**Exercise 1: Implementing the Singleton Pattern**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Program.cs

// See https://aka.ms/new-console-template for more information

// Console.WriteLine("Hello, World!");

using System;

namespace SingletonPatternExample

{

class Program

{

static void Main(string[] args)

{

// two instances of Logger

Logger logger1 = Logger.Instance;

Logger logger2 = Logger.Instance;

// logger to log messages

logger1.Log("First logger instance message");

logger2.Log("Second logger instance message");

// they are the same instance

if (logger1 == logger2)

{

Console.WriteLine("Both logger1 and logger2 are the SAME instance.");

}

else

{

Console.WriteLine("They are different instances — Singleton failed.");

}

}

}

}

Logger.cs

using System;

namespace SingletonPatternExample

{

public class Logger

{

// Create a private static instance of Logger

private static Logger? \_instance;

// Lock object to make thread-safe

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

// constructor

private Logger()

{

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

}

// global access to the instance

public static Logger Instance

{

get

{

// Double-checked locking

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

}

// log method

public void Log(string message)

{

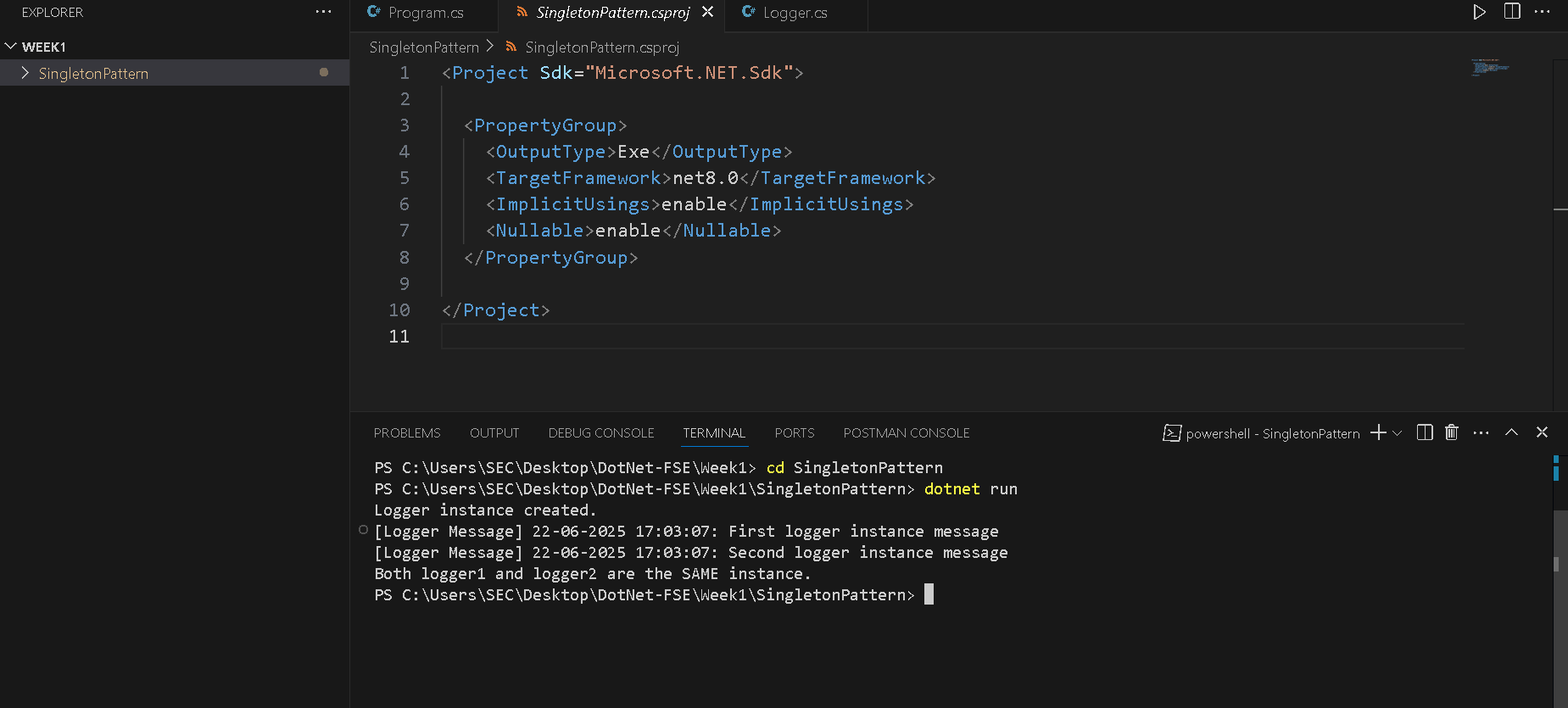
Console.WriteLine($"[Logger Message] {DateTime.Now}: {message}");

}

}

}

OUTPUT SCREENSHOT:



**Exercise 2: Implementing the Factory Method Pattern**

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.

IDocument.cs

namespace FactoryMethodPattern

{

public interface IDocument

{

void Create(string filename); // really creates the file

}

}

WordDocument.cs

using DocumentFormat.OpenXml.Packaging;

using DocumentFormat.OpenXml.Wordprocessing;

using System;

namespace FactoryMethodPattern

{

public class WordDocument : IDocument

{

public void Create(string filename)

{

string filepath = filename + ".docx";

// Create a Wordprocessing document.

using (WordprocessingDocument wordDocument =

WordprocessingDocument.Create(filepath, DocumentFormat.OpenXml.WordprocessingDocumentType.Document))

{

// Add a main document part.

MainDocumentPart mainPart = wordDocument.AddMainDocumentPart();

// Create the document structure and add some text.

mainPart.Document = new Document();

Body body = new Body();

Paragraph para = new Paragraph();

Run run = new Run();

Text text = new Text("This is a Word document created using Open XML SDK.");

run.Append(text);

para.Append(run);

body.Append(para);

mainPart.Document.Append(body);

mainPart.Document.Save();

}

Console.WriteLine(" Word document created using Open XML SDK.");

}

}

}

PdfDocument.cs

using PdfSharp.Pdf;

using PdfSharp.Drawing;

using System.Diagnostics;

namespace FactoryMethodPattern

{

public class PdfDocumentCreator : IDocument

{

public void Create(string filename)

{

PdfDocument document = new PdfDocument(); // This now refers to PdfSharp.Pdf.PdfDocument

document.Info.Title = "Created with PdfSharp";

PdfPage page = document.AddPage();

XGraphics gfx = XGraphics.FromPdfPage(page);

gfx.DrawString("This is a PDF document.",

new XFont("Verdana", 20, XFontStyle.Bold),

XBrushes.Black,

new XRect(0, 0, page.Width, page.Height),

XStringFormats.Center);

document.Save(filename + ".pdf");

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

}

}

}

ExcelDocument.cs

using OfficeOpenXml;

using System.IO;

namespace FactoryMethodPattern

{

public class ExcelDocument : IDocument

{

public void Create(string filename)

{

ExcelPackage.LicenseContext = LicenseContext.NonCommercial;

using (ExcelPackage package = new ExcelPackage())

{

var sheet = package.Workbook.Worksheets.Add("Sheet1");

sheet.Cells[1, 1].Value = "This is an Excel document.";

File.WriteAllBytes(filename + ".xlsx", package.GetAsByteArray());

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

}

}

}

}

DocumentFacotry.cs

namespace FactoryMethodPattern

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

WordFactory.cs

namespace FactoryMethodPattern

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

PdfFactory.cs

namespace FactoryMethodPattern

{

public class PdfFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocumentCreator();

}

}

}

ExcelFactory.cs

namespace FactoryMethodPattern

{

public class ExcelFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

}

Program.cs

using System;

using System.Text; // Needed for Encoding

namespace FactoryMethodPattern

{

class Program

{

static void Main(string[] args)

{

Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

Console.WriteLine("Select document type to create: Word / PDF / Excel");

string input = Console.ReadLine()?.Trim().ToLower();

DocumentFactory factory = input switch

{

"word" => new WordFactory(),

"pdf" => new PdfFactory(),

"excel" => new ExcelFactory(),

\_ => throw new Exception("Invalid type")

};

Console.Write("Enter filename (without extension): ");

string filename = Console.ReadLine();

IDocument document = factory.CreateDocument();

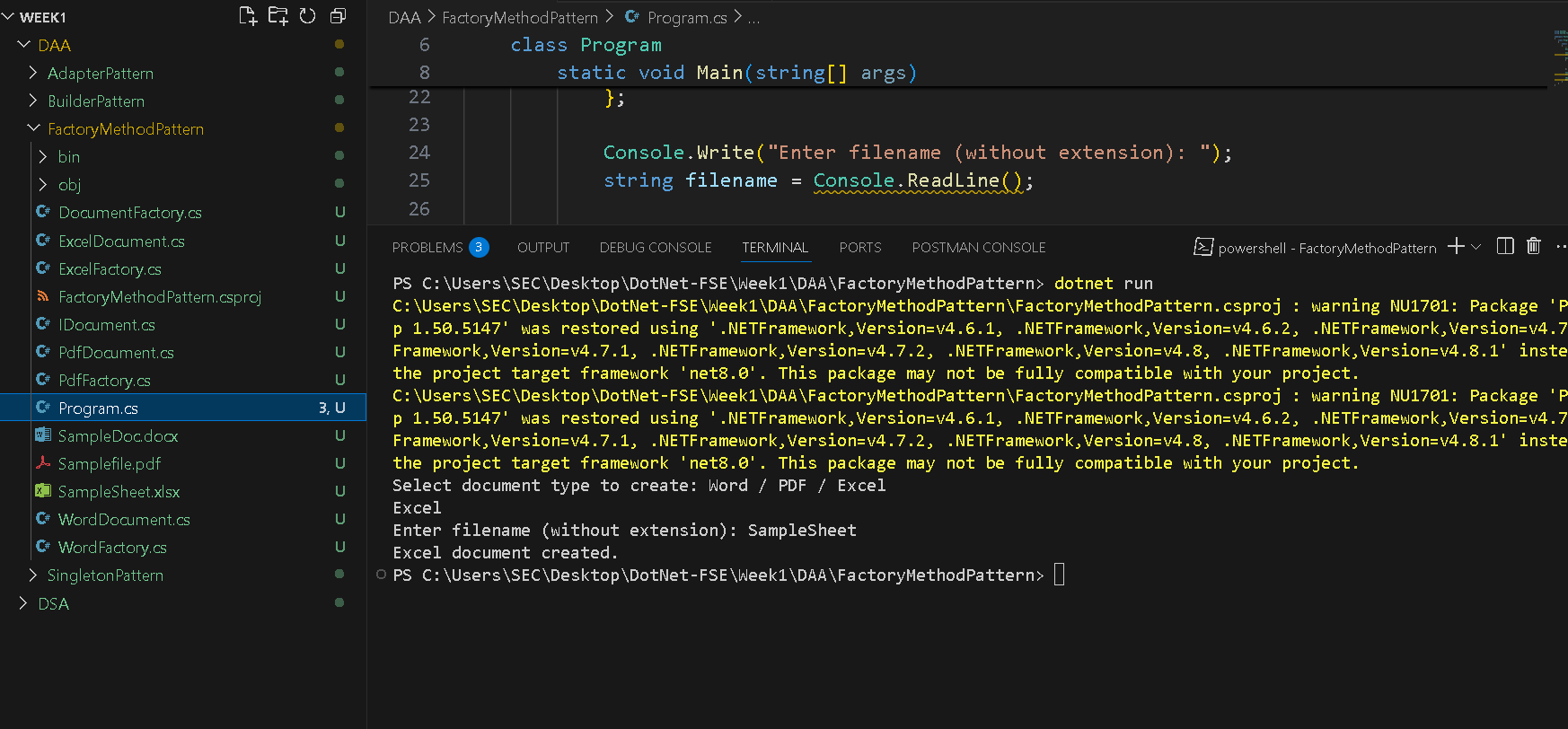
document.Create(filename);

}

}

}

OUTPUT SCREENSHOT:



**Exercise 3: Implementing the Builder Pattern**

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.

Program.cs

// See https://aka.ms/new-console-template for more information

// Console.WriteLine("Hello, World!");

using System;

namespace BuilderPattern

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Builder Pattern - Computer Configurations\n");

// Basic Computer (only CPU and RAM)

var basicComputer = new Computer.Builder()

.SetCPU("Intel i3")

.SetRAM("8GB")

.Build();

basicComputer.ShowConfiguration();

// Gaming Computer (Full config)

var gamingComputer = new Computer.Builder()

.SetCPU("AMD Ryzen 9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGPU("NVIDIA RTX 3080")

.Build();

gamingComputer.ShowConfiguration();

// Office Computer (no GPU)

var officeComputer = new Computer.Builder()

.SetCPU("Intel i5")

.SetRAM("16GB")

.SetStorage("512GB SSD")

.Build();

officeComputer.ShowConfiguration();

}

}

}

Computer.cs

namespace BuilderPattern

{

public class Computer

{

// Properties of the Computer

public string CPU { get; }

public string RAM { get; }

public string Storage { get; }

public string GPU { get; }

// Private constructor accepts the builder

private Computer(Builder builder)

{

CPU = builder.CPU;

RAM = builder.RAM;

Storage = builder.Storage;

GPU = builder.GPU;

}

// Static nested Builder class

public class Builder

{

// Properties with public setters

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GPU { get; private set; }

// Builder methods for setting each property

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;

}

// Final build method returns the fully built Computer

public Computer Build()

{

return new Computer(this);

}

}

// method to display computer configuration

public void ShowConfiguration()

{

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

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

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

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

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

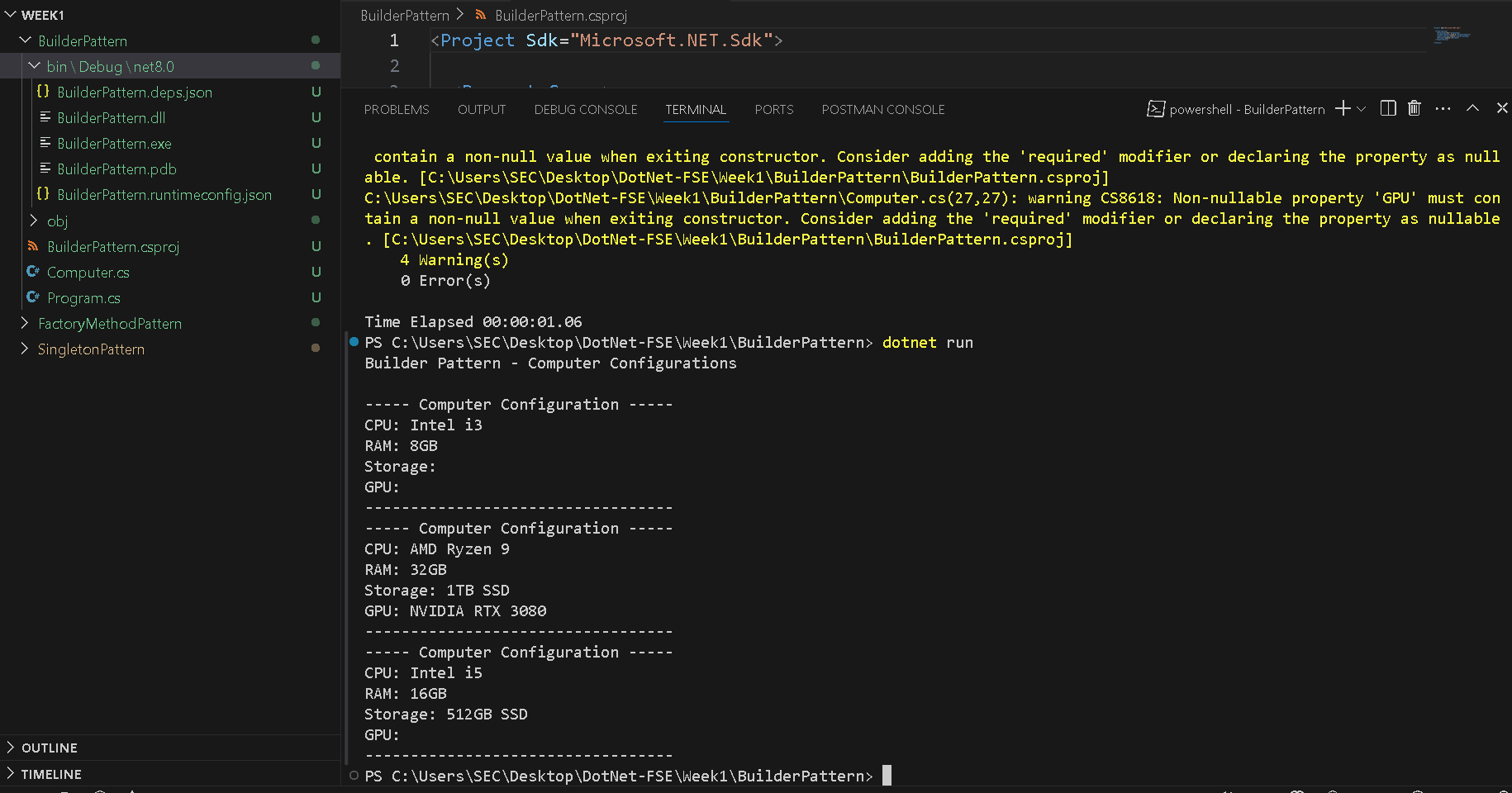
Console.WriteLine("----------------------------------");

}

}

}

OUTPUT SCREENSHOT:



**Exercise 4: Implementing the Adapter Pattern**

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.

PaymentProcessor.cs

namespace AdapterPattern

{

public interface IPaymentProcessor

{

void ProcessPayment(double amount);

}

}

GPayGateway.cs

namespace AdapterPattern

{

public class GPayGateway

{

public void PayNow(double amount)

{

Console.WriteLine($"GPay: Paid ₹{amount} successfully via UPI.");

}

}

}

PayPalGateway.cs

namespace AdapterPattern

{

public class PayPalGateway

{

public void MakeTransaction(double amount)

{

Console.WriteLine($"PayPal: Transacting ${amount}...");

}

}

}

RazorpayGateway.cs

namespace AdapterPattern

{

public class RazorpayGateway

{

public void SendMoney(double amount)

{

Console.WriteLine($"Razorpay: Sending ₹{amount}...");

}

}

}

GPayAdapter.cs

namespace AdapterPattern

{

public class GPayAdapter : IPaymentProcessor

{

private readonly GPayGateway \_gpay;

public GPayAdapter(GPayGateway gpay)

{

\_gpay = gpay;

}

public void ProcessPayment(double amount)

{

\_gpay.PayNow(amount);

}

}

}

PayPalAdapter.cs

namespace AdapterPattern

{

public class PayPalAdapter : IPaymentProcessor

{

private readonly PayPalGateway \_paypal;

public PayPalAdapter(PayPalGateway paypal)

{

\_paypal = paypal;

}

public void ProcessPayment(double amount)

{

\_paypal.MakeTransaction(amount);

}

}

}

RazorpayAdapter.cs

namespace AdapterPattern

{

public class RazorpayAdapter : IPaymentProcessor

{

private readonly RazorpayGateway \_razorpay;

public RazorpayAdapter(RazorpayGateway razorpay)

{

\_razorpay = razorpay;

}

public void ProcessPayment(double amount)

{

\_razorpay.SendMoney(amount);

}

}

}

Program.cs

// See https://aka.ms/new-console-template for more information

// Console.WriteLine("Hello, World!");

using System;

namespace AdapterPattern

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Adapter Pattern - Payment Processing System");

Console.WriteLine("Available gateways: razorpay, paypal, gpay");

Console.Write("Enter payment gateway: ");

string gateway = Console.ReadLine()?.ToLower();

Console.Write("Enter amount: ");

bool isValidAmount = double.TryParse(Console.ReadLine(), out double amount);

if (!isValidAmount || amount <= 0)

{

Console.WriteLine("Invalid amount entered.");

return;

}

IPaymentProcessor processor = null;

switch (gateway)

{

case "razorpay":

processor = new RazorpayAdapter(new RazorpayGateway());

break;

case "paypal":

processor = new PayPalAdapter(new PayPalGateway());

break;

case "gpay":

processor = new GPayAdapter(new GPayGateway());

break;

default:

Console.WriteLine("Unsupported payment gateway.");

return;

}

processor.ProcessPayment(amount);

}

}

}

OUTPUT SCREENSHOT:

