# **AbstractFactory**

Abstract Factory Design Pattern là một mẫu thiết kế phần mềm thuộc nhóm creational patterns (mẫu tạo đối tượng) trong lập trình hướng đối tượng. Mẫu thiết kế này cho phép tạo ra các đối tượng liên quan nhau mà không cần xác định cụ thể lớp cụ thể của chúng.

Trong Abstract Factory, một "abstract factory" (nhà máy trừu tượng) định nghĩa một giao diện để tạo ra các đối tượng, trong khi các "concrete factory" (nhà máy cụ thể) cung cấp cách thức cụ thể để tạo ra các đối tượng. Abstract Factory cho phép chúng ta tạo ra các đối tượng mà không phụ thuộc vào lớp cụ thể của chúng, đảm bảo tính linh hoạt và ràng buộc yếu tố thay đổi.

Trong C#, để triển khai Abstract Factory Design Pattern, chúng ta sử dụng các lớp abstract để định nghĩa giao diện cho factory và các lớp concrette để triển khai factory và tạo ra các đối tượng liên quan. Các lớp abstract factory có thể có nhiều phương thức để tạo ra các đối tượng khác nhau.

Dưới đây là một ví dụ về cách triển khai Abstract Factory Design Pattern trong C#:

```csharp

// Abstract Product A

public abstract class ProductA

{

    public abstract void MethodA();

}

// Concrete Product A1

public class ProductA1 : ProductA

{

    public override void MethodA()

    {

        Console.WriteLine("Product A1 - Method A");

    }

}

// Concrete Product A2

public class ProductA2 : ProductA

{

    public override void MethodA()

    {

        Console.WriteLine("Product A2 - Method A");

    }

}

// Abstract Product B

public abstract class ProductB

{

    public abstract void MethodB();

}

// Concrete Product B1

public class ProductB1 : ProductB

{

    public override void MethodB()

    {

        Console.WriteLine("Product B1 - Method B");

    }

}

// Concrete Product B2

public class ProductB2 : ProductB

{

    public override void MethodB()

    {

        Console.WriteLine("Product B2 - Method B");

    }

}

// Abstract Factory

public abstract class AbstractFactory

{

    public abstract ProductA CreateProductA();

    public abstract ProductB CreateProductB();

}

// Concrete Factory 1

public class ConcreteFactory1 : AbstractFactory

{

    public override ProductA CreateProductA()

    {

        return new ProductA1();

    }

    public override ProductB CreateProductB()

    {

        return new ProductB1();

    }

}

// Concrete Factory 2

public class ConcreteFactory2 : AbstractFactory

{

    public override ProductA CreateProductA()

    {

        return new ProductA2();

    }

    public override ProductB CreateProductB()

    {

        return new ProductB2();

    }

}

// Client

public class Client

{

    private ProductA \_productA;

    private ProductB \_productB;

    public Client(AbstractFactory factory

)

    {

        \_productA = factory.CreateProductA();

        \_productB = factory.CreateProductB();

    }

    public void Run()

    {

        \_productA.MethodA();

        \_productB.MethodB();

    }

}

// Usage

public static void Main(string[] args)

{

    AbstractFactory factory1 = new ConcreteFactory1();

    Client client1 = new Client(factory1);

    client1.Run();

    AbstractFactory factory2 = new ConcreteFactory2();

    Client client2 = new Client(factory2);

    client2.Run();

}

```

Trong ví dụ trên, chúng ta có hai loại sản phẩm là ProductA và ProductB, và có hai nhà máy cụ thể (Concrete Factory) là ConcreteFactory1 và ConcreteFactory2. Mỗi nhà máy cụ thể triển khai các phương thức để tạo ra các đối tượng cụ thể tương ứng. Client sử dụng một nhà máy cụ thể để tạo ra các sản phẩm và thực hiện các phương thức của chúng.

Abstract Factory Design Pattern giúp tách rời việc tạo đối tượng từ việc sử dụng chúng, tăng tính mở rộng và linh hoạt trong việc mở rộng hệ thống.

# **Adapter**

Adapter Design Pattern là một trong những mẫu thiết kế phần mềm (design pattern) thuộc nhóm cấu trúc (structural pattern). Mẫu thiết kế này cho phép các đối tượng không tương thích với nhau có thể làm việc cùng nhau thông qua việc chuyển đổi giao diện.

Trong ngôn ngữ lập trình C#, Adapter Design Pattern được sử dụng khi muốn kết nối giữa hai thành phần hoặc lớp có giao diện không tương thích. Pattern này tạo ra một lớp trung gian (adapter) để chuyển đổi giao diện của một lớp thành giao diện khác mà các lớp khác có thể sử dụng.

Cấu trúc của Adapter Design Pattern bao gồm ba thành phần chính:

1. Target: Đây là giao diện mà các đối tượng khách hàng mong muốn sử dụng.

2. Adapter: Lớp adapter chuyển đổi giao diện của một đối tượng thành giao diện mà khách hàng mong muốn sử dụng.

3. Adaptee: Đây là đối tượng hoặc lớp có giao diện không tương thích với Target, nhưng chúng ta muốn sử dụng thông qua Adapter.

Dưới đây là một ví dụ minh họa về Adapter Design Pattern trong C#:

```csharp

// Giao diện Target

interface ITarget

{

    void Request();

}

// Lớp Adaptee

class Adaptee

{

    public void SpecificRequest()

    {

        Console.WriteLine("Specific request from Adaptee.");

    }

}

// Lớp Adapter

class Adapter : ITarget

{

    private Adaptee adaptee;

    public Adapter(Adaptee adaptee)

    {

        this.adaptee = adaptee;

    }

    public void Request()

    {

        adaptee.SpecificRequest();

    }

}

// Client sử dụng Adapter

class Client

{

    public static void Main(string[] args)

    {

        Adaptee adaptee = new Adaptee();

        ITarget target = new Adapter(adaptee);

        target.Request();

    }

}

```

Trong ví dụ trên, giao diện `ITarget` đại diện cho giao diện mà Client mong muốn sử dụng. Lớp `Adaptee` có một phương thức `SpecificRequest` có giao diện không tương thích với `ITarget`. Lớp `Adapter` được tạo ra để chuyển đổi giao diện của `Adaptee` thành giao diện `ITarget`. Client sử dụng `Adapter` để gọi phương thức `Request`, và phương thức này sẽ chuyển tiếp yêu cầu đến `Adaptee` thông qua lớp `Adapter`.

Adapter Design Pattern cho phép chúng ta tích hợp các thành phần không tương thích với nhau một cách linh hoạt và tạo

 ra sự tương tác giữa chúng.

Ví dụ: Chúng ta có interface MediaPlayer và lớp AudioPlayer cụ thể implement MediaPlayer. AudioPlayer có thể phát các tệp âm thanh định dạng mp3 theo mặc định.

Chúng ta đang có một interface khác AdvancedMediaPlayer và các classes cụ thể implement interface AdvancedMediaPlayer. Các class này có thể phát các tệp định dạng vlc và mp4.

Chúng ta muốn tạo AudioPlayer để chơi các định dạng khác. Để đạt được điều này, chúng tôi đã tạo một class MediaAdapter, nó implements the MediaPlayer interface và sử dụng các đối tượng AdvancedMediaPlayer để chơi định dạng bắt buộc.

AudioPlayer sử dụng adapter của class MediaAdapter chuyển qua loại âm thanh mong muốn mà không cần biết class thực tế có thể phát định dạng mong muốn. AdapterPotypeDemo, class demo của chúng ta sẽ sử dụng class AudioPlayer để chơi các định dạng khác nhau.

# **Bridge**

Mẫu thiết kế Bridge là một mẫu thiết kế cấu trúc (structural design pattern) trong lập trình hướng đối tượng (OOP). Mẫu thiết kế này cho phép bạn tách biệt một lớp giao diện (abstraction) khỏi các lớp cụ thể thực hiện (implementations). Điều này giúp bạn có thể thay đổi lớp cụ thể mà không ảnh hưởng đến các lớp sử dụng lớp giao diện.

Trong C#, bạn có thể triển khai Bridge Design Pattern bằng cách sử dụng các lớp và giao diện. Hãy xem ví dụ sau:

Đầu tiên, chúng ta định nghĩa một giao diện "Implementor" để đại diện cho các lớp cụ thể thực hiện:

```csharp

public interface IImplementor

{

    void OperationImpl();

}

```

Tiếp theo, chúng ta có lớp giao diện "Abstraction" sử dụng giao diện "Implementor" để thực hiện một phương thức:

```csharp

public abstract class Abstraction

{

    protected IImplementor implementor;

    public Abstraction(IImplementor implementor)

    {

        this.implementor = implementor;

    }

    public abstract void Operation();

}

```

Sau đó, chúng ta triển khai các lớp cụ thể thực hiện "ImplementorA" và "ImplementorB" từ giao diện "IImplementor":

```csharp

public class ImplementorA : IImplementor

{

    public void OperationImpl()

    {

        Console.WriteLine("Implementor A operation");

    }

}

public class ImplementorB : IImplementor

{

    public void OperationImpl()

    {

        Console.WriteLine("Implementor B operation");

    }

}

```

Cuối cùng, chúng ta triển khai lớp "RefinedAbstraction" từ lớp "Abstraction" và triển khai phương thức "Operation" bằng cách sử dụng lớp cụ thể thực hiện tương ứng:

```csharp

public class RefinedAbstraction : Abstraction

{

    public RefinedAbstraction(IImplementor implementor) : base(implementor)

    {

    }

    public override void Operation()

    {

        implementor.OperationImpl();

    }

}

```

Bây giờ, chúng ta có thể sử dụng các lớp này như sau:

```csharp

// Tạo đối tượng RefinedAbstraction với ImplementorA

Abstraction abstraction = new RefinedAbstraction(new ImplementorA());

abstraction.Operation(); // Output: Implementor A operation

// Tạo đối tượng RefinedAbstraction với ImplementorB

abstraction = new RefinedAbstraction(new ImplementorB());

abstraction.Operation(); // Output: Implementor B operation

```

Khi chúng ta thay đổi lớp cụ thể thực hiện (implementor), ví dụ như từ "ImplementorA" sang "ImplementorB", lớp giao diện "

Abstraction" vẫn không bị ảnh hưởng và vẫn hoạt động như thông thường.

Mẫu thiết kế Bridge giúp chúng ta tách biệt các thành phần và giảm sự phụ thuộc giữa chúng. Điều này mang lại sự linh hoạt và dễ dàng mở rộng trong quá trình phát triển phần mềm.

Ví dụ: Trong một công ty có nhiều loại nhân viên. Mỗi loại nhân viên có một cách tính lương khác nhau. Chương trình quản lý nhân viên của bạn sẽ tính toán lương cho một nhân viên như thế nào họ chuyển từ loại này sang loại khác(hoặc chuyển từ phòng ban này sang phòng ban khác,). Ví dụ: Khi bạn apply vào vị trí developer của một công ty phần mềm, khi bắt đầu làm thì bạn là junior, một bậc cao hơn là senior, và cao hơn nữa là vai trò leader.Mỗi lần “lên cấp” như vậy là một lần thay đổi công thức tính lương.

# **ChainOfResponsibility**

Chain of Responsibility là một mẫu thiết kế phần mềm trong lập trình hướng đối tượng. Nó cho phép các đối tượng gửi yêu cầu của mình qua một chuỗi các đối tượng xử lý, cho đến khi yêu cầu được xử lý thành công hoặc không có đối tượng nào xử lý nó.

Trong C#, Chain of Responsibility có thể được triển khai như sau:

1. Đầu tiên, chúng ta cần xác định một interface hoặc một lớp cơ sở để đại diện cho các xử lý viên (handler) trong chuỗi. Ví dụ:

```csharp

public abstract class Handler

{

    protected Handler successor;

    public void SetSuccessor(Handler successor)

    {

        this.successor = successor;

    }

    public abstract void HandleRequest(Request request);

}

```

2. Tiếp theo, chúng ta cần triển khai các xử lý viên cụ thể kế thừa từ lớp cơ sở `Handler`. Mỗi xử lý viên sẽ quyết định xem liệu nó có thể xử lý yêu cầu hay không. Nếu có thể, nó sẽ xử lý yêu cầu đó. Nếu không, nó sẽ chuyển yêu cầu cho xử lý viên kế tiếp trong chuỗi bằng cách gọi phương thức `HandleRequest` của xử lý viên kế tiếp. Ví dụ:

```csharp

public class ConcreteHandler1 : Handler

{

    public override void HandleRequest(Request request)

    {

        if (request.Type == RequestType.Type1)

        {

            // Xử lý yêu cầu

        }

        else if (successor != null)

        {

            successor.HandleRequest(request);

        }

    }

}

public class ConcreteHandler2 : Handler

{

    public override void HandleRequest(Request request)

    {

        if (request.Type == RequestType.Type2)

        {

            // Xử lý yêu cầu

        }

        else if (successor != null)

        {

            successor.HandleRequest(request);

        }

    }

}

```

3. Cuối cùng, chúng ta có thể sử dụng chuỗi các xử lý viên để xử lý các yêu cầu. Đầu tiên, chúng ta cần xác định yêu cầu và sau đó gửi yêu cầu đó qua chuỗi các xử lý viên, bắt đầu từ xử lý viên đầu tiên. Ví dụ:

```csharp

public class Client

{

    private Handler handler;

    public Client()

    {

        // Tạo chuỗi các xử lý viên

        Handler handler1 = new ConcreteHandler1();

        Handler handler2 = new ConcreteHandler2();

        handler1.SetSuccessor(handler2);

        handler = handler1;

    }

    public void ProcessRequest(Request request)

    {

        handler.HandleRequest(request);

    }

}

```

Trong ví dụ trên, khi `Client` gọi phương thức `ProcessRequest` với một yêu

 cầu, nó sẽ gửi yêu cầu này qua chuỗi các xử lý viên. Nếu một xử lý viên có thể xử lý yêu cầu, nó sẽ xử lý nó. Nếu không, yêu cầu sẽ được chuyển cho xử lý viên kế tiếp cho đến khi yêu cầu được xử lý hoặc không còn xử lý viên nào trong chuỗi có thể xử lý nó.

Mẫu thiết kế Chain of Responsibility giúp chúng ta xây dựng các hệ thống linh hoạt và dễ mở rộng, vì chúng ta có thể thay đổi hoặc mở rộng chuỗi các xử lý viên mà không ảnh hưởng đến các thành phần khác trong hệ thống.

Ví dụ: https://viblo.asia/p/chain-of-responsibility-design-pattern-tro-thu-dac-luc-cua-developers-yMnKMBNDZ7P

# **Command**

Chain of Responsibility là một mẫu thiết kế phần mềm trong lập trình hướng đối tượng. Nó cho phép các đối tượng gửi yêu cầu của mình qua một chuỗi các đối tượng xử lý, cho đến khi yêu cầu được xử lý thành công hoặc không có đối tượng nào xử lý nó.

Trong C#, Chain of Responsibility có thể được triển khai như sau:

1. Đầu tiên, chúng ta cần xác định một interface hoặc một lớp cơ sở để đại diện cho các xử lý viên (handler) trong chuỗi. Ví dụ:

```csharp

public abstract class Handler

{

    protected Handler successor;

    public void SetSuccessor(Handler successor)

    {

        this.successor = successor;

    }

    public abstract void HandleRequest(Request request);

}

```

2. Tiếp theo, chúng ta cần triển khai các xử lý viên cụ thể kế thừa từ lớp cơ sở `Handler`. Mỗi xử lý viên sẽ quyết định xem liệu nó có thể xử lý yêu cầu hay không. Nếu có thể, nó sẽ xử lý yêu cầu đó. Nếu không, nó sẽ chuyển yêu cầu cho xử lý viên kế tiếp trong chuỗi bằng cách gọi phương thức `HandleRequest` của xử lý viên kế tiếp. Ví dụ:

```csharp

public class ConcreteHandler1 : Handler

{

    public override void HandleRequest(Request request)

    {

        if (request.Type == RequestType.Type1)

        {

            // Xử lý yêu cầu

        }

        else if (successor != null)

        {

            successor.HandleRequest(request);

        }

    }

}

public class ConcreteHandler2 : Handler

{

    public override void HandleRequest(Request request)

    {

        if (request.Type == RequestType.Type2)

        {

            // Xử lý yêu cầu

        }

        else if (successor != null)

        {

            successor.HandleRequest(request);

        }

    }

}

```

3. Cuối cùng, chúng ta có thể sử dụng chuỗi các xử lý viên để xử lý các yêu cầu. Đầu tiên, chúng ta cần xác định yêu cầu và sau đó gửi yêu cầu đó qua chuỗi các xử lý viên, bắt đầu từ xử lý viên đầu tiên. Ví dụ:

```csharp

public class Client

{

    private Handler handler;

    public Client()

    {

        // Tạo chuỗi các xử lý viên

        Handler handler1 = new ConcreteHandler1();

        Handler handler2 = new ConcreteHandler2();

        handler1.SetSuccessor(handler2);

        handler = handler1;

    }

    public void ProcessRequest(Request request)

    {

        handler.HandleRequest(request);

    }

}

```

Trong ví dụ trên, khi `Client` gọi phương thức `ProcessRequest` với một yêu

 cầu, nó sẽ gửi yêu cầu này qua chuỗi các xử lý viên. Nếu một xử lý viên có thể xử lý yêu cầu, nó sẽ xử lý nó. Nếu không, yêu cầu sẽ được chuyển cho xử lý viên kế tiếp cho đến khi yêu cầu được xử lý hoặc không còn xử lý viên nào trong chuỗi có thể xử lý nó.

Mẫu thiết kế Chain of Responsibility giúp chúng ta xây dựng các hệ thống linh hoạt và dễ mở rộng, vì chúng ta có thể thay đổi hoặc mở rộng chuỗi các xử lý viên mà không ảnh hưởng đến các thành phần khác trong hệ thống.

Ví dụ: https://viblo.asia/p/chain-of-responsibility-design-pattern-tro-thu-dac-luc-cua-developers-yMnKMBNDZ7P

# **Composite**

Composite Design Pattern là một mẫu thiết kế (design pattern) thuộc nhóm mẫu cấu trúc (structural pattern) trong lĩnh vực lập trình. Mẫu này cho phép bạn xây dựng cấu trúc cây hierarchial với các đối tượng tương tự và định nghĩa một giao diện chung cho chúng. Composite cho phép các đối tượng đơn lẻ và nhóm đối tượng được xử lý một cách đồng nhất.

Trong C#, Composite Design Pattern thường được sử dụng để xây dựng cây đối tượng có cấu trúc phân cấp. Mẫu này giúp tạo ra một sự phân loại rõ ràng giữa các đối tượng lá (leaf objects) và các đối tượng phức hợp (composite objects). Các đối tượng lá là các thành phần cơ bản không có con và các đối tượng phức hợp là nhóm các đối tượng, cả lá và phức hợp, có thể chứa các thành phần con khác.

Dưới đây là một ví dụ về cách triển khai Composite Design Pattern trong C#:

```csharp

using System;

using System.Collections.Generic;

// Giao diện chung cho tất cả các thành phần

public interface IComponent

{

    void Operation();

}

// Đối tượng lá (leaf object)

public class Leaf : IComponent

{

    public void Operation()

    {

        Console.WriteLine("Leaf operation");

    }

}

// Đối tượng phức hợp (composite object)

public class Composite : IComponent

{

    private List<IComponent> components = new List<IComponent>();

    public void AddComponent(IComponent component)

    {

        components.Add(component);

    }

    public void RemoveComponent(IComponent component)

    {

        components.Remove(component);

    }

    public void Operation()

    {

        Console.WriteLine("Composite operation");

        foreach (var component in components)

        {

            component.Operation();

        }

    }

}

// Sử dụng Composite Design Pattern

class Program

{

    static void Main(string[] args)

    {

        // Tạo đối tượng lá

        var leaf1 = new Leaf();

        var leaf2 = new Leaf();

        // Tạo đối tượng phức hợp

        var composite1 = new Composite();

        var composite2 = new Composite();

        // Thêm các đối tượng lá vào đối tượng phức hợp

        composite1.AddComponent(leaf1);

        composite2.AddComponent(leaf2);

        // Thêm đối tượng phức hợp vào đối tượng phức hợp khác

        composite1.AddComponent(composite2);

        // Gọi phương thức Operation trên đối tượng phức hợp gốc

        composite1.Operation();

        /\*

        Kết quả:

        Composite operation

        Leaf operation

        Composite operation

        Leaf operation

        \*/

    }

}

```

Trên đây là một ví dụ đơn giản về Composite Design Pattern trong C#. Bằng cách sử dụng mẫu thiết kế này, bạn có thể

 xây dựng các cây đối tượng phức tạp và xử lý chúng một cách đồng nhất. Mỗi đối tượng trong cây có thể được gọi để thực hiện một hành động cụ thể, không phụ thuộc vào việc đối tượng đó là lá hay phức hợp.

# **Decorator**

Decorator Design Pattern là một mẫu thiết kế (design pattern) trong lập trình, thuộc nhóm các mẫu thiết kế cấu trúc (structural design patterns). Mục tiêu của Decorator Design Pattern là định nghĩa một cách linh hoạt những chức năng mở rộng cho một đối tượng mà không làm thay đổi cấu trúc của lớp gốc.

Trong C#, Decorator Design Pattern cho phép bạn bọc (wrap) một đối tượng bên trong một đối tượng decorator để mở rộng hoặc thay đổi hành vi của đối tượng ban đầu mà không ảnh hưởng đến các đối tượng khác trong cùng một lớp.

Cấu trúc của Decorator Design Pattern bao gồm các thành phần sau:

- Component: là một interface hoặc lớp trừu tượng (abstract class) định nghĩa các phương thức chung cho các đối tượng gốc và decorator.

- ConcreteComponent: là lớp cài đặt giao diện Component. Đây là đối tượng ban đầu mà chúng ta muốn mở rộng hoặc thay đổi.

- Decorator: là một lớp trừu tượng (abstract class) kế thừa từ Component. Nó cũng chứa một tham chiếu đến một đối tượng Component. Decorator có thể thêm hoặc sửa đổi hành vi của Component.

- ConcreteDecorator: là lớp cài đặt giao diện Decorator. Nó bao gồm một đối tượng Component và triển khai các phương thức của Decorator. Nó cũng có thể thực hiện thêm logic bổ sung trước hoặc sau khi gọi đến phương thức của Component.

Dưới đây là một ví dụ minh họa Decorator Design Pattern trong C#:

```csharp

// Component

public interface ICar

{

    string GetDescription();

    double GetCost();

}

// ConcreteComponent

public class BasicCar : ICar

{

    public string GetDescription()

    {

        return "Basic Car";

    }

    public double GetCost()

    {

        return 10000;

    }

}

// Decorator

public abstract class CarDecorator : ICar

{

    protected ICar \_car;

    public CarDecorator(ICar car)

    {

        \_car = car;

    }

    public virtual string GetDescription()

    {

        return \_car.GetDescription();

    }

    public virtual double GetCost()

    {

        return \_car.GetCost();

    }

}

// ConcreteDecorator

public class LuxuryCar : CarDecorator

{

    public LuxuryCar(ICar car) : base(car)

    {

    }

    public override string GetDescription()

    {

        return base.GetDescription() + ", Luxury Features";

    }

    public override double GetCost()

    {

        return base.GetCost() + 5000;

    }

}

// ConcreteDecorator

public class SportsCar : CarDecorator

{

    public SportsCar(ICar car) : base(car)

    {

    }

    public override

 string GetDescription()

    {

        return base.GetDescription() + ", Sports Features";

    }

    public override double GetCost()

    {

        return base.GetCost() + 2000;

    }

}

// Usage

class Program

{

    static void Main(string[] args)

    {

        // Tạo một đối tượng cơ bản

        ICar basicCar = new BasicCar();

        Console.WriteLine(basicCar.GetDescription()); // Output: Basic Car

        Console.WriteLine(basicCar.GetCost()); // Output: 10000

        // Bọc đối tượng cơ bản vào các decorator để mở rộng chức năng

        ICar sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));

        Console.WriteLine(sportsLuxuryCar.GetDescription()); // Output: Basic Car, Luxury Features, Sports Features

        Console.WriteLine(sportsLuxuryCar.GetCost()); // Output: 17000

    }

}

```

Trong ví dụ trên, chúng ta có một đối tượng `BasicCar` làm đối tượng gốc và các decorator `LuxuryCar` và `SportsCar` để mở rộng chức năng. Khi chạy chương trình, chúng ta có thể thấy đối tượng cuối cùng `sportsLuxuryCar` có mô tả là "Basic Car, Luxury Features, Sports Features" và giá trị là 17000.

Decorator Design Pattern cho phép chúng ta mở rộng chức năng của một đối tượng một cách linh hoạt bằng cách bọc nó vào các decorator khác nhau. Điều này giúp chúng ta thay đổi hành vi của đối tượng mà không làm thay đổi cấu trúc của nó, đồng thời giúp tăng tính linh hoạt và dễ dàng bảo trì của mã nguồn.

# **Facade**

Mẫu thiết kế Facade là một mẫu thiết kế phần mềm thuộc nhóm mẫu cấu trúc (structural design pattern) trong lĩnh vực lập trình. Nó cung cấp một giao diện đơn giản để truy cập vào một hệ thống phức tạp hơn bằng cách đóng gói các lớp, đối tượng và phương thức phức tạp thành một giao diện đơn giản hơn.

Mục đích chính của Facade Design Pattern là cung cấp một giao diện thuận tiện và dễ sử dụng cho một phần của hệ thống, giảm độ phức tạp và sự phụ thuộc giữa các thành phần của hệ thống. Nó giúp che dấu sự phức tạp của hệ thống và cung cấp một cách tiếp cận trực quan hơn cho người sử dụng.

Trong C#, một Facade là một lớp hoặc một thành phần có trách nhiệm quản lý các giao tiếp và tương tác giữa các thành phần khác của hệ thống. Nó cung cấp một giao diện đơn giản cho người dùng cuối để tương tác với hệ thống mà không cần hiểu rõ về cấu trúc và chi tiết bên trong của nó.

Dưới đây là một ví dụ về cách sử dụng Facade Design Pattern trong C#:

```csharp

// Facade

public class PaymentFacade

{

    private PaymentProcessor paymentProcessor;

    private InventorySystem inventorySystem;

    private ShippingSystem shippingSystem;

    public PaymentFacade()

    {

        paymentProcessor = new PaymentProcessor();

        inventorySystem = new InventorySystem();

        shippingSystem = new ShippingSystem();

    }

    public void ProcessOrder(Order order)

    {

        paymentProcessor.ProcessPayment(order);

        inventorySystem.UpdateInventory(order);

        shippingSystem.ShipOrder(order);

    }

}

// Subsystems

public class PaymentProcessor

{

    public void ProcessPayment(Order order)

    {

        // Process payment logic

    }

}

public class InventorySystem

{

    public void UpdateInventory(Order order)

    {

        // Update inventory logic

    }

}

public class ShippingSystem

{

    public void ShipOrder(Order order)

    {

        // Ship order logic

    }

}

// Client

public class Client

{

    private PaymentFacade paymentFacade;

    public Client()

    {

        paymentFacade = new PaymentFacade();

    }

    public void PlaceOrder(Order order)

    {

        paymentFacade.ProcessOrder(order);

    }

}

// Usage

Order order = new Order();

Client client = new Client();

client.PlaceOrder(order);

```

Trong ví dụ trên, PaymentFacade là một lớp Facade. Nó giúp che dấu sự phức tạp của việc xử lý thanh toán, cập nhật kho hàng và vận chuyển đơn hàng bên trong hệ thống. Client chỉ cần tương tác với PaymentFacade để đặt hàng và không c

ần biết về các chi tiết bên trong.

Mẫu thiết kế Facade giúp tạo ra một cấu trúc hệ thống rõ ràng, giảm sự phụ thuộc giữa các thành phần và giúp tăng tính module hóa trong mã nguồn. Nó cũng cung cấp một giao diện đơn giản và dễ sử dụng cho người sử dụng cuối.

# **FactoryMethod**

Factory Method Design Pattern là một mẫu thiết kế phần mềm thuộc nhóm Creational Patterns trong lập trình hướng đối tượng. Mẫu này cho phép chúng ta tạo ra các đối tượng mà không cần chỉ định trực tiếp lớp cụ thể của chúng. Thay vào đó, nó tạo ra đối tượng thông qua một phương thức gọi là "factory method".

Trong C#, Factory Method Design Pattern được triển khai bằng cách tạo ra một lớp trừu tượng gọi là "Creator", chứa phương thức factory method. Lớp Creator định nghĩa phương thức để tạo ra đối tượng, nhưng không cung cấp cài đặt cụ thể cho phương thức này. Thay vào đó, các lớp con của Creator sẽ thừa kế từ Creator và cung cấp cài đặt cụ thể cho phương thức factory method.

Dưới đây là một ví dụ về cách triển khai Factory Method Design Pattern trong C#:

```csharp

// Creator abstract class

abstract class Creator

{

    public abstract Product FactoryMethod(); // Factory Method

    public void SomeOperation()

    {

        // Gọi factory method để tạo đối tượng

        Product product = FactoryMethod();

        // Sử dụng đối tượng đã tạo

        product.Operation();

    }

}

// Product abstract class

abstract class Product

{

    public abstract void Operation();

}

// ConcreteCreator class

class ConcreteCreator : Creator

{

    public override Product FactoryMethod()

    {

        return new ConcreteProduct();

    }

}

// ConcreteProduct class

class ConcreteProduct : Product

{

    public override void Operation()

    {

        Console.WriteLine("ConcreteProduct.Operation() called");

    }

}

// Sử dụng mẫu Factory Method

class Program

{

    static void Main(string[] args)

    {

        Creator creator = new ConcreteCreator();

        creator.SomeOperation();

        // Kết quả:

        // ConcreteProduct.Operation() called

    }

}

```

Trong ví dụ trên, lớp Creator là một lớp trừu tượng có một phương thức FactoryMethod. Lớp ConcreteCreator là một lớp con của Creator và triển khai phương thức FactoryMethod để tạo ra một đối tượng của ConcreteProduct. Lớp ConcreteProduct là một lớp con của Product và triển khai phương thức Operation.

Khi chạy, chương trình tạo một đối tượng của ConcreteCreator và gọi phương thức SomeOperation. Phương thức này sẽ sử dụng phương thức FactoryMethod để tạo ra một đối tượng của ConcreteProduct và gọi phương thức Operation của nó.

Factory Method Design Pattern cho phép chúng ta tách rời việc tạo đối tượng và sử dụng đối tượng, giúp linh hoạt trong việc mở rộng và thay đổi lớp cụ thể của đối tượng mà không ảnh hưởng

 đến mã gọi.

# **Iterator**

Iterator Design Pattern là một mẫu thiết kế phần mềm thuộc nhóm Behavioral Patterns. Nó cho phép duyệt qua một tập hợp các phần tử mà không cần tiết lộ cấu trúc nội bộ của tập hợp đó.

Trong C#, Iterator Design Pattern được triển khai bằng cách sử dụng interface `IEnumerable` và `IEnumerator` hoặc sử dụng từ khóa `yield`.

1. Triển khai Iterator Design Pattern bằng `IEnumerable` và `IEnumerator`:

   - `IEnumerable` là một interface định nghĩa một phương thức `GetEnumerator()` trả về một đối tượng `IEnumerator`.

   - `IEnumerator` là một interface định nghĩa các phương thức `MoveNext()`, `Reset()` và thuộc tính `Current` để duyệt qua các phần tử trong tập hợp.

   Ví dụ:

   ```csharp

   using System;

   using System.Collections;

   // Lớp tập hợp chứa các phần tử

   class MyCollection : IEnumerable

   {

       private int[] items;

       public MyCollection()

       {

           items = new int[5] { 1, 2, 3, 4, 5 };

       }

       // Triển khai phương thức GetEnumerator() của IEnumerable

       public IEnumerator GetEnumerator()

       {

           return new MyIterator(items);

       }

   }

   // Lớp duyệt qua các phần tử trong tập hợp

   class MyIterator : IEnumerator

   {

       private int[] items;

       private int currentIndex;

       public MyIterator(int[] collection)

       {

           items = collection;

           currentIndex = -1;

       }

       // Triển khai phương thức MoveNext() của IEnumerator

       public bool MoveNext()

       {

           currentIndex++;

           return (currentIndex < items.Length);

       }

       // Triển khai thuộc tính Current của IEnumerator

       public object Current

       {

           get { return items[currentIndex]; }

       }

       // Triển khai phương thức Reset() của IEnumerator

       public void Reset()

       {

           currentIndex = -1;

       }

   }

   // Sử dụng Iterator Design Pattern

   class Program

   {

       static void Main(string[] args)

       {

           MyCollection collection = new MyCollection();

           foreach (var item in collection)

           {

               Console.WriteLine(item);

           }

       }

   }

   ```

   Kết quả:

   ```

   1

   2

   3

   4

   5

   ```

2. Triển khai Iterator Design Pattern bằng `yield`:

   - `yield` là một từ khóa trong C# cho phép tạo một iterator mà không cần triển khai các phương thức `IEnumerator`.

   - Một phương thức chứa từ khóa `yield` trả về một `IEnumerable` hoặc `IEnumerator` và được gọi như một iterator khi được duyệt qua.

   Ví dụ:

   ```csharp

   using System;

   using System.Collections;

   class MyCollection

   {

       private int[] items;

       public MyCollection()

       {

 items = new int[5] { 1, 2, 3, 4, 5 };

       }

       // Sử dụng từ khóa yield để triển khai iterator

       public IEnumerable GetIterator()

       {

           foreach (var item in items)

           {

               yield return item;

           }

       }

   }

   // Sử dụng Iterator Design Pattern

   class Program

   {

       static void Main(string[] args)

       {

           MyCollection collection = new MyCollection();

           foreach (var item in collection.GetIterator())

           {

               Console.WriteLine(item);

           }

       }

   }

   ```

   Kết quả tương tự như trên.

Iterator Design Pattern cho phép duyệt qua các phần tử trong một tập hợp mà không cần biết cấu trúc nội bộ của tập hợp đó. Điều này giúp tách biệt việc duyệt qua các phần tử khỏi việc xử lý logic trong tập hợp, làm cho mã nguồn linh hoạt và dễ bảo trì.

# **Mediator**

Mediator pattern trong C# là một mẫu thiết kế (design pattern) dùng để giảm sự phụ thuộc (decoupling) giữa các thành phần (components) trong một hệ thống phần mềm. Mẫu này tạo ra một đối tượng trung gian (mediator) để quản lý và điều phối thông tin trao đổi giữa các thành phần khác nhau, thay vì các thành phần liên lạc trực tiếp với nhau.

Ví dụ:

Giả sử bạn đang xây dựng một ứng dụng chat đơn giản. Trong ứng dụng này, có nhiều thành phần như người dùng (User), hội thoại (ChatRoom) và tin nhắn (Message). Mỗi người dùng có thể gửi và nhận tin nhắn trong một hội thoại.

Sử dụng mẫu thiết kế Mediator, bạn có thể tạo ra một lớp Mediator trung gian, ví dụ là ChatMediator, để quản lý và điều phối thông tin giữa các người dùng và hội thoại. Các thành phần như người dùng và hội thoại sẽ giao tiếp thông qua giao diện của Mediator, không trực tiếp liên lạc với nhau.

Dưới đây là một ví dụ đơn giản về cách triển khai Mediator pattern trong C#:

// Giao diện Mediator

public interface IChatMediator

{

void SendMessage(string message, User user);

}

// Lớp Mediator

public class ChatMediator : IChatMediator

{

private List<User> users;

public ChatMediator()

{

users = new List<User>();

}

public void AddUser(User user)

{

users.Add(user);

}

public void SendMessage(string message, User sender)

{

foreach (var user in users)

{

if (user != sender)

user.ReceiveMessage(message);

}

}

}

// Lớp Người dùng

public class User

{

private string name;

private IChatMediator chatMediator;

public User(string name, IChatMediator chatMediator)

{

this.name = name;

this.chatMediator = chatMediator;

}

public void SendMessage(string message)

{

chatMediator.SendMessage(message, this);

}

public void ReceiveMessage(string message)

{

Console.WriteLine($"{name} received a message: {message}");

}

}

// Sử dụng Mediator pattern

public class Program

{

public static void Main(string[] args)

{

var chatMediator = new ChatMediator();

var user1 = new User("User 1", chatMediator);

var user2 = new User("User 2", chatMediator);

var user3 = new User("User 3", chatMediator);

chatMediator.AddUser(user1);

chatMediator.AddUser(user2);

chatMediator.AddUser(user3);

user1.SendMessage("Hello, everyone!");

/\*

Output:

User 2 received a message: Hello, everyone!

User 3 received a message: Hello, everyone!

\*/

}

}

Trong ví dụ trên, ChatMediator là lớp trung gian giữa các người dùng và quản lý việc gửi và nhận tin nhắn. Người dùng gửi tin nhắn thông qua phương thức SendMessage của Mediator, và Mediator sẽ chuyển tin nhắn cho các người dùng khác thông qua phương thức ReceiveMessage.

# **Observer**

Observer Design Pattern là một trong những mẫu thiết kế phần mềm (design pattern) thuộc nhóm hành vi (behavioral pattern) trong lập trình hướng đối tượng. Mẫu thiết kế này cho phép các đối tượng (observers) tự động nhận thông báo và cập nhật từ một đối tượng chủ đề (subject) khi có sự thay đổi trong trạng thái của nó. Điều này cho phép các đối tượng tương tác với nhau mà không cần biết thông tin chi tiết về nhau.

Trong C#, Observer Design Pattern có thể được triển khai bằng cách sử dụng một số thành phần cốt lõi như sau:

1. Subject (Chủ đề): Đây là đối tượng quản lý danh sách các observer và cung cấp các phương thức để thêm, xóa và thông báo cho các observer. Trong C#, một cách phổ biến để triển khai Subject là sử dụng một lớp trừu tượng hoặc giao diện để định nghĩa các phương thức chung.

```csharp

public abstract class Subject

{

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

    public void Attach(Observer observer)

    {

        observers.Add(observer);

    }

    public void Detach(Observer observer)

    {

        observers.Remove(observer);

    }

    public void Notify()

    {

        foreach (Observer observer in observers)

        {

            observer.Update();

        }

    }

}

```

2. Observer (Người quan sát): Đây là các đối tượng quan sát và nhận thông báo từ Subject. Trong C#, Observer thường được triển khai bằng cách sử dụng một lớp trừu tượng hoặc giao diện, và định nghĩa phương thức Update để xử lý thông báo từ Subject.

```csharp

public abstract class Observer

{

    public abstract void Update();

}

```

3. Concrete Subject (Chủ đề cụ thể): Lớp này triển khai các phương thức cụ thể của Subject và quản lý trạng thái của nó. Khi trạng thái thay đổi, nó gọi phương thức Notify để thông báo cho tất cả các Observer.

```csharp

public class ConcreteSubject : Subject

{

    private string state;

    public string State

    {

        get { return state; }

        set

        {

            state = value;

            Notify();

        }

    }

}

```

4. Concrete Observer (Người quan sát cụ thể): Lớp này triển khai phương thức Update để xử lý thông báo từ Subject. Nó có thể truy cập trực tiếp vào trạng thái của Subject để cập nhật và thực hiện các hành động phù hợp.

```csharp

public class ConcreteObserver : Observer

{

    private string name;

    private ConcreteSubject subject;

    public ConcreteObserver(string name,

 ConcreteSubject subject)

    {

        this.name = name;

        this.subject = subject;

    }

    public override void Update()

    {

        string subjectState = subject.State;

        Console.WriteLine("Observer {0} received the state update: {1}", name, subjectState);

    }

}

```

Ví dụ sử dụng Observer Design Pattern:

```csharp

ConcreteSubject subject = new ConcreteSubject();

ConcreteObserver observer1 = new ConcreteObserver("Observer 1", subject);

ConcreteObserver observer2 = new ConcreteObserver("Observer 2", subject);

subject.Attach(observer1);

subject.Attach(observer2);

subject.State = "State 1";

subject.State = "State 2";

subject.Detach(observer2);

subject.State = "State 3";

```

Kết quả:

```

Observer 1 received the state update: State 1

Observer 2 received the state update: State 1

Observer 1 received the state update: State 2

Observer 2 received the state update: State 2

Observer 1 received the state update: State 3

```

Trong ví dụ trên, khi trạng thái của ConcreteSubject thay đổi, tất cả các ConcreteObserver đính kèm đều nhận được thông báo và thực hiện hành động tương ứng.

# **Prototype**

Prototype Design Pattern là một mẫu thiết kế (design pattern) thuộc nhóm Creational Patterns trong lĩnh vực phát triển phần mềm. Mẫu thiết kế này cho phép tạo ra các đối tượng mới bằng cách sao chép các đối tượng đã tồn tại, thay vì tạo mới từ đầu.

Mục đích chính của Prototype Design Pattern là tạo ra các đối tượng mà không cần biết chi tiết cách khởi tạo. Thay vào đó, nó sử dụng một đối tượng đã tồn tại làm nguyên mẫu (prototype) và sao chép nó để tạo ra các đối tượng mới.

Trong C#, việc thực hiện Prototype Design Pattern có thể được thực hiện bằng cách sử dụng giao diện (interface) ICloneable hoặc sử dụng phương pháp tự viết để sao chép đối tượng. Dưới đây là một ví dụ minh họa về cách sử dụng giao diện ICloneable:

```csharp

using System;

// Giao diện ICloneable

public interface ICloneable

{

    object Clone();

}

// Lớp nguyên mẫu

public class Prototype : ICloneable

{

    public int Id { get; set; }

    public object Clone()

    {

        return MemberwiseClone();

    }

}

// Lớp sử dụng nguyên mẫu

public class Client

{

    public void Operation()

    {

        Prototype prototype = new Prototype();

        prototype.Id = 1;

        // Sao chép đối tượng

        Prototype clone = (Prototype)prototype.Clone();

        clone.Id = 2;

        Console.WriteLine($"Original: {prototype.Id}");

        Console.WriteLine($"Clone: {clone.Id}");

    }

}

public class Program

{

    public static void Main(string[] args)

    {

        Client client = new Client();

        client.Operation();

    }

}

```

Trong ví dụ trên, lớp `Prototype` là một lớp nguyên mẫu có thuộc tính `Id`. Lớp này cài đặt giao diện `ICloneable` và triển khai phương thức `Clone` để sao chép đối tượng bằng cách sử dụng `MemberwiseClone`.

Lớp `Client` sử dụng lớp nguyên mẫu để tạo ra một đối tượng ban đầu và sau đó sao chép nó để tạo ra một đối tượng mới. Kết quả khi chạy chương trình sẽ hiển thị giá trị `1` cho đối tượng ban đầu và `2` cho đối tượng được sao chép.

Prototype Design Pattern giúp tăng hiệu suất và linh hoạt trong việc tạo ra các đối tượng mới. Thay vì tạo mới các đối tượng từ đầu, mẫu thiết kế này cho phép tái sử dụ

ng đối tượng đã tồn tại và chỉ thay đổi những thuộc tính cần thiết. Điều này có thể giảm thiểu việc tạo ra nhiều đối tượng mới và giúp tối ưu hóa quá trình khởi tạo.

# **Proxy**

Proxy Pattern là một mẫu thiết kế (design pattern) trong lập trình được sử dụng để tạo ra một đối tượng proxy (đại diện) cho một đối tượng khác. Đối tượng proxy này cho phép bạn kiểm soát truy cập đến đối tượng gốc và cung cấp các chức năng bổ sung.

Trong C#, Proxy Pattern được triển khai bằng cách sử dụng một lớp proxy để đại diện cho đối tượng gốc và chuyển các yêu cầu từ client tới đối tượng gốc. Đối tượng proxy có cùng interface với đối tượng gốc, cho phép client giao tiếp với nó một cách như thể đang giao tiếp với đối tượng gốc.

Proxy Pattern thường được sử dụng trong các trường hợp sau:

1. Remote Proxy: Khi đối tượng gốc nằm ở một không gian bộ nhớ khác hoặc trên một máy chủ từ xa. Proxy Pattern cho phép bạn tạo ra một đối tượng proxy để đại diện cho đối tượng gốc và gửi yêu cầu tới đối tượng từ xa.

2. Virtual Proxy: Khi việc tạo đối tượng gốc mất nhiều tài nguyên hoặc thời gian. Proxy Pattern cho phép bạn tạo ra một đối tượng proxy để thực hiện các tác vụ như khởi tạo đối tượng gốc khi cần thiết.

3. Protection Proxy: Khi bạn muốn kiểm soát truy cập vào đối tượng gốc bằng cách thêm logic kiểm tra hoặc xác thực. Proxy Pattern cho phép bạn tạo ra một đối tượng proxy để kiểm soát việc truy cập đến đối tượng gốc.

Dưới đây là một ví dụ đơn giản về Proxy Pattern trong C#:

```csharp

// Interface chung cho đối tượng gốc và proxy

interface IImage

{

    void Display();

}

// Đối tượng gốc

class RealImage : IImage

{

    private string filename;

    public RealImage(string filename)

    {

        this.filename = filename;

        LoadFromDisk();

    }

    private void LoadFromDisk()

    {

        Console.WriteLine("Loading image from disk: " + filename);

    }

    public void Display()

    {

        Console.WriteLine("Displaying image: " + filename);

    }

}

// Proxy cho đối tượng gốc

class ProxyImage : IImage

{

    private string filename;

    private RealImage realImage;

    public ProxyImage(string filename)

    {

        this.filename = filename;

    }

    public void Display()

    {

        if (realImage == null)

        {

            realImage = new RealImage(filename);

        }

        realImage.Display();

    }

}

// Sử dụng Proxy Pattern

class Program

{

    static void Main(string[] args)

    {

        // Sử dụng đối tượng proxy để hi

ển thị hình ảnh

        IImage image = new ProxyImage("image.jpg");

        // Khi gọi phương thức Display(), hình ảnh chỉ được tải lên và hiển thị khi cần thiết

        image.Display();

        // Khi gọi lại phương thức Display(), hình ảnh đã được tải lên nên chỉ hiển thị mà không cần tải lại từ đĩa

        image.Display();

        Console.ReadLine();

    }

}

```

Trong ví dụ trên, `RealImage` là đối tượng gốc, thực hiện việc tải hình ảnh từ đĩa và hiển thị. `ProxyImage` là proxy cho `RealImage`, và chỉ tải và hiển thị hình ảnh khi cần thiết. Khi chạy chương trình, ta chỉ thấy thông điệp "Loading image from disk" xuất hiện một lần duy nhất, trong lần đầu tiên gọi phương thức `Display()`. Trong lần gọi phương thức thứ hai, hình ảnh đã được tải lên và hiển thị mà không cần tải lại từ đĩa.

# **Singleton**

Singleton Design Pattern là một mẫu thiết kế phổ biến trong lập trình hướng đối tượng. Nó đảm bảo rằng một lớp chỉ có duy nhất một thể hiện và cung cấp một cách tiếp cận toàn cục đến thể hiện đó. Trong C#, Singleton Design Pattern thường được sử dụng để tạo ra các đối tượng mà chỉ cần tồn tại duy nhất một thể hiện trong suốt quá trình thực thi ứng dụng.

Để triển khai Singleton Design Pattern trong C#, chúng ta có thể sử dụng cách triển khai Singleton thông qua một lớp tĩnh hoặc qua việc sử dụng một lớp thông thường kết hợp với cách triển khai bằng cách sử dụng Lazy<T>.

Dưới đây là một ví dụ về cách triển khai Singleton Design Pattern trong C# sử dụng Lazy<T>:

```csharp

public class Singleton

{

    private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new Singleton());

    private Singleton()

    {

        // Khởi tạo đối tượng Singleton

    }

    public static Singleton Instance

    {

        get { return instance.Value; }

    }

    // Các phương thức và thuộc tính của Singleton

}

```

Trong ví dụ trên, lớp Singleton có một thuộc tính tĩnh là `Instance` để truy cập vào thể hiện duy nhất của Singleton. Thể hiện này được khởi tạo bằng cách sử dụng một biến tĩnh `instance` có kiểu `Lazy<Singleton>`. Biến `instance` này sẽ được khởi tạo duy nhất khi được truy cập lần đầu tiên thông qua thuộc tính `Value` của `Lazy<T>`, sau đó nó sẽ trả về thể hiện đã được khởi tạo.

Với cách triển khai này, mọi nơi trong ứng dụng sử dụng `Singleton.Instance` để truy cập đến thể hiện duy nhất của Singleton. Khi gọi `Singleton.Instance`, nếu thể hiện chưa được khởi tạo, nó sẽ tự động khởi tạo và trả về thể hiện đã khởi tạo. Ngược lại, nếu thể hiện đã tồn tại, nó sẽ trực tiếp trả về thể hiện đó.

Điều quan trọng khi triển khai Singleton Design Pattern là đảm bảo rằng chỉ có duy nhất một thể hiện của lớp Singleton được tạo ra. Điều này thường được đảm bảo thông qua việc sử dụng khai báo constructor private hoặc protected để không cho phép tạo ra thể hiện từ bên ngoài

 lớp.

Tóm lại, Singleton Design Pattern trong C# giúp đảm bảo rằng chỉ có một thể hiện duy nhất của một lớp tồn tại và cung cấp một cách tiếp cận toàn cục đến thể hiện đó. Điều này có thể hữu ích trong nhiều tình huống, chẳng hạn như quản lý cấu hình ứng dụng, kết nối đến cơ sở dữ liệu hoặc quản lý các tài nguyên chia sẻ.

# **State**

State Design Pattern là một mẫu thiết kế hướng đối tượng trong lập trình, nó cho phép đối tượng thay đổi hành vi của mình dựa trên trạng thái nội tại của nó. Mẫu thiết kế này giúp tăng tính mô-đun, linh hoạt và dễ bảo trì cho mã nguồn.

Trong C#, State Design Pattern được triển khai bằng cách sử dụng các lớp và giao diện. Các yếu tố chính của mẫu thiết kế này là:

- Context: Đại diện cho đối tượng chứa trạng thái và triển khai các phương thức liên quan đến trạng thái. Nó giữ một tham chiếu đến một đối tượng trạng thái cụ thể.

- State: Đại diện cho giao diện hoặc lớp trạng thái cụ thể. Nó định nghĩa các phương thức mà context sẽ gọi khi nó ở trong trạng thái đó.

- ConcreteState: Các lớp cụ thể triển khai giao diện State. Mỗi lớp này đại diện cho một trạng thái cụ thể và triển khai các phương thức tương ứng.

Khi một sự kiện hoặc hành động xảy ra, context sẽ chuyển đổi trạng thái của nó bằng cách gọi phương thức tương ứng trên đối tượng trạng thái hiện tại. Các phương thức này có thể thay đổi trạng thái của context hoặc thực hiện các hành động khác.

Một ví dụ cụ thể để minh họa State Design Pattern trong C# là một ứng dụng quản lý tài khoản ngân hàng. Trạng thái của tài khoản có thể là "Mở", "Đóng" hoặc "Bị khóa". Mỗi trạng thái sẽ có các hành vi khác nhau như thực hiện giao dịch, rút tiền hoặc kiểm tra số dư.

Dưới đây là một mã nguồn minh họa cho ví dụ trên:

```csharp

// Giao diện định nghĩa các phương thức trạng thái

public interface IAccountState

{

    void Deposit(decimal amount);

    void Withdraw(decimal amount);

    void CheckBalance();

}

// Lớp trạng thái cụ thể "Mở"

public class OpenState : IAccountState

{

    private Account account;

    public OpenState(Account account)

    {

        this.account = account;

    }

    public void Deposit(decimal amount)

    {

        account.Balance += amount;

        Console.WriteLine($"{amount} deposited. Current balance: {account.Balance}");

    }

    public

 void Withdraw(decimal amount)

    {

        if (account.Balance >= amount)

        {

            account.Balance -= amount;

            Console.WriteLine($"{amount} withdrawn. Current balance: {account.Balance}");

        }

        else

        {

            Console.WriteLine("Insufficient balance.");

        }

    }

    public void CheckBalance()

    {

        Console.WriteLine($"Current balance: {account.Balance}");

    }

}

// Lớp trạng thái cụ thể "Đóng"

public class ClosedState : IAccountState

{

    public void Deposit(decimal amount)

    {

        Console.WriteLine("Account closed. Cannot deposit.");

    }

    public void Withdraw(decimal amount)

    {

        Console.WriteLine("Account closed. Cannot withdraw.");

    }

    public void CheckBalance()

    {

        Console.WriteLine("Account closed.");

    }

}

// Lớp trạng thái cụ thể "Bị khóa"

public class LockedState : IAccountState

{

    private Account account;

    public LockedState(Account account)

    {

        this.account = account;

    }

    public void Deposit(decimal amount)

    {

        Console.WriteLine("Account locked. Cannot deposit.");

    }

    public void Withdraw(decimal amount)

    {

        Console.WriteLine("Account locked. Cannot withdraw.");

    }

    public void CheckBalance()

    {

        Console.WriteLine("Account locked.");

    }

}

// Lớp chứa trạng thái và triển khai các phương thức liên quan

public class Account

{

    public decimal Balance { get; set; }

    private IAccountState currentState;

    public Account()

    {

        currentState = new OpenState(this);

    }

    public void Deposit(decimal amount)

    {

        currentState.Deposit(amount);

    }

    public void Withdraw(decimal amount)

    {

        currentState.Withdraw(amount);

    }

    public void CheckBalance()

    {

        currentState.CheckBalance();

    }

    public void ChangeState(IAccountState newState)

    {

        currentState = newState;

    }

}

// Sử dụng:

Account account = new Account();

account.Deposit(100);

account.Withdraw(50);

account.ChangeState(new ClosedState());

account.Deposit(100); // Kết quả: "Account closed. Cannot deposit."

account.CheckBalance(); // Kết quả: "Account closed."

account.ChangeState(new LockedState(account));

account.Withdraw(50); // Kết quả: "Account locked. Cannot withdraw."

```

Trong ví dụ trên, `Account` đóng vai trò là context và các lớp `OpenState`, `ClosedState`, `LockedState` đóng vai trò là các trạng thái cụ thể. Khi gọi các phương thức trên `Account`, chúng sẽ được chuyển tiếp đến trạng thái hiện tại để xử lý tương ứng.

State Design Pattern giúp tăng tính mô-đun và linh hoạt trong việc xử lý các trạng thái khác nhau của đối tượng, đồng thời giúp giảm sự phụ thuộc và dễ bảo trì.

# **Strategy**

Strategy Design Pattern là một trong các mẫu thiết kế phần mềm (design pattern) thuộc nhóm hành vi (behavioral patterns) trong lập trình hướng đối tượng. Mẫu này cho phép bạn xác định một tập hợp các thuật toán khác nhau và đóng gói chúng vào các đối tượng riêng biệt, từ đó có thể chọn lựa thuật toán phù hợp để sử dụng tại thời điểm thực thi.

Trong C#, Strategy Design Pattern thường được sử dụng để tách biệt logic xử lý từ các chi tiết cụ thể của thuật toán, từ đó giúp dễ dàng mở rộng, thay đổi và tái sử dụng code.

Dưới đây là cấu trúc của Strategy Design Pattern:

1. Strategy: Đây là một interface hoặc lớp trừu tượng chứa các phương thức mô tả các hành vi chung của các thuật toán. Ví dụ:

```csharp

public interface IStrategy

{

    void Execute();

}

```

2. Concrete Strategies: Đây là các lớp cụ thể thực hiện giao diện Strategy. Mỗi lớp cung cấp một cách thức cụ thể để thực hiện các thuật toán. Ví dụ:

```csharp

public class ConcreteStrategyA : IStrategy

{

    public void Execute()

    {

        // Thực hiện thuật toán A

    }

}

public class ConcreteStrategyB : IStrategy

{

    public void Execute()

    {

        // Thực hiện thuật toán B

    }

}

```

3. Context: Đây là lớp chứa một đối tượng Strategy và sử dụng nó để thực hiện công việc cụ thể. Lớp Context cho phép thay đổi Strategy tại thời điểm chạy. Ví dụ:

```csharp

public class Context

{

    private IStrategy \_strategy;

    public Context(IStrategy strategy)

    {

        \_strategy = strategy;

    }

    public void ExecuteStrategy()

    {

        \_strategy.Execute();

    }

}

```

Với cấu trúc trên, ta có thể sử dụng Strategy Design Pattern như sau:

```csharp

// Sử dụng ConcreteStrategyA

var context = new Context(new ConcreteStrategyA());

context.ExecuteStrategy();

// Sử dụng ConcreteStrategyB

context = new Context(new ConcreteStrategyB());

context.ExecuteStrategy();

```

Khi chạy, kết quả sẽ tùy thuộc vào Strategy được chọn, và Context sẽ thực hiện công việc thông qua Strategy đó.

Strategy Design Pattern cho phép linh hoạt thay đổi thuật toán trong quá trình chạy, mở rộng thuật toán bằng cách thêm Concrete Strategies mới, và giúp tách biệt logic xử lý, dễ dàng bảo trì và tái sử dụng code

.

# **TemplateMethod**

Template Method là một mẫu thiết kế (design pattern) thuộc nhóm mẫu thiết kế hành vi (behavioral design patterns) trong lập trình. Nó cho phép bạn xác định một khung (skeleton) cho một thuật toán trong một lớp cha, nhưng cho phép các lớp con triển khai cách thức cụ thể cho các bước của thuật toán đó.

Trong C#, Template Method được triển khai bằng cách sử dụng các lớp và phương thức ảo (virtual) trong lập trình hướng đối tượng. Các bước chung của thuật toán được xác định trong lớp cha, trong khi các bước cụ thể được triển khai trong các lớp con.

Dưới đây là một ví dụ đơn giản về cách triển khai Template Method Design Pattern trong C#:

```csharp

using System;

// Lớp cha chứa khung của thuật toán

abstract class AbstractClass

{

    // Phương thức Template Method

    public void TemplateMethod()

    {

        Step1();

        Step2();

        Step3();

    }

    // Các bước chung của thuật toán

    protected void Step1()

    {

        Console.WriteLine("Step 1");

    }

    protected void Step3()

    {

        Console.WriteLine("Step 3");

    }

    // Các bước cụ thể của thuật toán - để triển khai trong lớp con

    protected abstract void Step2();

}

// Lớp con 1 triển khai các bước cụ thể của thuật toán

class ConcreteClass1 : AbstractClass

{

    protected override void Step2()

    {

        Console.WriteLine("ConcreteClass1: Step 2");

    }

}

// Lớp con 2 triển khai các bước cụ thể khác của thuật toán

class ConcreteClass2 : AbstractClass

{

    protected override void Step2()

    {

        Console.WriteLine("ConcreteClass2: Step 2");

    }

}

class Program

{

    static void Main(string[] args)

    {

        AbstractClass class1 = new ConcreteClass1();

        AbstractClass class2 = new ConcreteClass2();

        // Gọi Template Method trên các đối tượng khác nhau

        class1.TemplateMethod();

        class2.TemplateMethod();

        Console.ReadKey();

    }

}

```

Kết quả khi chạy chương trình:

```

Step 1

ConcreteClass1: Step 2

Step 3

Step 1

ConcreteClass2: Step 2

Step 3

```

Trong ví dụ trên, lớp cha `AbstractClass` xác định khung của thuật toán thông qua phương thức `TemplateMethod()`, trong đó các bước chung `Step1()` và `Step3()` được triển khai sẵn. Các lớp con `ConcreteClass1` và `ConcreteClass2` triển khai bước cụ thể `Step2()` theo cách mà chúng muốn.

Qua việc sử dụng Template Method Design Pattern, bạn có thể x

ác định một khung chung cho một thuật toán và cho phép các lớp con triển khai các bước cụ thể mà không làm thay đổi cấu trúc của thuật toán. Điều này giúp tăng tính mô đun hóa và tái sử dụng trong mã nguồn của bạn.