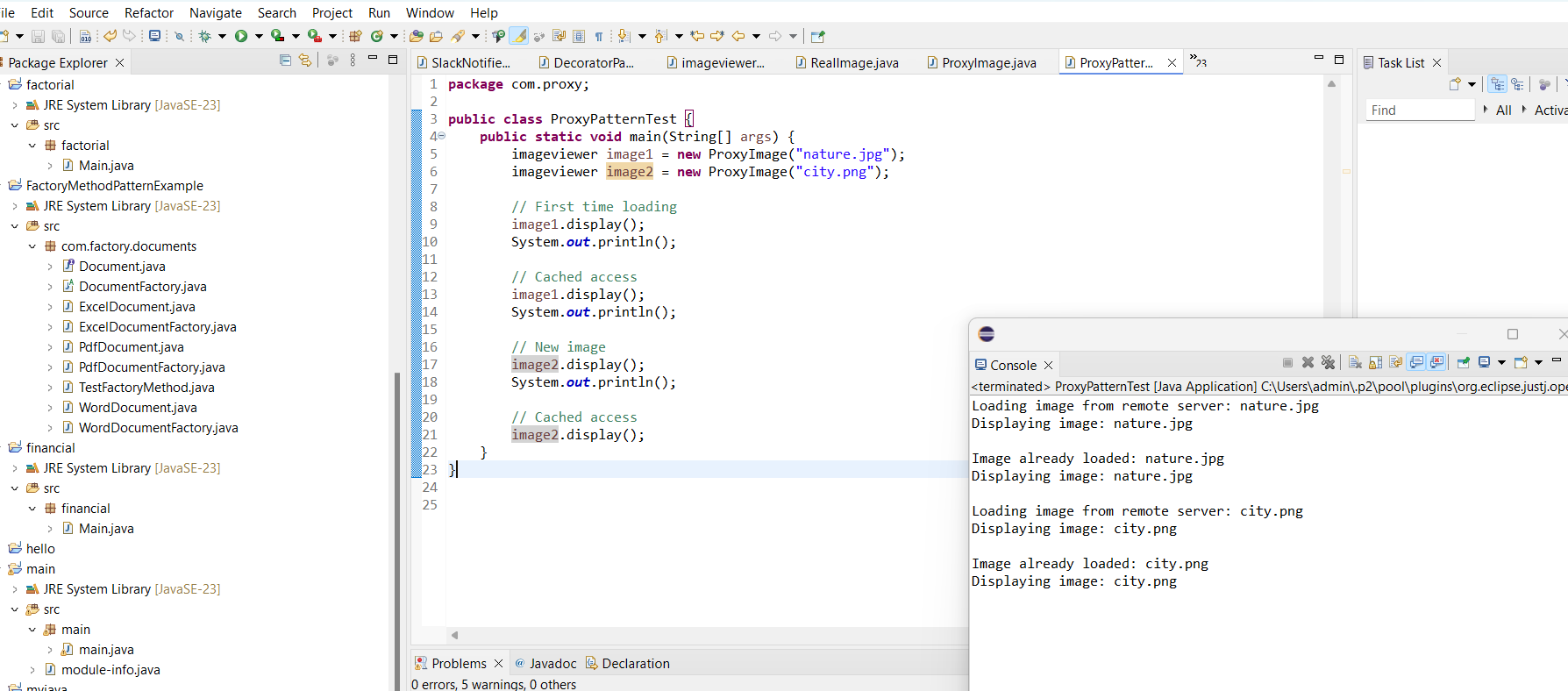
// Again from cache

image2.display(); // Just displays

}

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**CODE:**

// Package Declaration

package com.observer.stockmarket;

import java.util.ArrayList;

import java.util.List;

// Subject Interface

interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

// Observer Interface

interface Observer {

void update(String stockName, double stockPrice);

}

// Concrete Subject

class StockMarket implements Stock {

private List<Observer> observers;

private String stockName;

private double stockPrice;

public StockMarket(String stockName) {

this.stockName = stockName;

this.observers = new ArrayList<>();

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

System.out.println(observer + " registered for " + stockName);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

System.out.println(observer + " removed from " + stockName);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

// Concrete Observer: Mobile App

class MobileApp implements Observer {

private String user;

public MobileApp(String user) {

this.user = user;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println("MobileApp [" + user + "] received update: " + stockName + " is now $" + stockPrice);

}

@Override

public String toString() {

return "MobileApp(" + user + ")";

}

}

// Concrete Observer: Web App

class WebApp implements Observer {

private String user;

public WebApp(String user) {

this.user = user;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println("WebApp [" + user + "] received update: " + stockName + " is now $" + stockPrice);

}

@Override

public String toString() {

return "WebApp(" + user + ")";

}

}

// Main Test Class

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket appleStock = new StockMarket("AAPL");

Observer mobileUser = new MobileApp("Alice");

Observer webUser = new WebApp("Bob");

appleStock.registerObserver(mobileUser);

appleStock.registerObserver(webUser);

System.out.println("\n-- Stock price updated to $150.50 --");

appleStock.setStockPrice(150.50);

System.out.println("\n-- Stock price updated to $152.00 --");

appleStock.setStockPrice(152.00);

appleStock.removeObserver(webUser);

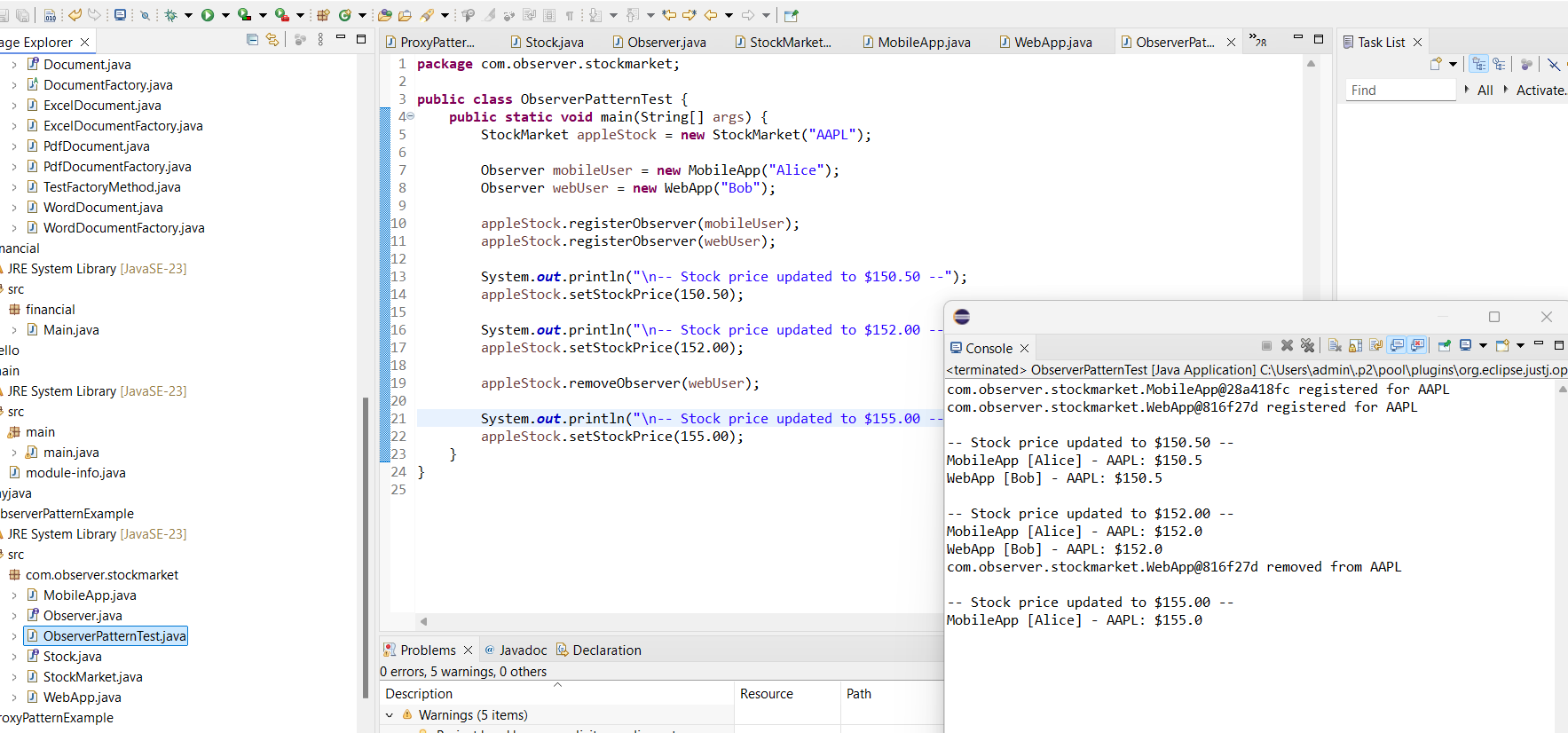
System.out.println("\n-- Stock price updated to $155.00 --");

appleStock.setStockPrice(155.00);

}

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**CODE:**

// Package Declaration

package com.strategy.payment;

// Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategy: Credit Card

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(double amount) {

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

}

}

// Concrete Strategy: PayPal

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

// Context Class

class PaymentContext {

private PaymentStrategy strategy;

// Set the strategy dynamically

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

// Execute the payment strategy

public void processPayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

// Test Class

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

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

context.processPayment(250.00);

System.out.println();

// Pay using PayPal

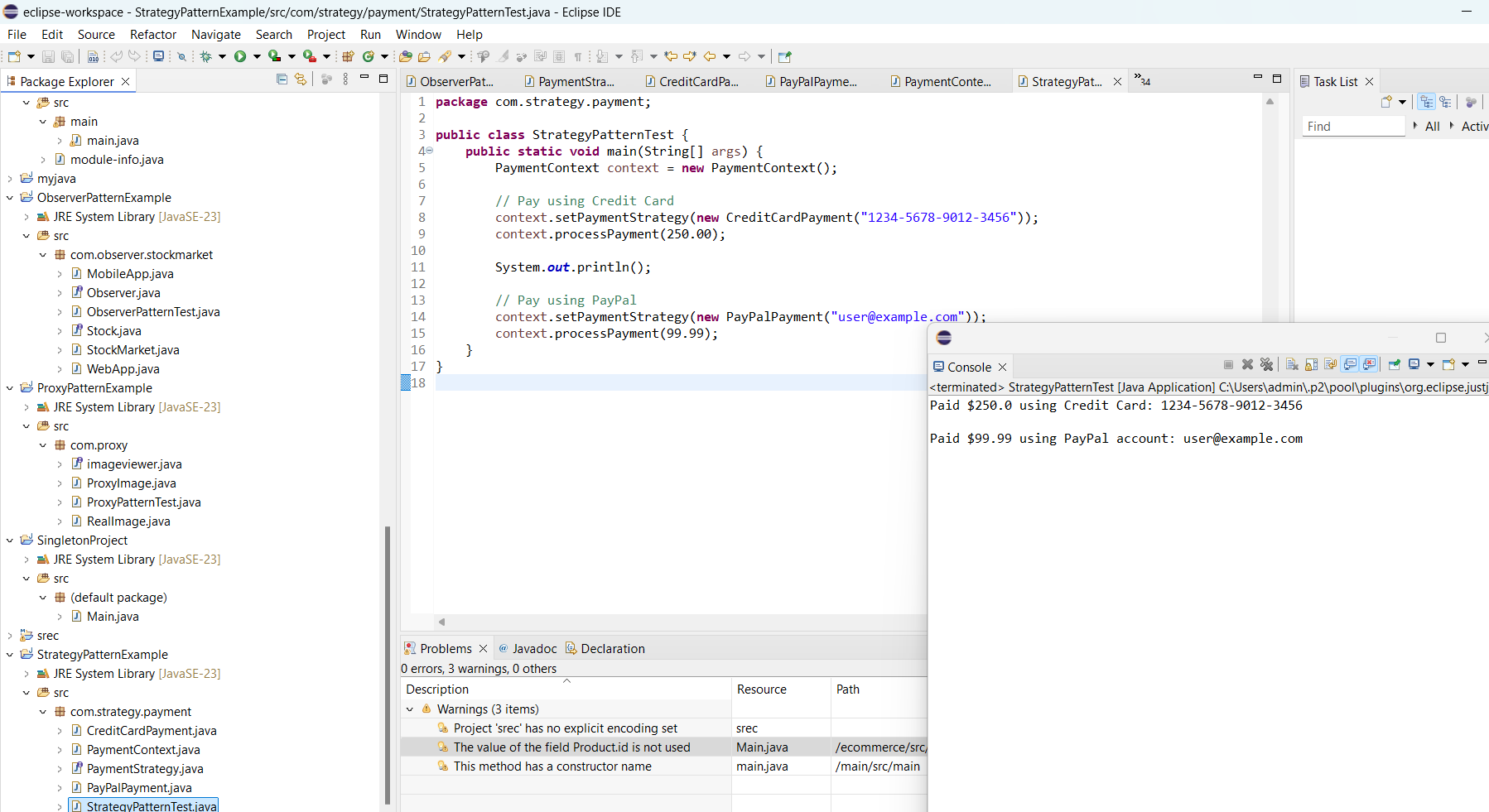
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(99.99);

}

}

**OUTPUT:**



**Exercise 9: Implementing the Command Pattern**

**CODE:**

// Package Declaration

package com.command.homeautomation;

// Command Interface

interface Command {

void execute();

}

// Receiver Class: Light

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Concrete Command: Turn Light ON

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// Concrete Command: Turn Light OFF

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

// Invoker Class: RemoteControl

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.");

}

}

}

// Main Test Class

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn the light ON

remote.setCommand(lightOn);

remote.pressButton();

// Turn the light OFF

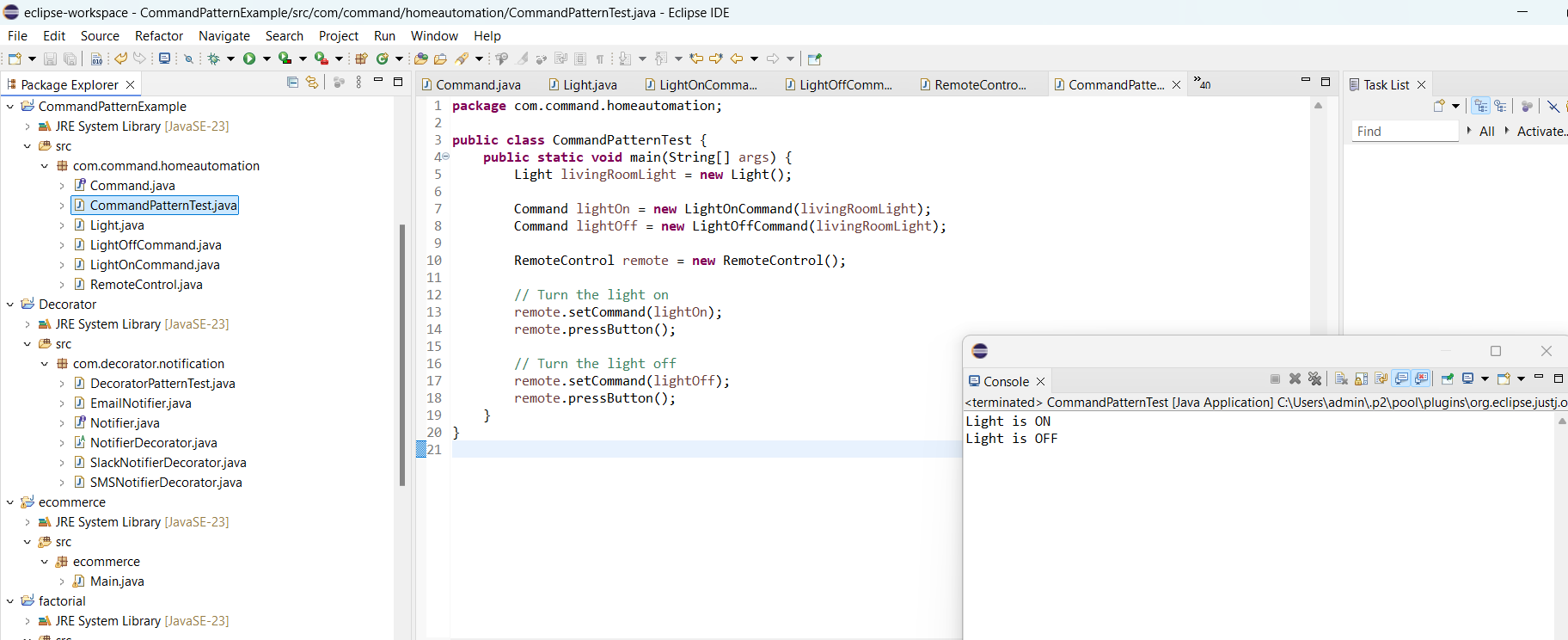
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**CODE:**

// Package Declaration

package com.mvc.studentapp;

// Model Class

class Student {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

// Getters and Setters

public String getId() {

return id;

}

public String getName() {

return name;

}

public String getGrade() {

return grade;

}

public void setId(String id) {

this.id = id;

}

public void setName(String name) {

this.name = name;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

// View Class

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);

System.out.println();

}

}

// Controller Class

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Setters for updating model

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Getters for accessing model data

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

// Update the view

public void updateView() {

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

}

}

// Main Class to test MVC

public class MVCPatternTest {

public static void main(String[] args) {

// Create model

Student student = new Student("S001", "John Doe", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Update student data

controller.setStudentName("Jane Smith");

controller.setStudentGrade("A+")

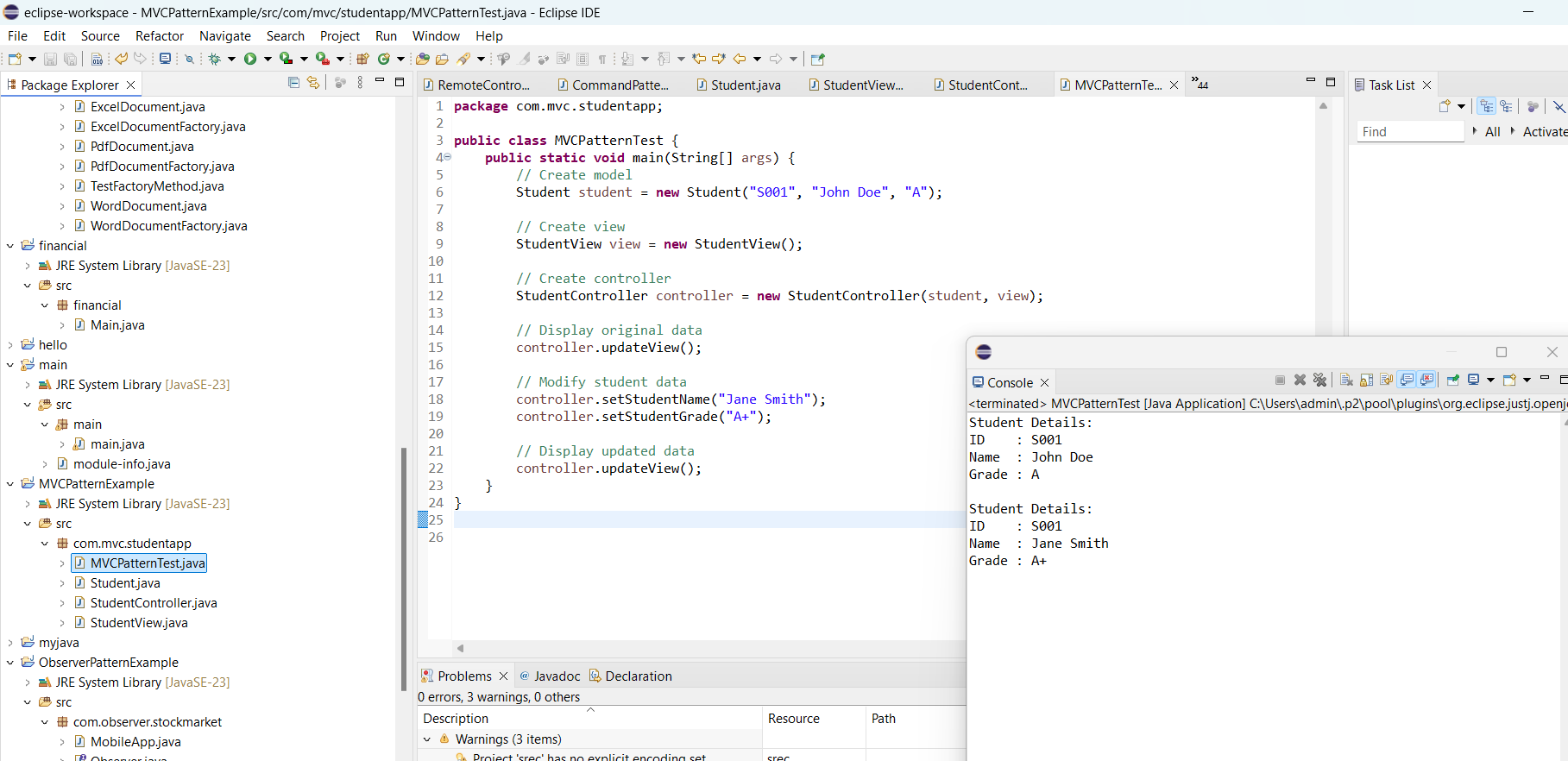
// Display updated details

controller.updateView();

}

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**CODE:**

// Package Declaration

package com.di.customerapp;

// Repository Interface

interface CustomerRepository {

String findCustomerById(String customerId);

}

// Concrete Repository Implementation

class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String customerId) {

// Simulating a database fetch

return "Customer[ID=" + customerId + ", Name=John Doe]";

}

}

// Service Class using Constructor Injection

class CustomerService {

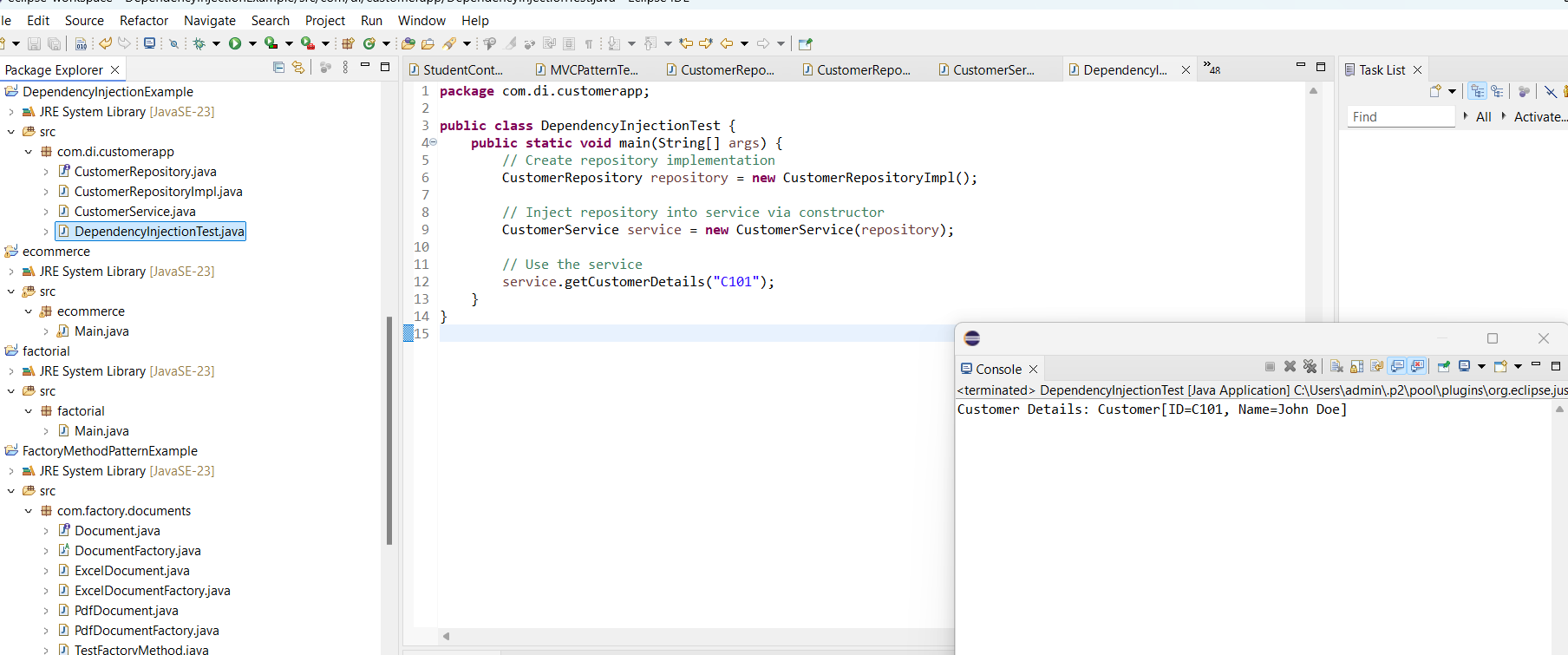
private CustomerRepository customerRepository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

**OUTPUT:**

****