**Week 1 Assignment Solutions  
Design Patterns**

**SuperSet ID: 6365365**

1. **Implementing Singleton Pattern**

Logger.cs

using System;

public class Logger

{

    private static Logger \_instance;

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

    // Private constructor to prevent external instantiation

    private Logger()

    {

        Console.WriteLine("Logger initialized.");

    }

    // Public method to access the single instance (Thread-safe)

    public static Logger GetInstance()

    {

        if (\_instance == null)

        {

            lock (\_lock)

            {

                if (\_instance == null)

                    \_instance = new Logger();

            }

        }

        return \_instance;

    }

    // Example method for logging

    public void Log(string message)

    {

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

    }

}

Program.cs

using System;

class Program

{

    static void Main()

    {

        Logger logger1 = Logger.GetInstance();

        logger1.Log("This is the first log message.");

        Logger logger2 = Logger.GetInstance();

        logger2.Log("This is the second log message.");

        // Check if both instances are the same

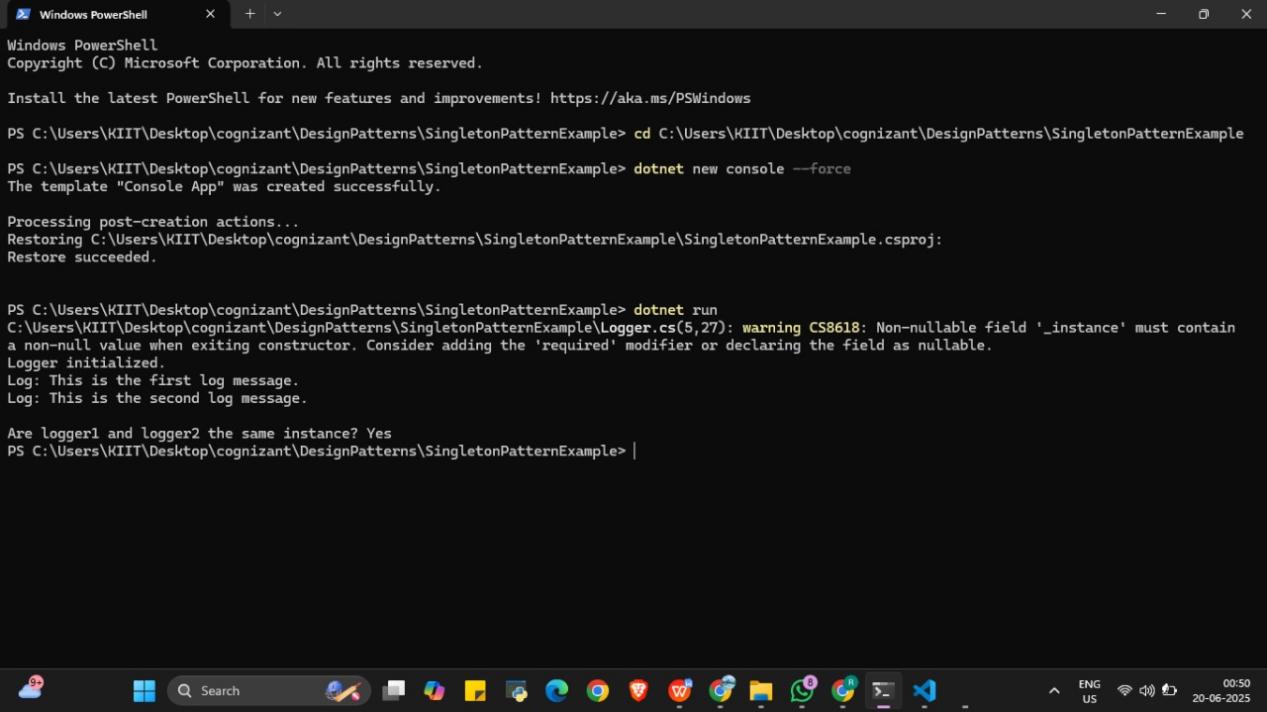
        Console.WriteLine("\nAre logger1 and logger2 the same instance? " +

            (logger1 == logger2 ? "Yes" : "No"));

    }

}

Output



1. **Implementing the Factory Method Pattern**

DocumentFactory.cs

public abstract class DocumentFactory

{

    public abstract IDocument CreateDocument();

}

ExcelDocument.cs

using System;

public class ExcelDocument : IDocument

{

    public void Open()

    {

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

    }

}

ExcelFactory.cs

public class ExcelFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new ExcelDocument();

    }

}

IDocument.cs

public interface IDocument

{

    void Open();

}

PdfDocument.cs

using System;

public class PdfDocument : IDocument

{

    public void Open()

    {

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

    }

}

PdfFactory.cs

public class PdfFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new PdfDocument();

    }

}

WordDocument.cs

using System;

public class WordDocument : IDocument

{

    public void Open()

    {

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

    }

}

WordFactory.cs

public class WordFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new WordDocument();

    }

}

Program.cs

using System;

class Program

{

    static void Main()

    {

        DocumentFactory wordFactory = new WordFactory();

        IDocument wordDoc = wordFactory.CreateDocument();

        wordDoc.Open();

        DocumentFactory pdfFactory = new PdfFactory();

        IDocument pdfDoc = pdfFactory.CreateDocument();

        pdfDoc.Open();

        DocumentFactory excelFactory = new ExcelFactory();

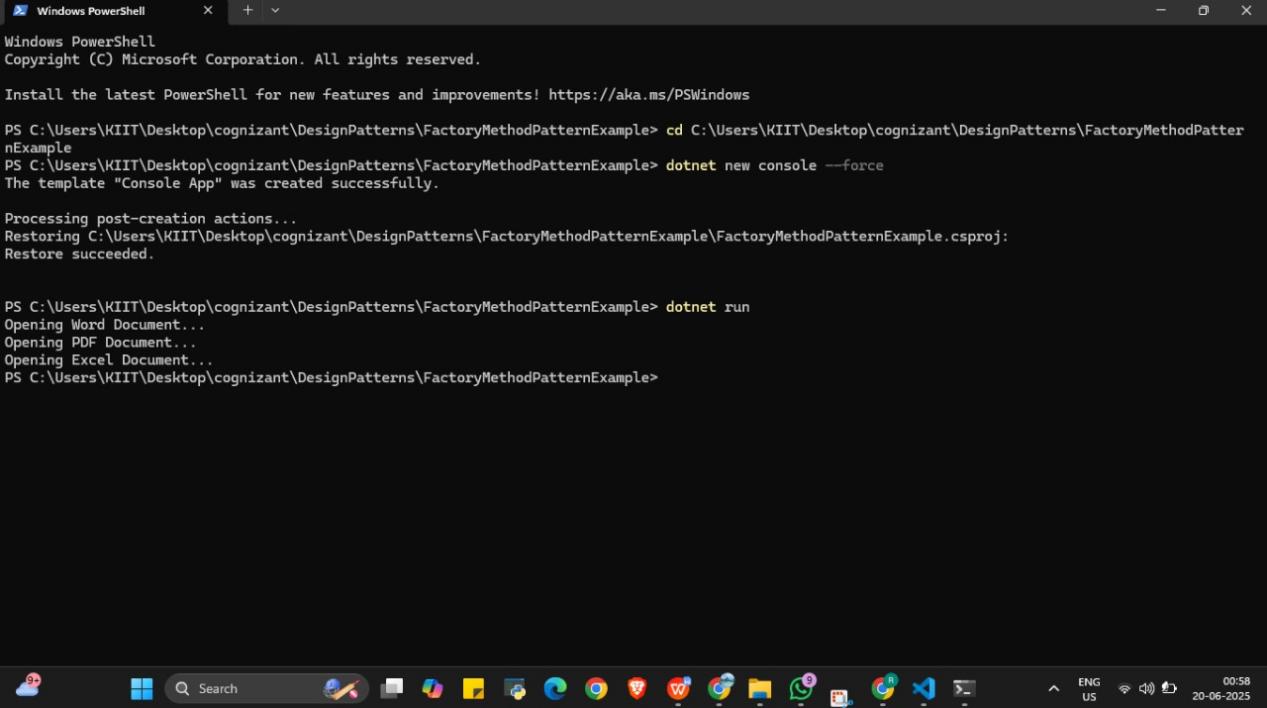
        IDocument excelDoc = excelFactory.CreateDocument();

        excelDoc.Open();

    }

}

Output



1. **Implementing The Builder Pattern**

Computer.cs

namespace BuilderPatternExample

{

    public class Computer

    {

        // Required attributes

        public string CPU { get; private set; }

        public string RAM { get; private set; }

        // Optional attributes

        public string Storage { get; private set; }

        public string GraphicsCard { get; private set; }

        public string OperatingSystem { get; private set; }

        private Computer(Builder builder)

        {

            CPU = builder.CPU;

            RAM = builder.RAM;

            Storage = builder.Storage;

            GraphicsCard = builder.GraphicsCard;

            OperatingSystem = builder.OperatingSystem;

        }

        public class Builder

        {

            // Required

            public string CPU { get; private set; }

            public string RAM { get; private set; }

            // Optional

            public string Storage { get; private set; }

            public string GraphicsCard { get; private set; }

            public string OperatingSystem { get; private set; }

            public Builder(string cpu, string ram)

            {

                CPU = cpu;

                RAM = ram;

            }

            public Builder SetStorage(string storage)

            {

                Storage = storage;

                return this;

            }

            public Builder SetGraphicsCard(string graphicsCard)

            {

                GraphicsCard = graphicsCard;

                return this;

            }

            public Builder SetOperatingSystem(string os)

            {

                OperatingSystem = os;

                return this;

            }

            public Computer Build()

            {

                return new Computer(this);

            }

        }

        public override string ToString()

        {

            return $"Computer Specs:\nCPU: {CPU}\nRAM: {RAM}\nStorage: {Storage}\nGraphics Card: {GraphicsCard}\nOperating System: {OperatingSystem}";

        }

    }

}

Program.cs

using System;

using BuilderPatternExample;

class Program

{

    static void Main(string[] args)

    {

        // Basic configuration

        var basicComputer = new Computer.Builder("Intel i5", "8GB")

                                .Build();

        Console.WriteLine("Basic Computer:\n" + basicComputer);

        Console.WriteLine("----------------------------");

        // Advanced configuration

        var gamingComputer = new Computer.Builder("Intel i9", "32GB")

                                .SetStorage("1TB SSD")

                                .SetGraphicsCard("NVIDIA RTX 3080")

                                .SetOperatingSystem("Windows 11")

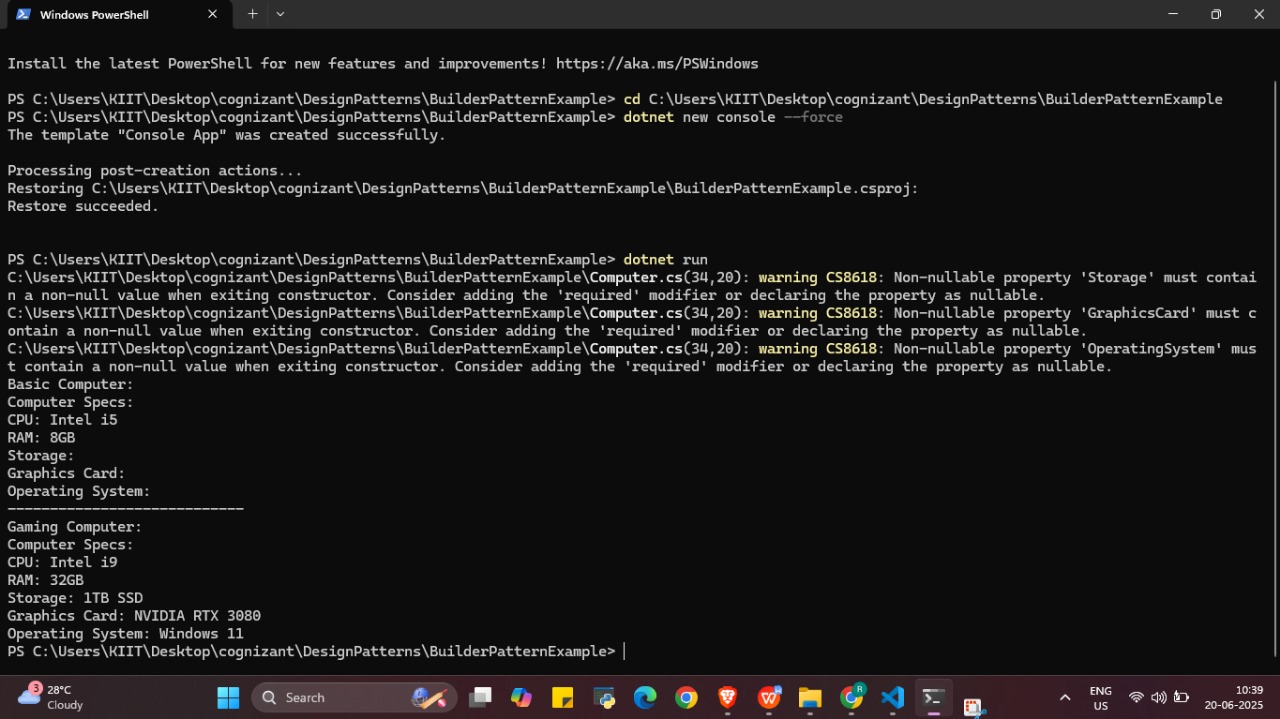
                                .Build();

        Console.WriteLine("Gaming Computer:\n" + gamingComputer);

    }

}

Output



1. **Implementing The Adapter Pattern**

NewPaymentAdapter.cs

namespace AdapterPatternExample

{

    public class NewPaymentAdapter : PaymentProcessor

    {

        private NewPaymentGateway \_newGateway;

        public NewPaymentAdapter(NewPaymentGateway newGateway)

        {

            \_newGateway = newGateway;

        }

        public void ProcessPayment(decimal amount)

        {

            \_newGateway.ExecuteTransaction(amount);

        }

    }

}

NewPaymentGateway.cs

namespace AdapterPatternExample

{

    public class NewPaymentGateway

    {

        public void ExecuteTransaction(decimal total)

        {

            Console.WriteLine($"[NewGateway] Transaction of ${total} executed.");

        }

    }

}

OldPaymentGateway.cs

namespace AdapterPatternExample

{

    public class OldPaymentGateway

    {

        public void MakePayment(double amt)

        {

            Console.WriteLine($"[OldGateway] Payment of ${amt} made successfully.");

        }

    }

}

OldPaymentAdapter.cs

namespace AdapterPatternExample

{

    public class OldPaymentAdapter : PaymentProcessor

    {

        private OldPaymentGateway \_oldGateway;

        public OldPaymentAdapter(OldPaymentGateway oldGateway)

        {

            \_oldGateway = oldGateway;

        }

        public void ProcessPayment(decimal amount)

        {

            \_oldGateway.MakePayment((double)amount);

        }

    }

}

PaymentProcessor.cs

namespace AdapterPatternExample

{

    public interface PaymentProcessor

    {

        void ProcessPayment(decimal amount);

    }

}

Program.cs

using System;

using AdapterPatternExample;

class Program

{

    static void Main(string[] args)

    {

        // Using OldPaymentGateway via Adapter

        PaymentProcessor oldAdapter = new OldPaymentAdapter(new OldPaymentGateway());

        oldAdapter.ProcessPayment(100.50m);

        Console.WriteLine("-----------------------");

        // Using NewPaymentGateway via Adapter

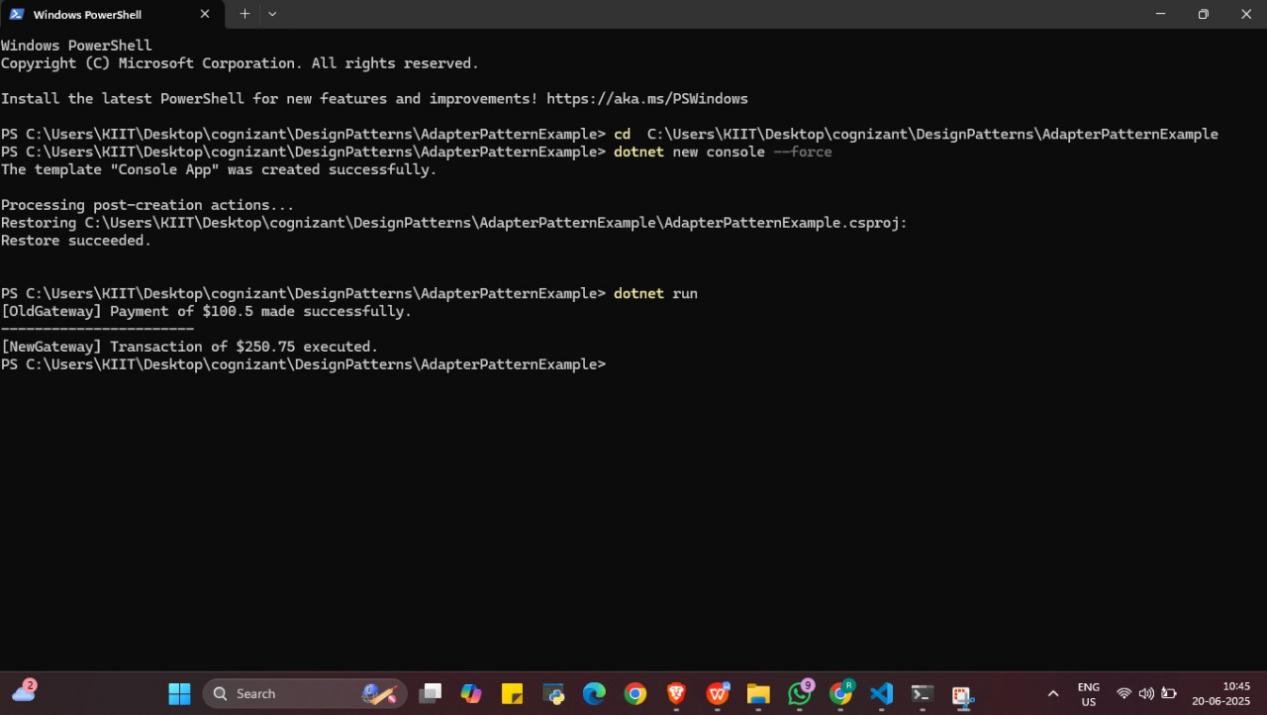
        PaymentProcessor newAdapter = new NewPaymentAdapter(new NewPaymentGateway());

        newAdapter.ProcessPayment(250.75m);

    }

}

Output



1. **Implementing The Decorator Pattern**

EmailNotifier.cs

namespace DecoratorPatternExample

{

    public class EmailNotifier : Notifier

    {

        public void Send(string message)

        {

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

        }

    }

}

Notifier.cs

namespace DecoratorPatternExample

{

    public interface Notifier

    {

        void Send(string message);

    }

}

NotifierDecorator.cs

namespace DecoratorPatternExample

{

    public abstract class NotifierDecorator : Notifier

    {

        protected Notifier \_notifier;

        protected NotifierDecorator(Notifier notifier)

        {

            \_notifier = notifier;

        }

        public virtual void Send(string message)

        {

            \_notifier.Send(message);

        }

    }

}

SlackNotifierDecorator.cs

namespace DecoratorPatternExample

{

    public class SlackNotifierDecorator : NotifierDecorator

    {

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

        public override void Send(string message)

        {

            base.Send(message);

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

        }

    }

}

SMSNotifierDecorator.cs

namespace DecoratorPatternExample

{

    public class SMSNotifierDecorator : NotifierDecorator

    {

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

        public override void Send(string message)

        {

            base.Send(message);

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

        }

    }

}

Program.cs

using System;

using DecoratorPatternExample;

class Program

{

    static void Main(string[] args)

    {

        // Basic email notification

        Notifier notifier = new EmailNotifier();

        // Add SMS notification

        notifier = new SMSNotifierDecorator(notifier);

        // Add Slack notification

        notifier = new SlackNotifierDecorator(notifier);

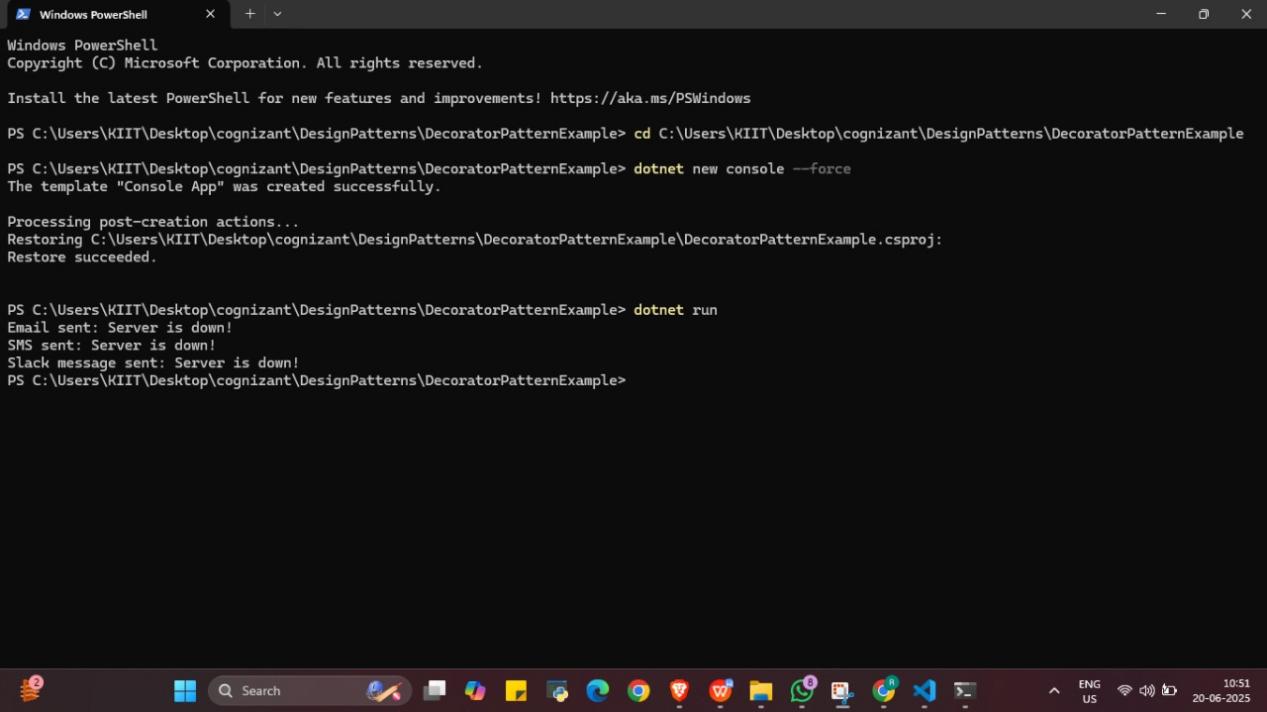
        // Send message through all channels

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

    }

}

Output



1. **Implementing the Proxy Pattern**

Image.cs

namespace ProxyPatternExample

{

    public interface Image

    {

        void Display();

    }

}

ProxyImage.cs

namespace ProxyPatternExample

{

    public class ProxyImage : Image

    {

        private RealImage \_realImage;

        private string \_fileName;

        public ProxyImage(string fileName)

        {

            \_fileName = fileName;

        }

        public void Display()

        {

            if (\_realImage == null)

            {

                \_realImage = new RealImage(\_fileName); // Lazy initialization

            }

            \_realImage.Display();

        }

    }

}

RealImage.cs

using System;

namespace ProxyPatternExample

{

    public class RealImage : Image

    {

        private string \_fileName;

        public RealImage(string fileName)

        {

            \_fileName = fileName;

            LoadFromServer();

        }

        private void LoadFromServer()

        {

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

        }

        public void Display()

        {

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

        }

    }

}

Program.cs

using System;

using ProxyPatternExample;

class Program

{

    static void Main(string[] args)

    {

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

        Image image2 = new ProxyImage("photo2.jpg");

        // First time: loads from server

        image1.Display();

        Console.WriteLine();

        // Second time: uses cached image

        image1.Display();

        Console.WriteLine();

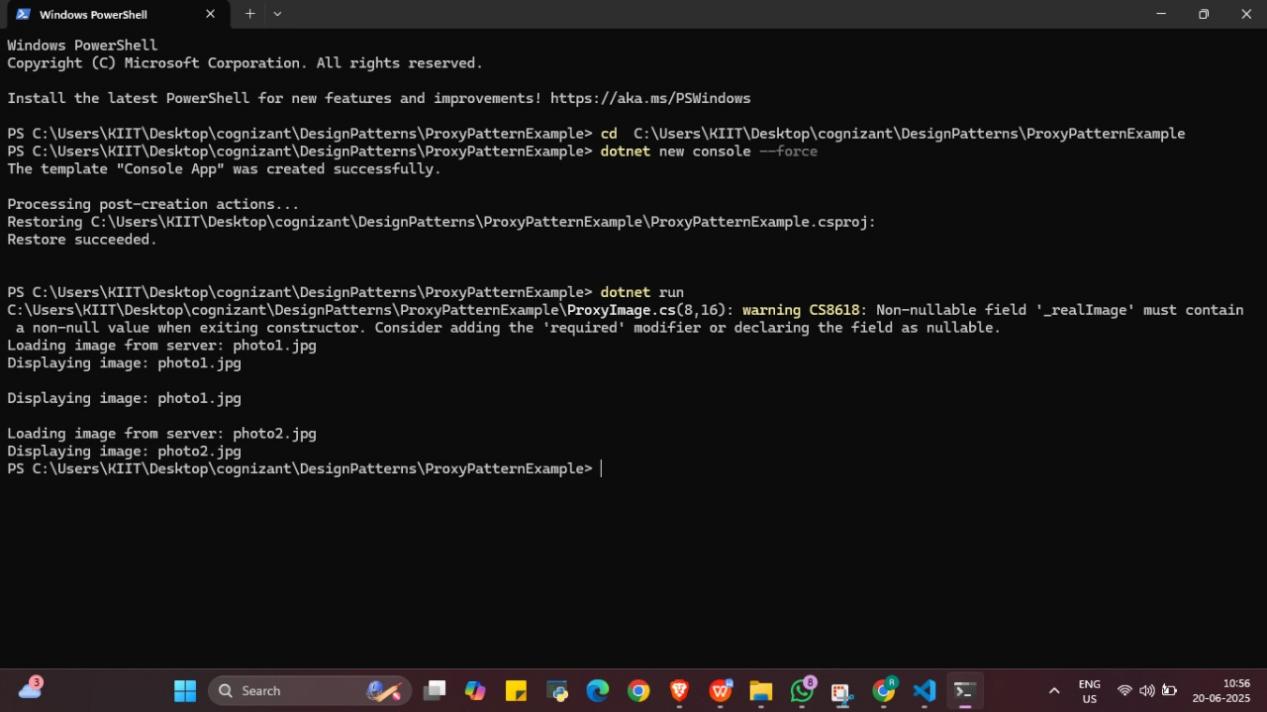
        // First time for another image

        image2.Display();

    }

}

Output



1. **Implementing the Observer Pattern**

IObserver.cs

namespace ObserverPatternExample

{

    public interface IObserver

    {

        void Update(string stockName, double price);

    }

}

IStock.cs

namespace ObserverPatternExample

{

    public interface IStock

    {

        void RegisterObserver(IObserver observer);

        void RemoveObserver(IObserver observer);

        void NotifyObservers();

    }

}

MobileApp.cs

using System;

namespace ObserverPatternExample

{

    public class MobileApp : IObserver

    {

        private string \_appName;

        public MobileApp(string appName)

        {

            \_appName = appName;

        }

        public void Update(string stockName, double price)

        {

            Console.WriteLine($"{\_appName} - Stock: {stockName} updated to ${price}");

        }

    }

}

StockMarket.cs

using System;

using System.Collections.Generic;

namespace ObserverPatternExample

{

    public class StockMarket : IStock

    {

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

        private string \_stockName;

        private double \_price;

        public void SetStockPrice(string stockName, double price)

        {

            \_stockName = stockName;

            \_price = 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(\_stockName, \_price);

            }

        }

    }

}

WebApp.cs

using System;

namespace ObserverPatternExample

{

    public class WebApp : IObserver

    {

        private string \_siteName;

        public WebApp(string siteName)

        {

            \_siteName = siteName;

        }

        public void Update(string stockName, double price)

        {

            Console.WriteLine($"{\_siteName} WebApp - Stock: {stockName} changed to ${price}");

        }

    }

}

Program.cs

using System;

using ObserverPatternExample;

class Program

{

    static void Main(string[] args)

    {

        var stockMarket = new StockMarket();

        var mobileApp = new MobileApp("MyStocks");

        var webApp = new WebApp("FinanceNow");

        stockMarket.RegisterObserver(mobileApp);

        stockMarket.RegisterObserver(webApp);

        stockMarket.SetStockPrice("AAPL", 180.50);

        Console.WriteLine();

        stockMarket.SetStockPrice("GOOG", 2725.00);

        Console.WriteLine();

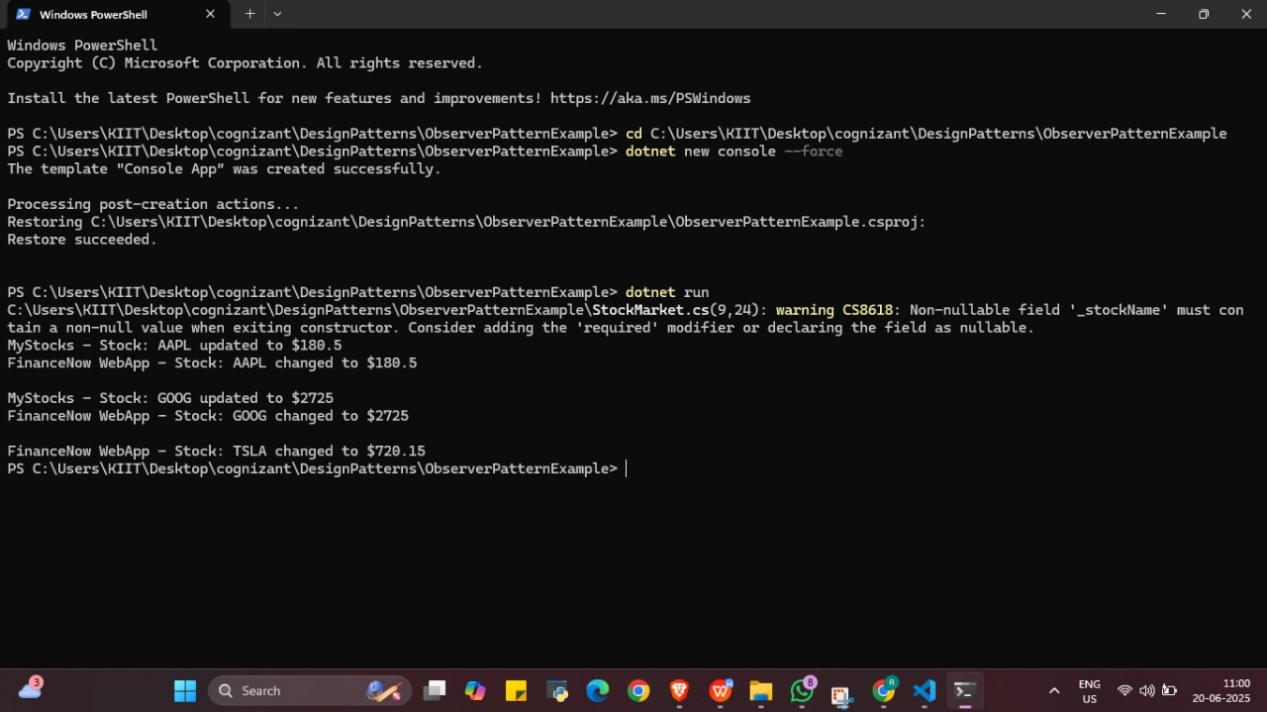
        stockMarket.RemoveObserver(mobileApp);

        stockMarket.SetStockPrice("TSLA", 720.15);

    }

}

Output



1. **Implementing the Strategy Pattern**

CreditCardPayment.cs

using System;

namespace StrategyPatternExample

{

    public class CreditCardPayment : IPaymentStrategy

    {

        public void Pay(decimal amount)

        {

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

        }

    }

}

IPaymentStrategy.cs

namespace StrategyPatternExample

{

    public interface IPaymentStrategy

    {

        void Pay(decimal amount);

    }

}

PaymentContext.cs

namespace StrategyPatternExample

{

    public class PaymentContext

    {

        private IPaymentStrategy \_paymentStrategy;

        public void SetPaymentStrategy(IPaymentStrategy strategy)

        {

            \_paymentStrategy = strategy;

        }

        public void ExecutePayment(decimal amount)

        {

            if (\_paymentStrategy == null)

            {

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

                return;

            }

            \_paymentStrategy.Pay(amount);

        }

    }

}

PayPalPayment.cs

using System;

namespace StrategyPatternExample

{

    public class PayPalPayment : IPaymentStrategy

    {

        public void Pay(decimal amount)

        {

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

        }

    }

}

Program.cs

using System;

using StrategyPatternExample;

class Program

{

    static void Main(string[] args)

    {

        var context = new PaymentContext();

        Console.WriteLine("Customer chooses Credit Card:");

        context.SetPaymentStrategy(new CreditCardPayment());

        context.ExecutePayment(150.00m);

        Console.WriteLine();

        Console.WriteLine("Customer chooses PayPal:");

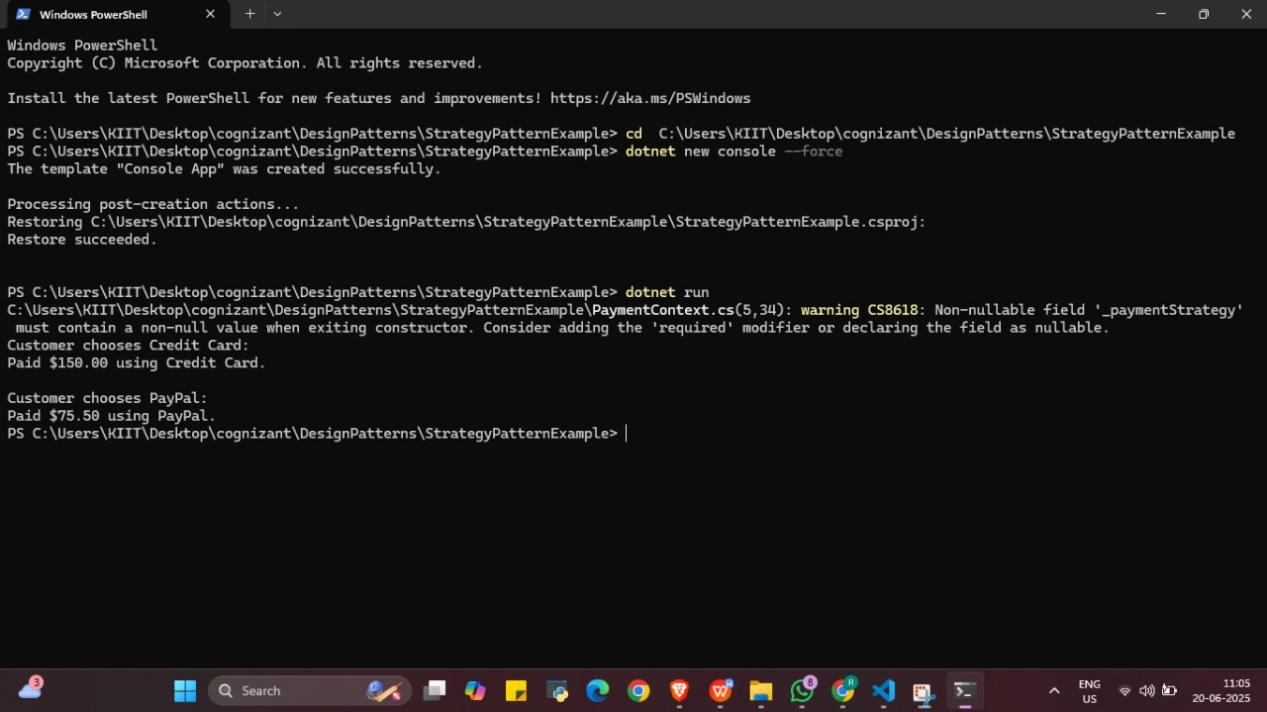
        context.SetPaymentStrategy(new PayPalPayment());

        context.ExecutePayment(75.50m);

    }

}

Output



1. **Implementing the Command pattern**

ICommand.cs

namespace CommandPatternExample

{

    public interface ICommand

    {

        void Execute();

    }

}

Light.cs

using System;

namespace CommandPatternExample

{

    public class Light

    {

        public void TurnOn()

        {

            Console.WriteLine("Light is ON.");

        }

        public void TurnOff()

        {

            Console.WriteLine("Light is OFF.");

        }

    }

}

LightOnCommand.cs

namespace CommandPatternExample

{

    public class LightOnCommand : ICommand

    {

        private Light \_light;

        public LightOnCommand(Light light)

        {

            \_light = light;

        }

        public void Execute()

        {

            \_light.TurnOn();

        }

    }

}

LightOffCommand.cs

namespace CommandPatternExample

{

    public class LightOffCommand : ICommand

    {

        private Light \_light;

        public LightOffCommand(Light light)

        {

            \_light = light;

        }

        public void Execute()

        {

            \_light.TurnOff();

        }

    }

}

RemoteControl.cs

namespace CommandPatternExample

{

    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.");

            }

        }

    }

}

Program.csusing System;

using CommandPatternExample;

class Program

{

    static void Main(string[] args)

    {

        // Receiver

        Light livingRoomLight = new Light();

        // Commands

        ICommand lightOn = new LightOnCommand(livingRoomLight);

        ICommand lightOff = new LightOffCommand(livingRoomLight);

        // Invoker

        RemoteControl remote = new RemoteControl();

        // Turn on the light

        remote.SetCommand(lightOn);

        remote.PressButton();

        // Turn off the light

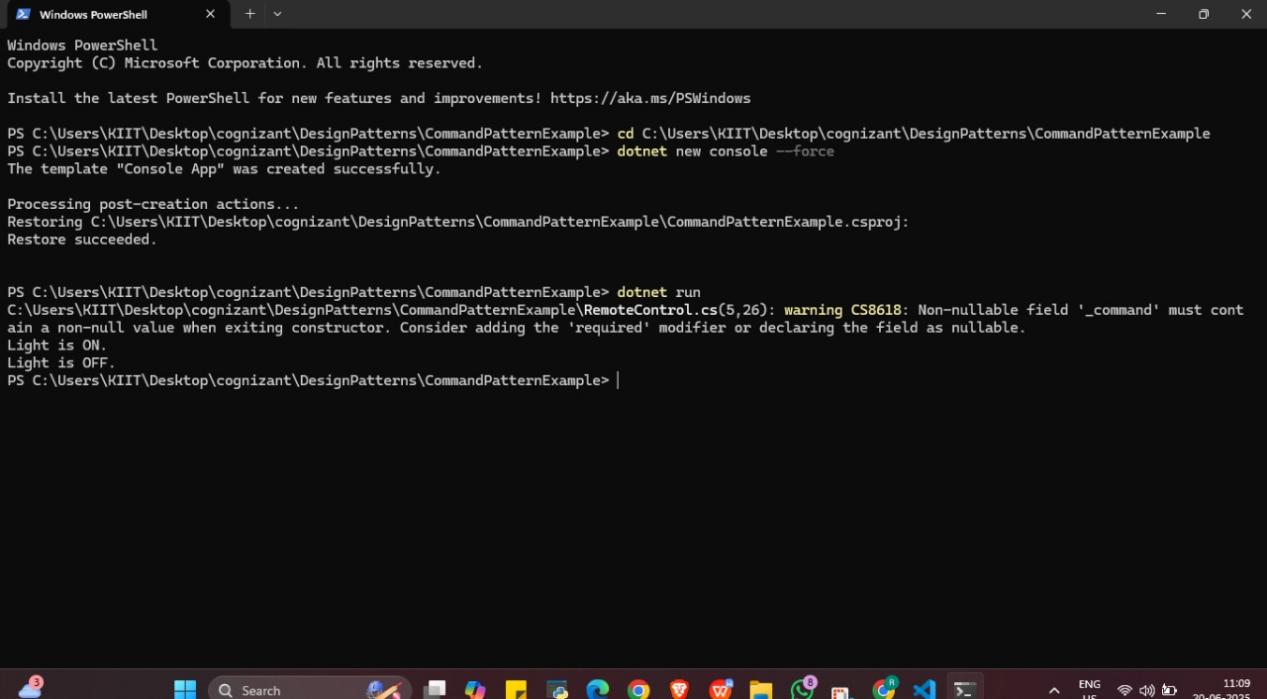
        remote.SetCommand(lightOff);

        remote.PressButton();

    }

}

Output

****

1. **Implementing the MVC pattern**

Student.cs

namespace MVCPatternExample

{

    public class Student

    {

        public string Name { get; set; }

        public int Id { get; set; }

        public string Grade { get; set; }

    }

}

StudentController.cs

namespace MVCPatternExample

{

    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 string GetStudentName()

        {

            return \_student.Name;

        }

        public void SetStudentGrade(string grade)

        {

            \_student.Grade = grade;

        }

        public string GetStudentGrade()

        {

            return \_student.Grade;

        }

        public void SetStudentId(int id)

        {

            \_student.Id = id;

        }

        public int GetStudentId()

        {

            return \_student.Id;

        }

        public void UpdateView()

        {

            \_view.DisplayStudentDetails(\_student);

        }

    }

}

StudentView.cs

using System;

namespace MVCPatternExample

{

    public class StudentView

    {

        public void DisplayStudentDetails(Student student)

        {

            Console.WriteLine("Student Details:");

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

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

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

            Console.WriteLine();

        }

    }

}

Program.cs

using System;

using MVCPatternExample;

class Program

{

    static void Main(string[] args)

    {

        // Model

        Student student = new Student

        {

            Name = "Alice",

            Id = 101,

            Grade = "A"

        };

        // View

        StudentView view = new StudentView();

        // Controller

        StudentController controller = new StudentController(student, view);

        // Display initial student details

        controller.UpdateView();

        // Update student data

        controller.SetStudentName("Alice Johnson");

        controller.SetStudentGrade("A+");

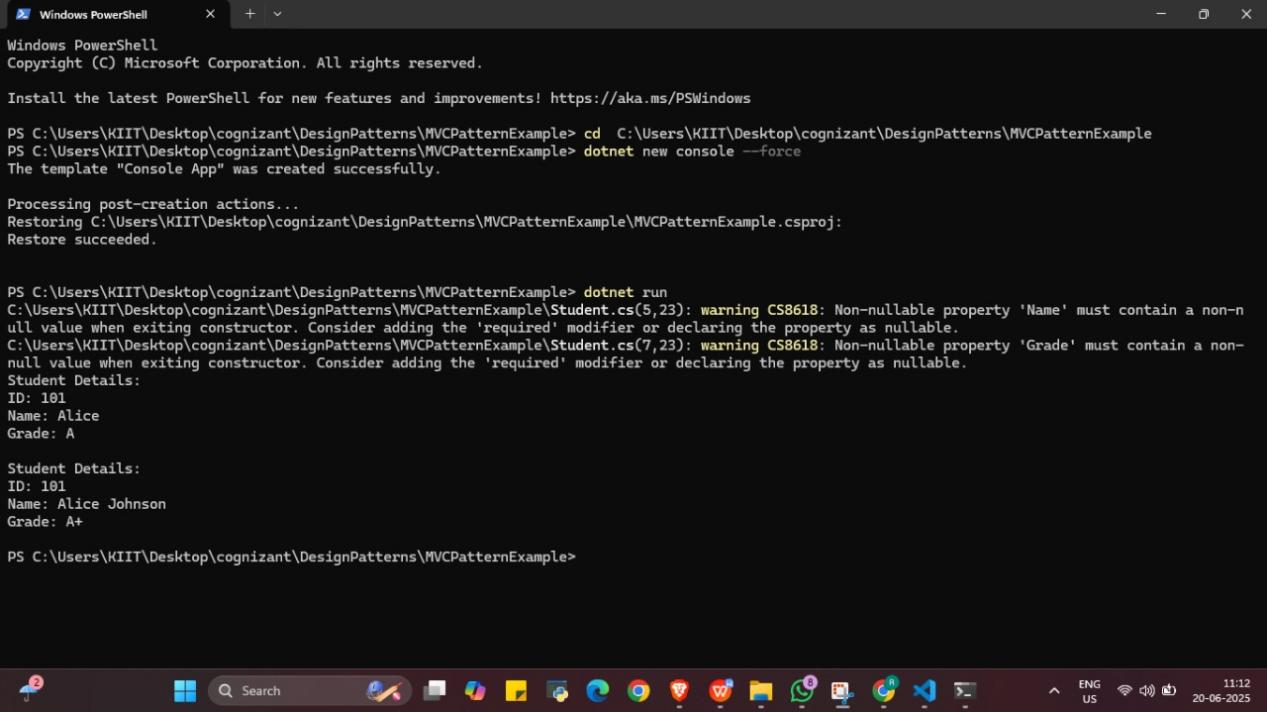
        // Display updated student details

        controller.UpdateView();

    }

}

Output.



1. **Implementing the Dependency Injection**

Customer.cs

namespace DependencyInjectionExample

{

    public class Customer

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

}

CustomerService.cs

using System;

namespace DependencyInjectionExample

{

    public class CustomerService

    {

        private readonly ICustomerRepository \_repository;

        public CustomerService(ICustomerRepository repository)

        {

            \_repository = repository;

        }

        public void DisplayCustomer(int id)

        {

            var customer = \_repository.FindCustomerById(id);

            if (customer != null)

            {

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

            }

            else

            {

                Console.WriteLine($"Customer with ID {id} not found.");

            }

        }

    }

}

CustomerRepositoryImpl.cs

using System;

using System.Collections.Generic;

namespace DependencyInjectionExample

{

    public class CustomerRepositoryImpl : ICustomerRepository

    {

        private Dictionary<int, Customer> \_customerDb = new Dictionary<int, Customer>

        {

            { 1, new Customer { Id = 1, Name = "Alice" } },

            { 2, new Customer { Id = 2, Name = "Bob" } }

        };

        public Customer FindCustomerById(int id)

        {

            \_customerDb.TryGetValue(id, out Customer customer);

            return customer;

        }

    }

}

ICustomerRepository.cs

namespace DependencyInjectionExample

{

    public interface ICustomerRepository

    {

        Customer FindCustomerById(int id);

    }

}

Program.cs

using System;

using DependencyInjectionExample;

class Program

{

    static void Main(string[] args)

    {

        // Create repository instance

        ICustomerRepository repository = new CustomerRepositoryImpl();

        // Inject repository into service using constructor injection

        CustomerService service = new CustomerService(repository);

        // Test service

        service.DisplayCustomer(1);  // should print Alice

        service.DisplayCustomer(3);  // not found

    }

}

Output

