

The Interface Segregation Principle (ISP)

“Clients should not be forced to depend upon methods that they do not use.”

ISP để giải quyết vấn đề của những interface quá lớn. Interface chứa nhiều nhóm method khác nhau không thực sự liên quan đến cùng một client. Khi đó mỗi client sẽ chỉ sử dụng một phần interface nhưng vẫn buộc phải triển khai toàn bộ các phần còn lại.

Ví dụ 1

```
public interface IPlayerService
{
    void Move();
    void Attack();

    void Save();
    void Load();

    void ShowInventory();
    void ShowSettings();
}
```

Với thiết kế này chúng ta có thể hình dung ra `IPlayerService` có 3 nhóm client:

- Gameplay sử dụng `Move()`, `Attack()`.
- SaveLoad sử dụng `Save()`, `Load()`.
- UI sử dụng `ShowInventory()`, `ShowSettings()`.

Nhưng cả 3 đều đang sử dụng `IPlayerService`, điều này tạo ra sự phụ thuộc lớn hơn nhu cầu cần thiết của từng client. Nếu thay đổi `Save()` thành `Save(string slot)` thì client Gameplay và UI cũng bị ảnh hưởng.

Đó chính là coupling mà ISP muốn loại bỏ.

Với ví dụ này chúng ta có thể thiết kế như sau:

```

public interface IPlayerController
{
    void Move();
    void Attack();
}

public interface IPlayerPersistence
{
    void Save();
    void Load();
}

public interface IPlayerUI
{
    void ShowInventory();
    void ShowSettings();
}

```

Với thiết kế này client hoàn toàn không cần biết và không bị kéo theo những dịch vụ không cần thiết. Khi cần một service chung client vẫn có thể triển khai `PlayerService : IPlayerController, IPlayerPersistence, IPlayerUI`

Điểm quan trọng của ISP không phải là chia nhỏ class hay interface mà là dependency của client phải nhỏ.

Ví dụ 2

Ban đầu, chúng ta bắt đầu với `IDoor`

```

public interface Door
{
    void Lock();
    void Unlock();
    bool IsDoorOpen();
}

```

Nhưng sau đó phát sinh thêm một loại cửa đặc biệt có thể tự động đóng/mở theo thời gian. Rất dễ xảy ra trường hợp chúng ta sẽ thêm method `TimeOut()` vào interface như dưới đây.

```

public interface Door
{
    void Lock();
    void Unlock();
    bool IsDoorOpen();

    void TimeOut();
}

```

Nhưng vấn đề là với thiết kế trên thì mọi cánh cửa đều cần phải có `TimeOut()` mặc dù nó không có tính năng tự động.

Door đã bị thêm một method chỉ vì một client đặc biệt cần đến nó. Đây gọi là **interface pollution**. Nếu tiếp tục với cách làm này. Mỗi nhu cầu mới của subclass lại thêm method vào base interface và interface đó sẽ càng ngày càng "fat".

Để giải quyết vấn đề trên chúng ta có thể thiết kế như sau:

```
public interface IDoor
{
    void Lock();
    void Unlock();
    bool IsDoorOpen();
}

public interface ITimerClient
{
    void TimeOut();
}

public class NormalDoor : IDoor
{
    public void Lock() { }
    public void Unlock() { }
    public bool IsDoorOpen() { return false; }

    // No timer dependency
}

public class TimedDoor : IDoor, ITimerClient
{
    public void Lock() { }
    public void Unlock() { }
    public bool IsDoorOpen() { return false; }

    public void TimeOut()
    {
        // Check and raise alarm
    }
}
```

Lý do nên thiết kế như thế này vì `IDoor` và `ITimerClient` phục vụ cho 2 nhóm client khác nhau. Code quản lý cửa sẽ quan tâm tới `IDoor` còn code quản lý thời gian sẽ quan tâm tới `ITimerClient`. Khi client khác nhau, interface cũng nên được tách ra theo nhu cầu của chúng.

Một dấu hiệu nhận biết interface chứa nhiều method **không phù hợp** là chúng ta bắt đầu thấy các **method rỗng** hoặc `throw new NotImplementedException();`

Thiết kế interface

Xét thêm ví dụ về UI của máy ATM dưới đây:

```
public interface IATMUI
{
    void RequestDepositAmount();
    void RequestWithdrawalAmount();
    void RequestTransferAmount();
}
```

Khi này mỗi transaction chỉ sử dụng một phần nhỏ của `IATMUI`

- `DepositTransaction` → `RequestDepositAmount()`

- WithdrawalTransaction → RequestWithdrawalAmount()
- TransferTransaction → RequestTransferAmount()

Nếu sau này thêm PayGasBillTransaction ta có thể sẽ lại thêm void RequestGasBillAmount(); vào interface. Nhưng tất cả transaction đều phụ thuộc vào IATMUI, thay đổi chỉ phục vụ PayGasBillTransaction lại ảnh hưởng đến cả DepositTransaction, WithdrawalTransaction, TransferTransaction và những module sử dụng interface cũ. **Đây là điều mà ISP muốn tránh.**

Để giải quyết nó ta có thể tách

```
public interface IDepositUI
{
    void RequestDepositAmount();
}

public interface IWithdrawalUI
{
    void RequestWithdrawalAmount();
}

public interface ITransferUI
{
    void RequestTransferAmount();
}
```

Giờ khi cần thêm transaction mới chúng ta sẽ thêm một interface mới không ảnh hưởng gì tới các transaction cũ.

```
public interface IGasBillUI
{
    void RequestGasBillAmount();
}
```

Triển khai thực tế vẫn có thể implement tất cả public class ATMUI : IDepositUI, IWithdrawalUI, ITransferUI, IGasBillUI

Một class lớn chưa chắc đã là vấn đề. Vấn đề là khi mỗi client bị buộc phải nhìn class đó qua một interface lớn chứa tất cả những thứ nó không cần.

Nhưng ISP không có nghĩa là 1 method - 1 interface hay 1 client - 1 interface. Nếu nhiều client có cùng nhu cầu chúng hoàn toàn có thể dùng chung một interface. Việc phải tạo tới hàng trăm interface là một cách quá cực đoan với ISP, nó sẽ tạo ra một lại complexity khác. Mục tiêu là tạo ra cohesive interface - mỗi interface đại diện cho một nhóm hành vi mà client thực sự cần.

Giá trị cốt lõi của ISP là khi client A thay đổi interface của nó thì client B, C... không bị thay đổi chỉ vì chúng tình cờ dùng chung một service.

Khi đánh giá một interface, câu hỏi hữu ích nhất là: **"Client này có thực sự cần sử dụng tất cả những gì interface bắt buộc nó phụ thuộc vào không?"**

Nếu câu trả lời là không thì bạn nên bắt đầu nghĩ tới ISP.

Revision #3
Created 2026-08-22 02:27:00 UTC by ThanhDV
Updated 2026-08-24 03:56:53 UTC by ThanhDV