**WEEK – 1 DESIGN PATTERNS AND PRINCIPLES**

**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 C# Project:**
   * Create a new C# project named **SingletonPatternExample**.
2. **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.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**SOLUTION:**

using System;

namespace SingletonPatternExample

{

public class Logger

{

private static Logger \_instance;

private static readonly object \_lock = new object();

private Logger()

{

Console.WriteLine("Logger instance created.");

}

public static Logger GetInstance()

{

if (\_instance == null)

{

lock (\_lock) {

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

public void Log(string message)

{

Console.WriteLine($"[LOG] {message}");

}

}

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

logger1.Log("First log message.");

Logger logger2 = Logger.GetInstance();

logger2.Log("Second log message.");

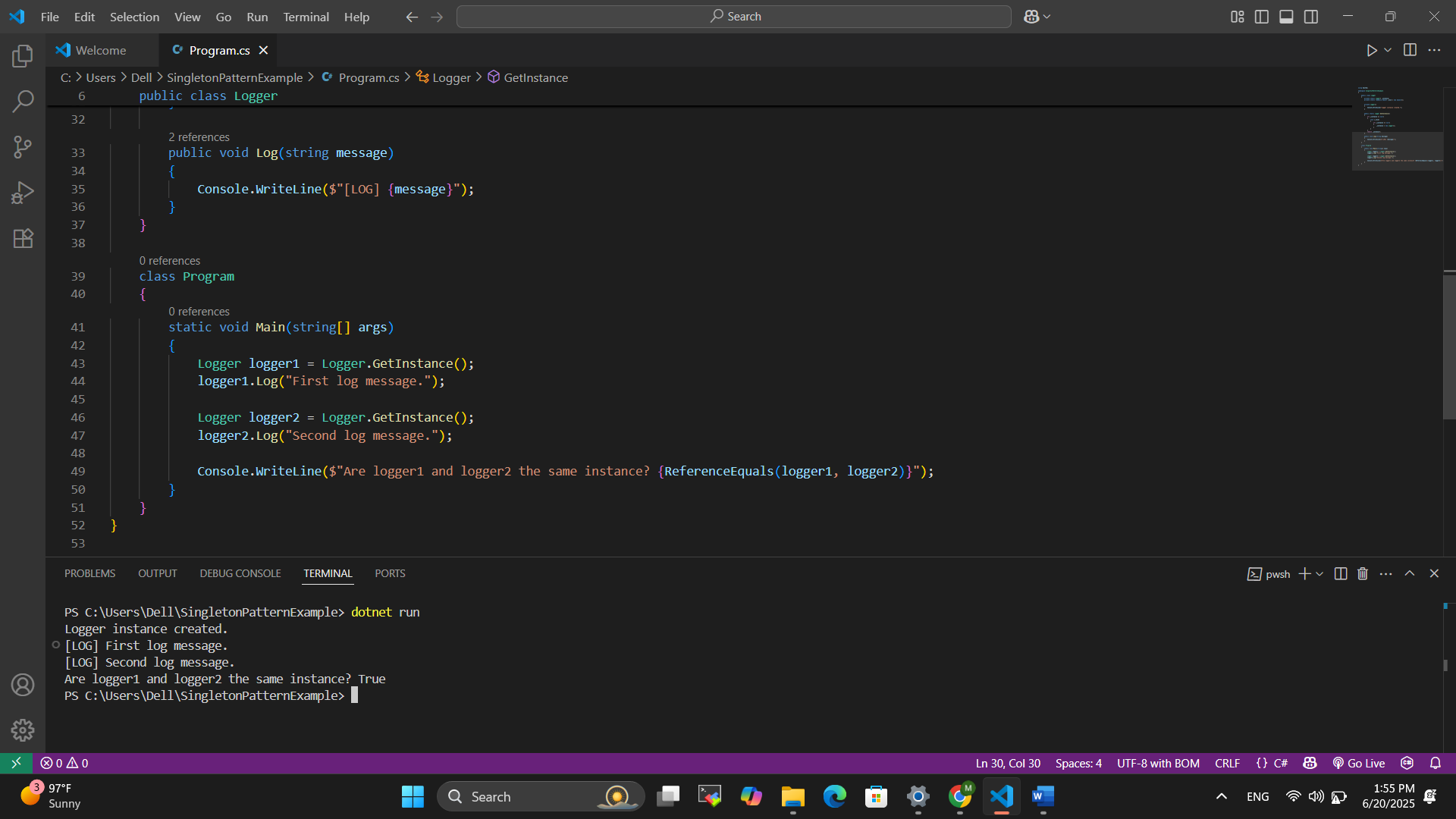
Console.WriteLine($"Are logger1 and logger2 the same instance? {ReferenceEquals(logger1, logger2)}");

}

}

}

**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 C# Project:**
   * Create a new C# project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **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.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**SOLUTION:**

**1. Document:**

public interface IDocument

{

void Open();

}

**2. Word Document:**

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening Word document.");

}

}

**3. Pdf Document:**

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening PDF document.");

}

}

**4. Excel Document:**

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening Excel document.");

}

}

**5. Document Factory:**

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

**6. Word Document Factory:**

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

**7. Pdf Document Factory:**

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

**8. Excel Document Factory:**

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

**9. Program:**

using System;

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

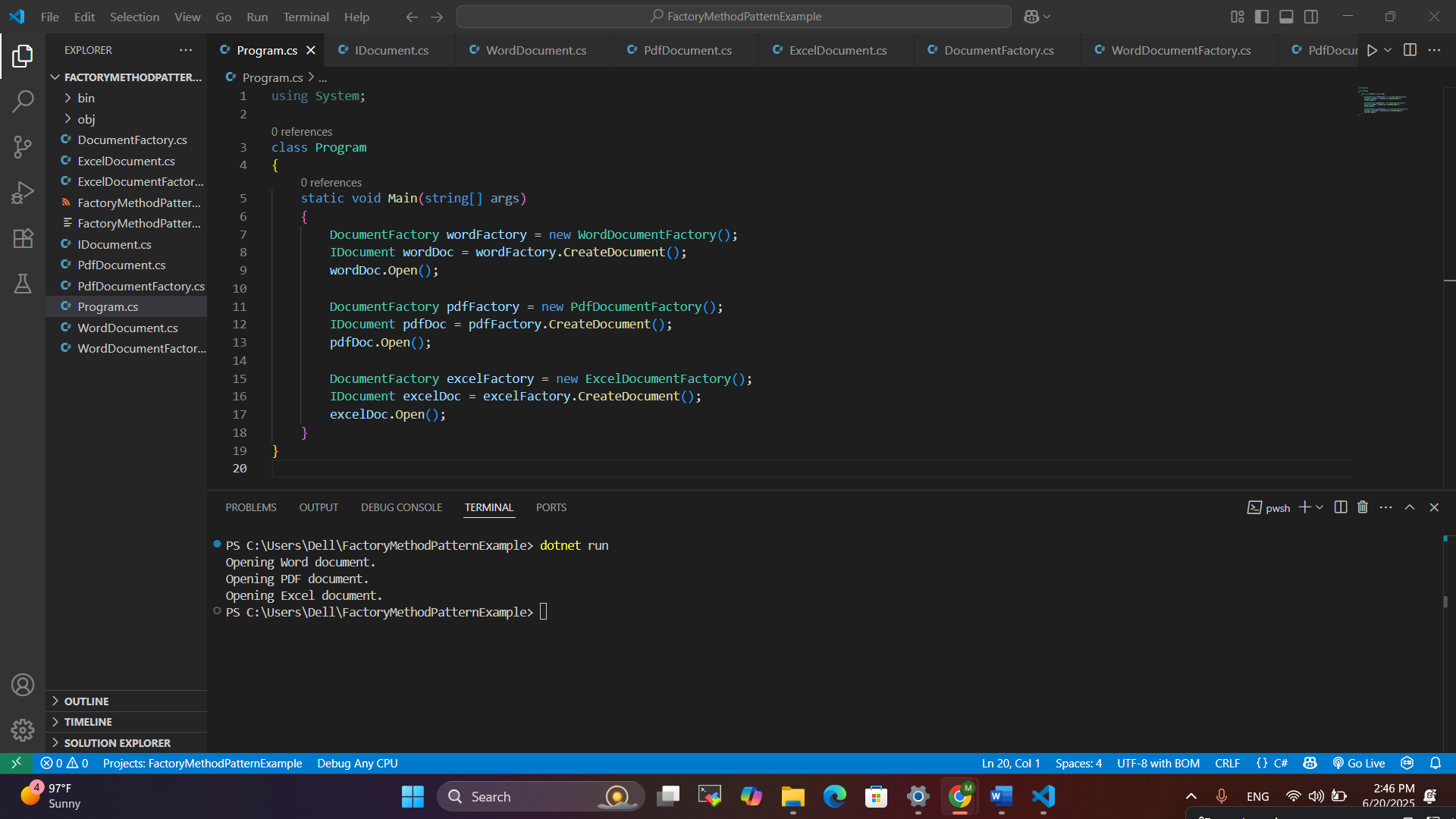
IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.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 C# Project:**
   * Create a new C# project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **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.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**SOLUTION :**

**Product Class:**

public class Computer

{

public string CPU { get; }

public string RAM { get; }

public string Storage { get; }

public string GPU { get; }

private Computer(Builder builder)

{

CPU = builder.CPU;

RAM = builder.RAM;

Storage = builder.Storage;

GPU = builder.GPU;

}

public void DisplayConfig()

{

Console.WriteLine("Computer Configuration:");

Console.WriteLine($"CPU: {CPU}");

Console.WriteLine($"RAM: {RAM}");

Console.WriteLine($"Storage: {Storage}");

Console.WriteLine($"GPU: {GPU}");

}

public class Builder

{

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GPU { get; private set; }

public Builder SetCPU(string cpu)

{

CPU = cpu;

return this;

}

public Builder SetRAM(string ram)

{

RAM = ram;

return this;

}

public Builder SetStorage(string storage)

{

Storage = storage;

return this;

}

public Builder SetGPU(string gpu)

{

GPU = gpu;

return this;

}

public Computer Build()

{

return new Computer(this);

}

}

}

**Test Class :**

using System;

class Program

{

static void Main(string[] args)

{

var gamingComputer = new Computer.Builder()

.SetCPU("Intel Core i9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGPU("NVIDIA RTX 4090")

.Build();

gamingComputer.DisplayConfig();

Console.WriteLine();

var officeComputer = new Computer.Builder()

.SetCPU("Intel Core i5")

.SetRAM("16GB")

.SetStorage("512GB SSD")

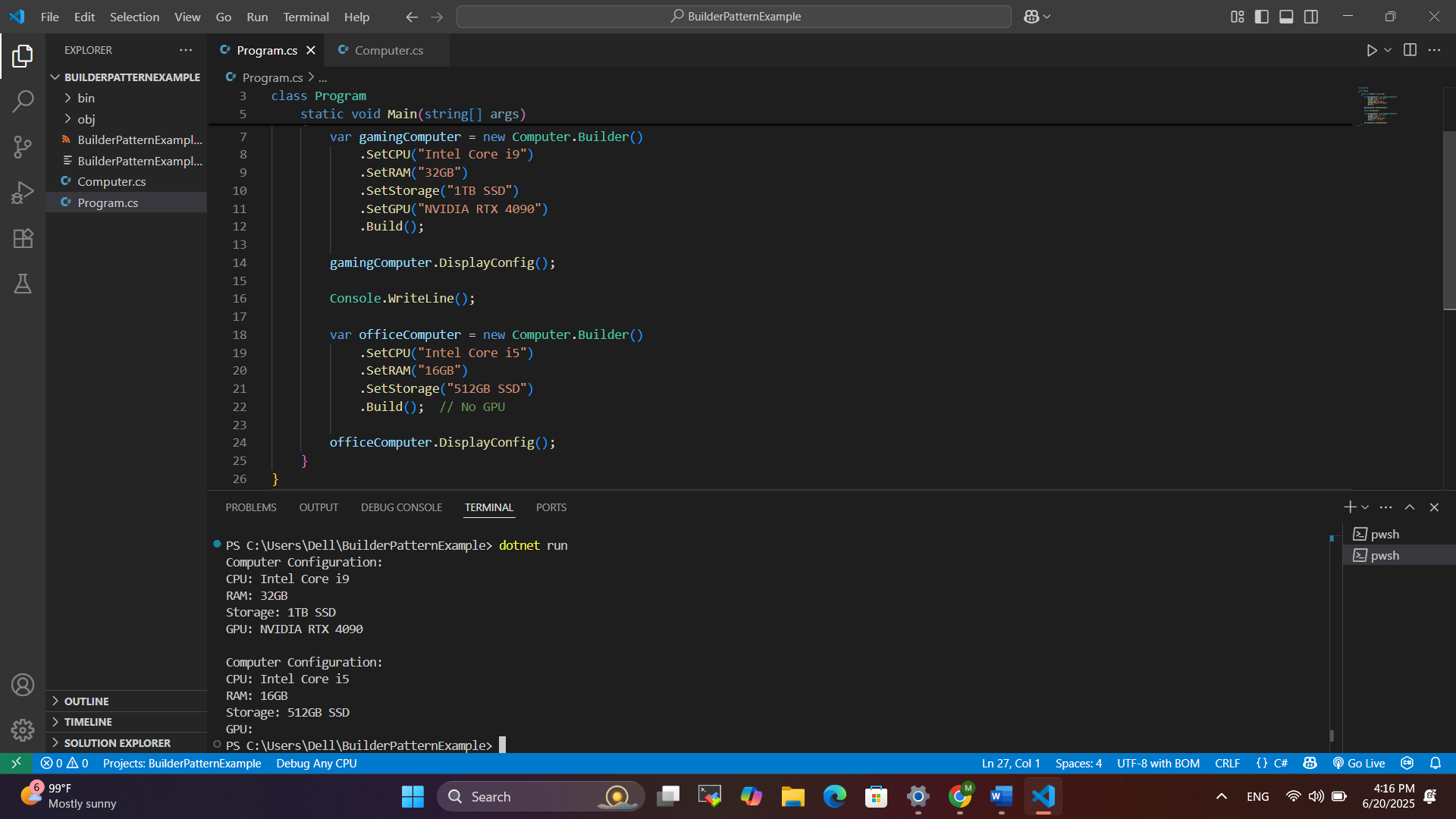
.Build();

officeComputer.DisplayConfig();

}

}

**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 C# Project:**
   * Create a new C# project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

**SOLUTION:**

**1. I Payment Processor**

public interface IPaymentProcessor

{

void ProcessPayment(double amount);

}

**2. PayPal Gateway**

public class PayPalGateway

{

public void MakePayment(double amount)

{

Console.WriteLine($"PayPal: Processing payment of ${amount}");

}

}

**3. Stripe Gateway**

public class StripeGateway

{

public void ExecutePayment(double total)

{

Console.WriteLine($"Stripe: Executing payment of ${total}");

}

}

**4. PayPal Adapter**

public class PayPalAdapter : IPaymentProcessor

{

private readonly PayPalGateway \_paypal = new PayPalGateway();

public void ProcessPayment(double amount)

{

\_paypal.MakePayment(amount);

}

}

**5. Stripe Adapter**

public class StripeAdapter : IPaymentProcessor

{

private readonly StripeGateway \_stripe = new StripeGateway();

public void ProcessPayment(double amount)

{

\_stripe.ExecutePayment(amount);

}

}

**6. Program**

using System;

class Program

{

static void Main(string[] args)

{

IPaymentProcessor paypal = new PayPalAdapter();

paypal.ProcessPayment(150.0);

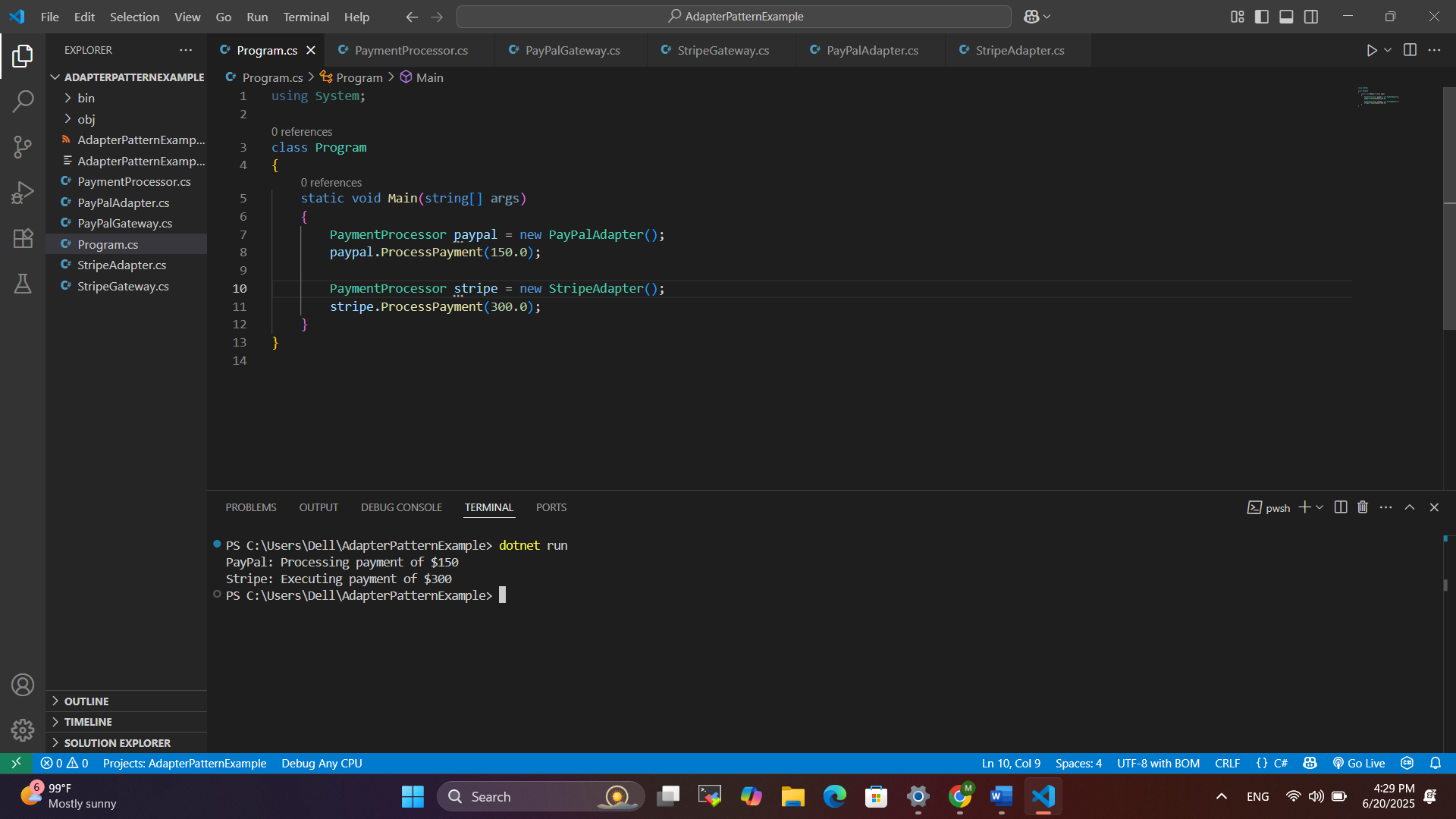
IPaymentProcessor stripe = new StripeAdapter();

stripe.ProcessPayment(300.0);

}

}

**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 C# Project:**
   * Create a new C# project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **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**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**SOLUTION :**

**1. Component Interface :**public interface INotifier

{

void Send(string message);

}

**2. Concrete Component :**

using System;

public class EmailNotifier : INotifier

{

public void Send(string message)

{

Console.WriteLine($"Sending Email: {message}");

}

}

**3. Base Decorator :**

public abstract class NotifierDecorator : INotifier

{

protected INotifier \_wrappee;

public NotifierDecorator(INotifier notifier)

{

\_wrappee = notifier;

}

public virtual void Send(string message)

{

\_wrappee.Send(message);

}

}

**4. Concrete Decorator :**

using System;

public class SMSNotifierDecorator : NotifierDecorator

{

public SMSNotifierDecorator(INotifier notifier) : base(notifier) { }

public override void Send(string message)

{

base.Send(message);

Console.WriteLine($"Sending SMS: {message}");

}

}

**5. Another Concrete Decorator :**

using System;

public class SlackNotifierDecorator : NotifierDecorator

{

public SlackNotifierDecorator(INotifier notifier) : base(notifier) { }

public override void Send(string message)

{

base.Send(message);

Console.WriteLine($"Sending Slack message: {message}");

}

}

**7. Test of Decorator Pattern :**

using System;

class Program

{

static void Main(string[] args)

{

INotifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

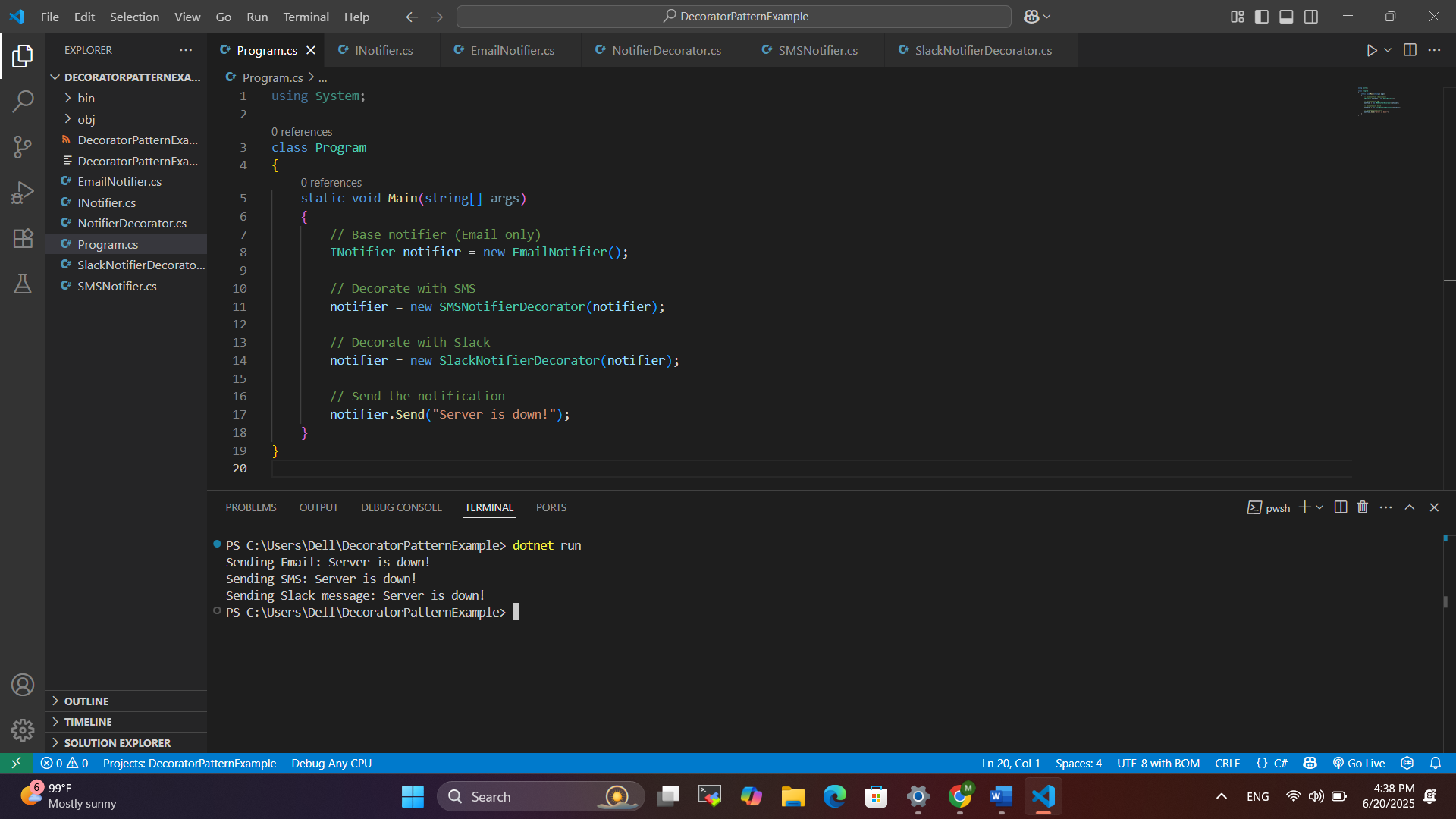
notifier = new SlackNotifierDecorator(notifier);

notifier.Send("Server is down!");

}

}

**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 C# Project:**
   * Create a new C# project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**SOLUTION :**

**1. Subject Interface:**

public interface Image {

void display();

}

**2. Real Subject :**

using System;

public class RealImage : IImage

{

private string \_fileName;

public RealImage(string fileName)

{

\_fileName = fileName;

LoadFromDisk();

}

private void LoadFromDisk()

{

Console.WriteLine($"Loading image from disk: {\_fileName}");

}

public void Display()

{

Console.WriteLine($"Displaying image: {\_fileName}");

}

}

**3. Proxy Class :**

public class ProxyImage : IImage

{

private RealImage? \_realImage;

private string \_fileName;

public ProxyImage(string fileName)

{

\_fileName = fileName;

}

public void Display()

{

if (\_realImage == null)

{

\_realImage = new RealImage(\_fileName);

}

\_realImage.Display();

}

}

**4. Test Case :**

using System;

class Program

{

static void Main(string[] args)

{

IImage image1 = new ProxyImage("image1.jpg");

IImage image2 = new ProxyImage("image2.png");

Console.WriteLine("First time displaying image1:");

image1.Display();

Console.WriteLine("\nSecond time displaying image1:");

image1.Display();

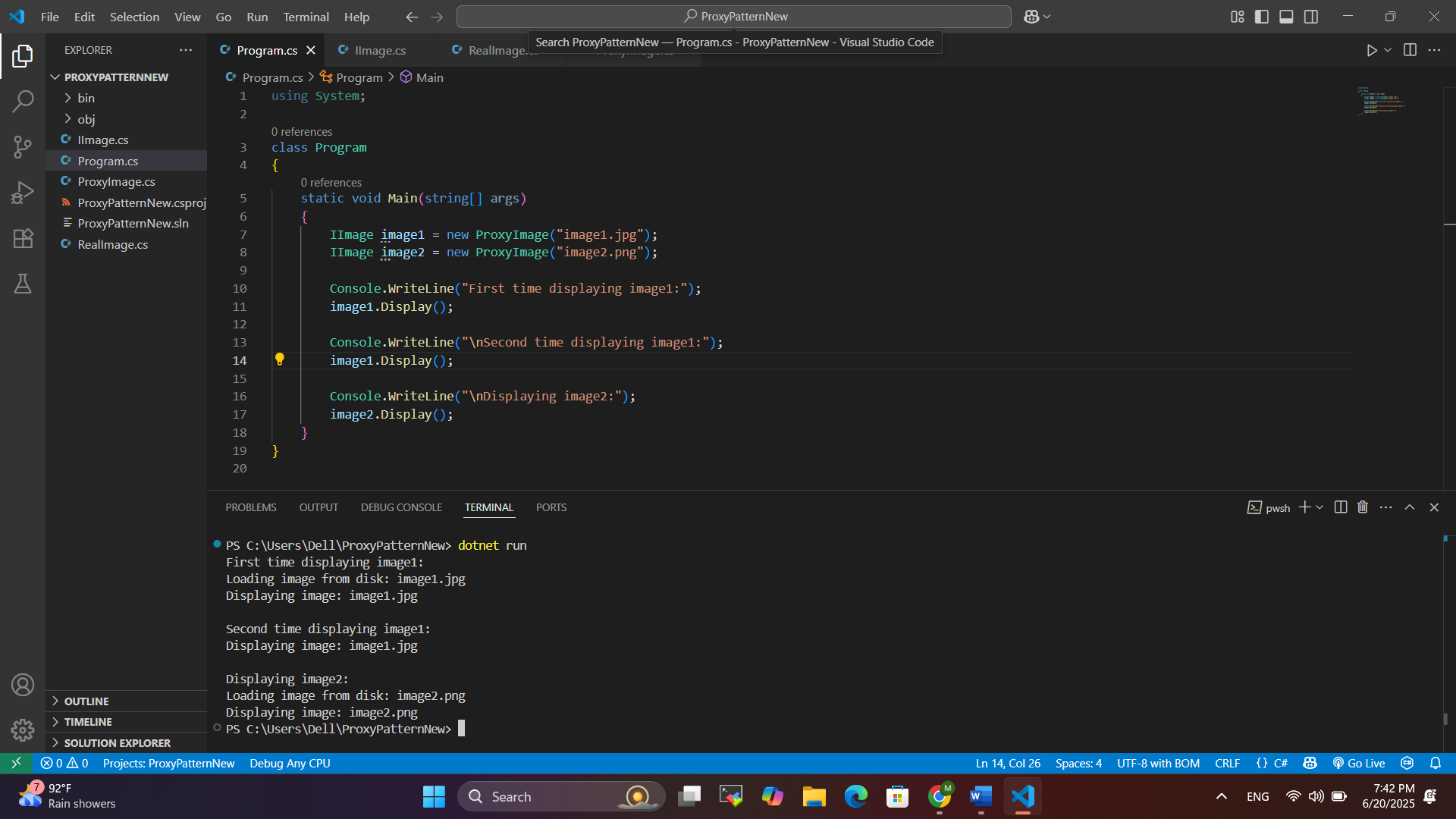
Console.WriteLine("\nDisplaying image2:");

image2.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 C# Project:**
   * Create a new C# project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**SOLUTION :**

**1. Subject Interace :**

public interface IStock

{

void RegisterObserver(IObserver observer);

void RemoveObserver(IObserver observer);

void NotifyObservers();

}

**2. Concrete Subject :**

using System;

using System.Collections.Generic;

public class StockMarket : IStock

{

private List<IObserver> \_observers = new List<IObserver>();

private double \_stockPrice;

public void SetStockPrice(double price)

{

\_stockPrice = price;

Console.WriteLine($"\nStock price updated to: ₹{price}");

NotifyObservers();

}

public void RegisterObserver(IObserver observer)

{

\_observers.Add(observer);

}

public void RemoveObserver(IObserver observer)

{

\_observers.Remove(observer);

}

public void NotifyObservers()

{

foreach (var observer in \_observers)

{

observer.Update(\_stockPrice);

}

}

}

**3.Observer Interface :**

public interface IObserver

{

void Update(double price);

}

**4.Concrete Observer :**

using System;

public class MobileApp : IObserver

{

public void Update(double price)

{

Console.WriteLine($"[Mobile App] New stock price: ₹{price}");

}

}

**5. Another Concrete Observer:**using System;

public class WebApp : IObserver

{

public void Update(double price)

{

Console.WriteLine($"[Web App] New stock price: ₹{price}");

}

}

**6. Test Case :**

using System;

class Program

{

static void Main(string[] args)

{

StockMarket stockMarket = new StockMarket();

IObserver mobileApp = new MobileApp();

IObserver webApp = new WebApp();

stockMarket.RegisterObserver(mobileApp);

stockMarket.RegisterObserver(webApp);

stockMarket.SetStockPrice(120.50);

stockMarket.SetStockPrice(125.75);

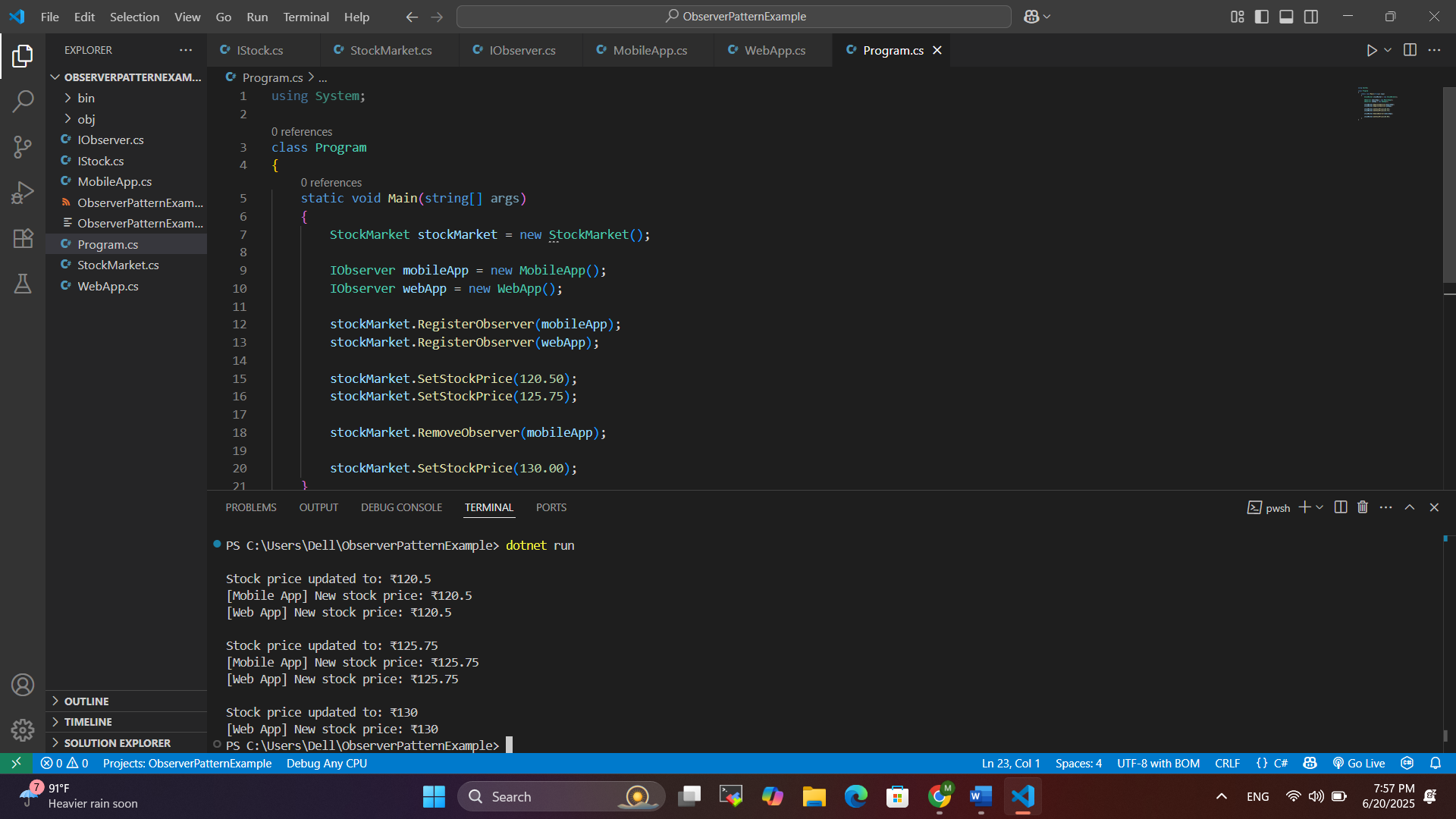
stockMarket.RemoveObserver(mobileApp);

stockMarket.SetStockPrice(130.00);

}

}

**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 C# Project:**
   * Create a new C# project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

**SOLUTION :**

**1. Strategy Interface :**

public interface IPaymentStrategy

{

void Pay(double amount);

}

**2. Concrete Strategies :**

using System;

public class CreditCardPayment : IPaymentStrategy

{

public void Pay(double amount)

{

Console.WriteLine($"Paid ₹{amount} using Credit Card.");

}

}

**3. Another Concrete Strategy:**

using System;

public class PayPalPayment : IPaymentStrategy

{

public void Pay(double amount)

{

Console.WriteLine($"Paid ₹{amount} using PayPal.");

}

}

**4. Context Class :**

public class PaymentContext

{

private IPaymentStrategy? \_strategy;

public void SetStrategy(IPaymentStrategy strategy)

{

\_strategy = strategy;

}

public void ExecutePayment(double amount)

{

if (\_strategy == null)

{

Console.WriteLine("Payment strategy not set.");

}

else

{

\_strategy.Pay(amount);

}

}

}

**5. Test Case :**

using System;

class Program

{

static void Main(string[] args)

{

PaymentContext context = new PaymentContext();

context.SetStrategy(new CreditCardPayment());

context.ExecutePayment(500.0);

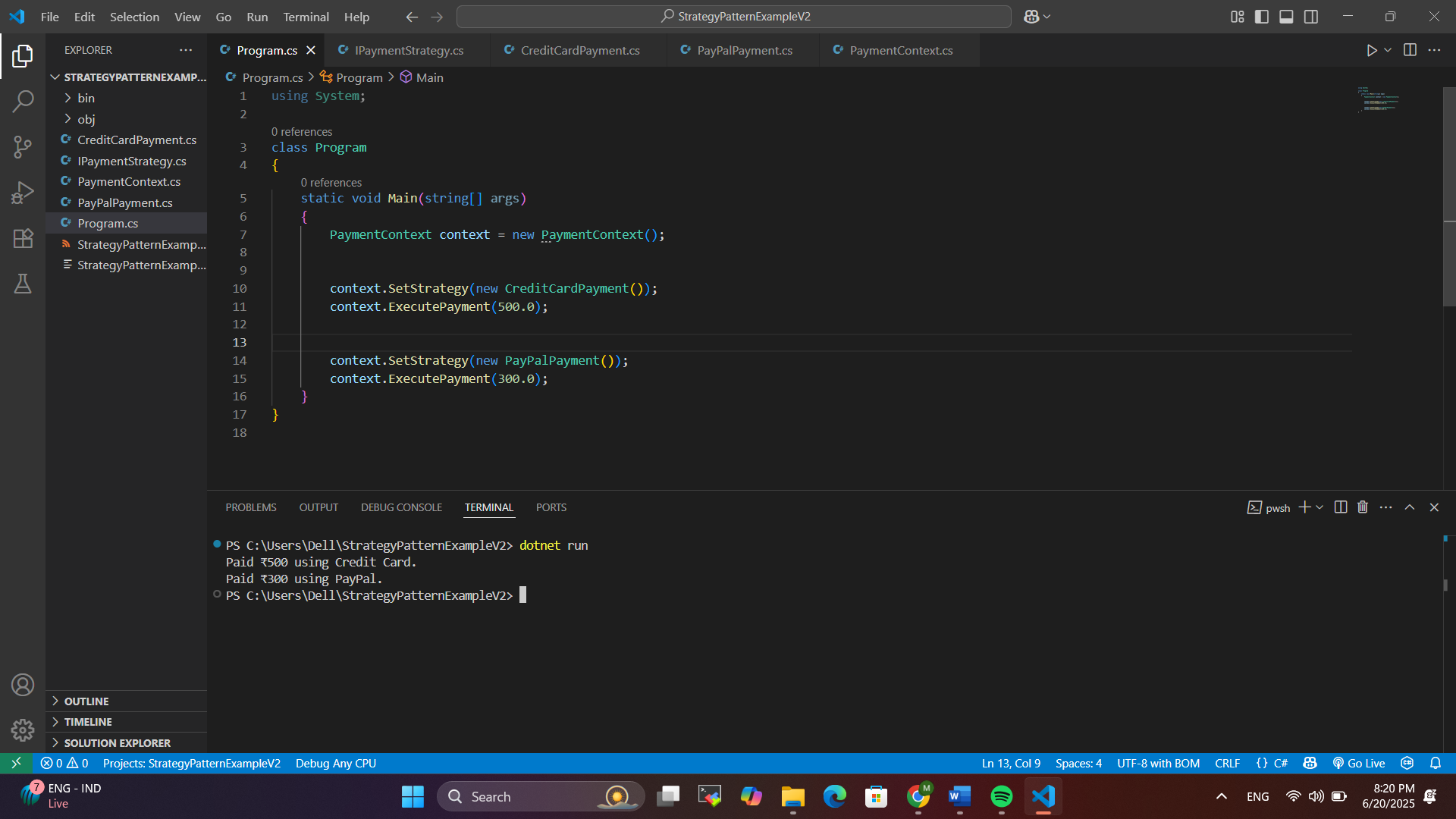
context.SetStrategy(new PayPalPayment());

context.ExecutePayment(300.0);

}

}

**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 C# Project:**
   * Create a new C# project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**SOLUTION :**

**1. Command Interface :**

public interface ICommand

{

void Execute();

}

**2. Receiver Class :**

using System;

public class Light

{

public void TurnOn()

{

Console.WriteLine("The light is ON.");

}

public void TurnOff()

{

Console.WriteLine("The light is OFF.");

}

}

**3. Concrete Commands :**

public class LightOnCommand : ICommand

{

private Light \_light;

public LightOnCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOn();

}

}

**4. Concrete Command:**

public class LightOffCommand : ICommand

{

private Light \_light;

public LightOffCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOff();

}

}

**5. Invoker :**

public class RemoteControl

{

private ICommand? \_command;

public void SetCommand(ICommand command)

{

\_command = command;

}

public void PressButton()

{

if (\_command != null)

{

\_command.Execute();

}

else

{

Console.WriteLine("No command set.");

}

}

}

**6. Test Command Pattern :**

using System;

class Program

{

static void Main(string[] args)

{

Light light = new Light();

ICommand lightOn = new LightOnCommand(light);

ICommand lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

Console.WriteLine("Turning light ON:");

remote.SetCommand(lightOn);

remote.PressButton();

Console.WriteLine("\nTurning light OFF:");

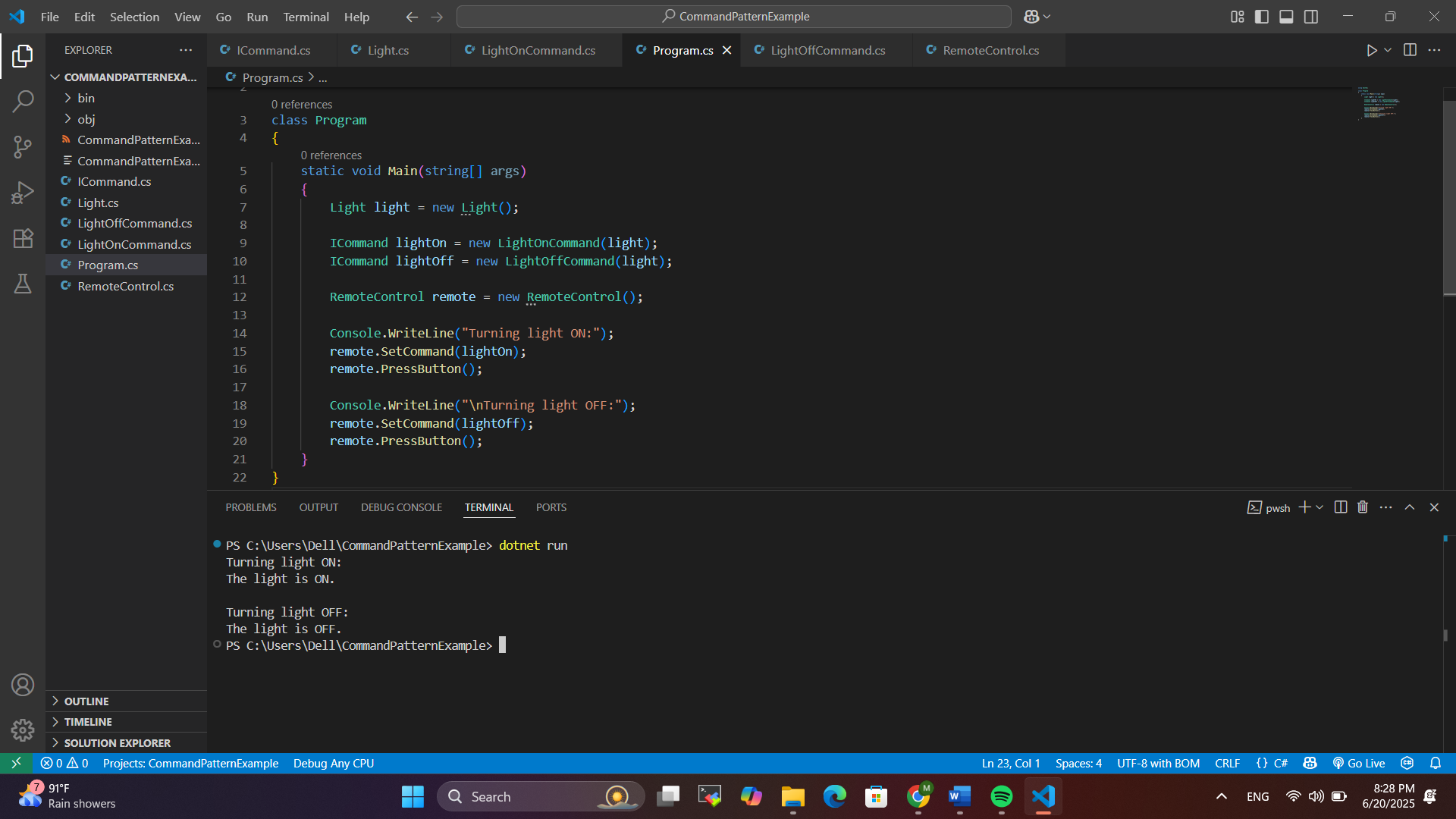
remote.SetCommand(lightOff);

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 C# Project:**
   * Create a new C# project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**SOLUTION :**

**1. Model Class :**

public class Student

{

public string Name { get; set; }

public string Id { get; set; }

public string Grade { get; set; }

}

**2. View Class :**

using System;

public class StudentView

{

public void DisplayStudentDetails(string name, string id, string grade)

{

Console.WriteLine("Student Details:");

Console.WriteLine($"Name : {name}");

Console.WriteLine($"ID : {id}");

Console.WriteLine($"Grade : {grade}");

}

}

**3. Controller Class :**

public class StudentController

{

private Student \_student;

private StudentView \_view;

public StudentController(Student student, StudentView view)

{

\_student = student;

\_view = view;

}

public void SetStudentName(string name) => \_student.Name = name;

public void SetStudentId(string id) => \_student.Id = id;

public void SetStudentGrade(string grade) => \_student.Grade = grade;

public string GetStudentName() => \_student.Name;

public string GetStudentId() => \_student.Id;

public string GetStudentGrade() => \_student.Grade;

public void UpdateView()

{

\_view.DisplayStudentDetails(\_student.Name, \_student.Id, \_student.Grade);

}

}

**4. Main Class :**

using System;

class Program

{

static void Main(string[] args)

{

Student student = new Student("Rajesh", "S1234", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.UpdateView();

controller.SetStudentName("Rajesh");

controller.SetStudentGrade("A+");

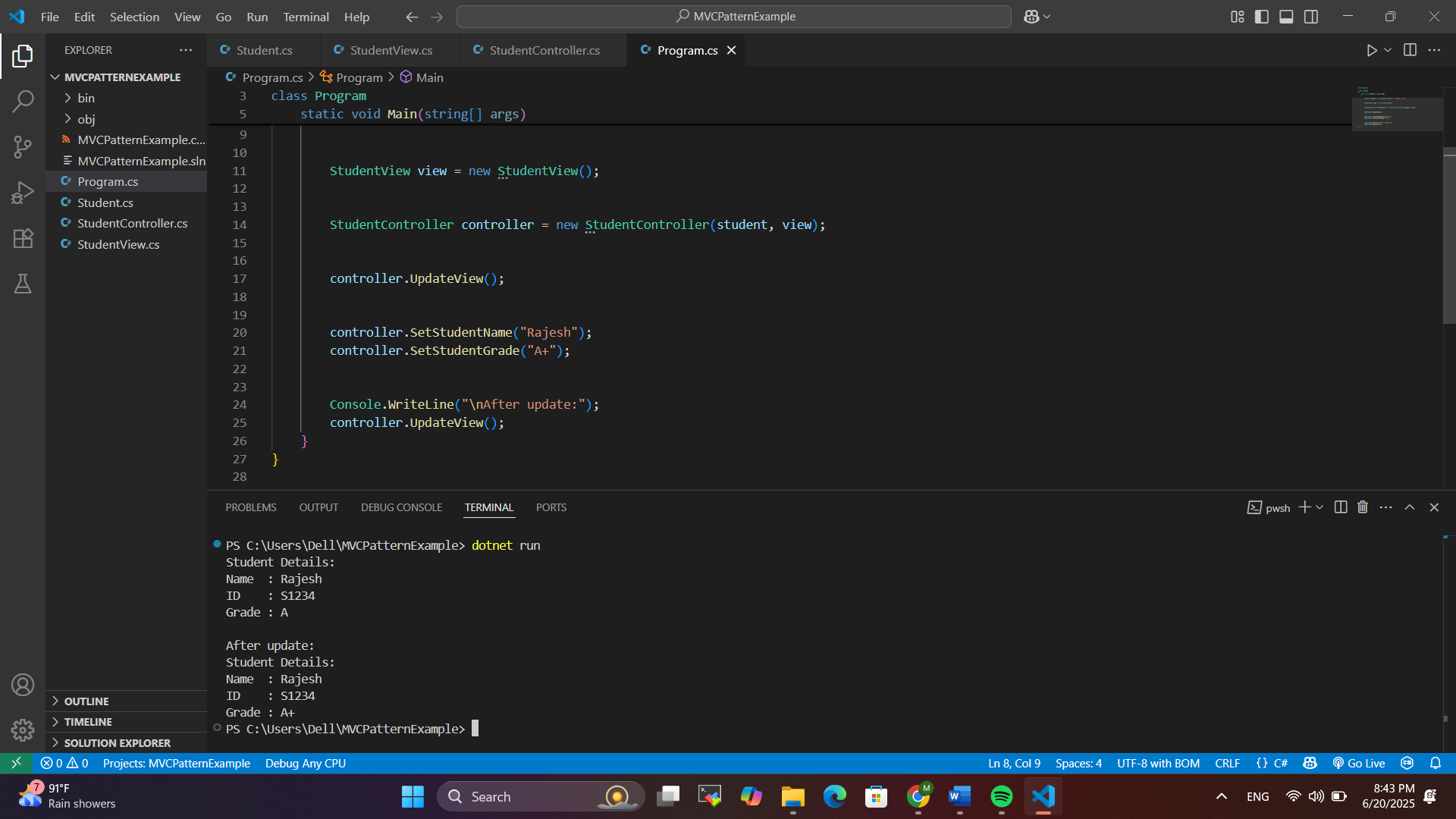
Console.WriteLine("\nAfter update:");

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 C# Project:**
   * Create a new C# project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**SOLUTION :**

**1. Model :**

public class Customer

{

public int Id { get; set; }

public string Name { get; set; }

public Customer(int id, string name)

{

Id = id;

Name = name;

}

}

**2. Interface**

public interface ICustomerRepository

{

Customer? FindCustomerById(int id);

}

**3. Concrete Repository :**

using System.Collections.Generic;

public class CustomerRepositoryImpl : ICustomerRepository

{

private readonly Dictionary<int, Customer> \_customers = new()

{

{ 1, new Customer(1, "Alice") },

{ 2, new Customer(2, "Bob") },

{ 3, new Customer(3, "Charlie") }

};

public Customer? FindCustomerById(int id)

{

\_customers.TryGetValue(id, out Customer? customer);

return customer;

}

}

**4. Service With Constructor Injection :**

using System;

public class CustomerService

{

private readonly ICustomerRepository \_repository;

public CustomerService(ICustomerRepository repository)

{

\_repository = repository;

}

public void PrintCustomerDetails(int id)

{

var customer = \_repository.FindCustomerById(id);

if (customer != null)

{

Console.WriteLine($"Customer Found: ID = {customer.Id}, Name = {customer.Name}");

}

else

{

Console.WriteLine("Customer not found.");

}

}

}

**5. Main Class :**

class Program

{

static void Main(string[] args)

{

ICustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.PrintCustomerDetails(1); // Should print Alice

service.PrintCustomerDetails(4); // Not found

}

}

**OUTPUT :**

