

SOLID Principles

SOLID Principles là một bộ tiêu chuẩn lập trình mà mọi lập trình viên cần phải nắm vững để phát triển phần mềm đúng cách nhằm tránh các thiết kế tệ. Bộ nguyên lý này được phổ biến bởi Robert C. Martin (Uncle Bob) và được áp dụng rộng rãi trong toàn bộ phạm vi thiết kế hướng đối tượng. Khi được áp dụng đúng cách nó sẽ giúp source code của bạn dễ mở rộng, rõ ràng và dễ đọc hơn.

1. **S**ingle responsibility principle
2. **O**pen/closed principle
3. **L**iskov substitution principle
4. **I**nterface segregation principle
5. **D**ependency inversion principle

- [The Single-Responsibility Principle \(SRP\)](#)
- [The Open/Closed Principle \(OCP\)](#)
- [The Liskov Substitution Principle \(LSP\)](#)
- [The Interface Segregation Principle \(ISP\)](#)
- [The Dependency Inversion Principle \(DIP\)](#)

The Single-Responsibility Principle (SRP)

“A class should have only one reason to change.”

Đó là cách mà Robert C. Martin nói về SRP. Điểm quan trọng ở đây là **"reason to change"**. Trong ngữ cảnh của SRP, một responsibility được định nghĩa là một lý do khiến class phải thay đổi. Nếu chúng ta có thể nghĩ ra được nhiều hơn 1 lý do độc lập khiến class phải thay đổi tức là class đó đang đảm nhận nhiều hơn một trách nhiệm.

Có nghĩa là SRP không đơn giản là mỗi class chỉ làm một việc, càng không có nghĩa là một class chỉ nên có 1 method. Câu hỏi chính xác nên là: **"Có những yêu cầu nào từ hệ thống khiến tôi phải sửa class này?"**.

Responsibility là gì?

Với SRP, Responsibility có thể hiểu là một lý do để class đó phải thay đổi.

Ví dụ 1

```
public class Player
{
    public int CalculateDamage()
    {
        // Gameplay logic
    }

    public void Save()
    {
        // Save player
    }
}
```

Trong ví dụ này, 2 method đều liên quan đến player nhưng chúng thay đổi vì những lý do khác nhau.

- Khi gameplay thay đổi, công thức tính sát thương thay đổi thì cần sửa lại `CalculateDamage()`
- Khi logic save thay đổi. Save local sang cloud, save json sang binary... thì cần sửa lại `Save()`

Như vậy, Player đang có 2 responsibility.

Ví dụ 2

```
public class Rectangle
{
    public double GetArea()
    {
        // Calculate area
    }

    public void Draw()
    {
        // Draw on screen
    }
}
```

Với ví dụ này chúng ta cũng có thể thấy có 2 phần là logic tính toán và render. Khi sử dụng Rectangle trong tính toán chúng ta mặc nhiên bị kéo theo cả GUI một cách không cần thiết và ngược lại.

SRP được sinh ra để tránh các coupling không cần thiết.

Khi nào nên tách class

Trong thực tế không phải lúc nào việc tách class và đảm bảo SPR cũng đem lại lợi ích. Tùy vào dự án thực tế để lựa chọn tách hoặc không tách class cho phù hợp với yêu cầu dự án.

Ví dụ 1

```
public interface Modem
{
    void Dial(string phoneNumber);
    void Hangup();

    void Send(char data);
    char Receive();
}
```

Nhìn vào ví dụ này chúng ta có thể thấy được 2 lý do để Modem phải thay đổi.

- Quản lý kết nối: Dial(), Hangup()
- Truyền dữ liệu: Send(), Receive()

Mặc dù 2 nhóm này có thể thay đổi độc lập nhưng điều đó **không có nghĩa là cần phải tách chúng ngay lập tức.**

Nếu trong hệ thống thực tế 2 nhóm này luôn thay đổi cùng nhau. Ví dụ như chỉ cần cập nhật khi hạ tầng phần cứng thay đổi. Khi đó cả 2 nhóm sẽ thay đổi cùng nhau. Khi này việc tạo thêm interface và class chỉ để đảm bảo đúng SPR khiến cho dự án trở nên phức tạp một cách không cần thiết.

SRP nên được áp dụng khi vấn đề thay đổi thực sự xuất hiện, không phải để dự đoán mọi khả năng có thể xảy ra trong tương lai.

Ví dụ 2

```
public class Employee
{
    public decimal CalculateSalary()
    {
        // Business rules
    }

    public void Save()
    {
        // Database
    }
}
```

Ví dụ này cũng tương tự như ví dụ về player ở phần trên. Quy tắc tính lương và cách lưu dữ liệu thay đổi vì những lý do hoàn toàn khác nhau.

Quy tắc tính lương có thể thay đổi theo nhân sự, theo chính sách công ty... còn save sẽ chỉ cần thay đổi khi thay đổi hạ tầng. Vậy nên chúng có thể tách ra như sau:

```
public class Employee
{
    public decimal CalculateSalary()
    {
        // Business rules
    }
}
```

```
public class EmployeeRepository
{
    public void Save(Employee employee)
    {
        // Persistence
    }
}
```

Việc tách ra này giúp cho việc thay đổi cách tính lương không kéo theo cả việc cập nhật lại cách lưu dữ liệu.

Cách nhận biết SRP

SRP không nói về độ lớn file code, số dòng code, số method hay kích thước class.

Nó nói về **lý do mà class cần thay đổi**. Khi đọc một class hãy thử hoàn thành câu: "Class này sẽ phải thay đổi nếu..."

Nếu bạn có câu là lời kiểu như *Player sẽ cần thay đổi nếu UI thay đổi, gameplay thay đổi, save format thay đổi...* thì rất có khả năng Player đang chứa quá nhiều responsibility.

Khi đó chúng ta có thể cân nhắc tách Player thành:

- Player → Gameplay

- PlayerRepository → Persistence
- PlayerView → Presentation
- ...

Mục tiêu cuối cùng không phải là càng nhiều class càng tốt, class càng ngắn càng tốt mà là không để thay đổi phải kéo theo những phần không liên quan.

Nhớ rằng, **một class chỉ nên có một lý do để thay đổi**. Nhưng cái khó khi áp dụng SRP là làm sao nhận biết được đâu mới thực sự là responsibility.

The Open/Closed Principle (OCP)

“The software entities should be open for extension, but closed for modification.

OCP không có nghĩa là "**code đã viết thì tuyệt đối không được sửa**". Mục tiêu là khi hệ thống cần thêm một hành vi mới mà chúng ta **đã lường trước** thì có thể thêm code mới thay vì sửa code cũ đang hoạt động ổn định.

Đặc điểm chính của một module tuân thủ OCP:

- **Open for extension:** Có thể thêm hành vi mới mà không cần sửa code cũ.
- **Closed for modification:** Việc bổ sung tính năng không gây ảnh hưởng đến module đã có.

OCP giải quyết vấn đề gì?

Trong ví dụ này chúng ta xét đến một game có nhiều loại enemy khác nhau.

```
public enum EnemyType
{
    Zombie,
    Robot
}

public void Attack(Enemy enemy)
{
    switch (enemy.Type)
    {
        case EnemyType.Zombie:
            AttackZombie(enemy);
            break;
        case EnemyType.Robot:
            AttackRobot(enemy);
            break;
    }
}
```

Trong ví dụ này, vấn đề sẽ xảy ra khi chúng ta cần thêm các loại enemy khác như Alien, Mutant, Boss... khi đó chúng ta sẽ cần quay cho trở lại `Attack()` để thêm. Nếu vẫn với thiết kế như trên thì chúng ta còn phải sửa cả `Move()`, `TakeDamage()`, `PlayAnimation()`... kéo theo là phải test lại cả enemies cũ vốn đã hoạt động ổn.

Để giải quyết vấn đề trên chúng ta có thể thử thiết kế dưới đây:

```
public interface IEnemy
{
    void Attack();
}

public class Zombie : IEnemy
{
    public void Attack()
    {
        // Zombie attack
    }
}

public class Robot : IEnemy
{
    public void Attack()
    {
        // Robot attack
    }
}
```

Với thiết kế này khi cần thêm enemy chúng ta chỉ cần triển khai interface `IEnemy`. Các enemy khác hay việc sử dụng `Attack()` không thay đổi.

```
public class Alien : IEnemy
{
    public void Attack()
    {
        // Alien attack
    }
}
```

Với thiết kế trên, **hành vi của hệ thống được mở rộng bằng cách thêm các triển khai mới trong khi phần code cũ đã ổn định không bị thay đổi.**

Abstraction và **polymorphism** là những công cụ quan trọng để đạt được điều này. Module cấp cao làm việc với abstraction ổn định, còn các hành vi khác nhau được cung cấp bởi những triển khai của abstraction đó.

Ứng dụng OCP

Với ví dụ trên rất dễ để chúng ta nhầm lẫn việc sử dụng `switch`, `if/else` là vi phạm OCP. Nhưng thực tế không phải như vậy.

Để ứng dụng đúng OCP chúng ta cần biết hệ thống cần được đóng trước loại thay đổi nào và mở với hành vi nào.

Với ví dụ trên, thiết kế như trên là rất tốt nếu enemies hoạt động riêng lẻ và độc lập với nhau. Nhưng nếu GDD yêu cầu ưu tiên Alien luôn đánh trước Robot thì thiết kế interface như trên lại không giải quyết được nhiều vấn đề.

Vậy nên **đừng cố thiết kế một module có thể đóng với mọi loại thay đổi. Điều đó là không thể.**

Áp dụng OCP cũng cần có tính chiến lược. Chúng ta cần xác định những thay đổi có khả năng xảy ra và xây dựng abstraction cho vấn đề đó. Cũng không có nghĩa là chúng ta sẽ trừu tượng hóa mọi thứ. Làm thế đồng nghĩa với việc chúng ta cần bỏ thêm thời gian, công sức và tiền bạc cho những thay đổi sẽ không bao giờ xuất hiện.

Nếu mỗi lần thêm một object, một behaviour hay một policy mới bạn phải quay lại cùng một nhóm hoặc cùng module để sửa thì đó là dấu hiệu để cân nhắc sử dụng OCP.

Tổng kết

OCP sinh ra không phải để ngăn việc sửa code mà là nguyên lý về việc **kiểm soát phạm vi của thay đổi.**

Để áp dụng OCP hãy tự hỏi: "**Loại thay đổi nào có thể thường xuyên xảy ra và làm thế nào để khi nó xảy ra chúng ta chỉ cần thêm code mới mà không phải quay lại sửa code cũ.**"

The Liskov Substitution Principle (LSP)

“Subtypes must be substitutable for their base types.”

Đơn gì thì nếu code của bạn đang hoạt động đúng với object `Base` thì việc truyền vào một object `Derived`: `Base` không được làm đoạn code đó gặp vấn đề.

Ví dụ kinh điển Square và Rectangle

Về mặt toán học chúng ta đều biết hình vuông cũng là một hình chữ nhật. Thế nên rất dễ dẫn tới thiết kế như sau:

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int Area()
    {
        return Width * Height;
    }
}
public class Square : Rectangle
{
    public override int Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override int Height
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

Nhìn qua thì thiết kế này vẫn đảm bảo `width == Height`. Nhìn riêng thì có vẻ như 2 class này hoàn toàn hợp lý. Nhưng hãy xem cách sử dụng dưới đây:

```
public int CalculateArea(Rectangle rectangle)
{
    rectangle.Width = 5;
    rectangle.Height = 4;

    return rectangle.Area();
}

public void Main()
{
    Console.WriteLine(CalculateArea(new Rectangle())); // 20
    Console.WriteLine(CalculateArea(new Square())); // 16
}
```

Như ví dụ trên chúng ta có thể thấy kết quả khác nhau và không đúng với `Square`. Hay nói cách khác `Square` không thể thay thế cho `Rectangle`.

Điểm đáng chú ý là nếu chỉ xét riêng `Square` thì nó vẫn là một hình vuông hợp lệ với `width == Height`.

Vấn đề nằm ở **contract** của `Rectangle`. Client sử dụng `Rectangle` có quyền kỳ vọng rằng `rectangle.Width = 5`; chỉ thay đổi `Width` chứ không âm thầm thay đổi cả `Height`. `Square` phá vỡ kỳ vọng đó.

Do đó trong thiết kế hướng đối tượng, quan hệ "Square **IS-A** Rectangle" không nên chỉ được quyết định dựa trên quan hệ ngoài đời thực.

“ LSP thực sự quan tâm tới **IS-A về behavior** về mặt behavior. Nếu behavior của subtype không tương thích với những kỳ vọng hợp lý của client với base type thì tức là inheritance đó không phù hợp.

Contract quan trọng hơn inheritance

Có thể xem mỗi public API là một contract giữa object và client.

Contract thường gồm:

- **Precondition:** điều kiện client phải đáp ứng trước khi gọi.
- **Postcondition:** điều object đảm bảo sau khi thực hiện.

Nói đơn giản **Derived class không được đòi hỏi nhiều hơn và không được cung cấp ít hơn base class**.

Một ví dụ cho việc subtype đòi hỏi precondition mạnh hơn base type:

```
public class Account
{
    public virtual void Withdraw(int amount)
    {
        if (balance < amount) return;
    }
}
```

```

        // Withdraw
    }
}

public class ChildAccount : Account
{
    public override void Withdraw(int amount)
    {
        if (balance < amount || amount > 100) return;

        // Withdraw
    }
}

```

Trong ví dụ này code vốn chạy đúng với `Account` nhưng có thể thất bại với `ChildAccount`.

Một ví dụ khác, chúng ta có `Bird.Fly()` client có quyền kỳ vọng object đó có thể bay nhưng `Penguin`: `Bird` và `Penguin.Fly()` lại không thực hiện được kỳ vọng đó thì tức là hierarchy đang có vấn đề.

LSP liên quan trực tiếp tới OCP

Một dấu hiệu rất phổ biến khi vi phạm LSP là client bắt đầu phải kiểm tra subtype

```

public void Process(Rectangle rectangle)
{
    if (rectangle is Square)
    {
        // Special case for Square
    }
    else
    {
        // Normal Rectangle behavior
    }
}

```

Ban đầu abstraction này được tạo ra để client chỉ cần biết `Rectangle` nhưng giờ client cần phải đọc lại để biết thêm về `Square`. Với mỗi subtype lại cần thêm một nhánh để xử lý riêng. Khi đó abstraction gần như không còn ý nghĩa.

Đây cũng là lý do mà LSP có quan hệ chặt chẽ với OCP. Khi subtype không thể thay thế base type, client thường phải thêm type checking để xử lý ngoại lệ. Mỗi subtype mới buộc client phải sửa và khiến code không còn **closed for modification**.

Tổng kết

LSP không đơn giản là quy tắc về inheritance. Nó là quy tắc về đảm bảo **abstraction thực sự đáng tin cậy**.

Câu hỏi hữu ích khi đánh giá một quan hệ inheritance là: "**Nếu thay Base bằng Derived, những đoạn code đang sử dụng Base có còn hoạt động đúng theo kỳ vọng ban đầu không?**"

Nếu câu trả lời là không thì chưa chắc subtype đã đã thỏa mãn **IS-A về mặt behavior** và đó là điều mà LSP muốn chúng ta tránh.

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.

The Dependency Inversion Principle (DIP)

“High-level modules should not depend on low-level modules. Both should depend on abstractions.

“Abstractions should not depend upon details. Details should depend upon abstractions.

Điểm quan trọng nhất của DIP là **hướng của dependency**.

Trong một thiết kế trực tiếp chúng ta thường viết: `Business Logic` → `Database` → `File System` → `Framework`. Module cấp cao chứa business rule nhưng lại phụ thuộc trực tiếp vào những implementation detail bên dưới. Vấn đề gặp phải là khi detail thay đổi, module chứa business rule cũng có thể bị ảnh hưởng.

Các module cấp cao chứa những quyết định và business logic quan trọng của ứng dụng. Nếu chúng phụ thuộc vào những module cấp thấp, thay đổi ở tầng thấp có thể lan ngược lên và buộc tầng cao phải thay đổi.

DIP đảo ngược chiều dependency này: `High-level Policy` → `Abstraction` ← `Low-level Detail`. Cả 2 đều hướng về **abstraction**.

Ví dụ: Button và Lamp.

Dưới đây là một cách đơn giản nhất để một `Button` điều khiển một `Lamp`

```
public class Lamp
{
    public void TurnOn()
    {
        // Turn lamp on
    }

    public void TurnOff()
    {
        // Turn lamp off
    }
}

public class Button
{
```

```

private Lamp lamp;

public Button(Lamp lamp)
{
    this.lamp = lamp;
}

public void Press()
{
    lamp.TurnOn();
}
}

```

Chúng ta có thể thấy dependency hiện tại là `Button` → `Lamp`. `Button` phục thuộc trực tiếp vào một implementation cụ thể. Điều này đồng nghĩa với việc `Button` được thiết kế để điều khiển **Lamp và chỉ Lamp**. Nếu sau này muốn dùng `Button` để điều khiển `Door`, `Fan`, `TV`... thì `Button` buộc phải thay đổi.

Chúng ta có thể dễ dàng nhìn thấy vấn đề với thiết kế này. Khi `Lamp` thay đổi có thể kéo theo `Button` cũng cần thay đổi và `Button` cũng khó để có thể tái sử dụng với một thiết bị khác.

Trong ví dụ này chúng ta không nên hiểu "`Button` điều khiển `Lamp`" mà là "`Button` nhận thao tác từ người dùng và chuyển thao tác đó thành tín hiệu bật/tắt cho `Lamp` (hoặc một đối tượng bất kỳ có thể bật tắt)."

Chúng ta có thể thay thế bằng thiết kế sau đây:

```

public interface ISwitchableDevice
{
    void TurnOn();
    void TurnOff();
}

public class Lamp : ISwitchableDevice
{
    public void TurnOn()
    {
        // Turn lamp on
    }

    public void TurnOff()
    {
        // Turn lamp off
    }
}

public class Button
{
    private readonly ISwitchableDevice device;

    public Button(ISwitchableDevice device)
    {
        this.device = device;
    }

    public void Press()
    {
        device.TurnOn();
    }
}

```

Với thiết kế này dependency sẽ là `Button` → `ISwitchableDevice` ← `Lamp`

Ngược lại, `Lamp` cũng phải tương thích với abstraction mà `Button` cần.

Đó chính là **Dependency Inversion**.

Giờ chúng ta có thể dùng `Button` để điều khiển một thiết bị bất kỳ mà không cần chỉnh sửa `Button` và cũng không ảnh hưởng tới các thiết bị khác.

```
public class Motor : ISwitchableDevice
{
    public void TurnOn()
    {
        // Start motor
    }

    public void TurnOff()
    {
        // Stop motor
    }
}
```

Abstraction nên thuộc về phía nào?

Abstraction không nhất thiết phải được thiết kế từ phía implementation.

Ví dụ `Gameplay` → `SaveSystem`. Chúng ta có thể nghĩ rằng `SaveSystem` nên tạo `ISaveSystem` với tất cả những khả năng mà nó cung cấp.

Nhưng DIP hướng chúng ta theo hướng **client nên định nghĩa abstraction dựa theo dịch vụ mà nó cần**.

Ví dụ `gameplay` chỉ cần `IPlayerStorage`. Sau đó, một implementation cụ thể có thể đáp ứng contract đó `JsonPlayerStorage : IPlayerStorage` hoặc `CloudPlayerStorage : IPlayerStorage`

Dependency của thiết kế này sẽ trở thành `Gameplay` → `IPlayerStorage` ← `JsonPlayerStorage`

Thay vì để module bậc cao phải phụ thuộc vào API của module cấp thấp, module cấp thấp nên triển khai abstraction mà logic cấp cao yêu cầu.

Đây cũng là lý do mà interface nên được định nghĩa dựa trên nhu cầu của client thay vì dựa trên những gì server có thể cung cấp. Module cấp cao quyết định mình cần những thao tác nào còn module cấp thấp chỉ việc triển khai các thao tác đó.

Điều quan trọng với DIP là abstraction mô tả nhu cầu ổn định của logic bậc cao, còn triển khai chi tiết sẽ phụ thuộc vào các abstraction đó.

Có được phụ thuộc vào concrete class?

Vấn đề của DIP không phải là loại bỏ hoàn toàn mọi dependency vào concrete class mà là xác định dependency nào dễ gây thay đổi và nên được đảo chiều.

DIP có thể được diễn giải là "**Depend on abstractions**". Nhưng nếu áp dụng tuyệt đối thì gần như mọi object đều phải nằm sau một interface. Khi đó code trở nên phức tạp mà không mang lại lợi ích thực tế.

Hơn nữa, vẫn phải có nơi tạo concrete object và không phải concrete class nào cũng đáng lo.

Vì vậy, câu hỏi quan trọng không phải là "có phụ thuộc vào concrete class hay không?" mà là "**concrete class đó có dễ thay đổi và có gây ảnh hưởng lan truyền hay không?**".

Những class có khả năng thay đổi cao (**concrete class volatile**) là nơi dependency trực tiếp dễ tạo ra vấn đề, vì khi class thay đổi các class phụ thuộc vào nó cũng có thể phải thay đổi theo. Ngược lại, những concrete type ổn định, ít có khả năng thay đổi không nhất thiết phải được bọc bằng abstraction chỉ để tuân thủ DIP.

Tổng kết

Vấn đề không phải là concrete class luôn xấu. Vấn đề là business logic có đang phụ thuộc trực tiếp vào một detail dễ thay đổi hay không.

DIP không yêu cầu mọi thứ phải có interface. Nó yêu cầu chúng ta bảo vệ business logic khỏi những chi tiết dễ thay đổi.

Hãy hướng tới `Business Logic` → `Abstraction` ← `Implementation Detail`

Khi đó, database, file system, hardware... có thể thay đổi mà không kéo theo thay đổi trong phần logic quan trọng của hệ thống.

Câu hỏi hữu ích khi áp dụng DIP là: "**Business logic đang phụ thuộc vào abstraction chung hay đang phụ thuộc trực tiếp vào một class cụ thể?**"

Nếu business logic buộc phải thay đổi chỉ vì một triển khai chi tiết bên dưới thay đổi, dependency rất có thể đang đi sai hướng.