

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.

Revision #2

Created 2026-08-22 06:45:34 UTC by ThanhDV

Updated 2026-08-24 03:57:00 UTC by ThanhDV