Design Patterns

BnD Document version 1.0

Mục lục

Pattern là gì ?

Pattern mô tả một giải pháp chung đối với một vấn đề nào đó trong thiết kế thường được “lặp lại” trong nhiều dự án. Nói một cách khác, một pattern có thể được xem như một “khuôn mẫu” có sẵn áp dụng được cho nhiều tình huống khác nhau để giải quyết một vấn đề cụ thể.

Định nghĩa

- Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày.
- Là một mẫu hoặc mô tả cách giải quyết 1 vấn đề
- Là một kỹ thuật đã được kiểm chứng, Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác , nó là một kĩ thuật lập trình

Khi nào nên sử dụng Design patterns

Khi bạn muốn giữ cho chương trình của mình thực sự đơn giản. Việc sử dụng các design pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các cách giải quyết cho những vấn đề đã có lời giải.

Ưu điểm

1. Tăng tốc độ phát triển phần mềm

Loại bỏ thời gian thừa của developer khi suy nghĩ giải pháp cho một vấn đề, Design Pattern đưa ra các mô hình test và mô hình phát triển đã qua kiểm nghiệm giúp developer có được hướng giải quyết nhanh chóng và hiệu quả.

2. Hạn chế lỗi tiềm ẩn

Sử dụng giải pháp đã được chứng minh và công nhận thì hẳn là sẽ giảm bớt rủi ro hơn là tự mình thử nghiệm giải pháp mới.

Ưu điểm

3. Hỗ trợ tái sử dụng mã lệnh

Các mẫu thiết kế có thể được sử dụng hàng triệu lần mà không nảy sinh bất cứ vấn đề nào. Developer cũng dễ dàng mở rộng, nâng cấp và bảo trì để đáp ứng được các yêu cầu thay đổi liên tục của dự án.

4. Giúp code dễ đọc hơn

Việc sử dụng Design Pattern giúp cho code dễ đọc hơn, developer khi làm việc nhóm cũng giao tiếp thuận lợi hơn vì có được tiếng nói chung.

Nhược điểm

Lý thuyết

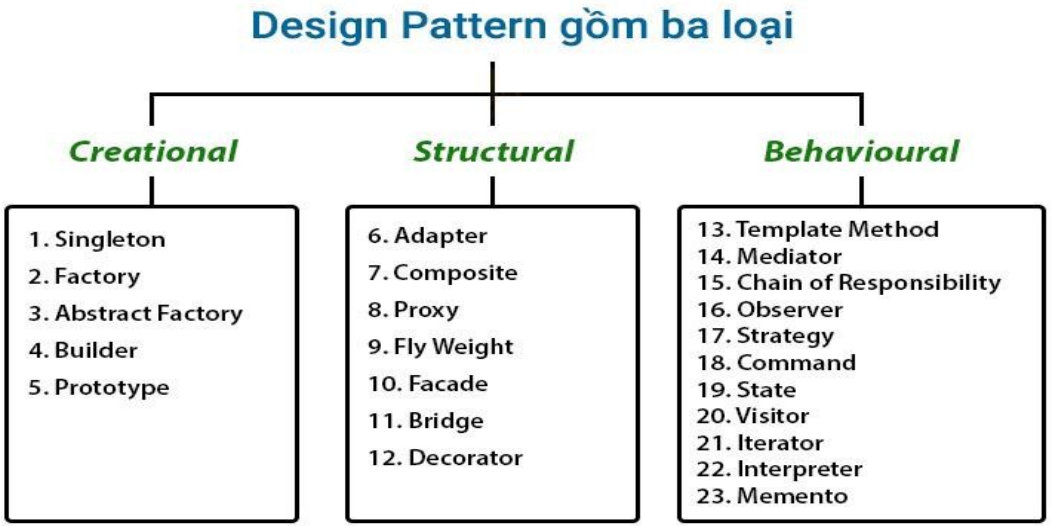
Đây là một lĩnh vực khó nhằn và hơi trừu tượng. Sẽ rất dễ dàng nếu bạn code từ đầu và ứng dụng các mẫu thiết kế vào sản phẩm, tuy nhiên với các code cũ thì sẽ rất khó khăn

Ứng dụng

Design Pattern có thể sẽ gây ảnh hưởng tới quá trình làm việc của sản phẩm, ví dụ như các code có thể sẽ chạy chậm. Do đó, hãy nắm chắc bạn hiểu toàn bộ về mã nguồn trước khi bắt đầu sử dụng các mẫu thiết kế.

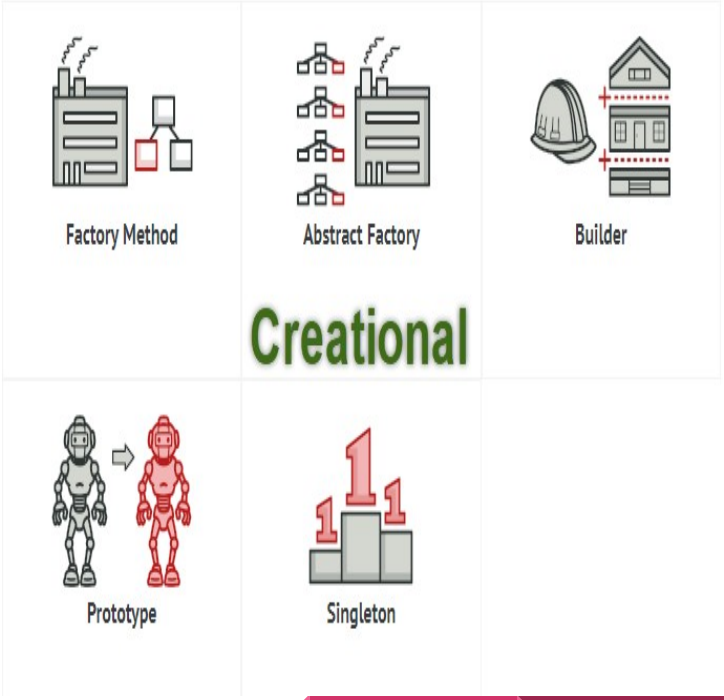
Phân loại

Hệ thống các mẫu Design pattern hiện có **23 mẫu** và được chia thành **3 nhóm**:



Nhóm Creational

01 (nhóm khởi tạo – 5 mẫu)

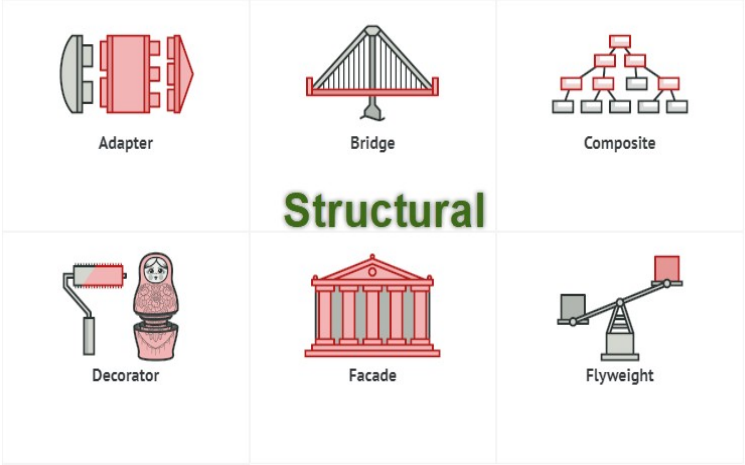


Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.

- **Singleton**: Tần suất sử dụng: **Cao**.
- **Abstract Factory**: Tần suất sử dụng: **Cao**.
- **Factory Method**: Tần suất sử dụng: **Cao**.
- **Builder**: Tần suất sử dụng: **Trung bình thấp.**
- **Prototype:** Tần suất sử dụng: **Trung bình.**

Nhóm Structural
02 (nhóm cấu trúc – 7 mẫu)

Những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

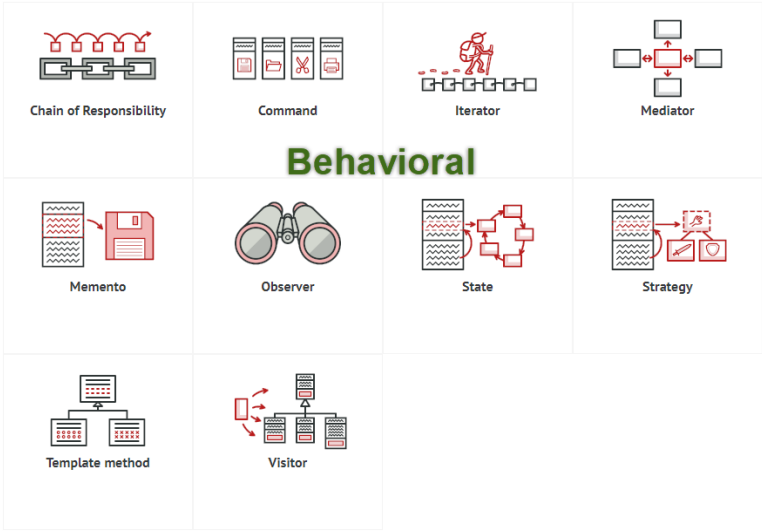


- **Adapter:** Tần suất sử dụng: **Cao trung bình.**
- **Bridge:** Tần suất sử dụng: **Trung bình.**
- **Composite:** Tần suất sử dụng: **Cao trung bình**
- **Decorator:** Tần suất sử dụng: **Trung bình**
- **Facade:** Tần suất sử dụng: **Cao.**
- **Flyweight:** Tần suất sử dụng: **Thấp.**
- **Proxy:** Tần suất sử dụng: **Cao trung bình.**

Nhóm Behavioral
03 (nhóm tương tác/ hành vi – 11 mẫu)

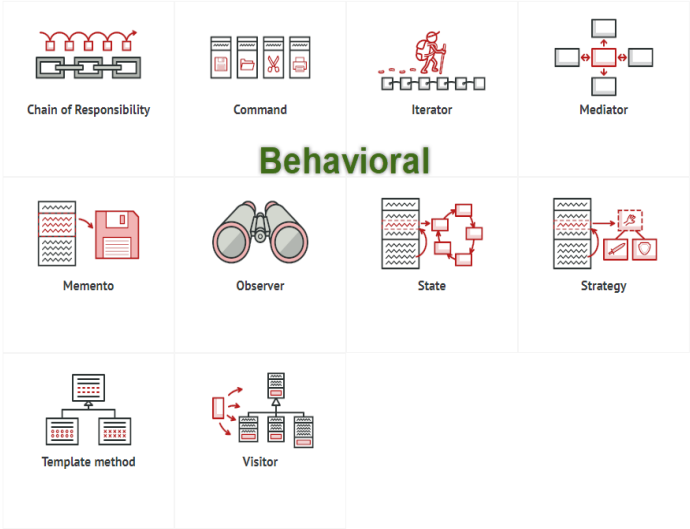
Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.

- **Chain of Responsibility:** Tần suất sử dụng: **Trung bình thấp.**
- **Command:** Tần suất sử dụng: **Cao trung bình.**
- **Interpreter:** Tần suất sử dụng: **Thấp.**
- **Iterator:** Tần suất sử dụng: **Cao.**
- **Mediator:** Tần suất sử dụng: **Trung bình thấp.**



Nhóm Behavioral
03 (nhóm tương tác/ hành vi – 11 mẫu)

- **Memento:** Tần suất sử dụng: **Thấp.**
- **Observer**: Tần suất sử dụng: **Cao.**
- **State**: Tần suất sử dụng: **Trung bình.**
- **Strategy**: Tần suất sử dụng: **Cao trung bình.**
- **Template method**: Tần suất sử dụng: **Trung bình.**
- **Visitor**: Tần suất sử dụng: **Thấp.**



Một số Design patterns hay sử dụng

BnD Document version 1.0

Singleton Pattern

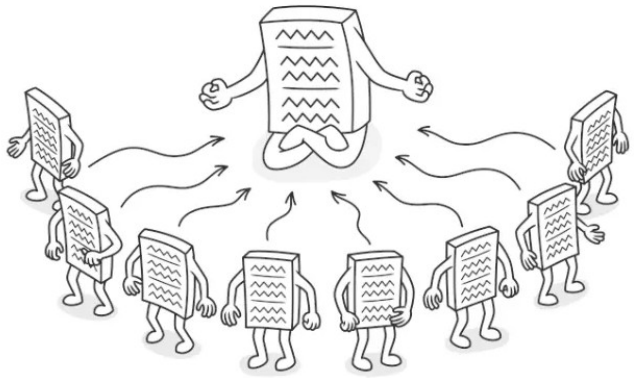
Vấn đề đặt ra

Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn?

Ví dụ

Trong cuộc sống

Tại mỗi thời điểm, công ty có một và chỉ một Ban giám đốc mà cách thức hoạt động, quyền hạn,... đã được quy định trước. Bất kể giám đốc tên họ là gì, nam hay nữ, theo tôn giáo nào hay không,... ta luôn "truy cập" thông qua chức danh giám đốc trong toàn bộ công ty.



Trong lĩnh vực phần mềm

Khi bạn cần đối tượng thực hiện chức năng cụ thể, như ghi log cho chương trình, nó nên được viết thành Singleton. Bất kể tại phần nào trong chương trình, chỉ có một thể hiện, chỉ có một cách truy cập đến Logger.

Singleton Pattern

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SingletonDesignPattern
{
    8 references
    public sealed class Singleton
    {
        private static int counter = 0;
        private static Singleton instance = null;
        2 references
        public static Singleton GetInstance
        {
            get
            {
                if (instance == null)
                    instance = new Singleton();
                return instance;
            }
        }

        1 reference
        private Singleton()
        {
            counter++;
            Console.WriteLine("Counter Value " + counter.ToString());
        }
        2 references
        public void PrintDetails(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

```
namespace SingletonPatternDemo
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Singleton fromWebsite = Singleton.GetInstance;
            fromWebsite.PrintDetails("From Website Vitanedu");
            Singleton fromCms = Singleton.GetInstance;
            fromCms.PrintDetails("From CMS");

            Console.Write("\n-----------------------------\n");
            Console.ReadLine();
        }
    }
}
```

```
Counter Value 1
From Website Vitanedu
From CMS

-----------------------------
```

Singleton Pattern

```
const httpClient = (function () {
  // private method
  function sendRequest(url, config) {
    // send request
  }
  return {
    get(url, config) {
      return sendRequest(url, config);
    },
    post(url, config) {
      return sendRequest(url, config);
    },
  };
})();
```

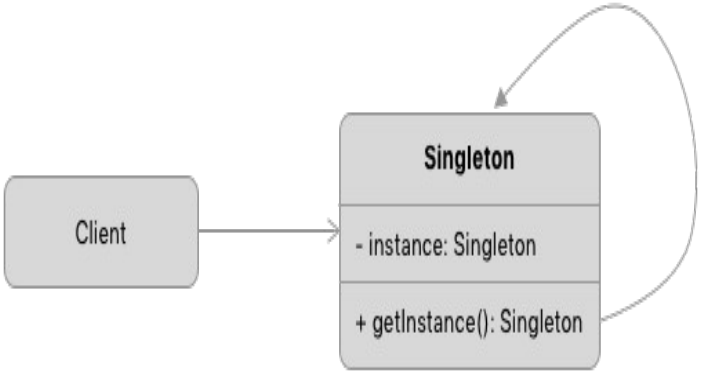
Singleton Pattern

Định nghĩa

01 | Đảm bảo class chỉ có một thể hiện (instance) duy nhất

02 | Đảm bảo có thể truy cập mọi lúc, mọi nơi

03 | Khởi tạo ở lần gọi đầu tiên



Hạn chế khởi tạo đối tượng, giảm bớt được khai báo đối tượng dư thừa, chỉ khởi tạo một lần duy nhất và có thể truy cập toàn cục.

Singleton Pattern

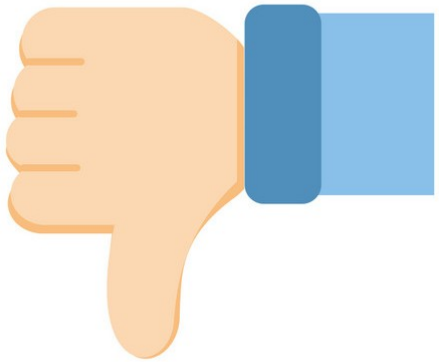
Ưu điểm

- Singleton quản lý rất tốt việc đồng thời chia sẻ tài nguyên cho nhiều Object.
- Có thể chia sẻ dữ liệu chung, nghĩa là dữ liệu chính và dữ liệu cấu hình không bị thay đổi thường xuyên trong một ứng dụng.
- Rất dễ bảo trì, thay vì phải đi chỉnh sửa tất cả các object, ta chỉ cần chỉnh một lần duy nhất ở Singleton Object mà thôi.
- Giảm thiểu bộ nhớ, thời gian, bởi vì chỉ cần tạo một lần và sử dụng nhiều lần.

Singleton Pattern

Nhược điểm

- Tạo ra quá nhiều phụ thuộc, không thể sử dụng đa hình và dễ tạo ra các bug
- Singleton yêu cầu xử lý đặc biệt trong một trường đa luồng, để nhiều luồng không tạo ra một đối tượng Singleton nhiều lần.
- Singleton Design Pattern có thể ẩn đi những thiết kế xấu cho instance khi các component trong chương trình biết rõ về nhau.



Singleton Pattern

Khi nào sử dụng

- Vì class dùng Singleton chỉ tồn tại 1 Instance nên nó thường được dùng cho các trường hợp giải quyết những bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Thread pool,..

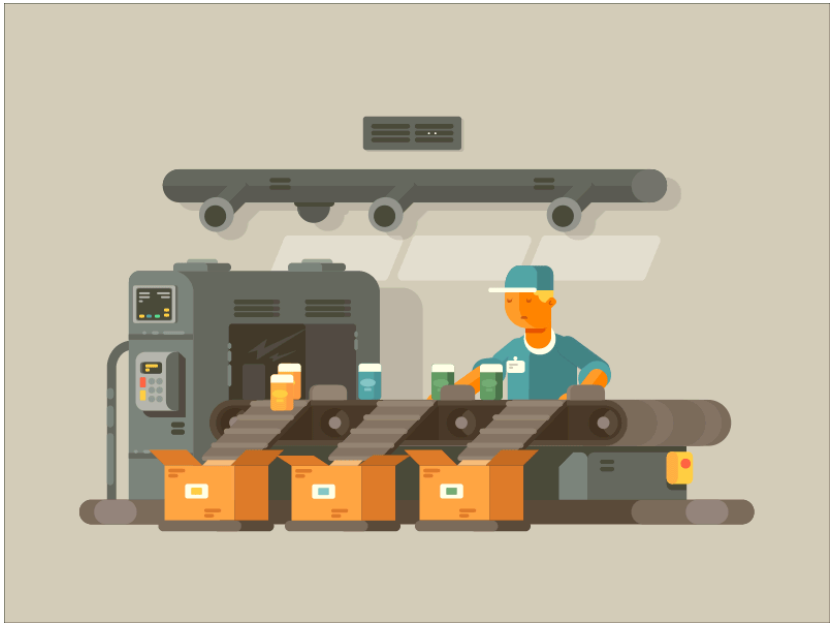
Singleton Pattern

Ứng dụng trong dự án

- Kết nối database
- Logger
- Configuration

Factory Pattern

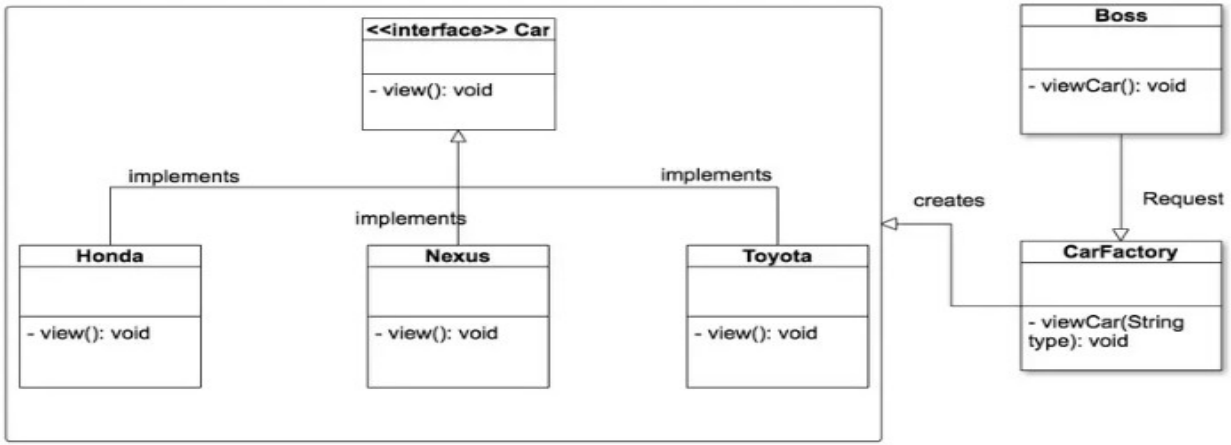
Vấn đề đặt ra



Factory Pattern : quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn.

Ví dụ

Giả sử bạn muốn mua một chiếc xe ô tô, bạn sẽ phải đến các cửa hàng (Honda, Lexus, Toyota..) để xem xét các xe trước khi mua. Các cửa hàng sẽ đưa xe ra cho bạn xem.
Việc làm này mất thời gian và công sức của bạn khi đến từng cửa hàng để xem. Tuy nhiên có một cách khác đơn giản hơn, đó là đến một đại lý ô tô có bán nhiều hãng để xem xét hết tất cả các xe.



Factory Pattern

Bước 1: Tạo interface
Tạo interface Car.cs

```
3 references
public interface Car
{

    3 references
    void view();
}
```

Bước 2 : Tạo những Class cụ thể implement interface
Honda.cs
Toyota.cs
Lexus.cs

```
0 references
public class Honda : Car
{
    1 reference
    public void view()
    {
        Console.WriteLine("Honda view");
    }
}
```

```
0 references
public class Toyota : Car
{
    1 reference
    public void view()
    {
        Console.WriteLine("Toyota view");
    }
}
```

Factory Pattern

Bước 3: Tạo class Factory để khởi tạo các lớp cụ thể dựa vào thông tin đã cho
CarFactory.cs

```
2 references
public class CarFactory
{
    3 references
    public Car viewCar(String carType)
    {
        if (carType == null)
        {
            return null;
        }

        if (carType == "HONDA")
        {
            return new Honda();
        }
        else if (carType == "LEXUS")
        {
            return new Lexus();
        }
        else if (carType == "TOYOTA")
        {
            return new Toyota();
        }
    }
}
```

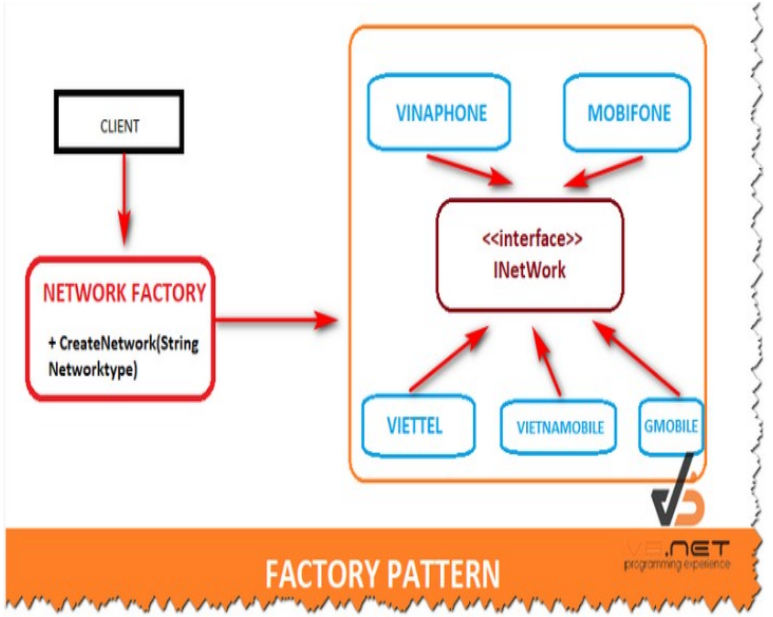
Bước 4 : Sử dụng
Boss.cs

```
0 references
public class Boss
{
    0 references
    public void viewCar()
    {
        CarFactory carFactory = new CarFactory();
        carFactory.viewCar("HONDA");
        carFactory.viewCar("LEXUS");
        carFactory.viewCar("TOYOTA");
    }

}
```

Factory Pattern

Định nghĩa

- Là thiết kế mẫu hướng đối tượng trong việc thiết kế phần mềm cho máy tính, nhằm giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp nào sẽ được tạo.
- Nói chung, "factory method" thường được áp dụng cho những phương thức mà nhiệm vụ chính của nó là tạo ra đối tượng



Factory Pattern

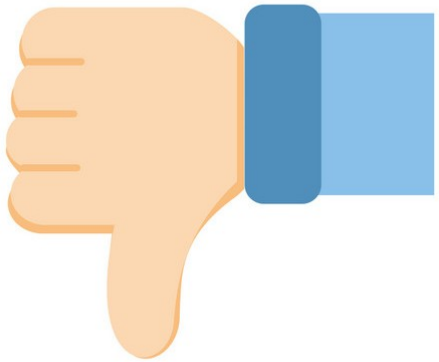
Ưu điểm

- Tránh việc gắn chặt việc tạo sản phẩm với bất kỳ một loại sản phẩm cụ thể nào.
- Factory Method Pattern giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.
- Dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần phá vỡ các đối tượng ban đầu

Factory Pattern

Nhược điểm

- Source code có thể trở nên phức tạp hơn mức bình thường do đòi hỏi phải sử dụng nhiều class mới có thể cài đặt được pattern này.
- Việc refactoring (tái cấu trúc) một class bình thường có sẵn thành một class có Factory Method có thể dẫn đến nhiều lỗi trong hệ thống



Factory Pattern

Khi nào sử dụng

- Sử dụng Factory Method khi bạn không biết trước kiểu và các phụ thuộc của object mà code sẽ làm việc với nó.
- Sử dụng Factory Method khi bạn muốn cung cấp cho người dùng thư viện hoặc framework của bạn một cách để mở rộng các thành phần sẵn có bên trong nó.
- Sử dụng Factory Method khi bạn muốn tiết kiệm tài nguyên hệ thống bằng việc tái sử dụng các object đã có thay vì xây dựng lại mỗi lần có thêm sản phẩm mới.

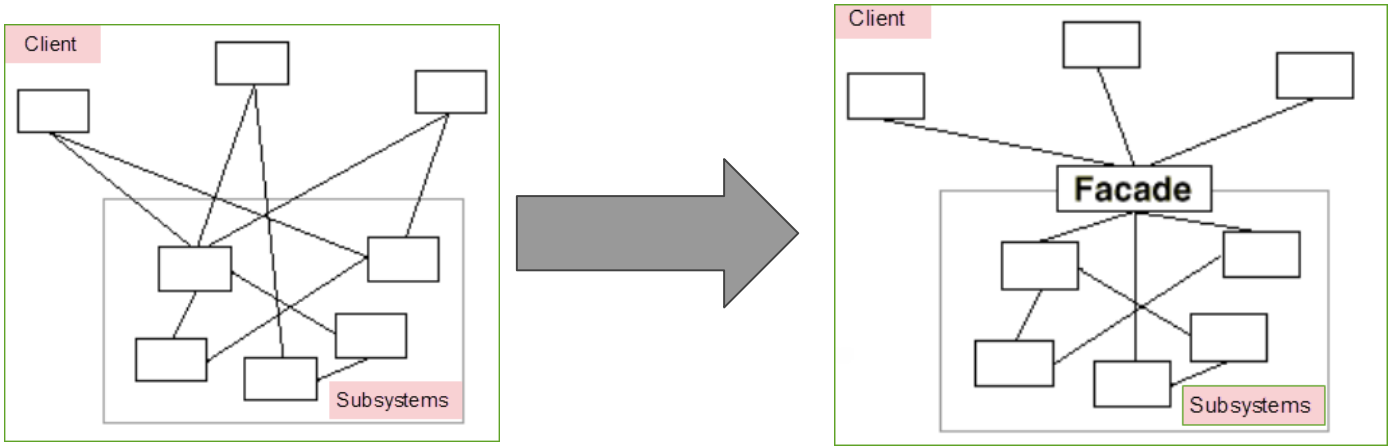
Factory Pattern

Ứng dụng trong dự án

- Thư viện dùng chung trong dự án
- Sdk, Sdk Reaction....

Facade Pattern

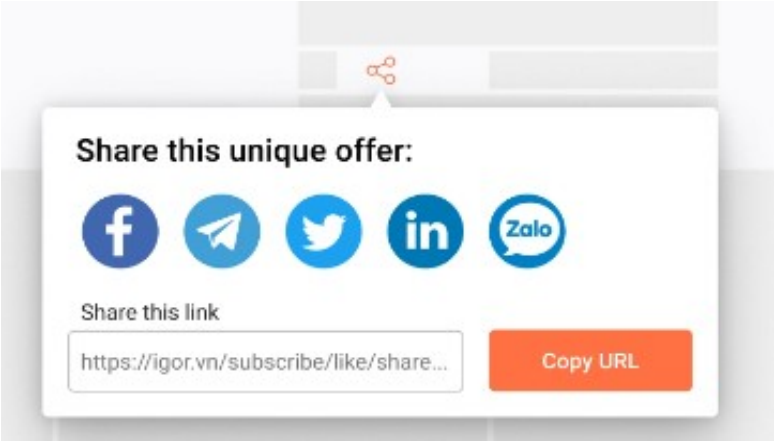
Vấn đề đặt ra



Chúng ta cần thiết kế một Facade, và trong đó phương thức của Facade sẽ xử lý các đoạn code được sử dụng với nhiều, lặp lại. Với Facade, chúng ta sẽ chỉ cần gọi Facade để thực thi các hành động dựa trên các parameters được cung cấp.

Ví dụ

Cung cấp 1 API đơn giản chia sẻ lên các nền tảng mạng xã hội



Facade Pattern

Bước 1: Tạo interface
Tạo interface IShare.cs

```
public interface IShare
{

    0 references
    void share();
    0 references
    void setMessage(string Message);
}
```

Facade Pattern

Bước 2 : Tạo những Class cụ thể implement interface
FacebookShare.cs
ZaloShare.cs

```
3 references
public class FacebookShare : IShare
{
    private string message;
    2 references
    public void setMessage(string message)
        {
            this.message = message;
        }
    2 references
    public void share()
    {
        //Logic connect API Facebook
        Console.WriteLine("Share to facebook" + message);
    }
}
```

```
3 references
public class ZaloShare : IShare
{
    private string message;
    2 references
    public void setMessage(string message)
    {
        this.message = message;
    }
    2 references
    public void share()
    {
        //Logic connect API Zalo
        Console.WriteLine("Share to Zalo" + message);
    }
}
```

Facade Pattern

Bước 3: Tạo class SocialMediaShare để khởi tạo các lớp cụ thể dựa vào thông tin đã cho SocialMediaShare.cs

```
public class SocialMediaShare
{
    private FacebookShare facebookShare;
    private ZaloShare zaloShare;

    0 references
    public SocialMediaShare (FacebookShare facebookShare, ZaloShare zaloShare)
    {
        this.facebookShare = facebookShare;
        this.zaloShare = zaloShare;
    }
    0 references
    public void share (string message)
    {
        this.facebookShare.setMessage(message);
        this.zaloShare.setMessage(message);

        this.facebookShare.share();
        this.zaloShare.share();

    }
}
```

Facade Pattern

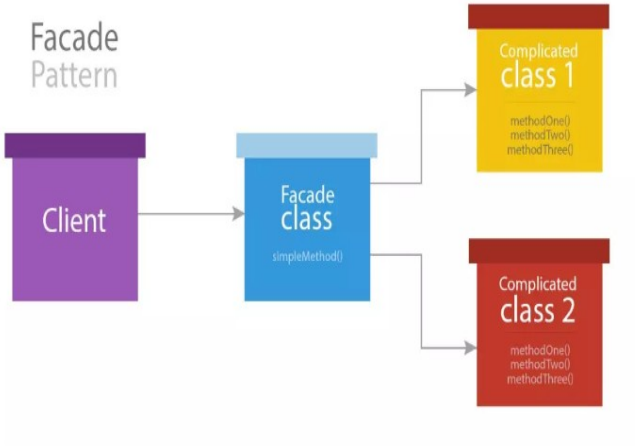
Bước 4 : Sử dụng
Boss.cs

```
0 references
public class Boss
{
    0 references
    public void share()
    {
        SocialMediaShare socialMediaShare = new SocialMediaShare(new FacebookShare(), new ZaloShare());
        socialMediaShare.share("Chia sẻ vitanedu");

    }

}
```

Facade Pattern

Định nghĩa

- Facade Pattern cung cấp cho chúng ta một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.



Facade Pattern

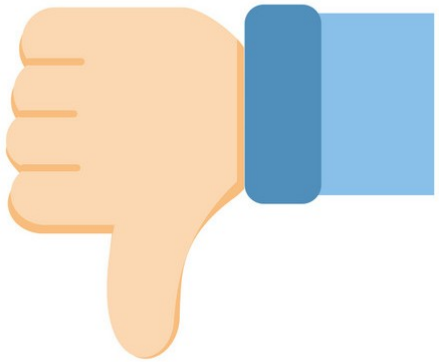
Ưu điểm

- Ta có thể tách mã nguồn của mình ra khỏi sự phức tạp của hệ thống con
- Hệ thống tích hợp thông qua Facade sẽ đơn giản hơn vì chỉ cần tương tác với Facade thay vì hàng loạt đối tượng khác.
- Tăng khả năng độc lập và khả chuyển, giảm sự phụ thuộc.
- Có thể đóng gói nhiều hàm được thiết kế không tốt bằng 1 hàm có thiết kế tốt hơn.

Facade Pattern

Nhược điểm

- Class Facade của bạn có thể trở lên quá lớn, làm quá nhiều nhiệm vụ với nhiều hàm chức năng trong nó.
- Việc sử dụng Facade cho các hệ thống đơn giản, ko quá phức tạp trở nên dư thừa.



Facade Pattern

Khi nào sử dụng

- Muốn gom nhóm chức năng lại để Client dễ sử dụng
- Giảm sự phụ thuộc. Khi bạn muốn phân lớp các hệ thống con. Dùng Facade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con
- Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Facade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác
- Đóng gói nhiều chức năng, che giấu thuật toán phức tạp.

Facade Pattern

Ứng dụng trong dự án

- Chia sẻ mạng xã hội
- Đặt hàng thanh toán

Observer Pattern

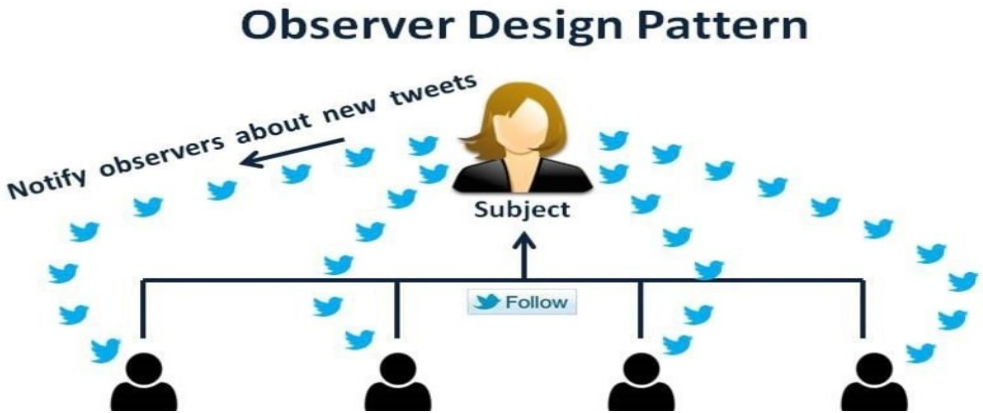
Vấn đề đặt ra

Nếu bạn subscribe một kênh youtube, bạn không cần phải vào mục tìm kiếm xem đã có video mới hay không. Thay vào đó, chủ kênh sẽ gửi các video mới trực tiếp đến hộp thư của bạn ngay sau khi xuất bản hoặc thậm chí trước.

Chủ kênh giữ danh sách những người đăng ký . Người đăng ký có thể rời khỏi danh sách bất kỳ lúc nào khi họ không muốn chủ kênh gửi các thông tin video mới cho họ nữa.

Ví dụ

Giả sử chúng ta có 1 hotgirl và một đám con trai crush cô ấy. Hiển nhiên, mỗi lần cô ta đăng status than ế thì cả đám lập tức sồn sồn lên. Ở trường hợp này, cô gái chính là `Subject`, đám con trai kia là `Observer`, sự kiện cho là post facebook đi



Observer Pattern

Bước 1: Khai báo Publisher interface

Interface Hotgirl.cs

```
1 reference
public class HotGirl
{
    private bool needAttention = false;

    // Some of boys crushing this instance :))
    public IList<Boy> FriendZone = new List<Boy>();

    1 reference
    public void PostFacebook()
    {
        Console.WriteLine("Hôm nay em buồn....");
        NeedAttention = true;
    }

    // State of instance. When state change, observe will know and react
    1 reference
    private bool NeedAttention
    {
        get => needAttention;
        set
        {
            needAttention = value;
            Notify();
        }
    }

    1 reference
    public void Notify()
    {
        foreach (var b in FriendZone)
        {
            b.Care();
        }
    }

    // Register observer.
    2 references
    public void AddToZone(Boy b)
    {
        FriendZone.Add(b);
    }
}
```

Observer Pattern

Bước 2: Khai báo publisher interface

Tạo interface Boy.cs

```
0 references
public class Boy
{
    public string Name;

    2 references
    public Boy(string name)
    {
        Name = name;
    }

    1 reference
    public void Care()
    {
        Console.WriteLine($"{Name}: Em có chuyện gì thế?");
    }
}
```

Observer Pattern

Bước 3 : Sử dụng
Main.cs

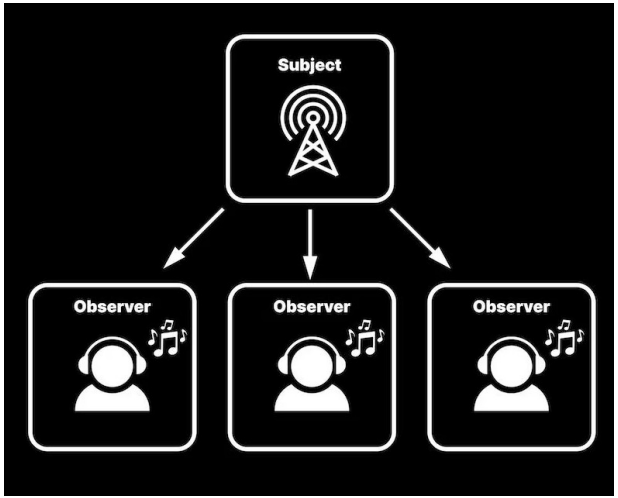
```
0 references
static void Main(string[] args)
{
    // Create a girl instance as subject (publisher)
    var sexyGirl = new HotGirl();

    //// Add 2 boys to her friend zone.
    sexyGirl.AddToZone(new Boy("TiepNV"));
    sexyGirl.AddToZone(new Boy("MinhNT"));

    // One day. She need some attention, so she post to facebook.
    // After she complete post facebook. These poor boys react.
    sexyGirl.PostFacebook();
    Console.ReadKey();
}
```

```
Hôm nay em buồn....
TiepNV: Em có chuyện gì thế?
MinhNT: Em có chuyện gì thế?
```

Observer Pattern

Định nghĩa

- Định nghĩa mối phụ thuộc một - nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất cả các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.
- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác



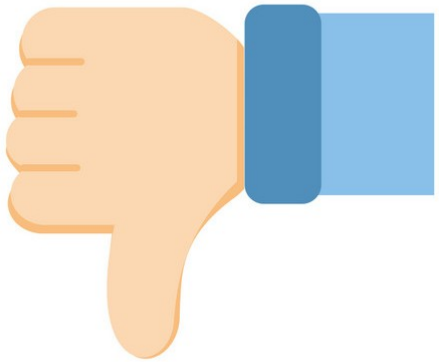
Observer Pattern

Ưu điểm

- Cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác.
- Thiết lập mối quan hệ giữa các objects trong thời gian chạy.
- Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
- Không giới hạn số lượng Observer

Observer Pattern

Nhược điểm



- Rò rỉ bộ nhớ gây ra bởi sự cố vì đăng ký lắng nghe thì nhiều và rõ ràng nhưng việc hủy người đăng ký thì không có, dẫn tới nhiều chỗ không còn dùng tới việc đăng ký mà vẫn đang kết nối với Chủ đề.
- Không kiểm soát được thứ tự subscriber nhận thông báo.

Observer Pattern

Khi nào sử dụng

- Sự thay đổi trạng thái ở 1 đối tượng cần được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
- Cần mở rộng dự án với ít sự thay đổi nhất.
- Khi thay đổi một đối tượng yêu cầu việc thay đổi đến các đối tượng khác, và bạn không biết số lượng đối tượng cần thay đổi.

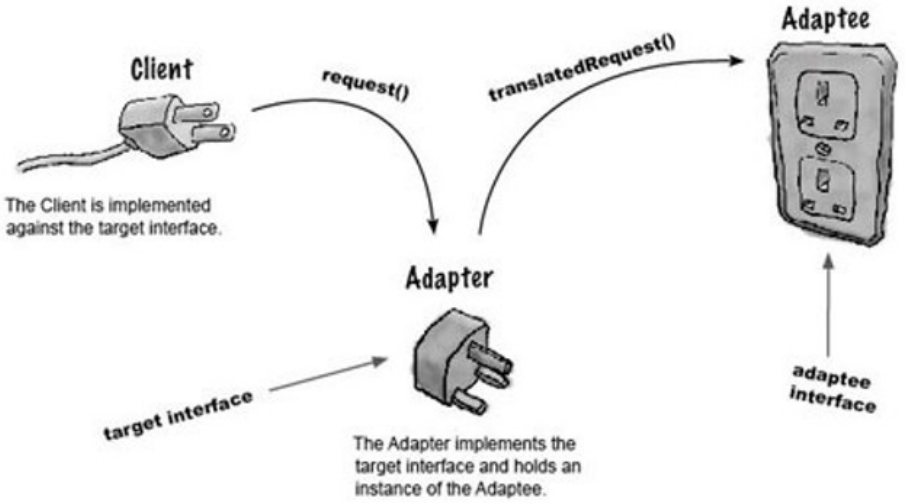
Observer Pattern

Ứng dụng trong dự án

- Thông báo
- Messenger
- Tài khoản hết hạn...
- Theo dõi tin tức....

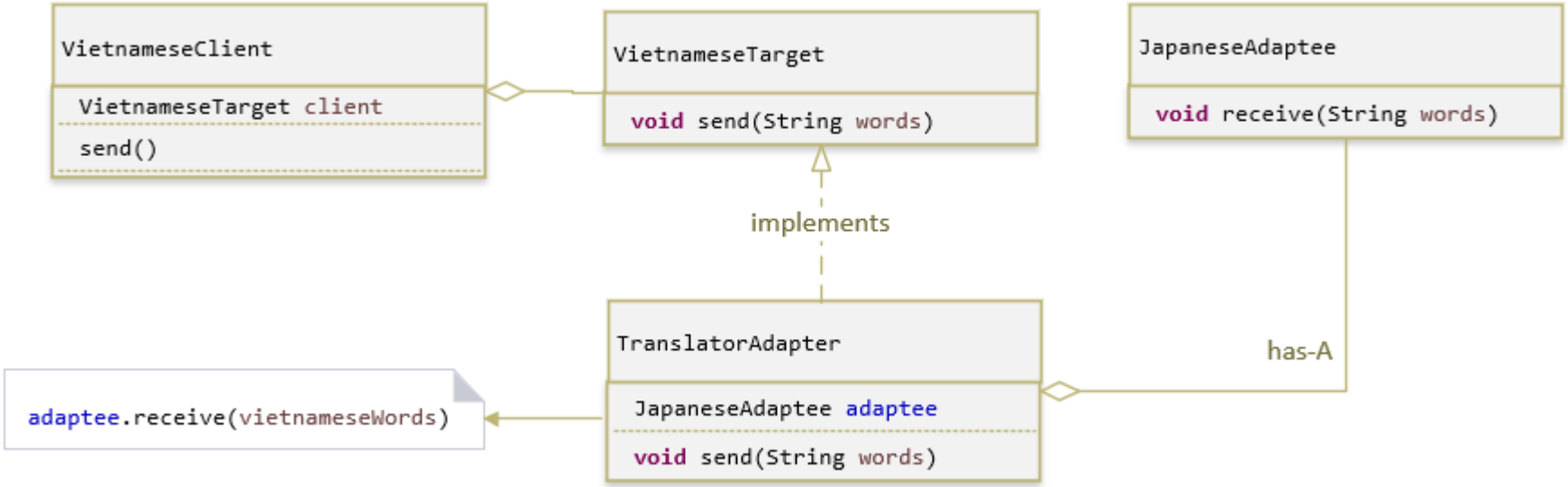
Adapter Pattern

Vấn đề đặt ra



Ví dụ

Một người Việt muốn trao đổi với một người Nhật. Tuy nhiên, 2 người này không biết ngôn ngữ của nhau nên cần phải có một người để chuyển đổi từ ngôn ngữ tiếng Việt sang ngôn ngữ tiếng Nhật



Adapter Pattern

Bước 1: Khai báo Target
VietnameseTarget.cs

```
/// <summary>
/// The 'VietnameseTarget' class
/// </summary>
2 references
public class VietnameseTarget
{
    2 references
    public virtual void Request(String vietnameseWords)
    {
    }
}
```

Adapter Pattern

Bước 2: Khai báo Adaptee

Tạo JapaneseAdaptee .cs

```
/// <summary>
/// The 'JapaneseAdaptee' class
/// </summary>
2 references
public class JapaneseAdaptee
{
    1 reference
    public void SpecificRequest(String words)
    {
        Console.WriteLine("Result Translation..." + words);
    }
}
```

Adapter Pattern

Bước 3: Khai báo Adapter
Adapter.cs

```
/// <summary>
/// The 'Adapter' class
/// </summary>
1 reference
public class Adapter : VietnameseTarget
{
    private JapaneseAdaptee JapaneseAdaptee = new JapaneseAdaptee();
    2 references
    public override void Request(String words)
    {
        // Possibly do some other work
        // and then call SpecificRequest
        Console.WriteLine("Reading Words ...");
        Console.WriteLine(words);
        String vietnameseWords = translate(words);
        Console.WriteLine("Sending Words ...");
        JapaneseAdaptee.SpecificRequest(vietnameseWords);
    }
    1 reference
    private String translate(String vietnameseWords)
    {
        Console.WriteLine("Translated!");
        return "Hello";
    }
}
```

Adapter Pattern

Bước 4 : Sử dụng
Main.cs

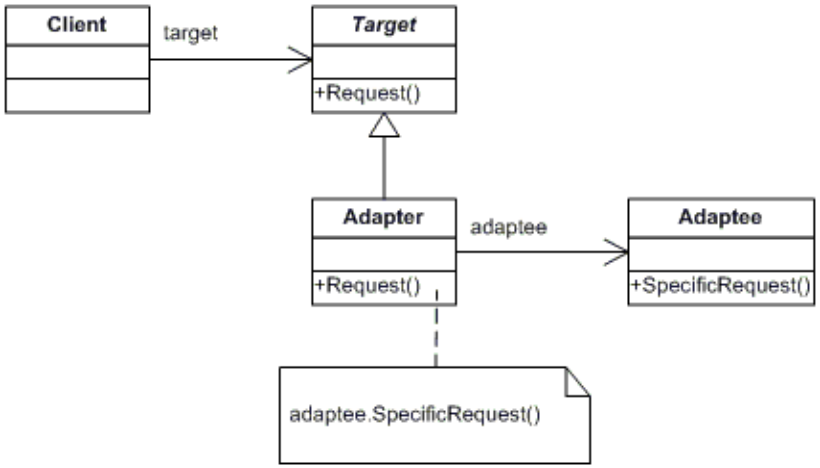
```
0 references
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        // Create adapter and place a request
        VietnameseTarget VietnameseTarget = new Adapter();
        VietnameseTarget.Request("Xin chào");
        // Wait for user
        Console.ReadKey();
    }
}
```

```
Reading Words ...
Xin chào
Translated!
Sending Words ...
Result Translation...Hello
```

Adapter Pattern

Định nghĩa

Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian



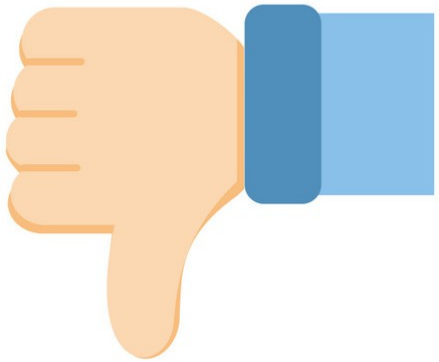
Adapter Pattern

Ưu điểm

- Làm việc với adapter class mà không thay đổi dữ liệu bên trong adapter, điều này giúp thuận tiện cho việc mở rộng.
- Có thể sử dụng adapter với các class con của adaptee.
- Cho phép nhiều interface khác nhau giao tiếp với nhau

Adapter Pattern

Nhược điểm



- Tất cả các yêu cầu phải được thông qua adapter, điều này làm tốn thêm chi phí.
- Làm tăng thêm độ phức tạp của code, vì phải tạo thêm interface và các class.
- Dễ gây ra các exception.

Adapter Pattern

Khi nào sử dụng

- Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
- Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.

Adapter Pattern

Ứng dụng trong dự án

- Chuyển đổi ngôn ngữ
- Chuyển đổi tiền tệ
- Thay đổi nhà cung cấp phương thức thanh toán ...

Tài liệu tham khảo

Sách:

- Làm chủ các mẫu Thiết kế kinh điển trong lập trình - Tạ Văn Dũng
- Design Patterns For Dummies
- Pattern Hatching: Design Patterns Applied.
- Refactoring to Patterns.
- Patterns of Enterprise Application Architecture.

Các Website viết về Design Pattern:

- https://sourcemaking.com/design_patterns
- https://refactoring.guru
- https://www.tutorialspoint.com/design_pattern/index.htm
- https://gpcoder.com/4164-gioi-thieu-design-patterns/#De_hoc_Design_Patterns_can_co_gi
- dotNet Design Patterns: https://www.dofactory.com/net/design-patterns
- Javascript Design Patterns: https://www.dofactory.com/javascript/design-pattern
- Flutter Design Patterns: https://flutterdesignpatterns.com/

Xin cảm ơn!.