**Week-1 Hands On Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Logger.cs**

using System;

public class Logger

{

private static Logger \_instance;

private Logger()

{

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

}

public static Logger GetInstance()

{

if (\_instance == null)

{

\_instance = new Logger();

}

return \_instance;

}

public void Log(string message)

{

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

}

}

**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

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

Logger logger2 = Logger.GetInstance();

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

if (logger1 == logger2)

{

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

}

else

{

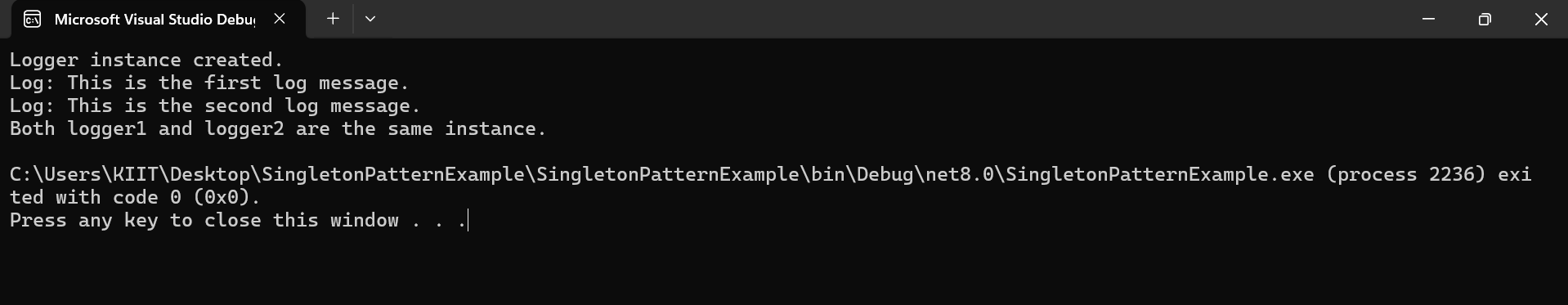
Console.WriteLine("Different instances - Singleton pattern failed.");

}

}

}

**ScreenShot of Exercise-1 Output**



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

**WordDocument.cs**

using System;

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Word Document Opened");

}

}

**IDocument.cs**

public interface IDocument

{

void Open();

}

**PdfDocumentFactory.cs**

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

**WordDocumentFactory.cs**

public class WordDocumentFactory :DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

**DocumentFactory.cs**

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

**ExcelDocument.cs**

using System;

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Excel Document Opened");

}

}

**PdfDocument.cs**

using System;

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("PDF Document Opened");

}

}

**Program.cs**

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();

}

}

**ExcelDocumentFactory.cs**

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

**Screenshot of Exercise-2 Output**



**Exercise 3: Implementing the Builder Pattern**

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);

}

}

}

**Computer.cs**

using System;

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; }

Continue……..

**ScreenShot of Exercise-3 Output**



**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

Computer gamingPC = new Computer.Builder()

.SetCPU("Intel Core i9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGPU("NVIDIA RTX 4090")

.Build();

gamingPC.DisplayConfig();

Console.WriteLine();

Computer officePC = new Computer.Builder()

.SetCPU("Intel Core i5")

.SetRAM("16GB")

.SetStorage("512GB SSD")

.Build();

officePC.DisplayConfig();

}

}

**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.cs**

public interface PaymentProcessor

{ void ProcessPayment(double amount);

}

**PayPalGateway.cs**

using System;

public class PayPalGateway

{ public void MakePayment(double amount)

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

} }

**Exercise 5: Implementing the Decorator Pattern**

**PayPalAdapter.cs**

public class PayPalAdapter : PaymentProcessor

{ private PayPalGateway \_paypal = new PayPalGateway();

public void ProcessPayment(double amount)

{ paypal.MakePayment(amount);

} }

**Program.cs**

using System;

class Program

{ static void Main(string[] args)

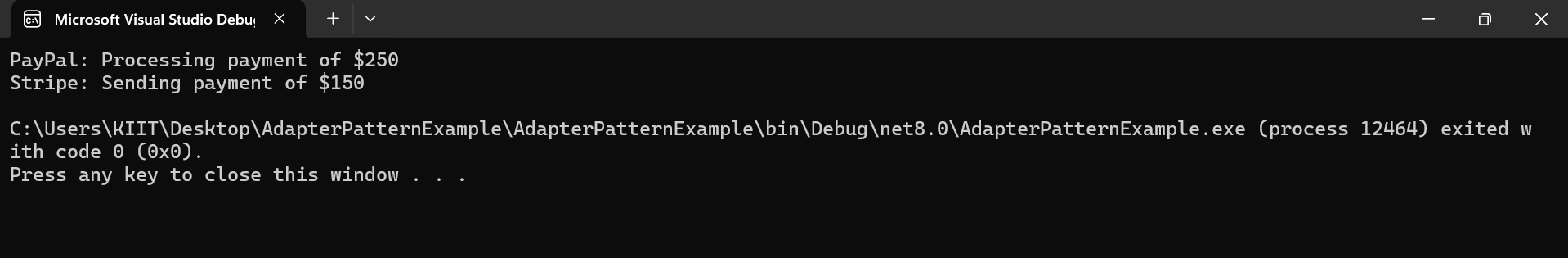
{ PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.ProcessPayment(250.00);

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.ProcessPayment(150.00); } }

**ScreenShot of Exercise-4 Output**



**StripeAdapter.cs**

public class StripeAdapter : PaymentProcessor

{ private StripeGateway \_stripe = new StripeGateway();

public void ProcessPayment(double amount)

{ \_stripe.SendPayment(amount);

}

}

**StripeGateway.cs**

using System;

public class StripeGateway

{ public void SendPayment(double amount)

{ Console.WriteLine($"Stripe: Sending payment of ${amount}");

}

}

**EmailNotifier.cs**

using System;

public class EmailNotifier : Notifier

{ public void Send(string message)

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

}

}

**Notifier.cs**

public interface Notifier

{

void Send(string message);

}

**SlackNotifierDecorator.cs**

using System;

public class SlackNotifierDecorator : NotifierDecorator

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

public override void Send(string message)

{ base.Send(message);

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

}

}

**SMSNotifierDecorator.cs**

using System;

public class SMSNotifierDecorator : NotifierDecorator

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

public override void Send(string message)

{ base.Send(message);

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

}

}

**NotifierDecorator.cs**

public abstract class NotifierDecorator : Notifier

{ protected Notifier \_notifier;

public NotifierDecorator(Notifier notifier)

{ \_notifier = notifier;

}

public virtual void Send(string message)

{ \_notifier.Send(message);

}

}

**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

Notifier notifier = new EmailNotifier(); notifier = new SMSNotifierDecorator(notifier);

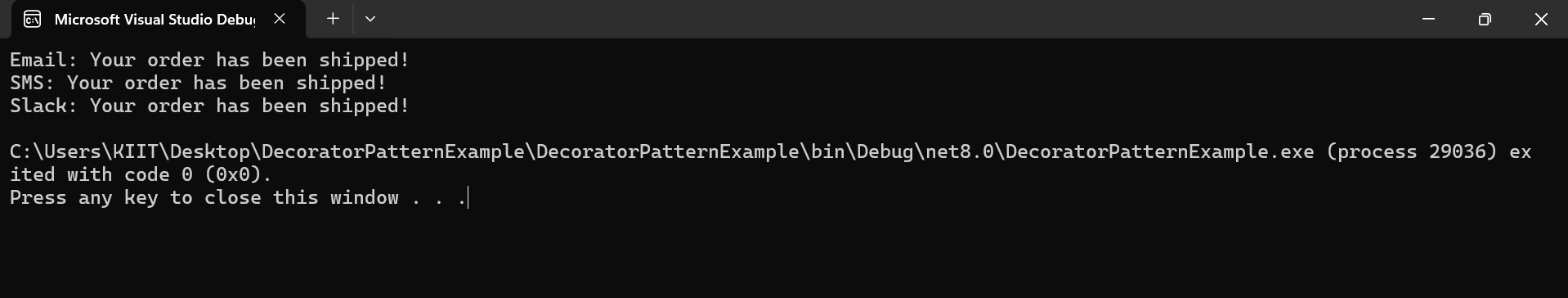
notifier = new SlackNotifierDecorator(notifier);

notifier.Send("Your order has been shipped!");

}

}

**ScreenShot of Exercise-5 Output**



**Exercise 6: Implementing the Proxy Pattern**

**ProxyImage.cs**

using System;

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);

}

\_realImage.Display();

}

}

**RealImage.cs**

using System;

public class RealImage : Image

{ private string \_fileName;

public RealImage(string fileName)

{ \_fileName = fileName;

LoadFromRemoteServer();

}

private void LoadFromRemoteServer()

{

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

}

public void Display()

{

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

}

}

**Image.cs**

public interface Image

{

void Display();

}

**Program.cs**

using System;

class Program

{ static void Main(string[] args)

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

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

Console.WriteLine("First call:");

image1.Display();

Console.WriteLine("\nSecond call:");

image1.Display();

Console.WriteLine("\nNew image:");

image2.Display();

}

}

**ScreenShot of Exercise-6 Output**



**Exercise 7: Implementing the Observer Pattern**

**IStock.cs**

public interface IStock

{ void Register(IObserver observer);

void Deregister(IObserver observer);

void NotifyObservers();

void SetPrice(double price);

}

**IObserver.cs**

public interface IObserver

{ void Update(string stockName, double price);

}

**WebApp.cs**

using System;

public class WebApp : IObserver

{

public void Update(string stockName, double price)

{

Console.WriteLine($"[WebApp] {stockName} updated to ${price}");

}

}

**MobileApp.cs**

using System;

public class MobileApp : IObserver

{

public void Update(string stockName, double price)

{

Console.WriteLine($"[MobileApp] {stockName} updated to ${price}");

}

}

**StockMarket.cs**

using System;

using System.Collections.Generic;

public class StockMarket : IStock

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

private string \_stockName;

private double \_price;

public StockMarket(string stockName)

{ \_stockName = stockName;

}

public void Register(IObserver observer)

{ \_observers.Add(observer);

}

public void Deregister(IObserver observer)

{ \_observers.Remove(observer);

}

public void NotifyObservers()

{ foreach (var observer in \_observers)

{ observer.Update(\_stockName, \_price);

}

}

public void SetPrice(double price)

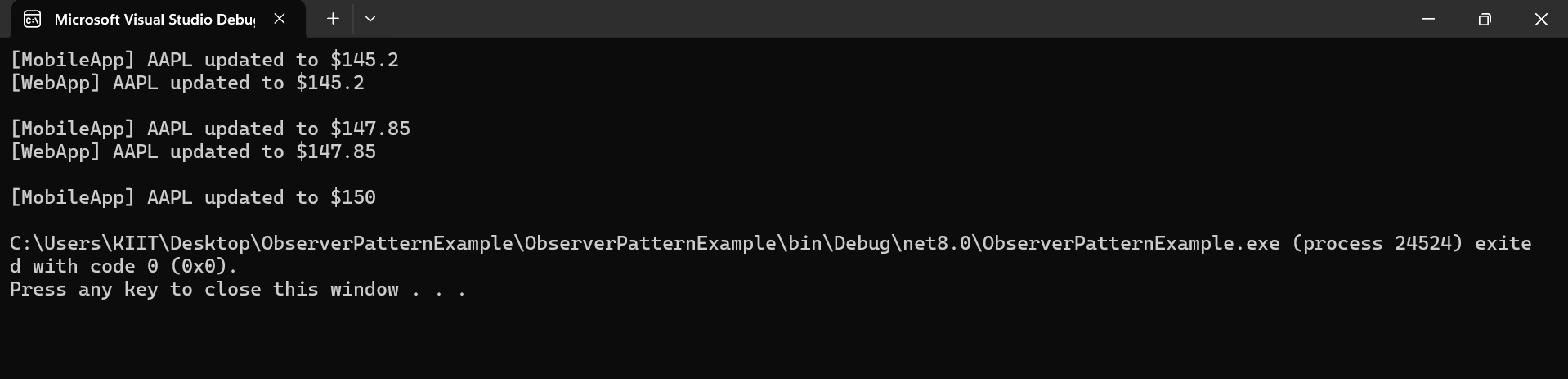
{ \_price = price;

NotifyObservers();

}

}

**ScreenShot of Exercise-7 Output**



**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

StockMarket stockMarket = new StockMarket("AAPL");

IObserver mobileApp = new MobileApp();

IObserver webApp = new WebApp();

stockMarket.Register(mobileApp);

stockMarket.Register(webApp);

stockMarket.SetPrice(145.20);

Console.WriteLine();

stockMarket.SetPrice(147.85);

Console.WriteLine();

stockMarket.Deregister(webApp);

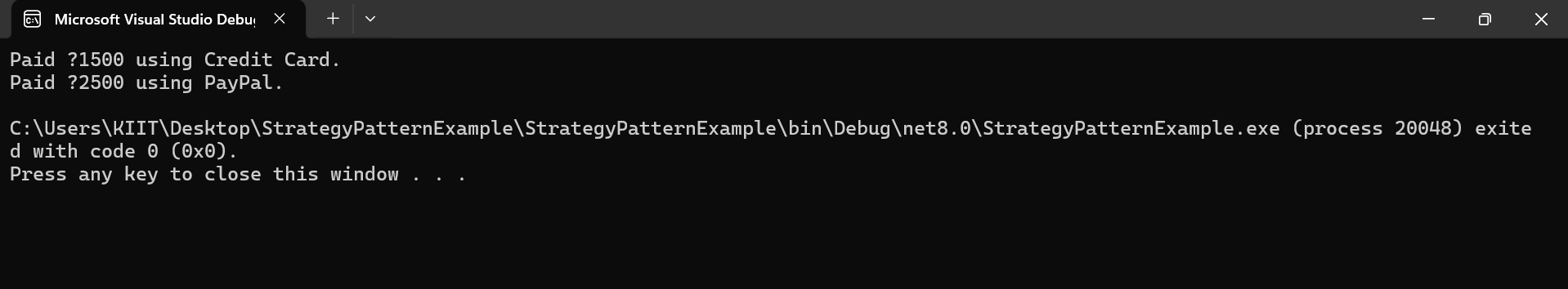
stockMarket.SetPrice(150.00);

}

}

**Exercise 8: Implementing the Strategy Pattern**

**ScreenShot of Exercise-8 Output**



**CreditCardPayment.cs**

using System;

public class CreditCardPayment : IPaymentStrategy

{ public void Pay(double amount)

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

}

}

**PaymentContext.cs**

public class PaymentContext

{ private IPaymentStrategy \_paymentStrategy;

public void SetPaymentStrategy(IPaymentStrategy strategy)

{ \_paymentStrategy = strategy;

}

public void ExecutePayment(double amount)

{ \_paymentStrategy.Pay(amount);

}

}

**Program.cs**

using System;

class Program

{ static void Main(string[] args)

{

PaymentContext context = new PaymentContext();

context.SetPaymentStrategy(new CreditCardPayment());

context.ExecutePayment(1500.00);

context.SetPaymentStrategy(new PayPalPayment());

context.ExecutePayment(2500.00);

}

}

**PayPalPayment.cs**

using System;

public class PayPalPayment : IPaymentStrategy

{ public void Pay(double amount)

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

}

}

**IPaymentStrategy.cs**

public interface IPaymentStrategy

{ void Pay(double amount);

}

**Exercise 9: Implementing the Command Pattern**

**RemoteControl.cs**

public class RemoteControl

{ private ICommand \_command;

public void SetCommand(ICommand command)

{ \_command = command;

}

public void PressButton()

{ \_command.Execute();

}

}

**LightOffCommand.cs**

public class LightOffCommand : ICommand

{ private Light \_light;

public LightOffCommand(Light light)

{ \_light = light;

}

public void Execute()

{ \_light.TurnOff();

}

}

**LightOnCommand.cs**

public class LightOnCommand : ICommand

{ private Light \_light;

public LightOnCommand(Light light)

{ \_light = light;

}

public void Execute()

{ \_light.TurnOn();

}

}

**Light.cs**

using System;

public class Light

{ public void TurnOn()

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

}

public void TurnOff()

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

}

}

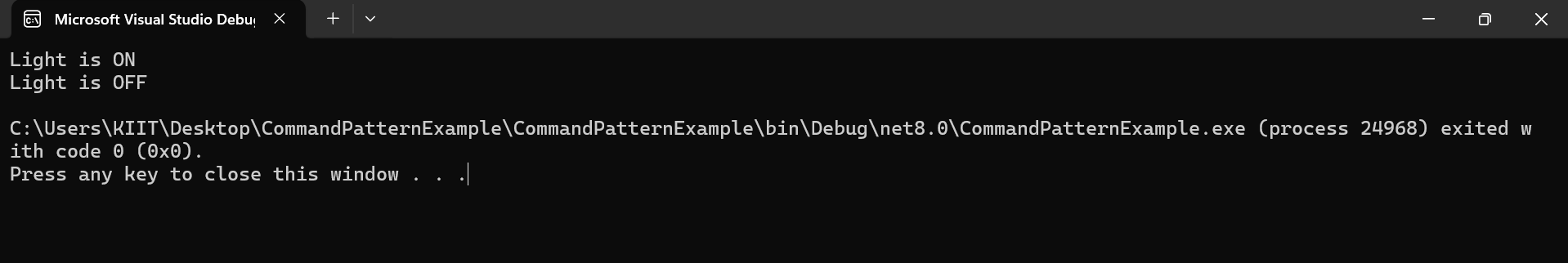
**ICommand.cs**

public interface ICommand

{ void Execute();

}

**ScreenShot of Exercise-9 Output**



**Program.cs**

using System;

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();

}

}

**Exercise 10: Implementing the MVC Pattern**

**Program.cs**

using System;

class Program

{ static void Main(string[] args)

{ Student student = new Student();

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.SetStudentName("Jyotim Mandi");

controller.SetStudentId("S12345");

controller.SetStudentGrade("A");

controller.UpdateView();

controller.SetStudentGrade("A+");

Console.WriteLine("\nAfter Grade Update:");

controller.UpdateView();

}

}

**StudentController.cs**

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);

}

}

**StudentView.cs**

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}");

}

}

**Student.cs**

public class Student

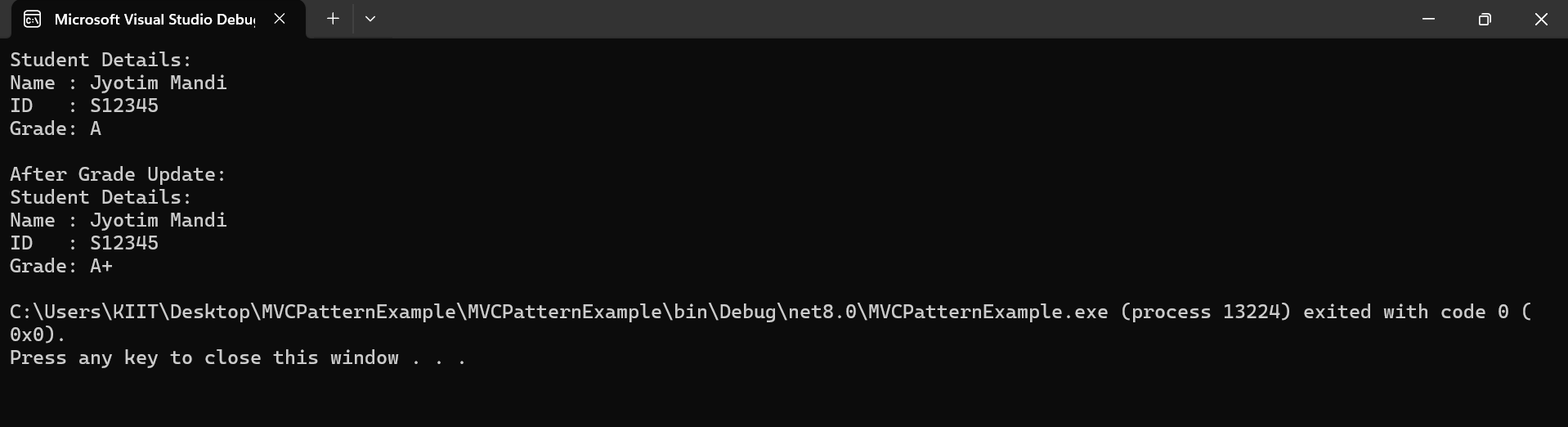
{ public string Name { get; set; }

public string Id { get; set; }

public string Grade { get; set; }

}

**ScreenShot of Exercise-10 Output**



**Exercise 11: Implementing Dependency Injection**

**Program.cs**

using System;

class Program

{ static void Main(string[] args)

{

ICustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.GetCustomerInfo(1);

service.GetCustomerInfo(3);

}

}

**CustomerRepositoryImpl.cs**

using System;

using System.Collections.Generic;

public class CustomerRepositoryImpl : ICustomerRepository

{ private Dictionary<int, Customer> \_customers = new Dictionary<int, Customer>

{

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

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

};

public Customer FindCustomerById(int id)

{ return \_customers.ContainsKey(id) ? \_customers[id] : null;

}

}

**CustomerService.cs**

using System;

public class CustomerService

{ private readonly ICustomerRepository \_repository;

public CustomerService(ICustomerRepository repository)

{ \_repository = repository;

}

public void GetCustomerInfo(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.");

}

}

}

**Customer.cs**

public class Customer

{ public int Id { get; set; }

public string Name { get; set; }

}

**ICustomerRepository.cs**

public interface ICustomerRepository

{ Customer FindCustomerById(int id);

}

**ScreenShot of Exercise-11 Output**

