**DESIGN PATTERNS AND PRINCIPLES SOLUTION:**

**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****.*

1. ***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.*

1. ***Implement the Singleton Pattern:***

* *Write code to ensure that the Logger class follows the Singleton design pattern.*

1. ***Test the Singleton Implementation:***

* *Create a test class to verify that only one instance of Logger is created and used across the application.*

**SOLUTION:**

public class Logger {

private static Logger instance;

private Logger() {

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

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();}

return instance;}

public void log(String msg) {

System.out.println("Log: " + msg);}}

public class LoggerTest {

public static void main(String[] args) {

Logger l1 = Logger.getInstance();

Logger l2 = Logger.getInstance();

l1.log("First log");

l2.log("Second log");

if (l1 == l2) {

System.out.println("Same instance");

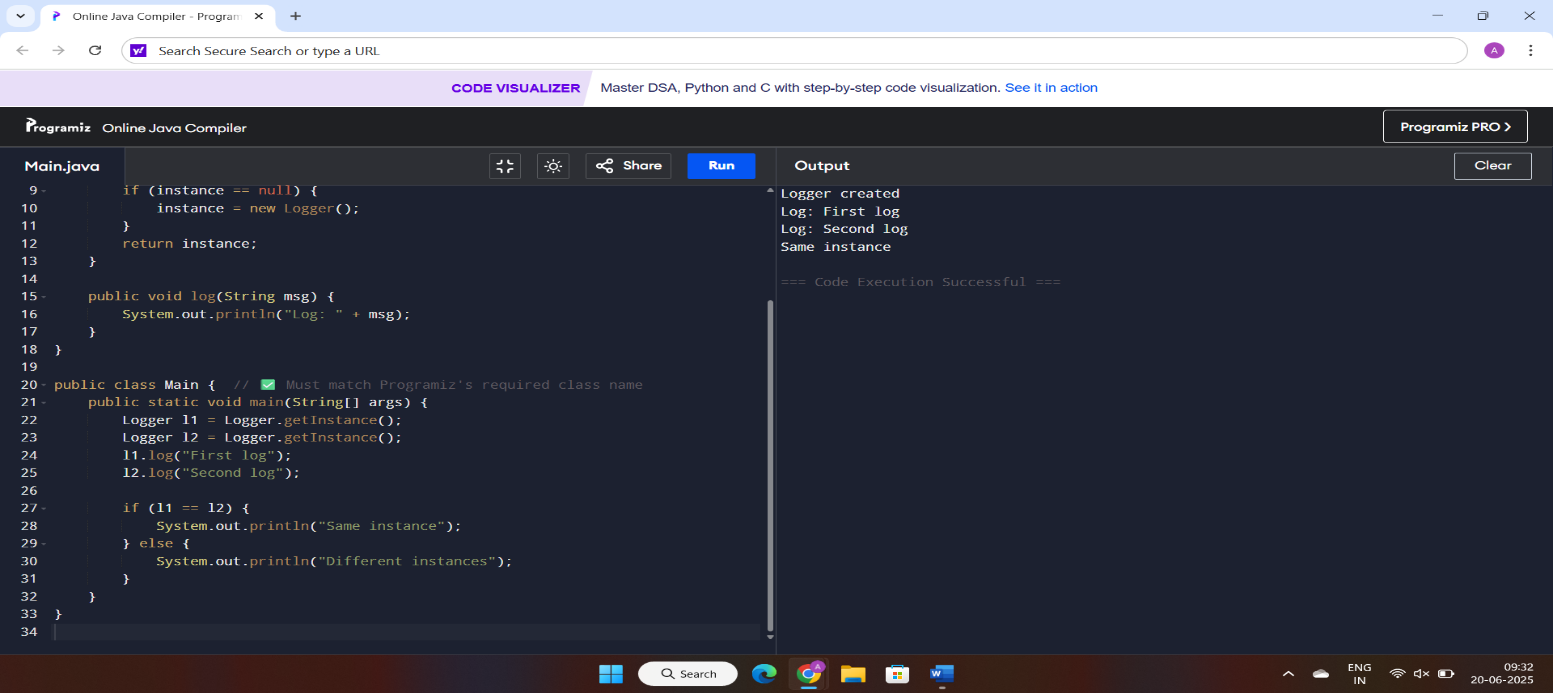
} else {

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

}

}

}

**OUTPUT:**

**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****.*

1. ***Define Document Classes:***

* *Create interfaces or abstract classes for different document types such as* ***WordDocument****,* ***PdfDocument****, and* ***ExcelDocument****.*

1. ***Create Concrete Document Classes:***

* *Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.*

1. ***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.*

1. ***Test the Factory Method Implementation:***

* *Create a test class to demonstrate the creation of different document types using the factory method*

**SOLUTION:**

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("Opening Word Document");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening PDF Document");

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

Document createDocument() {

return new ExcelDocument();

}

}

public class DocumentTest {

public static void main(String[] args) {

DocumentFactory factory;

factory = new WordFactory();

Document doc1 = factory.createDocument();

doc1.open();

factory = new PdfFactory();

Document doc2 = factory.createDocument();

doc2.open();

factory = new ExcelFactory();

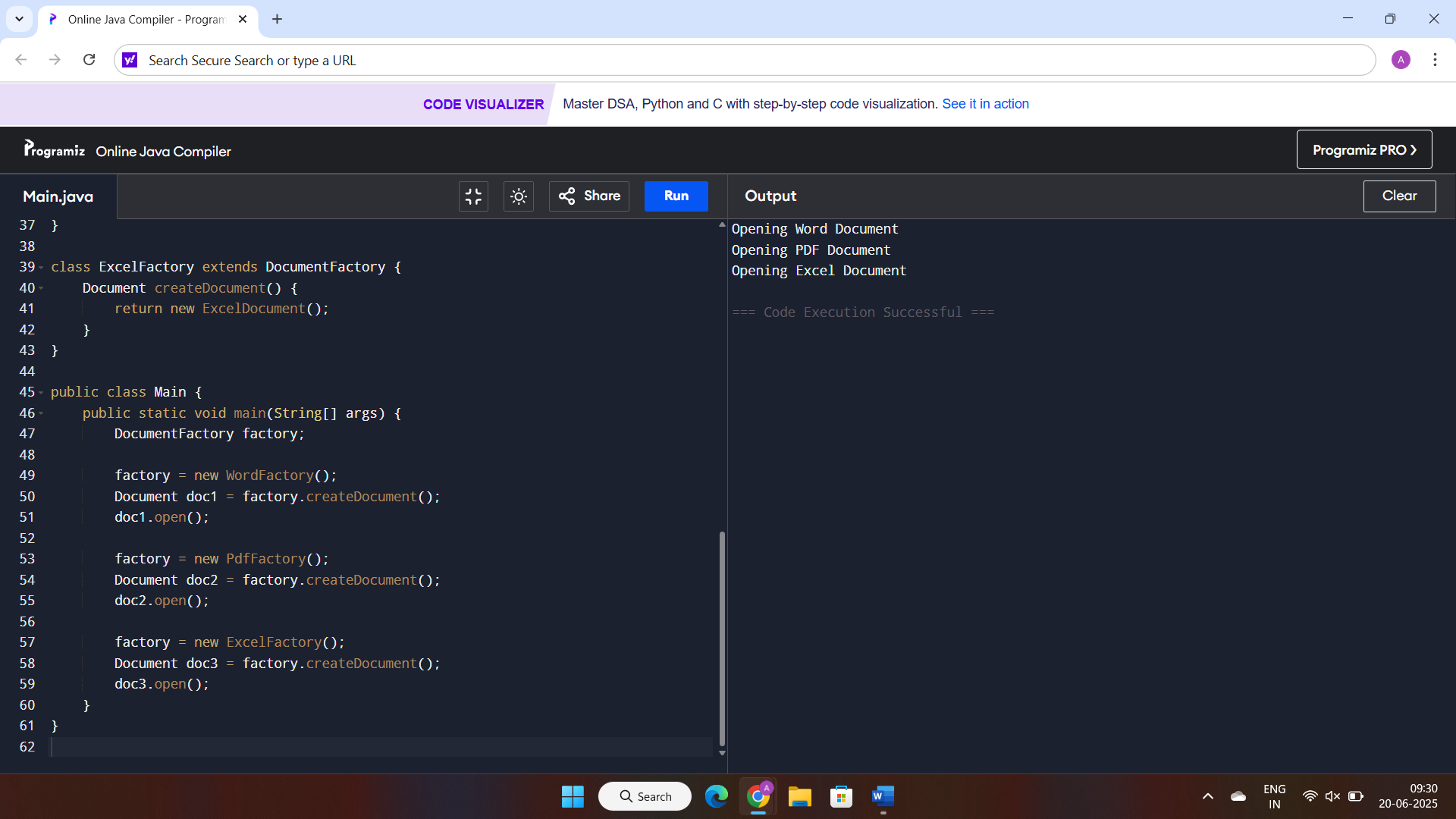
Document doc3 = factory.createDocument();

doc3.open();

}

}

**OUTPUT:**



**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****.*

1. ***Define a Product Class:***

* *Create a class* ***Computer*** *with attributes like* ***CPU****,* ***RAM****,* ***Storage****, etc.*

1. ***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.*

1. ***Implement the Builder Pattern:***

* *Ensure that the* ***Computer*** *class has a private constructor that takes the* ***Builder*** *as a parameter.*

1. ***Test the Builder Implementation:***

* *Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.*

**SOLUTION:**

class Computer {

private String cpu;

private String ram;

private String storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage; }

public String toString() {

return "Computer [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage + "]";}

static class Builder {

private String cpu;

private String ram;

private String storage;

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 Computer build() {

return new Computer(this); } }}

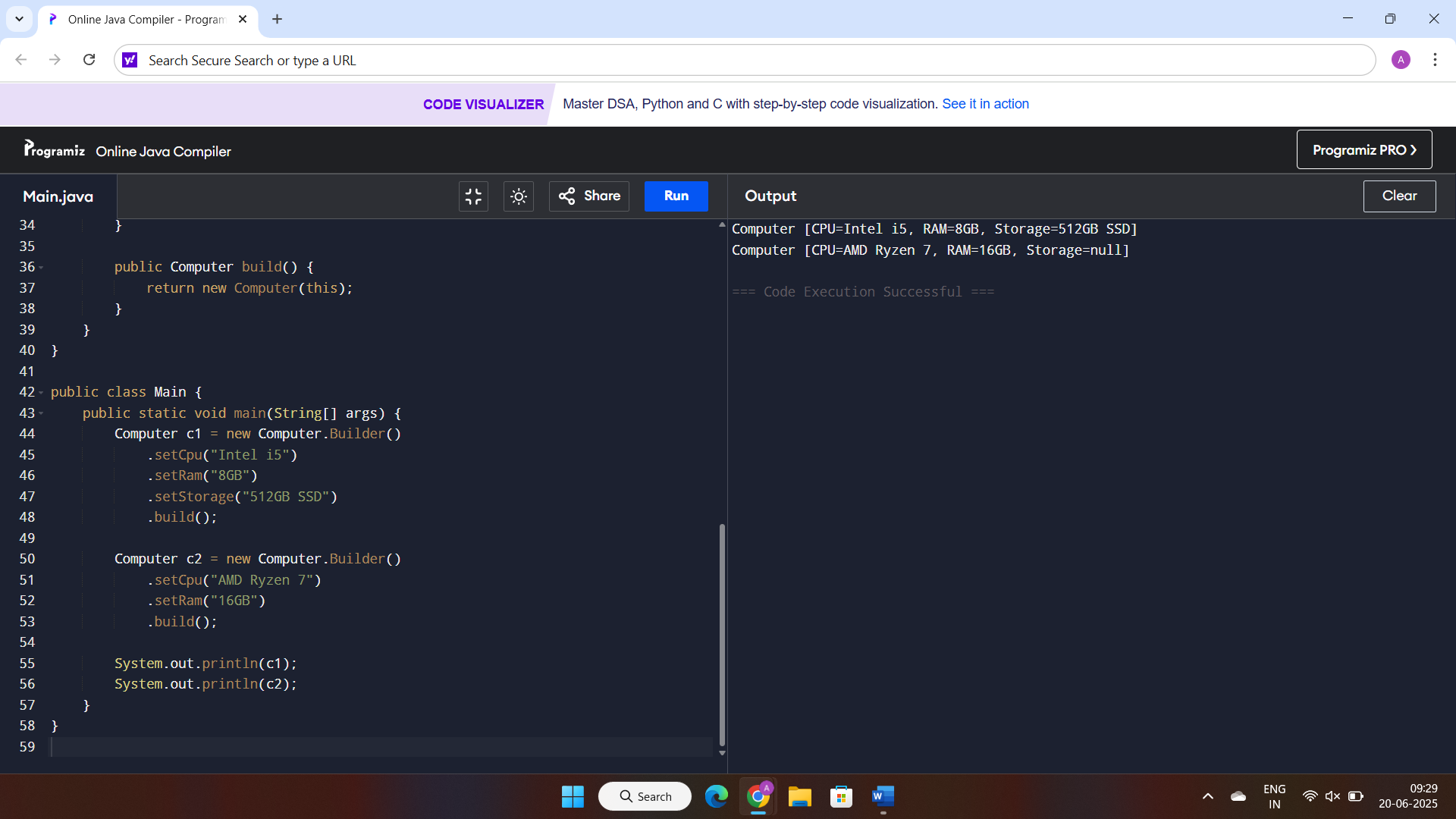
public class ComputerTest {

public static void main(String[] args) {

Computerc1=newComputer.Builder().setCpu("Intel i5").setRam("8GB").setStorage("512GB SSD").build();

Computer c2 = new Computer.Builder().setCpu("AMD Ryzen 7").setRam("16GB").build();

System.out.println(c1);System.out.println(c2);}}

**OUTPUT:**

**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****.*

1. ***Define Target Interface:***

* *Create an interface* ***PaymentProcessor*** *with methods like* ***processPayment()****.*

1. ***Implement Adaptee Classes:***

* *Create classes for different payment gateways with their own methods.*

1. ***Implement the Adapter Class:***

* *Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.*

1. ***Test the Adapter Implementation:***

* *Create a test class to demonstrate the use of different payment gateways through the adapter.*

**SOLUTION:**

interface PaymentProcessor {

void processPayment(double amount);}

class PayPalGateway {

public void makePayment(double amount) {

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

class StripeGateway {

public void pay(double amount) {

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

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;}

public void processPayment(double amount) {

paypal.makePayment(amount);

}}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe; }

public void processPayment(double amount) {

stripe.pay(amount);}}

public class PaymentTest {

public static void main(String[] args) {

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

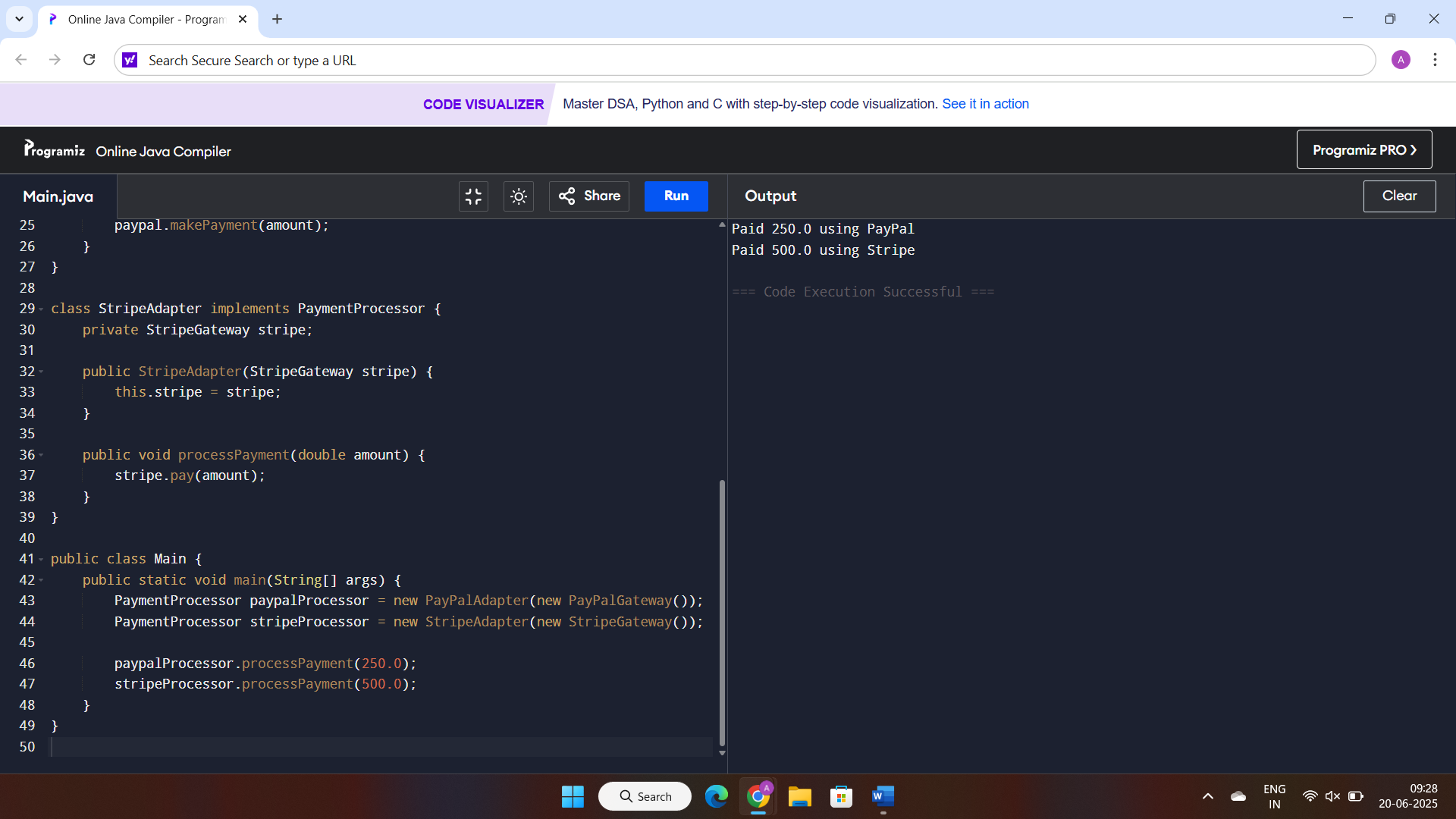
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

paypalProcessor.processPayment(250.0);

stripeProcessor.processPayment(500.0);

}

}

**OUTPUT:**

**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****.*

1. ***Define Component Interface:***

* *Create an interface* ***Notifier*** *with a method* ***send()****.*

1. ***Implement Concrete Component:***

* *Create a class* ***EmailNotifier*** *that implements Notifier.*

1. ***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****.*

1. ***Test the Decorator Implementation:***

* *Create a test class to demonstrate sending notifications via multiple channels using decorators.*

**SOLUTON:**

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class NotificationTest {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

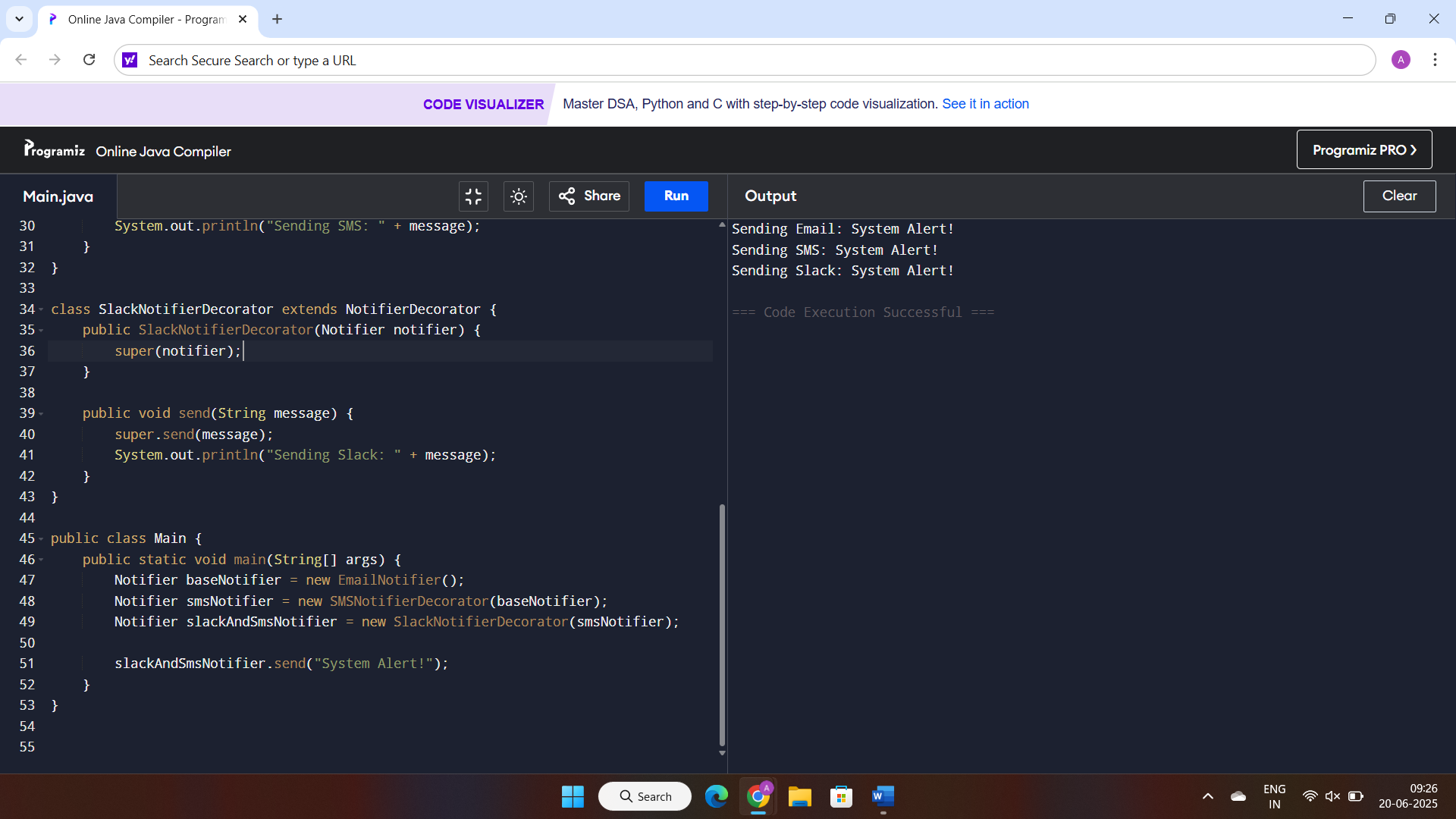
Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

slackAndSmsNotifier.send("System Alert!");

}

}

**OUTPUT:**

**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****.*

1. ***Define Subject Interface:***

* *Create an interface Image with a method* ***display()****.*

1. ***Implement Real Subject Class:***

* *Create a class* ***RealImage*** *that implements Image and loads an image from a remote server.*

1. ***Implement Proxy Class:***

* *Create a class* ***ProxyImage*** *that implements Image and holds a reference to RealImage.*
* *Implement lazy initialization and caching in* ***ProxyImage****.*

1. ***Test the Proxy Implementation:***

* *Create a test class to demonstrate the use of* ***ProxyImage*** *to load and display images.*

**SOLUTION:**

interface Image {

void display();

}

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading " + fileName + " from remote server...");

}

public void display() {

System.out.println("Displaying " + fileName);}}

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();}}

public class ImageViewer {

public static void main(String[] args) {

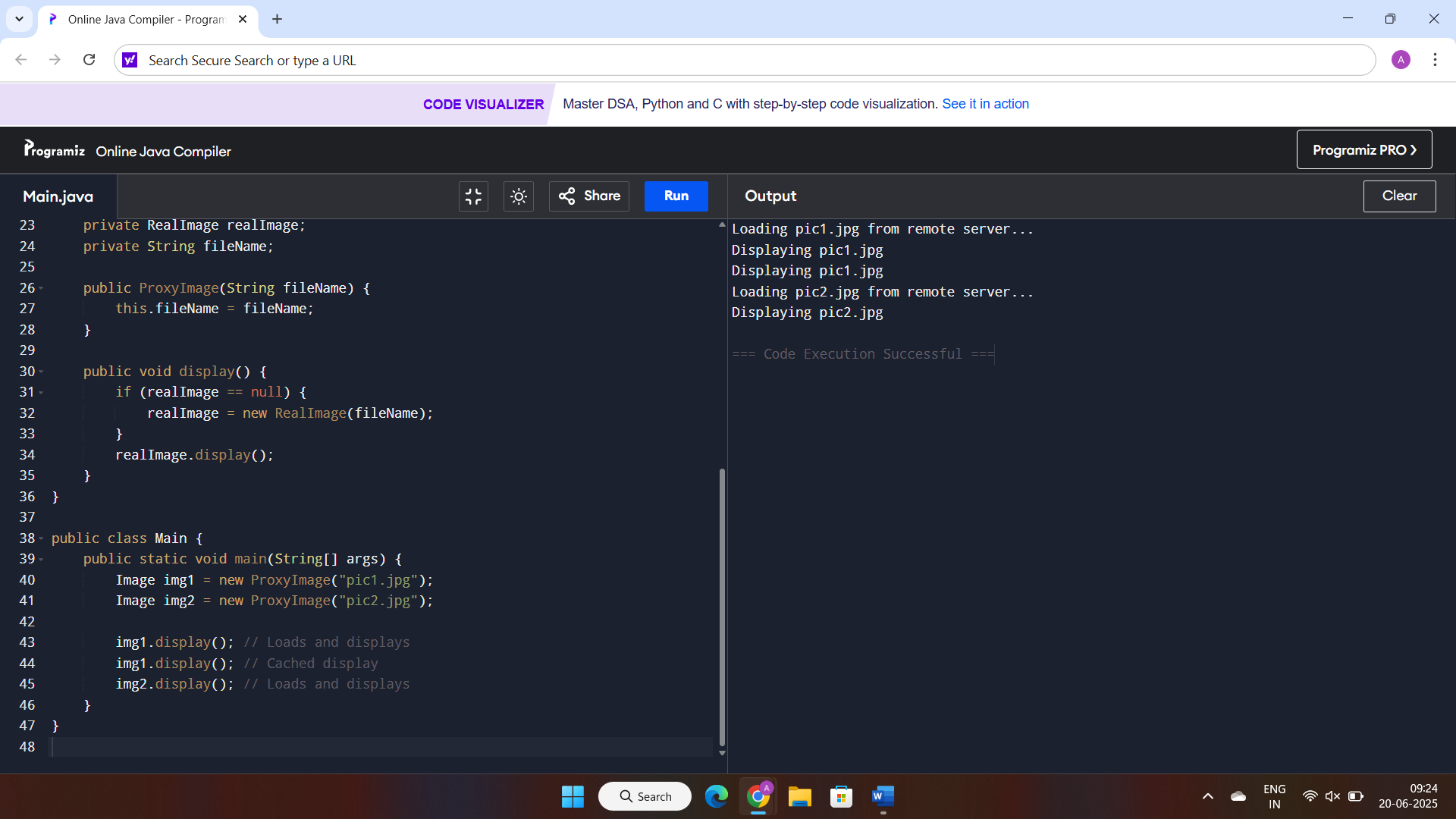
Image img1 = new ProxyImage("pic1.jpg");

Image img2 = new ProxyImage("pic2.jpg");

img1.display();

img1.display();

img2.display(); }}

**OUTPUT:**

**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****.*

1. ***Define Subject Interface:***

* *Create an interface* ***Stock*** *with methods to* ***register****,* ***deregister****, and* ***notify*** *observers.*

1. ***Implement Concrete Subject:***

* *Create a class* ***StockMarket*** *that implements* ***Stock*** *and maintains a list of observers.*

1. ***Define Observer Interface:***

* *Create an interface Observer with a method* ***update().***

1. ***Implement Concrete Observers:***

* *Create classes* ***MobileApp****,* ***WebApp*** *that implement Observer.*

1. ***Test the Observer Implementation:***

* *Create a test class to demonstrate the registration and notification of observers.*

**SOLUTION:**

import java.util.\*;

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private double price;

public void setPrice(double price) {

this.price = price;

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(price);

}

}

}

interface Observer {

void update(double price);

}

class MobileApp implements Observer {

public void update(double price) {

System.out.println("Mobile App - New stock price: " + price);

}

}

class WebApp implements Observer {

public void update(double price) {

System.out.println("Web App - New stock price: " + price);

}

}

public class StockAppTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

market.setPrice(100.5);

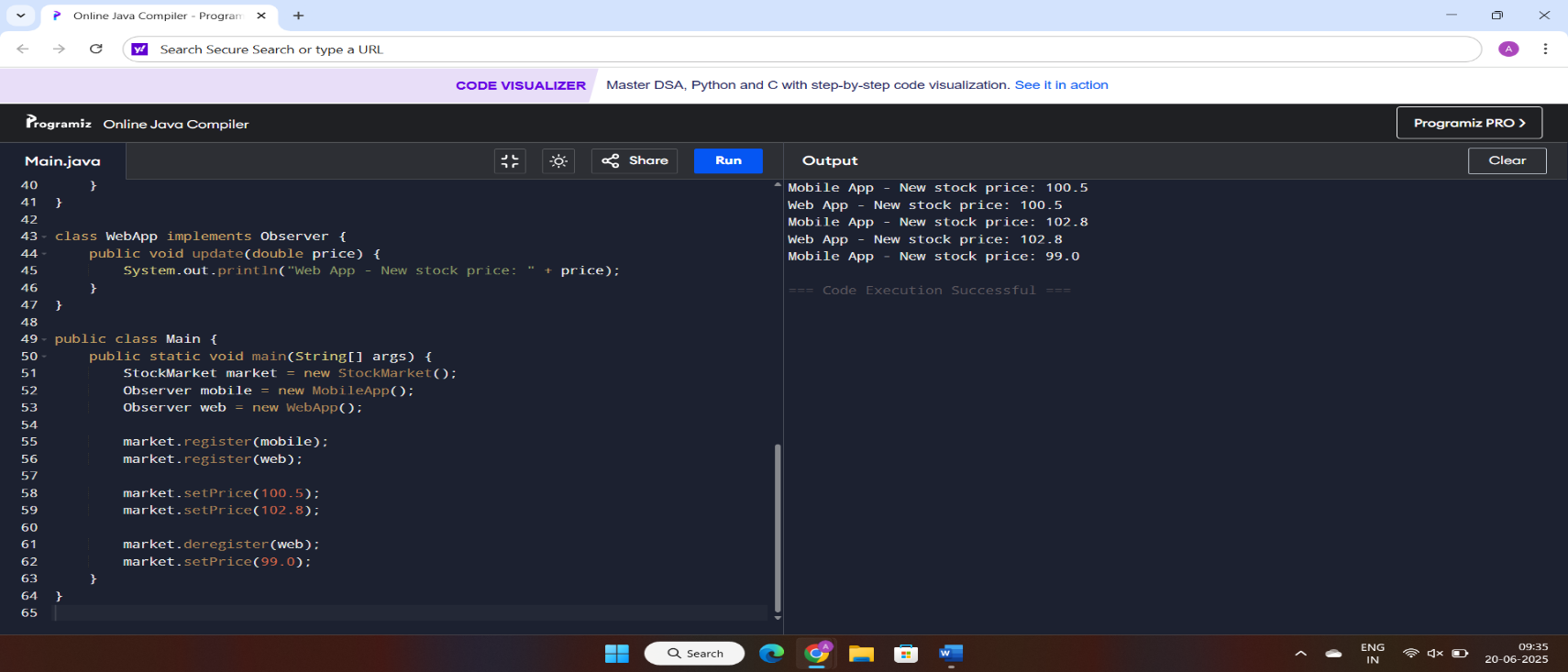
market.setPrice(102.8);

market.deregister(web);

market.setPrice(99.0);

}}

**OUTPUT:**



**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****.*

1. ***Define Strategy Interface:***

* *Create an interface PaymentStrategy with a method* ***pay()****.*

1. ***Implement Concrete Strategies:***

* *Create classes* ***CreditCardPayment****,* ***PayPalPayment*** *that implement* ***PaymentStrategy****.*

1. ***Implement Context Class:***

* *Create a class* ***PaymentContext*** *that holds a reference to* ***PaymentStrategy*** *and a method to execute the strategy.*

1. ***Test the Strategy Implementation:***

* *Create a test class to demonstrate selecting and using different payment strategies.*

**SOLUTION:**

interface PaymentStrategy {

void pay(double amount);}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy; }

public void payAmount(double amount) {

strategy.pay(amount);}}

public class PaymentTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

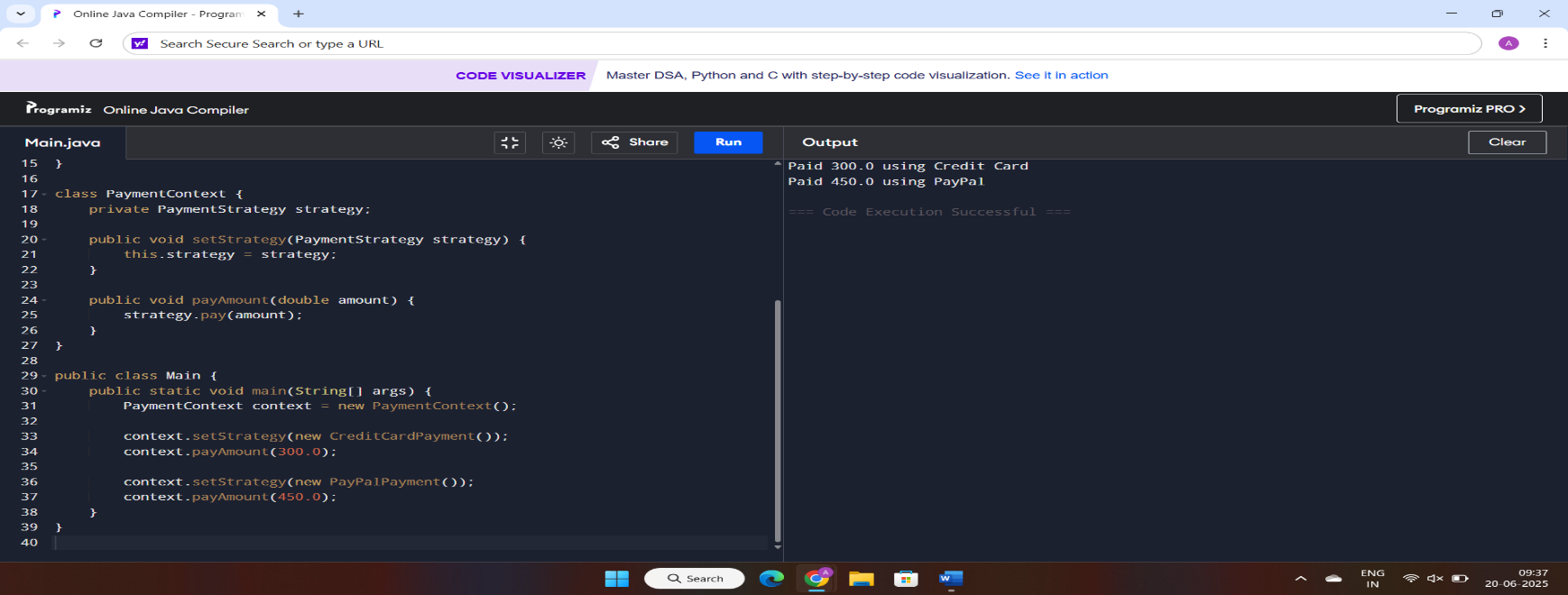
context.setStrategy(new CreditCardPayment());

context.payAmount(300.0);

context.setStrategy(new PayPalPayment());

context.payAmount(450.0);

}}

**OUTPUT:**

**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****.*

1. ***Define Command Interface:***

* *Create an interface Command with a method* ***execute()****.*

1. ***Implement Concrete Commands:***

* *Create classes* ***LightOnCommand****,* ***LightOffCommand*** *that implement Command.*

1. ***Implement Invoker Class:***

* *Create a class* ***RemoteControl*** *that holds a reference to a Command and a method to execute the command.*

1. ***Implement Receiver Class:***

* *Create a class* ***Light*** *with methods to turn on and off.*

1. ***Test the Command Implementation:***

* *Create a test class to demonstrate issuing commands using the* ***RemoteControl****.*

**SOLUTION:**

interface Command {

void execute();

}

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class HomeAutomationTest {

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);

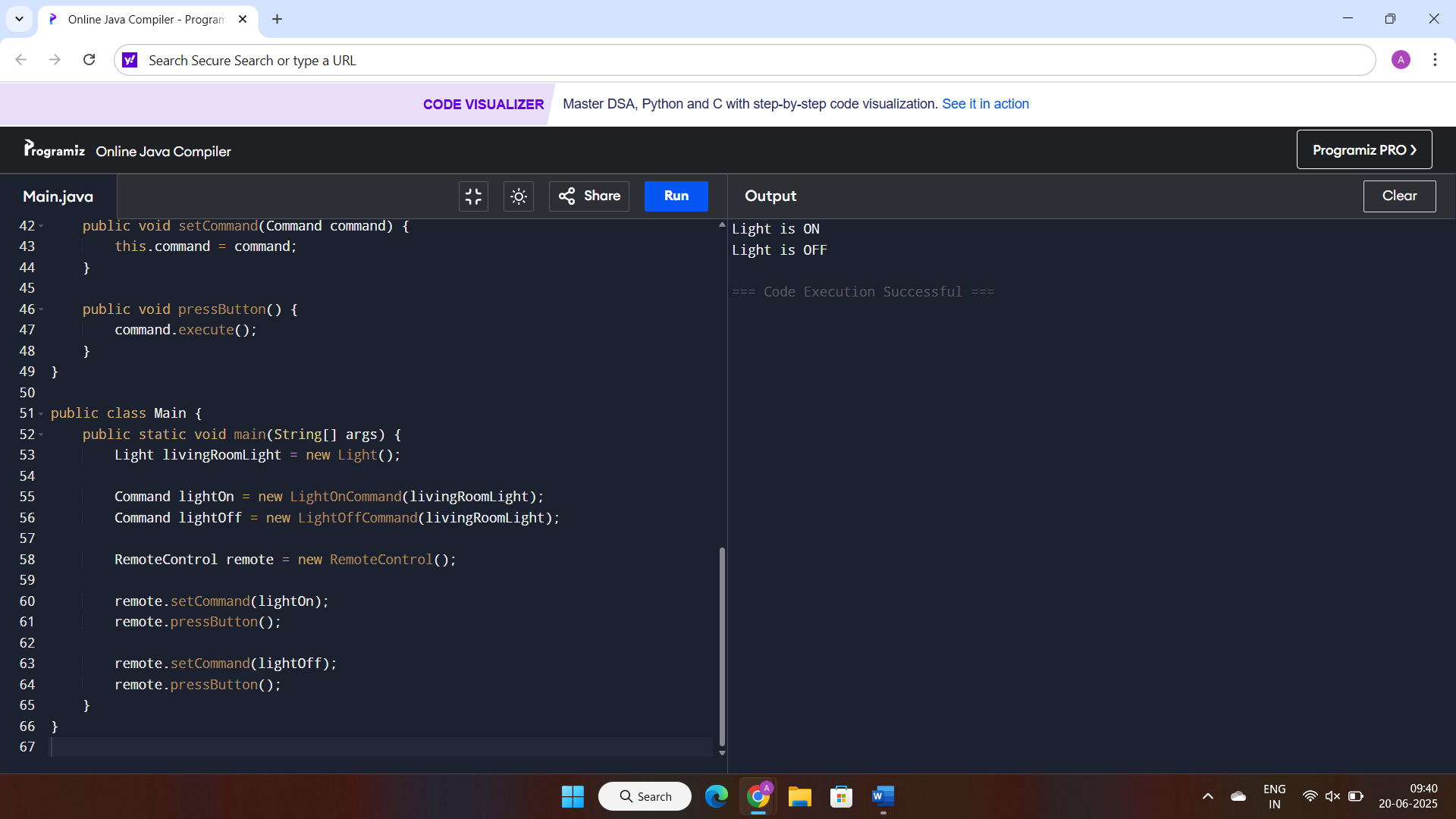
remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT:**

**Exercise 10: Implementing the MVC Pattern**

***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****.*

1. ***Define Model Class:***

* *Create a class* ***Student*** *with attributes like* ***name, id, and grade****.*

1. ***Define View Class:***

* *Create a class* ***StudentView*** *with a method* ***displayStudentDetails()****.*

1. ***Define Controller Class:***

* *Create a class* ***StudentController*** *that handles the communication between the model and the view.*

1. ***Test the MVC Implementation:***

* *Create a main class to demonstrate creating a* ***Student****, updating its details using* ***StudentController****, and displaying them using* ***StudentView****.*

**SOLUTION:**

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;5}

public String getGrade() {

return grade;}

public void setGrade(String grade) {

this.grade = grade;}}

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);}}

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());

}}

public class MVCTest {

public static void main(String[] args) {

Student student = new Student();

student.setName("John");

student.setId("S101");

student.setGrade("A");

StudentView view = new StudentView();

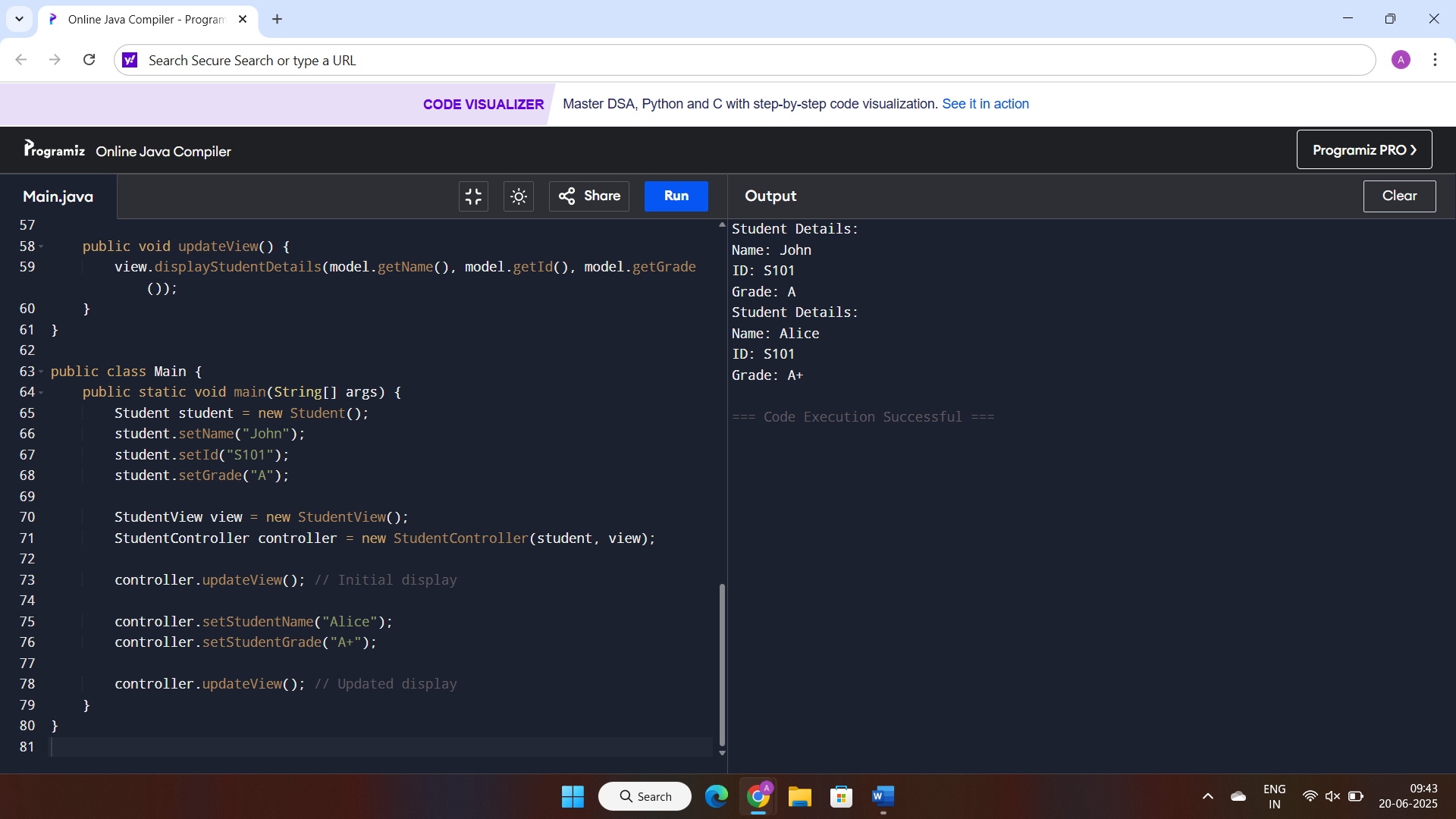
StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Alice");

controller.setStudentGrade("A+");

controller.updateView();}}

**OUTPUT:**

**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****.*

1. ***Define Repository Interface:***

* *Create an interface* ***CustomerRepository*** *with methods like* ***findCustomerById()****.*

1. ***Implement Concrete Repository:***

* *Create a class* ***CustomerRepositoryImpl*** *that implements* ***CustomerRepository****.*

1. ***Define Service Class:***

* *Create a class* ***CustomerService*** *that depends on* ***CustomerRepository****.*

1. ***Implement Dependency Injection:***

* *Use constructor injection to inject* ***CustomerRepository*** *into* ***CustomerService****.*

1. ***Test the Dependency Injection Implementation:***

* *Create a main class to demonstrate creating a* ***CustomerService*** *with* ***CustomerRepositoryImpl*** *and using it to find a customer.*

**SOLUTION:**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer with ID: " + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String result = repository.findCustomerById(id);

System.out.println(result);

}

}

public class CustomerApp {

public static void main(String[] args) {

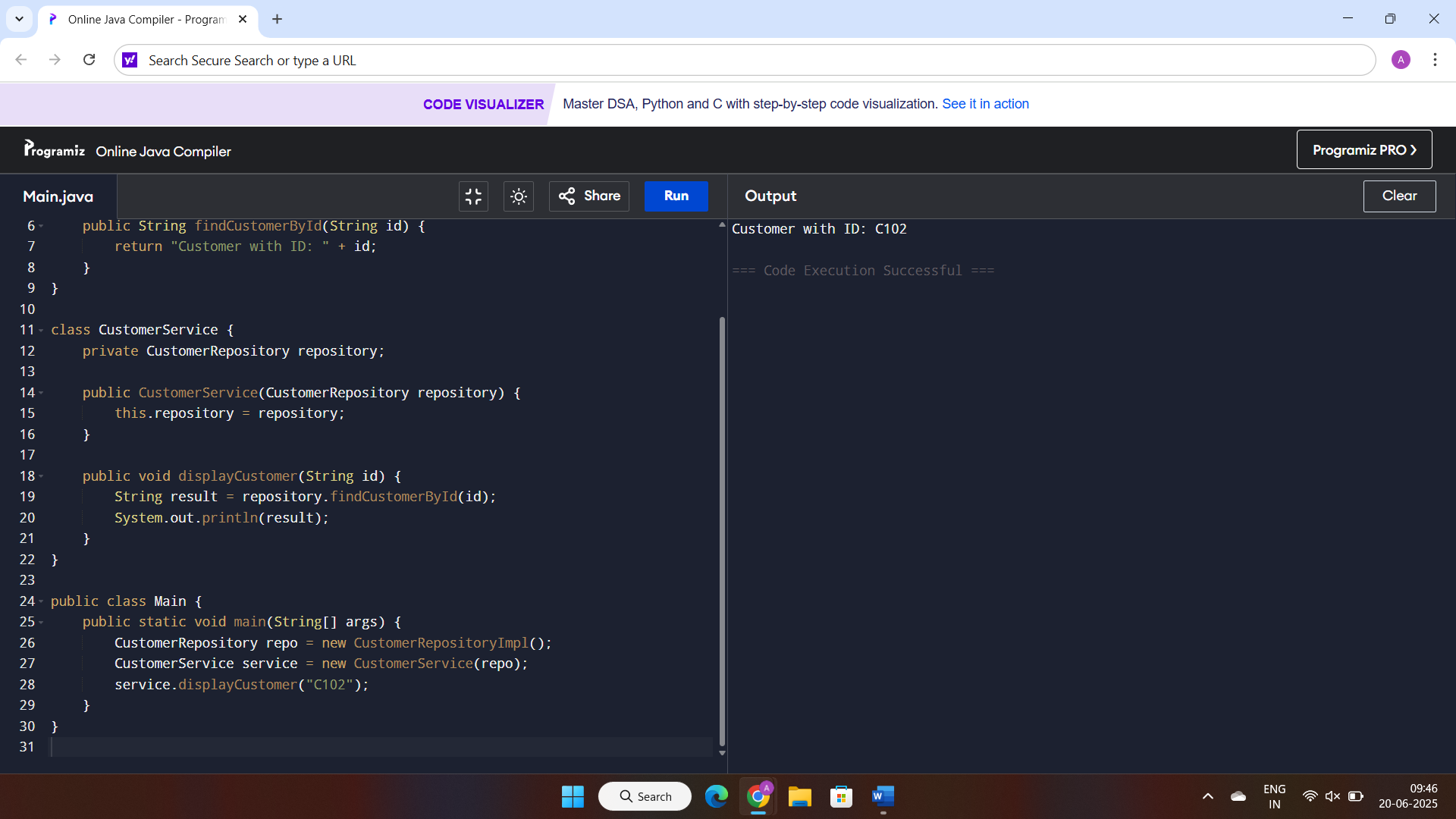
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer("C102");

}

}

**OUTPUT:**