Lab Exercise 1: Implementing the Strategy Pattern

Exercise:

Implement the Strategy Pattern to create a payment system where the client can choose different payment methods (e.g., Credit Card, PayPal, Bitcoin).

Solution:

```csharp
public interface IPaymentStrategy
{
    void Pay(double amount);
}

public class CreditCardPayment : IPaymentStrategy
{
    public void Pay(double amount)
    {
        Console.WriteLine($"Paid {amount} using Credit Card.");
    }
}

public class PayPalPayment : IPaymentStrategy
{
    public void Pay(double amount)
    {
        Console.WriteLine($"Paid {amount} using PayPal.");
    }
}

public class BitcoinPayment : IPaymentStrategy
{
    public void Pay(double amount)
```

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

public class PaymentContext
{
    private IPaymentStrategy _paymentStrategy;

    public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)
    {
        _paymentStrategy = paymentStrategy;
    }

    public void Pay(double amount)
    {
        _paymentStrategy.Pay(amount);
    }
}

// Usage
var paymentContext = new PaymentContext();
paymentContext.SetPaymentStrategy(new CreditCardPayment());
paymentContext.Pay(100.0);

paymentContext.SetPaymentStrategy(new PayPalPayment());
paymentContext.Pay(200.0);

paymentContext.SetPaymentStrategy(new BitcoinPayment());
paymentContext.Pay(300.0);
```

Lab Exercise 2: Observer Pattern for Stock Prices

Exercise:

Create an Observer Pattern to notify different investors when a stock price changes.

Solution:

```csharp
public interface IInvestor
{
    void Update(string stock, decimal price);
}

public class Stock
{
    private decimal _price;
    private readonly List<IInvestor> _investors = new List<IInvestor>();
    public string Symbol { get; }

    public Stock(string symbol)
    {
        Symbol = symbol;
    }

    public void Attach(IInvestor investor)
    {
        _investors.Add(investor);
    }

    public void Detach(IInvestor investor)
    {
        _investors.Remove(investor);
```

```
    }

    public decimal Price
    {
        get => _price;
        set
        {
            _price = value;
            Notify();
        }
    }

    private void Notify()
    {
        foreach (var investor in _investors)
        {
            investor.Update(Symbol, _price);
        }
    }
}

public class Investor : IInvestor
{
    private readonly string _name;

    public Investor(string name)
    {
        _name = name;
    }

    public void Update(string stock, decimal price)
```

```
    {
        Console.WriteLine($"{_name} notified. {stock} is now {price:C}");
    }
}

// Usage
var appleStock = new Stock("AAPL");
var investor1 = new Investor("Alice");
var investor2 = new Investor("Bob");

appleStock.Attach(investor1);
appleStock.Attach(investor2);

appleStock.Price = 150.00m;
appleStock.Price = 155.00m;
```

Lab Exercise 3: Command Pattern for Remote Control

Exercise:

Implement the Command Pattern to simulate a remote control that can turn on and off a light.

Solution:

```csharp
public interface ICommand
{
    void Execute();
}

public class Light
{
    public void On()
```

```
    {
        Console.WriteLine("Light is on.");
    }

    public void Off()
    {
        Console.WriteLine("Light is off.");
    }
}

public class LightOnCommand : ICommand
{
    private readonly Light _light;

    public LightOnCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.On();
    }
}

public class LightOffCommand : ICommand
{
    private readonly Light _light;

    public LightOffCommand(Light light)
    {
```

```
        _light = light;
    }

    public void Execute()
    {
        _light.Off();
    }
}

public class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        _command.Execute();
    }
}

// Usage
var light = new Light();
var lightOn = new LightOnCommand(light);
var lightOff = new LightOffCommand(light);

var remote = new RemoteControl();
remote.SetCommand(lightOn);
```

```
remote.PressButton();

remote.SetCommand(lightOff);
remote.PressButton();
```

Lab Exercise 4: Chain of Responsibility for Authentication

Exercise:

Implement a Chain of Responsibility Pattern to handle authentication steps (e.g., username/password, two-factor authentication).

Solution:

```csharp
public abstract class AuthenticationHandler
{
    protected AuthenticationHandler _nextHandler;

    public void SetNextHandler(AuthenticationHandler nextHandler)
    {
        _nextHandler = nextHandler;
    }

    public abstract void HandleRequest(string username, string password);
}

public class UsernamePasswordHandler : AuthenticationHandler
{
    public override void HandleRequest(string username, string password)
    {
        if (username == "user" && password == "pass")
        {
```

```
            Console.WriteLine("Username and Password authenticated.");
            _nextHandler?.HandleRequest(username, password);
        }
        else
        {
            Console.WriteLine("Invalid Username or Password.");
        }
    }
}

public class TwoFactorAuthenticationHandler : AuthenticationHandler
{
    public override void HandleRequest(string username, string password)
    {
        Console.WriteLine("Two-factor authentication successful.");
        _nextHandler?.HandleRequest(username, password);
    }
}

// Usage
var authHandler = new UsernamePasswordHandler();
var twoFactorHandler = new TwoFactorAuthenticationHandler();

authHandler.SetNextHandler(twoFactorHandler);

authHandler.HandleRequest("user", "pass");
```

Lab Exercise 5: Mediator Pattern for Chat Room

Exercise:

Implement a Mediator Pattern for a chat room where users can send messages to each other through a central chat room.

Solution:

```csharp
public class ChatRoom
{
    private readonly Dictionary<string, User> _users = new Dictionary<string, User>();

    public void Register(User user)
    {
        if (!_users.ContainsKey(user.Name))
        {
            _users[user.Name] = user;
            user.ChatRoom = this;
        }
    }

    public void Send(string from, string to, string message)
    {
        var user = _users[to];
        user?.Receive(from, message);
    }
}

public class User
{
    public string Name { get; }
    public ChatRoom ChatRoom { get; set; }

    public User(string name)
```

```
    {
        Name = name;
    }

    public void Send(string to, string message)
    {
        ChatRoom.Send(Name, to, message);
    }

    public void Receive(string from, string message)
    {
        Console.WriteLine($"{from} to {Name}: {message}");
    }
}

// Usage
var chatRoom = new ChatRoom();
var user1 = new User("Alice");
var user2 = new User("Bob");

chatRoom.Register(user1);
chatRoom.Register(user2);

user1.Send("Bob", "Hello Bob!");
user2.Send("Alice", "Hi Alice!");
```

Lab Exercise 6: Strategy Pattern for Tax Calculation

Exercise:

Create a Strategy Pattern for calculating tax based on different regions.

Solution:

```csharp
public interface ITaxStrategy
{
    double CalculateTax(double amount);
}

public class USTaxStrategy : ITaxStrategy
{
    public double CalculateTax(double amount)
    {
        return amount * 0.1;
    }
}

public class EUTaxStrategy : ITaxStrategy
{
    public double CalculateTax(double amount)
    {
        return amount * 0.2;
    }
}

public class TaxContext
{
    private ITaxStrategy _taxStrategy;

    public void SetTaxStrategy(ITaxStrategy taxStrategy)
    {
        _taxStrategy = taxStrategy;
    }
```

```
    public double CalculateTax(double amount)
    {
        return _taxStrategy.CalculateTax(amount);
    }
}

// Usage
var taxContext = new TaxContext();

taxContext.SetTaxStrategy(new USTaxStrategy());
Console.WriteLine(taxContext.CalculateTax(1000)); // Output: 100

taxContext.SetTaxStrategy(new EUTaxStrategy());
Console.WriteLine(taxContext.CalculateTax(1000)); // Output: 200
```

Lab Exercise 7: Observer Pattern for Weather Station

Exercise:

Implement an Observer Pattern for a weather station that notifies different displays (e.g., phone display, TV display) when the temperature changes.

Solution:

```csharp
public interface IDisplay
{
    void Update(double temperature);
}

public class WeatherStation
{
```

```
    private double _temperature;
    private readonly List<IDisplay> _displays = new List<IDisplay>();

    public void Attach(IDisplay display)
    {
        _displays.Add(display);
    }

    public void Detach(IDisplay display)
    {
        _displays.Remove(display);
    }

    public double Temperature
    {
        get => _temperature;
        set
        {
            _temperature = value;
            Notify();
        }
    }

    private void Notify()
    {
        foreach (var display in _displays)
        {
            display.Update(_temperature);
        }
    }
}
```

```
public class PhoneDisplay : IDisplay
{
    public void Update(double temperature)
    {
        Console.WriteLine($"Phone display: Temperature is {temperature} degrees.");
    }
}

public class TVDisplay : IDisplay
{
    public void Update(double temperature)
    {
        Console.WriteLine($"TV display: Temperature is {temperature} degrees

.");
    }
}

// Usage
var weatherStation = new WeatherStation();
var phoneDisplay = new PhoneDisplay();
var tvDisplay = new TVDisplay();

weatherStation.Attach(phoneDisplay);
weatherStation.Attach(tvDisplay);

weatherStation.Temperature = 25.0;
weatherStation.Temperature = 30.0;
```

Lab Exercise 8: Command Pattern for Text Editor

Exercise:

Implement the Command Pattern to handle text editing operations like AddText, RemoveText, and Undo.

Solution:

```csharp
public interface ICommand
{
    void Execute();
    void Undo();
}

public class TextEditor
{
    public string Text { get; private set; } = string.Empty;

    public void AddText(string text)
    {
        Text += text;
    }

    public void RemoveText(int length)
    {
        Text = Text.Substring(0, Text.Length - length);
    }
}

public class AddTextCommand : ICommand
{
    private readonly TextEditor _textEditor;
```

```
    private readonly string _text;

    public AddTextCommand(TextEditor textEditor, string text)
    {
        _textEditor = textEditor;
        _text = text;
    }

    public void Execute()
    {
        _textEditor.AddText(_text);
    }

    public void Undo()
    {
        _textEditor.RemoveText(_text.Length);
    }
}

public class CommandManager
{
    private readonly Stack<ICommand> _commands = new Stack<ICommand>();

    public void ExecuteCommand(ICommand command)
    {
        command.Execute();
        _commands.Push(command);
    }

    public void Undo()
    {
```

```
        if (_commands.Count > 0)
        {
            var command = _commands.Pop();
            command.Undo();
        }
    }
}

// Usage
var textEditor = new TextEditor();
var commandManager = new CommandManager();

var addTextCommand = new AddTextCommand(textEditor, "Hello, World!");

commandManager.ExecuteCommand(addTextCommand);
Console.WriteLine(textEditor.Text); // Output: Hello, World!

commandManager.Undo();
Console.WriteLine(textEditor.Text); // Output: (empty string)
```

Lab Exercise 9: Chain of Responsibility for Support Ticket System

Exercise:

Implement a Chain of Responsibility Pattern to handle different levels of customer support tickets (e.g., Level 1, Level 2, Level 3).

Solution:

```csharp
public abstract class SupportHandler
{
    protected SupportHandler _nextHandler;
```

```
    public void SetNextHandler(SupportHandler nextHandler)
    {
        _nextHandler = nextHandler;
    }

    public abstract void HandleRequest(string issue);
}

public class LevelOneSupport : SupportHandler
{
    public override void HandleRequest(string issue)
    {
        if (issue == "Basic Issue")
        {
            Console.WriteLine("Level One Support handled the issue.");
        }
        else if (_nextHandler != null)
        {
            _nextHandler.HandleRequest(issue);
        }
    }
}

public class LevelTwoSupport : SupportHandler
{
    public override void HandleRequest(string issue)
    {
        if (issue == "Intermediate Issue")
        {
            Console.WriteLine("Level Two Support handled the issue.");
```

```
        }
        else if (_nextHandler != null)
        {
            _nextHandler.HandleRequest(issue);
        }
    }
}

public class LevelThreeSupport : SupportHandler
{
    public override void HandleRequest(string issue)
    {
        if (issue == "Complex Issue")
        {
            Console.WriteLine("Level Three Support handled the issue.");
        }
        else
        {
            Console.WriteLine("Issue could not be handled.");
        }
    }
}

// Usage
var levelOne = new LevelOneSupport();
var levelTwo = new LevelTwoSupport();
var levelThree = new LevelThreeSupport();

levelOne.SetNextHandler(levelTwo);
levelTwo.SetNextHandler(levelThree);
```

```
levelOne.HandleRequest("Basic Issue");

levelOne.HandleRequest("Intermediate Issue");

levelOne.HandleRequest("Complex Issue");

levelOne.HandleRequest("Unknown Issue");
```

Lab Exercise 10: Mediator Pattern for Air Traffic Control

Exercise:

Implement a Mediator Pattern for an air traffic control system where multiple planes communicate with the control tower to land.

Solution:

```csharp
public class ControlTower
{
    private readonly List<Plane> _planes = new List<Plane>();

    public void Register(Plane plane)
    {
        if (!_planes.Contains(plane))
        {
            _planes.Add(plane);
            plane.ControlTower = this;
        }
    }

    public void RequestLanding(Plane plane)
    {
        foreach (var p in _planes)
        {
            if (p != plane)
```

```
            {
                p.ReceiveMessage($"Plane {plane.Id} is landing.");
            }
        }
    }
}

public class Plane
{
    public string Id { get; }
    public ControlTower ControlTower { get; set; }

    public Plane(string id)
    {
        Id = id;
    }

    public void RequestLanding()
    {
        ControlTower.RequestLanding(this);
    }

    public void ReceiveMessage(string message)
    {
        Console.WriteLine($"Plane {Id} received message: {message}");
    }
}

// Usage
var controlTower = new ControlTower();
var plane1 = new Plane("A1");
```

```
var plane2 = new Plane("B2");

controlTower.Register(plane1);
controlTower.Register(plane2);

plane1.RequestLanding();
```

Lab Exercise 11: Strategy Pattern for Data Compression

Exercise:

Implement the Strategy Pattern to create a data compression system where the client can choose between different compression algorithms (e.g., ZIP, RAR, GZIP).

Solution:

```csharp
public interface ICompressionStrategy
{
    void Compress(string fileName);
}

public class ZipCompression : ICompressionStrategy
{
    public void Compress(string fileName)
    {
        Console.WriteLine($"Compressing {fileName} using ZIP.");
    }
}

public class RarCompression : ICompressionStrategy
{
    public void Compress(string fileName)
```

```
    {
        Console.WriteLine($"Compressing {fileName} using RAR.");
    }
}

public class GzipCompression : ICompressionStrategy
{
    public void Compress(string fileName)
    {
        Console.WriteLine($"Compressing {fileName} using GZIP.");
    }
}

public class CompressionContext
{
    private ICompressionStrategy _compressionStrategy;

    public void SetCompressionStrategy(ICompressionStrategy compressionStrategy)
    {
        _compressionStrategy = compressionStrategy;
    }

    public void CompressFile(string fileName)
    {
        _compressionStrategy.Compress(fileName);
    }
}

// Usage
var compressionContext = new CompressionContext();
```

```
compressionContext.SetCompressionStrategy(new ZipCompression());
compressionContext.CompressFile("file.txt");

compressionContext.SetCompressionStrategy(new RarCompression());
compressionContext.CompressFile("file.txt");

compressionContext.SetCompressionStrategy(new GzipCompression());
compressionContext.CompressFile("file.txt");
```

Lab Exercise 12: Observer Pattern for Auction System

Exercise:

Create an Observer Pattern for an auction system where bidders are notified when the auction price changes.

Solution:

```csharp
public interface IBidder
{
    void Update(decimal newPrice);
}

public class Auction
{
    private decimal _price;
    private readonly List<IBidder> _bidders = new List<IBidder>();

    public void Attach(IBidder bidder)
    {
        _bidders.Add(bidder);
    }
```

```
    public void Detach(IBidder bidder)
    {
        _bidders.Remove(bidder);
    }

    public decimal Price
    {
        get => _price;
        set
        {
            _price = value;
            Notify();
        }
    }

    private void Notify()
    {
        foreach (var bidder in _bidders)
        {
            bidder.Update(_price);
        }
    }
}

public class Bidder : IBidder
{
    private readonly string _name;

    public Bidder(string name)
    {
```

```
        _name = name;
    }

    public void Update(decimal newPrice)
    {
        Console.WriteLine($"{_name} has been notified of new price: {newPrice:C}");
    }
}

// Usage
var auction = new Auction();
var bidder1 = new Bidder("Alice");
var bidder2 = new Bidder("Bob");

auction.Attach(bidder1);
auction.Attach(bidder2);

auction.Price = 100.00m;
auction.Price = 150.00m;
```

Lab Exercise 13: Command Pattern for Media Player

Exercise:

Implement the Command Pattern to control a media player with commands like Play, Pause, and Stop.

Solution:

```csharp
public interface ICommand
{
    void Execute();
```

```
}

public class MediaPlayer
{
    public void Play()
    {
        Console.WriteLine("Playing media.");
    }

    public void Pause()
    {
        Console.WriteLine("Pausing media.");
    }

    public void Stop()
    {
        Console.WriteLine("Stopping media.");
    }
}

public class PlayCommand : ICommand
{
    private readonly MediaPlayer _mediaPlayer;

    public PlayCommand(MediaPlayer mediaPlayer)
    {
        _mediaPlayer = mediaPlayer;
    }

    public void Execute()
    {
```

```
        _mediaPlayer.Play();
    }
}

public class PauseCommand : ICommand
{
    private readonly MediaPlayer _mediaPlayer;

    public PauseCommand(MediaPlayer mediaPlayer)
    {
        _mediaPlayer = mediaPlayer;
    }

    public void Execute()
    {
        _mediaPlayer.Pause();
    }
}

public class StopCommand : ICommand
{
    private readonly MediaPlayer _mediaPlayer;

    public StopCommand(MediaPlayer mediaPlayer)
    {
        _mediaPlayer = mediaPlayer;
    }

    public void Execute()
    {
        _mediaPlayer.Stop();
```

```
    }
}

public class RemoteControl
{
    private ICommand

 _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        _command.Execute();
    }
}

// Usage
var mediaPlayer = new MediaPlayer();
var playCommand = new PlayCommand(mediaPlayer);
var pauseCommand = new PauseCommand(mediaPlayer);
var stopCommand = new StopCommand(mediaPlayer);

var remote = new RemoteControl();
remote.SetCommand(playCommand);
remote.PressButton();

remote.SetCommand(pauseCommand);
```

```
remote.PressButton();

remote.SetCommand(stopCommand);
remote.PressButton();
```

Lab Exercise 14: Chain of Responsibility for Data Validation

Exercise:

Implement a Chain of Responsibility Pattern to validate user input (e.g., check if not null, check length, check format).

Solution:

```csharp
public abstract class ValidationHandler
{
    protected ValidationHandler _nextHandler;

    public void SetNextHandler(ValidationHandler nextHandler)
    {
        _nextHandler = nextHandler;
    }

    public abstract void Validate(string input);
}

public class NotNullValidation : ValidationHandler
{
    public override void Validate(string input)
    {
        if (input != null)
        {
```

```
            _nextHandler?.Validate(input);
        }
        else
        {
            Console.WriteLine("Input cannot be null.");
        }
    }
}

public class LengthValidation : ValidationHandler
{
    public override void Validate(string input)
    {
        if (input.Length >= 5)
        {
            _nextHandler?.Validate(input);
        }
        else
        {
            Console.WriteLine("Input length must be at least 5 characters.");
        }
    }
}

public class FormatValidation : ValidationHandler
{
    public override void Validate(string input)
    {
        if (input.Contains("@"))
        {
            _nextHandler?.Validate(input);
```

```
        }
        else
        {
            Console.WriteLine("Input must contain '@' character.");
        }
    }
}

// Usage
var notNullValidation = new NotNullValidation();
var lengthValidation = new LengthValidation();
var formatValidation = new FormatValidation();

notNullValidation.SetNextHandler(lengthValidation);
lengthValidation.SetNextHandler(formatValidation);

notNullValidation.Validate("hello@example.com");
notNullValidation.Validate("short");
notNullValidation.Validate(null);
```

Lab Exercise 15: Mediator Pattern for E-commerce System

Exercise:

Implement a Mediator Pattern for an e-commerce system where different components (e.g., inventory, payment, shipping) interact through a central mediator.

Solution:

```csharp
public class EcommerceMediator
{
    public Inventory Inventory { get; set; }
```

```
    public Payment Payment { get; set; }
    public Shipping Shipping { get; set; }

    public void PlaceOrder(string item, int quantity)
    {
        if (Inventory.CheckStock(item, quantity))
        {
            if (Payment.ProcessPayment())
            {
                Shipping.ShipOrder(item, quantity);
            }
        }
    }
}

public class Inventory
{
    public bool CheckStock(string item, int quantity)
    {
        Console.WriteLine($"Checking stock for {quantity} of {item}.");
        return true; // Assume item is in stock
    }
}

public class Payment
{
    public bool ProcessPayment()
    {
        Console.WriteLine("Processing payment.");
        return true; // Assume payment is successful
    }
```

```
}

public class Shipping
{
    public void ShipOrder(string item, int quantity)
    {
        Console.WriteLine($"Shipping {quantity} of {item}.");
    }
}

// Usage
var mediator = new EcommerceMediator();
mediator.Inventory = new Inventory();
mediator.Payment = new Payment();
mediator.Shipping = new Shipping();

mediator.PlaceOrder("Laptop", 1);
```

Lab Exercise 16: Strategy Pattern for Sorting Algorithms

Exercise:

Implement the Strategy Pattern to allow a list of numbers to be sorted using different algorithms (e.g., Bubble Sort, Quick Sort).

Solution:

```csharp
public interface ISortStrategy
{
    void Sort(List<int> list);
}
```

```
public class BubbleSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        Console.WriteLine("Sorting using Bubble Sort.");
        // Implement Bubble Sort algorithm
    }
}

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        Console.WriteLine("Sorting using Quick Sort.");
        // Implement Quick Sort algorithm
    }
}

public class SortContext
{
    private ISortStrategy _sortStrategy;

    public void SetSortStrategy(ISortStrategy sortStrategy)
    {
        _sortStrategy = sortStrategy;
    }

    public void Sort(List<int> list)
    {
        _sortStrategy.Sort(list);
    }
```

```
}

// Usage
var sortContext = new SortContext();
var list = new List<int> { 5, 2, 9, 1, 5, 6 };

sortContext.SetSortStrategy(new BubbleSort());
sortContext.Sort(list);

sortContext.SetSortStrategy(new QuickSort());
sortContext.Sort(list);
```

Lab Exercise 17: Observer Pattern for News System

Exercise:

Implement an Observer Pattern to notify subscribers when new news articles are published.

Solution:

```csharp
public interface ISubscriber
{
    void Update(string article);
}

public class NewsPublisher
{
    private readonly List<ISubscriber> _subscribers = new List<ISubscriber>();
    private string _article;

    public void Attach(ISubscriber subscriber)
    {
```

```
        _subscribers.Add(subscriber);
    }

    public void Detach(ISubscriber subscriber)
    {
        _subscribers.Remove(subscriber);
    }

    public void PublishArticle(string article)
    {
        _article = article;
        Notify();
    }

    private void Notify()
    {
        foreach (var subscriber in _subscribers)
        {
            subscriber.Update(_article);
        }
    }
}

public class Subscriber : ISubscriber
{
    private readonly string _name;

    public Subscriber(string name)
    {
        _name = name;
    }
```

```
    public void Update(string article)
    {
        Console.WriteLine($"{_name} received new article: {article}");
    }
}

// Usage
var newsPublisher = new NewsPublisher();
var subscriber1 = new Subscriber("Alice");
var subscriber2 = new Subscriber("Bob");

newsPublisher.Attach(subscriber1);
newsPublisher.Attach(subscriber2);

newsPublisher.PublishArticle("Breaking News: New C# Version Released!");
```

Lab Exercise 18: Command Pattern for Smart Home System

Exercise:

Implement the Command Pattern to control various smart home devices (e.g., Lights, Thermostat, Security System).

Solution:

```csharp
public interface ICommand
{
    void Execute();
}

public class Light
```

```
{
    public void On()
    {
        Console.WriteLine("Light is on.");
    }

    public void Off()
    {
        Console.WriteLine("Light is off.");
    }
}

public class Thermostat
{
    public void SetTemperature(int temperature)
    {
        Console.WriteLine($"Setting temperature to {temperature} degrees.");
    }
}

public class SecuritySystem
{
    public void Arm()
    {
        Console.WriteLine("Security system armed.");
    }

    public void Disarm()
    {
        Console.WriteLine("Security system disarmed.");
    }
```

```
}

public class LightOnCommand : ICommand
{
    private readonly Light _light;

    public LightOnCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.On();
    }
}

public class LightOffCommand : ICommand
{
    private readonly Light _light;

    public LightOffCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.Off();
    }
}
```

```
public class SetThermostatCommand : ICommand
{
    private readonly Thermostat _thermostat;
    private readonly int _temperature;

    public SetThermostatCommand(Thermostat thermostat, int temperature)
    {
        _thermostat = thermostat;
        _temperature = temperature;
    }

    public void Execute()
    {
        _thermostat.SetTemperature(_temperature);
    }
}

public class ArmSecuritySystemCommand : ICommand
{
    private readonly SecuritySystem _securitySystem;

    public ArmSecuritySystemCommand(SecuritySystem securitySystem)
    {
        _securitySystem = securitySystem;
    }

    public void Execute()
    {
        _securitySystem.Arm();
    }
```

```
}

public class SmartHomeController
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        _command.Execute();
    }
}

// Usage
var light = new Light();
var thermostat = new Thermostat();
var securitySystem = new SecuritySystem();

var lightOnCommand = new LightOnCommand(light);
var lightOffCommand = new LightOffCommand(light);
var setThermostatCommand = new SetThermostatCommand(thermostat, 22);
var armSecuritySystemCommand = new ArmSecuritySystemCommand(securitySystem);

var smartHomeController = new SmartHomeController();
smartHomeController.SetCommand(lightOnCommand);
smartHomeController.PressButton();
```

```
smartHomeController.SetCommand(setThermostatCommand);
smartHomeController.PressButton();

smartHomeController.SetCommand(armSecuritySystemCommand);
smartHomeController.PressButton();
```

Lab Exercise 19: Chain of Responsibility for Order Processing

Exercise:

Implement a Chain of Responsibility Pattern to process orders through different stages (e.g., Validation, Payment, Shipping).

Solution:

```csharp
public abstract class OrderHandler
{
    protected OrderHandler _nextHandler;

    public void SetNextHandler(OrderHandler nextHandler)
    {
        _nextHandler = nextHandler;
    }

    public abstract void Handle(Order order);
}

public class ValidationHandler : OrderHandler
{


    public override void Handle(Order order)
```

```
    {
        if (order.IsValid)
        {
            Console.WriteLine("Order validated.");
            _nextHandler?.Handle(order);
        }
        else
        {
            Console.WriteLine("Order validation failed.");
        }
    }
}

public class PaymentHandler : OrderHandler
{
    public override void Handle(Order order)
    {
        if (order.IsPaid)
        {
            Console.WriteLine("Payment processed.");
            _nextHandler?.Handle(order);
        }
        else
        {
            Console.WriteLine("Payment failed.");
        }
    }
}

public class ShippingHandler : OrderHandler
{
```

```
    public override void Handle(Order order)
    {
        Console.WriteLine("Order shipped.");
    }
}

public class Order
{
    public bool IsValid { get; set; }
    public bool IsPaid { get; set; }
}

// Usage
var validationHandler = new ValidationHandler();
var paymentHandler = new PaymentHandler();
var shippingHandler = new ShippingHandler();

validationHandler.SetNextHandler(paymentHandler);
paymentHandler.SetNextHandler(shippingHandler);

var order = new Order { IsValid = true, IsPaid = true };
validationHandler.Handle(order);
```

Lab Exercise 20: Mediator Pattern for Chat Application

Exercise:

Implement the Mediator Pattern to manage chat communication between users in a chat application.

Solution:

```csharp
```

```
public class ChatRoom
{
    private readonly Dictionary<string, User> _users = new Dictionary<string, User>();

    public void Register(User user)
    {
        if (!_users.ContainsKey(user.Name))
        {
            _users[user.Name] = user;
            user.ChatRoom = this;
        }
    }

    public void SendMessage(string from, string to, string message)
    {
        if (_users.ContainsKey(to))
        {
            _users[to].ReceiveMessage(from, message);
        }
        else
        {
            Console.WriteLine($"User {to} not found.");
        }
    }
}

public class User
{
    public string Name { get; }
    public ChatRoom ChatRoom { get; set; }
```

```
    public User(string name)
    {
        Name = name;
    }

    public void SendMessage(string to, string message)
    {
        ChatRoom.SendMessage(Name, to, message);
    }

    public void ReceiveMessage(string from, string message)
    {
        Console.WriteLine($"{from} to {Name}: {message}");
    }
}

// Usage
var chatRoom = new ChatRoom();
var user1 = new User("Alice");
var user2 = new User("Bob");

chatRoom.Register(user1);
chatRoom.Register(user2);

user1.SendMessage("Bob", "Hello Bob!");
user2.SendMessage("Alice", "Hi Alice!");
```