**WEEK 1**

**Design Patterns and Principles**

**Name: Rajesh A  
Superset ID: 6384182**

**Exercise 1: Implementing the Singleton Pattern (.NET)**

**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.

### **What I Learned:**

I learned how to implement the Singleton Design Pattern in C#. This pattern ensures that only one instance of a class is created throughout the application's lifecycle. I also understood how to make it thread-safe using the lock keyword. The main goal was to build a global logging utility that maintains consistent logging behavior.

**C# Code:**

using System;

public class Logger

{

private static Logger? instance;

private static readonly object lockObj = new object();

private Logger() { }

public static Logger GetInstance()

{

if (instance == null)

{

lock (lockObj)

{

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();

Logger logger2 = Logger.GetInstance();

logger1.Log("First log message");

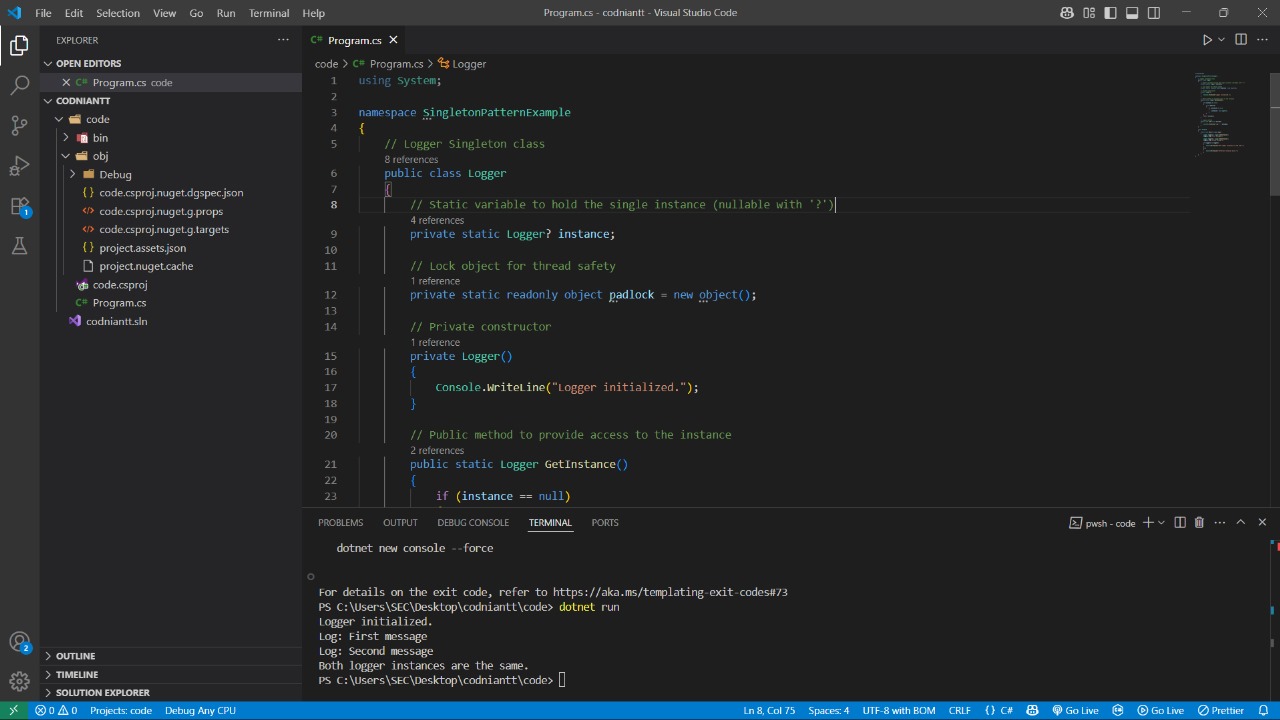
logger2.Log("Second log message");

Console.WriteLine("Are both loggers the same? " + (logger1 == logger2));

}

}

**Output:-**



### Conclusion:

The Singleton pattern was successfully used to create a single shared instance of the Logger class. This ensured that all logging operations were consistent across the program. The use of the lock mechanism provided thread safety, making the implementation safe for multi-threaded applications. This pattern is useful in scenarios where only one instance of a class should exist, such as logging or configuration management.

**Exercise 2: Implementing the Factory Method Pattern (.NET)**

**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.

### What I Learned:

I learned how to implement the Factory Method Design Pattern in C#. This pattern provides a way to delegate the instantiation logic to subclasses, allowing the creation of objects without specifying the exact class. It is useful for building scalable and extensible applications where object creation is dependent on runtime conditions.

**# CODE:**

using System;

public interface IDocument

{

void Open();

}

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a Word Document.");

}

}

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a PDF Document.");

}

}

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening an Excel Document.");

}

}

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

public class WordFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

public class PdfFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

public class ExcelFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordFactory();

IDocument word = wordFactory.CreateDocument();

word.Open();

DocumentFactory pdfFactory = new PdfFactory();

IDocument pdf = pdfFactory.CreateDocument();

pdf.Open();

DocumentFactory excelFactory = new ExcelFactory();

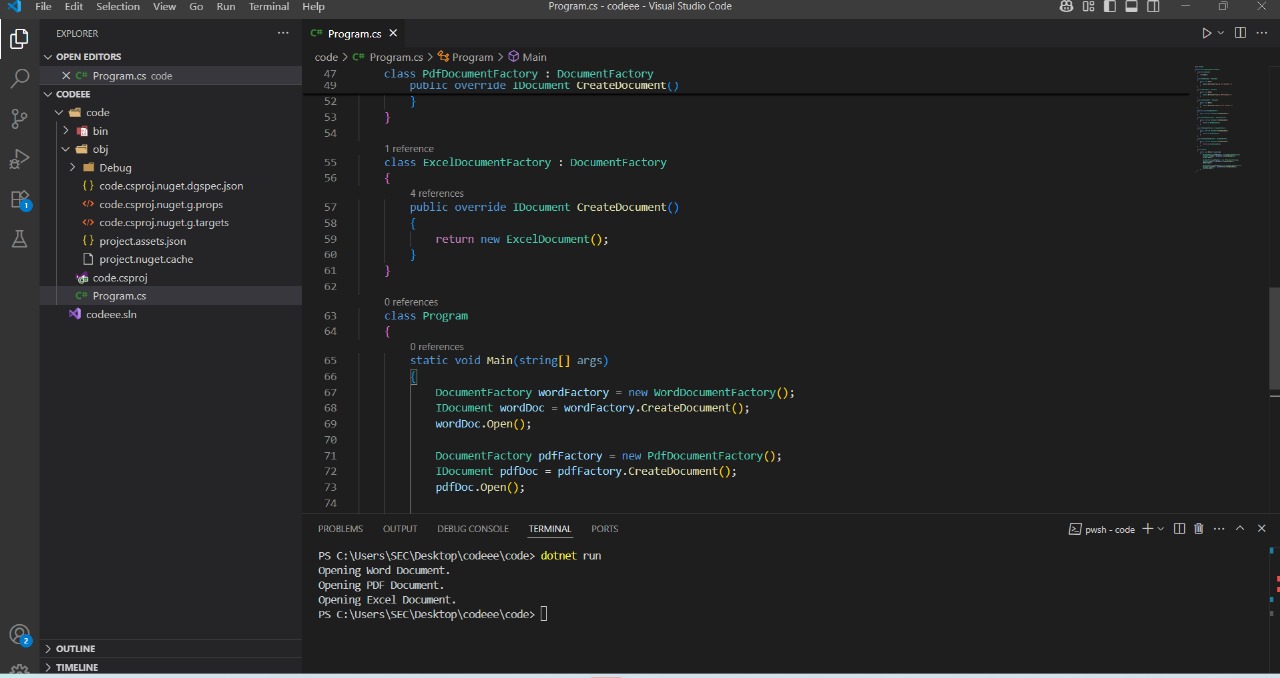
IDocument excel = excelFactory.CreateDocument();

excel.Open();

}

}

**OUTPUT:**



### Conclusion:

The Factory Method Pattern was successfully implemented to create different types of documents (Word, PDF, Excel) using dedicated factory classes. Each factory returns an object of its corresponding document type, making the code extensible and easy to maintain. This pattern helps decouple the object creation logic from the client, adhering to the principles of object-oriented design.

**Exercise 3: Implementing the Builder Pattern (.NET)**

**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.

### What I Learned

I learned how to implement the Builder Design Pattern in C#. This pattern helps construct complex objects step by step using a builder class. It allows setting only the required properties and provides flexibility in object creation without the need for multiple constructors. It is useful when creating objects with many optional parameters.

**CODE:**

using System;

public class Computer

{

public string CPU { get; }

public string RAM { get; }

public string Storage { get; }

public string GraphicsCard { get; }

private Computer(Builder builder)

{

CPU = builder.CPU;

RAM = builder.RAM;

Storage = builder.Storage;

GraphicsCard = builder.GraphicsCard;

}

public void ShowConfig()

{

Console.WriteLine("Computer Configuration:");

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

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

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

Console.WriteLine($"Graphics Card: {GraphicsCard}");

}

public class Builder

{

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GraphicsCard { 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 SetGraphicsCard(string graphicsCard)

{

GraphicsCard = graphicsCard;

return this;

}

public Computer Build()

{

return new Computer(this);

}

}

}

class Program

{

static void Main(string[] args)

{

Computer gamingPC = new Computer.Builder()

.SetCPU("Intel Core i9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGraphicsCard("NVIDIA RTX 4080")

.Build();

gamingPC.ShowConfig();

Console.WriteLine();

Computer officePC = new Computer.Builder()

.SetCPU("Intel Core i5")

.SetRAM("8GB")

.SetStorage("512GB SSD")

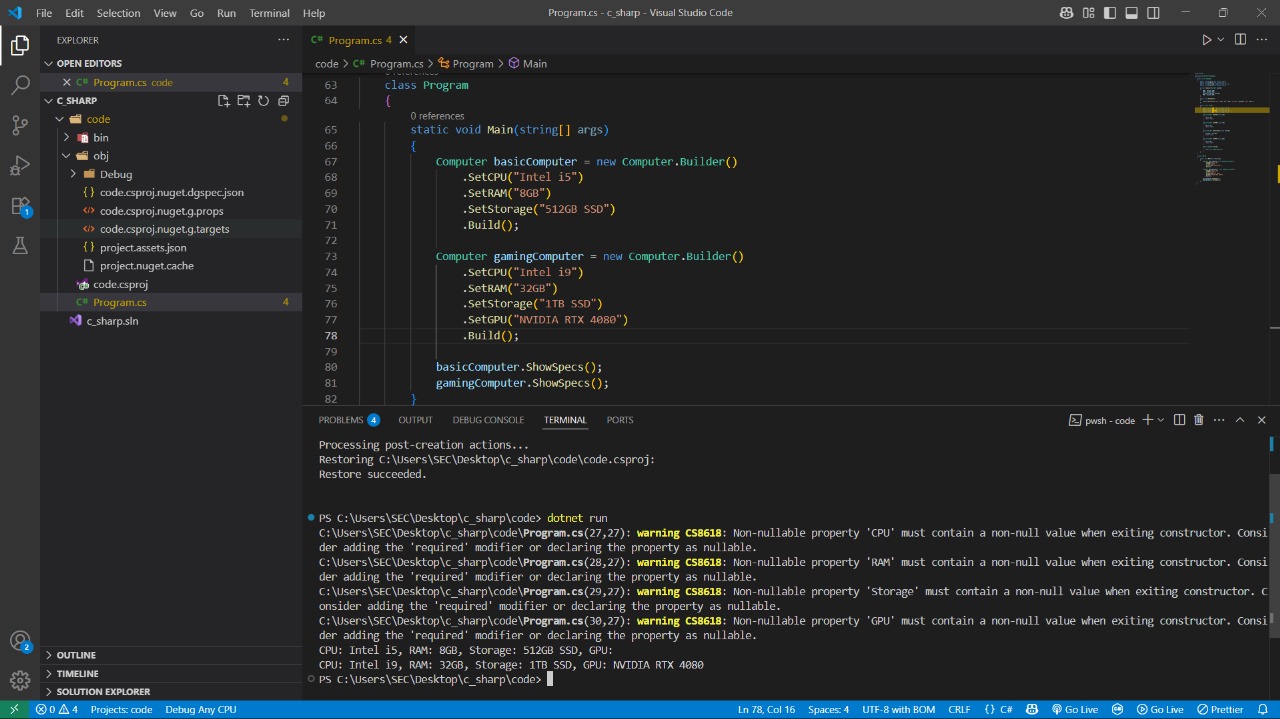
.Build();

officePC.ShowConfig();

}

}

**OUTPUT:**



### Conclusion:

The Builder pattern was implemented successfully to construct different configurations of a Computer object. This approach simplifies object creation by allowing the client to configure only the necessary properties, improving code readability and maintainability. It avoids the need for complex constructors and supports optional parameters effectively.

**Exercise 4: Implementing the Adapter Pattern(.NET)**

**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.

### What I Learned:

I learned how to implement the Adapter Design Pattern in C#. This pattern is used to bridge the gap between incompatible interfaces. It allows an existing class to be used as another interface. In this example, different third-party payment gateways are adapted to a common interface so that the client code can process payments uniformly without depending on specific gateway implementations.

**CODE:**

using System;

namespace AdapterPatternExample

{

public interface IPaymentProcessor

{

void ProcessPayment(decimal amount);

}

public class StripeGateway

{

public void MakeStripePayment(decimal amount)

{

Console.WriteLine($"Stripe payment processed: {amount} INR");

}

}

public class PayPalGateway

{

public void PayWithPayPal(decimal amount)

{

Console.WriteLine($"PayPal payment processed: {amount} INR");

}

}

public class StripeAdapter : IPaymentProcessor

{

private readonly StripeGateway stripe;

public StripeAdapter()

{

stripe = new StripeGateway();

}

public void ProcessPayment(decimal amount)

{

stripe.MakeStripePayment(amount);

}

}

public class PayPalAdapter : IPaymentProcessor

{

private readonly PayPalGateway paypal;

public PayPalAdapter()

{

paypal = new PayPalGateway();

}

public void ProcessPayment(decimal amount)

{

paypal.PayWithPayPal(amount);

}

}

class Program

{

static void Main(string[] args)

{

IPaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.ProcessPayment(1500);

IPaymentProcessor paypalProcessor = new PayPalAdapter();

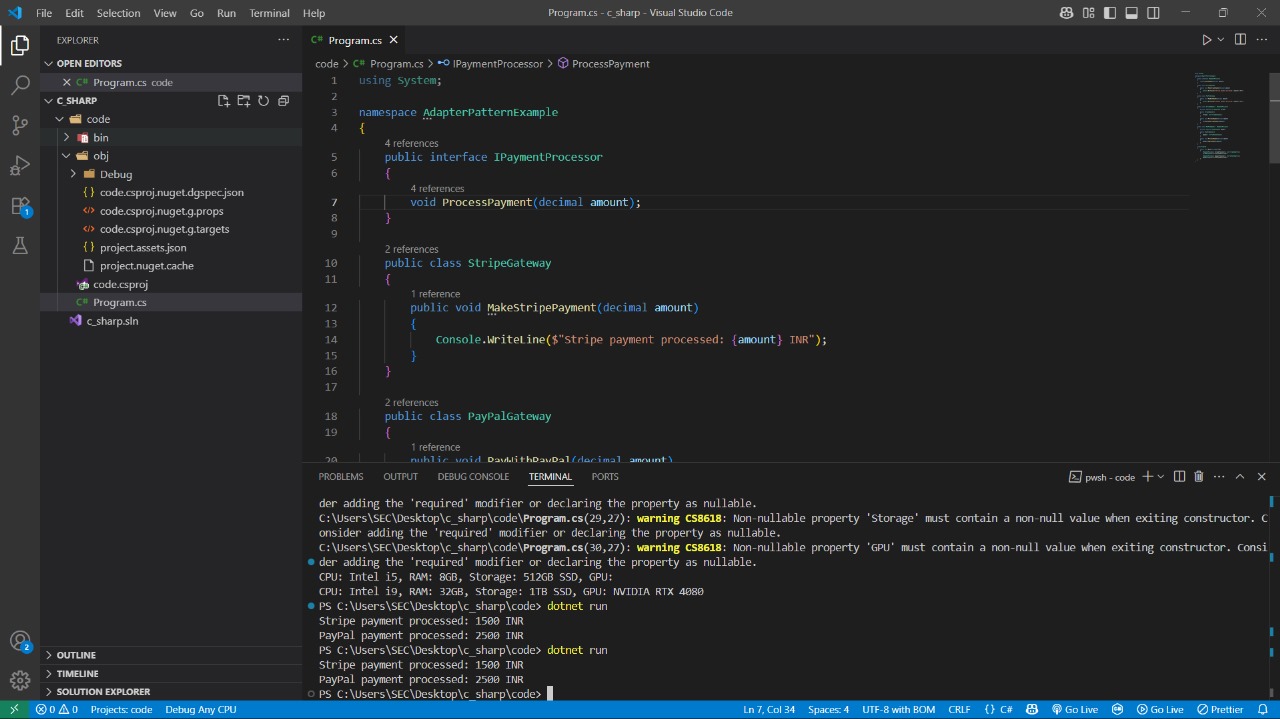
paypalProcessor.ProcessPayment(2500);

}

}

}

**OUTPUT:-**



### Conclusion

The Adapter pattern was implemented to adapt multiple payment gateway implementations to a common interface. This allows the client code to use different payment methods interchangeably without knowing the internal details of each gateway. This pattern is helpful when integrating third-party libraries or legacy code with new systems that expect a specific interface.

**Exercise 5: Implementing the Decorator Pattern(.NET)**

**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.

### What I Learned:

I learned how to implement the Decorator Design Pattern in C#. This pattern is used to add new functionalities to an existing object without modifying its structure. It follows the Open/Closed Principle by allowing behaviors to be extended without altering the original code. In this example, multiple notification methods like Email, SMS, and Slack are combined dynamically using decorators.

**CODE:**

using System;

namespace DecoratorPatternExample

{

public interface INotifier

{

void Send(string message);

}

public class EmailNotifier : INotifier

{

public void Send(string message)

{

Console.WriteLine("Email sent: " + message);

}

}

public abstract class NotifierDecorator : INotifier

{

protected INotifier notifier;

public NotifierDecorator(INotifier notifier)

{

this.notifier = notifier;

}

public virtual void Send(string message)

{

notifier.Send(message);

}

}

public class SMSNotifierDecorator : NotifierDecorator

{

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

public override void Send(string message)

{

base.Send(message);

Console.WriteLine("SMS sent: " + message);

}

}

public class SlackNotifierDecorator : NotifierDecorator

{

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

public override void Send(string message)

{

base.Send(message);

Console.WriteLine("Slack message sent: " + message);

}

}

class Program

{

static void Main(string[] args)

{

INotifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

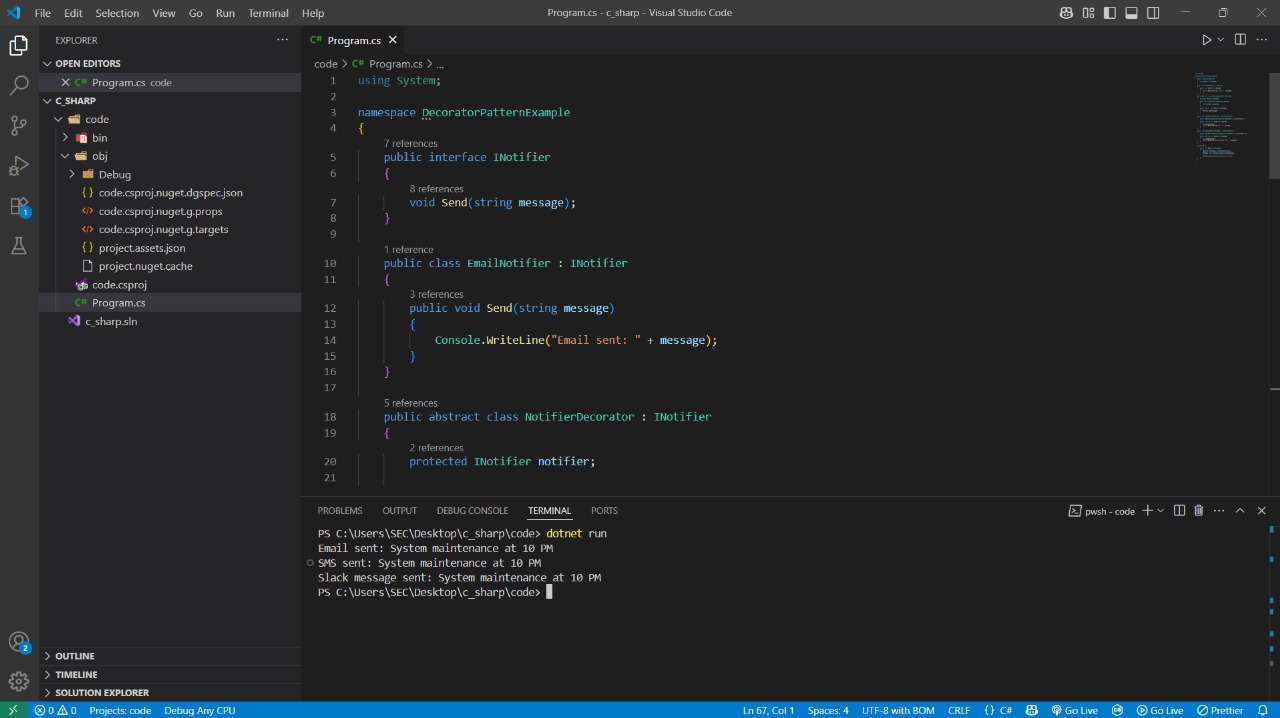
notifier.Send("System maintenance at 10 PM");

}

}

}

**OUTPUT:**



Conclusion:

The Decorator pattern allows dynamic addition of notification features like SMS and Slack without modifying the original EmailNotifier class.

**Exercise 6: Implementing the Proxy Pattern (.NET)**

**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.

### What I Learned

I learned how to implement the Proxy Design Pattern in C#. This pattern provides a substitute or placeholder for another object to control access to it. It is useful when dealing with resource-intensive objects. In this example, image loading is deferred until display time, optimizing performance by avoiding unnecessary object creation.

**CODE:**

using System;

namespace ProxyPatternExample

{

public interface IImage

{

void Display();

}

public class RealImage : IImage

{

private string fileName;

public RealImage(string fileName)

{

this.fileName = fileName;

LoadFromServer(fileName);

}

private void LoadFromServer(string fileName)

{

Console.WriteLine("Loading " + fileName + " from remote server...");

}

public void Display()

{

Console.WriteLine("Displaying " + fileName);

}

}

public class ProxyImage : IImage

{

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();

}

}

class Program

{

static void Main(string[] args)

{

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

image1.Display();

Console.WriteLine();

IImage image2 = new ProxyImage("photo1.jpg");

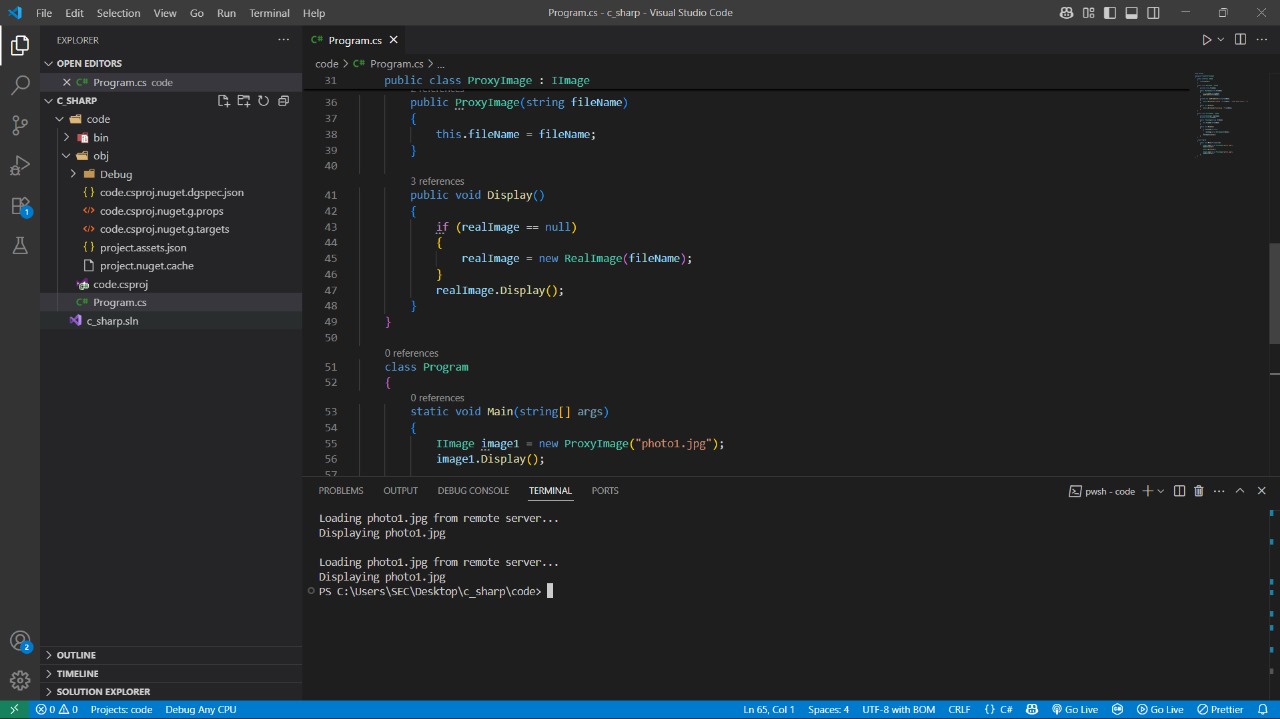
image2.Display();

}

}

}

**OUTPUT:**



### Conclusion:

The Proxy pattern helps reduce resource usage by loading the image only when needed, without changing the client code.

**Exercise 7: Implementing the Observer Pattern(.NET)**

**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.

### What I Learned:

I learned how to implement the Observer Design Pattern in C#. This pattern establishes a one-to-many relationship between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. It is useful in event-driven systems like stock price tracking, news broadcasting, and UI updates.

**CODE:**

using System;

using System.Collections.Generic;

namespace ObserverPatternExample

{

public interface IObserver

{

void Update(string stockName, double price);

}

public interface IStock

{

void RegisterObserver(IObserver observer);

void RemoveObserver(IObserver observer);

void NotifyObservers();

}

public class StockMarket : IStock

{

private List<IObserver> observers = new List<IObserver>();

private string stockName;

private double stockPrice;

public StockMarket(string name, double price)

{

stockName = name;

stockPrice = price;

}

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(stockName, stockPrice);

}

}

public void SetPrice(double price)

{

stockPrice = price;

NotifyObservers();

}

}

public class MobileApp : IObserver

{

public void Update(string stockName, double price)

{

Console.WriteLine($"Mobile App: {stockName} price updated to {price}");

}

}

public class WebApp : IObserver

{

public void Update(string stockName, double price)

{

Console.WriteLine($"Web App: {stockName} price updated to {price}");

}

}

class Program

{

static void Main(string[] args)

{

StockMarket stock = new StockMarket("TCS", 3200);

IObserver mobileApp = new MobileApp();

IObserver webApp = new WebApp();

stock.RegisterObserver(mobileApp);

stock.RegisterObserver(webApp);

stock.SetPrice(3220);

stock.SetPrice(3250);

stock.RemoveObserver(webApp);

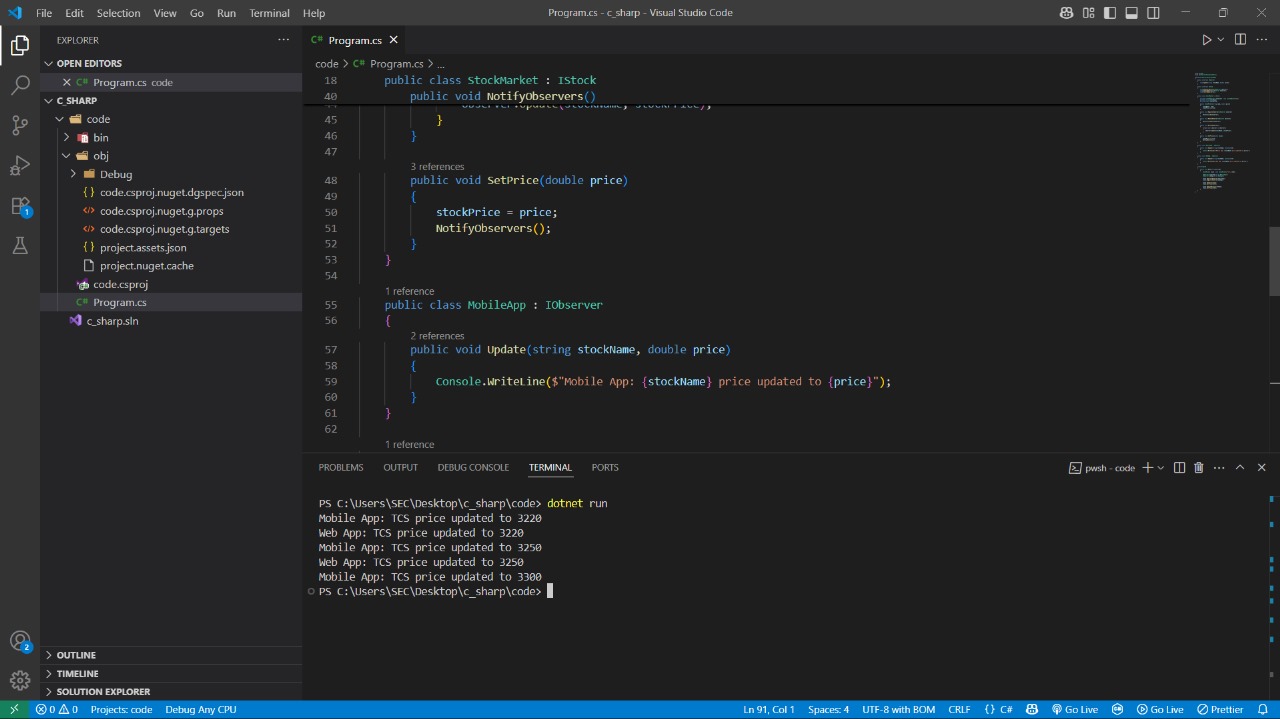
stock.SetPrice(3300);

}

}

}

**OUTPUT:-**



### Conclusion

The Observer pattern enables automatic updates to registered observers whenever the stock price changes.

**Exercise 8: Implementing the Strategy Pattern (.NET)**

**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.

### What I Learned

I learned how to implement the Strategy Design Pattern in C#. This pattern allows selecting an algorithm's behavior at runtime. It enables the use of different payment methods (Credit Card, PayPal) without changing the code in the main context class. This promotes flexibility, reusability, and adherence to the Open/Closed Principle.

**CODE:**

using System;

namespace StrategyPatternExample

{

public interface IPaymentStrategy

{

void Pay(decimal amount);

}

public class CreditCardPayment : IPaymentStrategy

{

public void Pay(decimal amount)

{

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

}

}

public class PayPalPayment : IPaymentStrategy

{

public void Pay(decimal amount)

{

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

}

}

public class PaymentContext

{

private IPaymentStrategy strategy;

public PaymentContext(IPaymentStrategy strategy)

{

this.strategy = strategy;

}

public void SetStrategy(IPaymentStrategy strategy)

{

this.strategy = strategy;

}

public void ExecutePayment(decimal amount)

{

strategy.Pay(amount);

}

}

class Program

{

static void Main(string[] args)

{

PaymentContext context = new PaymentContext(new CreditCardPayment());

context.ExecutePayment(1000);

context.SetStrategy(new PayPalPayment());

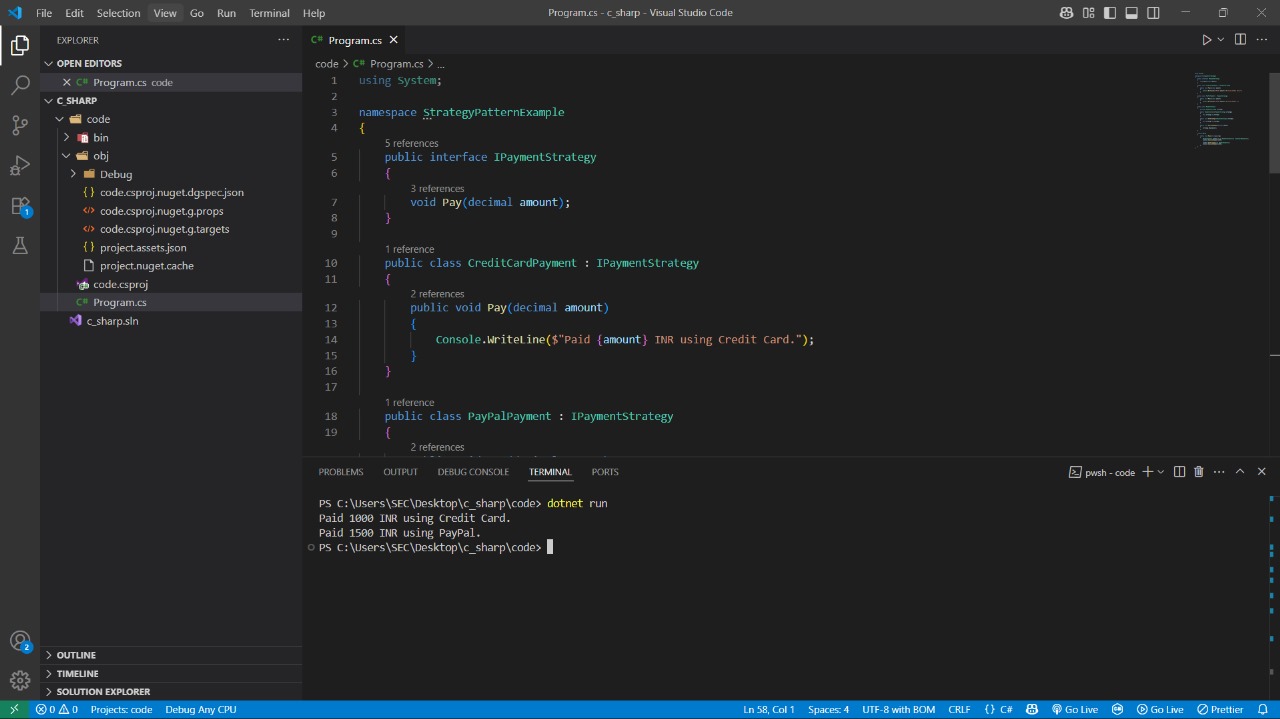
context.ExecutePayment(1500);

}

}

}

**OUTPUT:**



### Conclusion

The Strategy pattern allows switching between payment methods at runtime without modifying the context class.

**Exercise 9: Implementing the Command Pattern (.NET)**

**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.

### What I Learned

I learned how to implement the Command Design Pattern in C#. This pattern encapsulates a request as an object, thereby allowing parameterization of clients with different requests, queuing, and logging operations. It decouples the object that invokes the operation from the one that knows how to perform it.

**CODE:**

using System;

namespace CommandPatternExample

{

public interface ICommand

{

void Execute();

}

public class Light

{

public void TurnOn()

{

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

}

public void TurnOff()

{

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

}

}

public class LightOnCommand : ICommand

{

private Light light;

public LightOnCommand(Light light)

{

this.light = light;

}

public void Execute()

{

light.TurnOn();

}

}

public class LightOffCommand : ICommand

{

private Light light;

public LightOffCommand(Light light)

{

this.light = light;

}

public void Execute()

{

light.TurnOff();

}

}

public class RemoteControl

{

private ICommand command;

public void SetCommand(ICommand command)

{

this.command = command;

}

public void PressButton()

{

command.Execute();

}

}

class Program

{

static void Main(string[] args)

{

Light livingRoomLight = new Light();

ICommand lightOn = new LightOnCommand(livingRoomLight);

ICommand lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.SetCommand(lightOn);

remote.PressButton();

remote.SetCommand(lightOff);

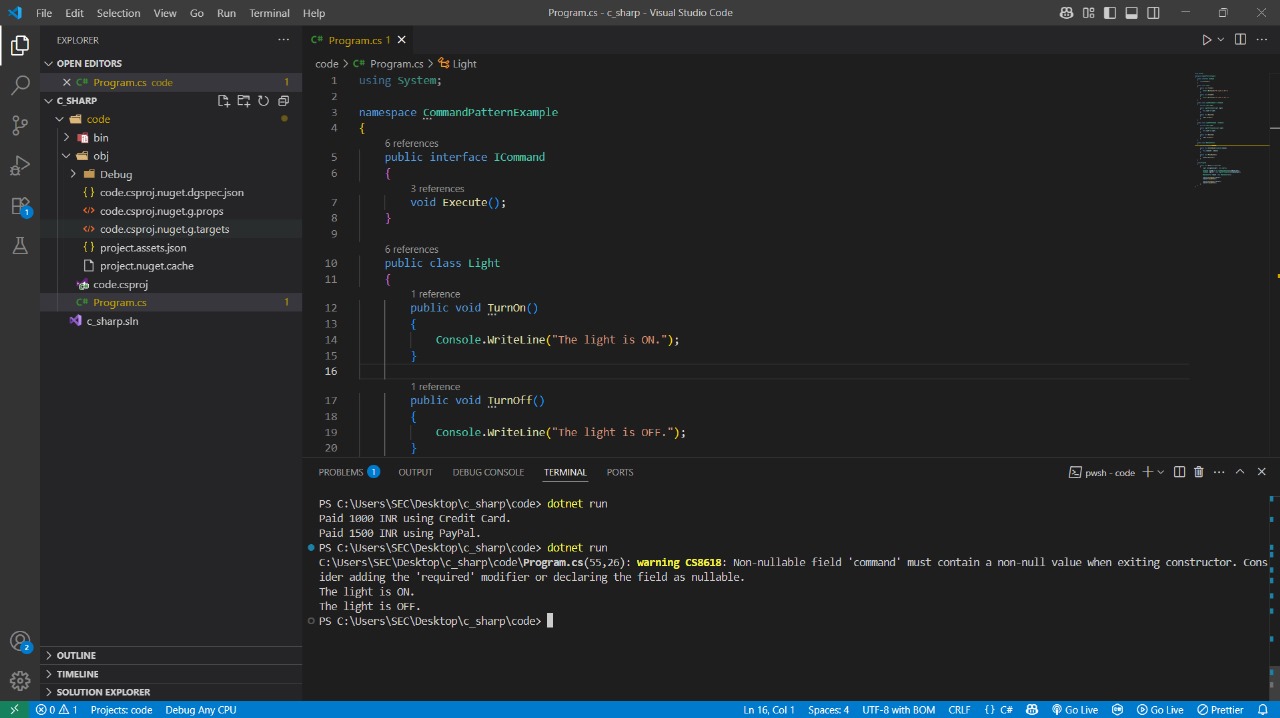
remote.PressButton();

}

}

}

**OUTPUT:**



### Conclusion

The Command pattern encapsulates light control actions as command objects, enabling flexible and decoupled execution through a remote control interface.

**Exercise 10: Implementing the MVC Pattern (.NET)**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

### What I Learned

I learned how to implement the Model-View-Controller (MVC) architectural pattern in C#. This pattern separates the application into three interconnected components: Model (data), View (UI), and Controller (logic). It helps in organizing code better and makes it easier to manage, test, and scale.

**CODE:**

using System;

namespace MVCPatternExample

{

public class Student

{

public string Name { get; set; }

public int Id { get; set; }

public string Grade { get; set; }

}

public class StudentView

{

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

{

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

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

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

}

}

public class StudentController

{

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view)

{

this.student = student;

this.view = view;

}

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

public void SetStudentId(int id) => student.Id = id;

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

public string GetStudentName() => student.Name;

public int GetStudentId() => student.Id;

public string GetStudentGrade() => student.Grade;

public void UpdateView()

{

view.DisplayStudentDetails(student.Name, student.Id, student.Grade);

}

}

class Program

{

static void Main(string[] args)

{

Student student = new Student { Name = "Alice", Id = 101, Grade = "A" };

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.UpdateView();

controller.SetStudentName("Bob");

controller.SetStudentGrade("A+");

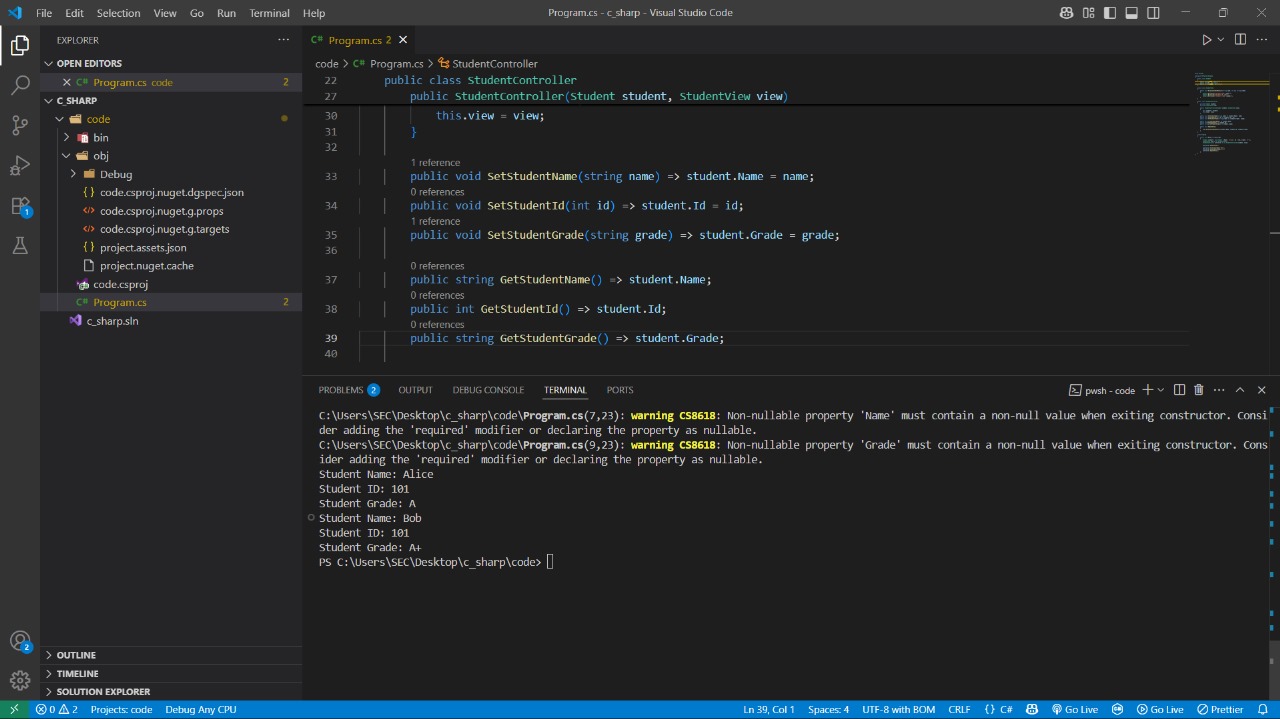
controller.UpdateView();

}

}

}

**OUTPUT:**



### Conclusion

The MVC pattern separates data, presentation, and logic clearly, making the application easier to maintain and extend.

**Exercise 11: Implementing Dependency Injection(.NET)**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

### What I Learned

I learned how to implement Dependency Injection (DI) in C#. This pattern promotes loose coupling between classes by injecting dependencies from outside rather than creating them internally. It improves code flexibility, testability, and scalability.

**CODE:**

using System;

namespace DependencyInjectionExample

{

public interface ICustomerRepository

{

string FindCustomerById(int id);

}

public class CustomerRepositoryImpl : ICustomerRepository

{

public string FindCustomerById(int id)

{

return $"Customer #{id} - John Doe";

}

}

public class CustomerService

{

private readonly ICustomerRepository repository;

public CustomerService(ICustomerRepository repository)

{

this.repository = repository;

}

public void DisplayCustomer(int id)

{

string customer = repository.FindCustomerById(id);

Console.WriteLine(customer);

}

}

class Program

{

static void Main(string[] args)

{

ICustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

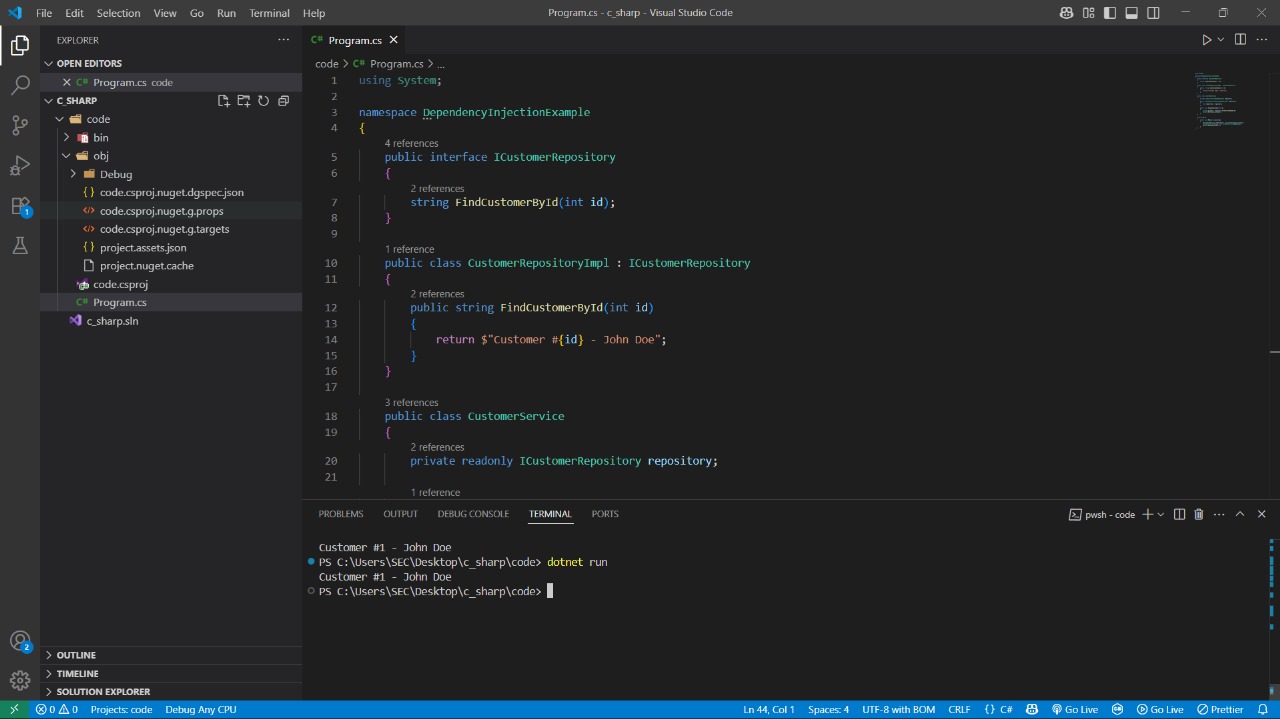
service.DisplayCustomer(1);

}

}

}

**OUTPUT:**



### Conclusion

Dependency Injection decouples service classes from concrete implementations, improving modularity and maintainability.

**Thank you**