Dependency Inversion, Inversion of Control và Dependency Injection, Một cái nhìn tổng quan

Giới thiệu

Trong quá trình phát triển phần mềm, việc thiết kế hệ thống một cách linh hoạt và dễ bảo trì là một trong những mục tiêu hàng đầu. Các nguyên tắc thiết kế hướng đối tượng (OOP) đóng vai trò quan trọng trong việc đạt được mục tiêu này. Trong số đó, **Dependency Inversion (DI)** là một nguyên tắc cốt lõi, giúp giảm thiểu sự kết hợp chặt chẽ giữa các module trong một hệ thống.

Vấn đề

Một điện thoại smartphone sẽ bao gồm **Màn hình**, **Pin**, **Chip**. Ví dụ, điện thoại iPhone:

- **Màn hình** được sản xuất bởi **Samsung** hoặc **LG**.
- **Pin** được sản xuất bởi **Remax** hoặc **Orizin** hoặc **Pisen**.
- **Chip** được sản xuất bởi **TSMC** hoặc **Samsung**.

```
using System;
using System.Reflection;

public class Ram
{
    public string Producer { get; }
    public string Specification { get; }

    public Ram(string producer, string specification)
    {
        Producer = producer;
        Specification = specification;
    }
}

public class Cpu
{
    public string Producer { get; }
    public string Specification { get; }

    public Cpu(string producer, string specification)
    {
        Producer = producer;
        Specification = specification;
    }
}
```

```
public class SmartPhone
{
    private Ram _ram;
    private Cpu _cpu;

    public void PrintSpecification()
    {
        Console.WriteLine($"RAM: {_ram.Specification} is produced by: {_ram.Producer}");
        Console.WriteLine($"CPU: {_cpu.Specification} is produced by: {_cpu.Producer}");
    }
}

public class Application
{
    public static void Main(string[] args)
    {
        var iphone = CreateNewIphone();
        iphone.PrintSpecification();
    }

    private static SmartPhone CreateNewIphone()
    {
        return new SmartPhone();
    }
}
```

Khi chạy đoạn code trên thì kết quả là `NullReferenceException`. Chúng ta cần **khởi tạo** giá trị cho 2 thuộc tính `ram` và `chip`.

Một cách thức ăn liền là tạo constructor và sử dụng keywork `new` để init `ram` và `chip`. Sửa đoạn code trên như sau.

```
public class SmartPhone
{
    private Ram _ram;
    private Cpu _cpu;

    SmartPhone()
    {
        _ram = new Ram("Kingston", "8GB");
        _cpu = new Cpu("TSMC", "2.8Ghz");
    }
    public void PrintSpecification()
    {
        Console.WriteLine($"RAM: {_ram.Specification} is produced by: {_ram.Producer}");
        Console.WriteLine($"CPU: {_cpu.Specification} is produced by: {_cpu.Producer}");
    }
}
```

Vấn đề là `SmartPhone` đang phụ thuộc trực tiếp vào `Ram` và `Cpu`. Rất khó thay đổi giá trị trong quá trình *runtime*.

Giải pháp

```
using System;

public class Ram
{
    public string Producer { get; }
    public string Specification { get; }

    public Ram(string producer, string specification)
    {
        Producer = producer;
        Specification = specification;
    }

    public string GetSpecification()
    {
        return Specification;
    }

    public string GetProducer()
    {
        return Producer;
    }
}

public class Cpu
{
    public string Producer { get; }
    public string Specification { get; }

    public Cpu(string producer, string specification)
    {
        Producer = producer;
        Specification = specification;
    }

    public string GetSpecification()
    {
        return Specification;
    }

    public string GetProducer()
    {
        return Producer;
    }
}
```

```
public class SmartPhone
{
    private Ram _ram;
    private Cpu _cpu;

    // Constructor injection
    public SmartPhone(Ram ram, Cpu cpu)
    {
        _ram = ram;
        _cpu = cpu;
    }

    // Setter injection
    public void SetRam(Ram ram)
    {
        _ram = ram;
    }

    public void SetCpu(Cpu cpu)
    {
        _cpu = cpu;
    }

    public void PrintSpecification()
    {
        Console.WriteLine($"RAM: {_ram.GetSpecification()} is produced by: {_ram.GetProducer()}");
        Console.WriteLine($"CPU: {_cpu.GetSpecification()} is produced by: {_cpu.GetProducer()}");
    }
}

public class Application
{
    public static void Main(string[] args)
    {
        // Initiate dependency
        var ram = new Ram("Kingston", "8GB");
        var cpu = new Cpu("TSMC", "2.8Ghz");

        var iPhone = CreateNewIphone(ram, cpu);
        iPhone.PrintSpecification();

        // Initiate new ram and replace the current one
        var newRam = new Ram("Kingston", "64GB");
        iPhone.SetRam(newRam);
        iPhone.PrintSpecification();
    }

    private static SmartPhone CreateNewIphone(Ram ram, Cpu cpu)
    {
        return new SmartPhone(ram, cpu);
```

```
    }
}
```

Dependency Inversion (DI) - Đảo ngược sự phụ thuộc

DI là một nguyên tắc thiết kế hướng đối tượng nhằm giảm thiểu sự kết hợp chặt chẽ giữa các module trong một hệ thống. Theo nguyên tắc này, các module cấp cao không nên phụ thuộc trực tiếp vào các module cấp thấp. Thay vào đó, cả hai nên phụ thuộc vào một abstraction (ví dụ: interface). Điều này giúp tăng cường tính linh hoạt và khả năng tái sử dụng của code.

Ví dụ:

Giả sử chúng ta có một class `ShapeManager` để tính toán chu vi và diện tích của các hình học. Thay vì để `ShapeManager` phụ thuộc trực tiếp vào các class cụ thể như `Circle`, `Square`, chúng ta sẽ tạo một interface `IShape` và để `ShapeManager` phụ thuộc vào interface này. Các class `Circle`, `Square` sẽ implement interface `IShape`.

```
interface IShape
{
    double GetPerimeter();
    double GetArea();
}

class Circle : IShape
{
    // ...
}

class Square : IShape
{
    // ...
}

class ShapeManager
{
    private IShape shape;

    // ...
}
```

Inversion of Control (IoC) - Đảo ngược sự điều khiển

IoC là một kỹ thuật thực hiện nguyên tắc DI. IoC chuyển giao việc tạo và quản lý các đối tượng từ code của chúng ta sang một container hoặc framework. Điều này cho phép code của chúng ta tập trung vào logic nghiệp vụ chính.

IoC Container là một thành phần trung tâm trong IoC. Nó chịu trách nhiệm:

- **Tạo các đối tượng:** Container sẽ tạo ra các đối tượng dựa trên cấu hình hoặc các annotation.
- **Quản lý vòng đời của đối tượng:** Container sẽ quản lý việc khởi tạo, kết nối và hủy các đối tượng.
- **Tiêm các dependency:** Container sẽ tự động tiêm các dependency vào các đối tượng khi chúng được tạo ra.

Dependency Injection (DI) - Tiêm sự phụ thuộc

DI là một pattern cụ thể của IoC. Trong DI, các dependency (phụ thuộc) của một đối tượng được cung cấp từ bên ngoài thông qua constructor, setter hoặc interface.

Các cách thực hiện DI:

- **Constructor Injection:** Các dependency được truyền vào thông qua constructor của một class.
- **Setter Injection:** Các dependency được thiết lập thông qua các phương thức setter.
- **Interface Injection:** Các dependency được thiết lập thông qua một interface.

Ví dụ về Constructor Injection:

```
public class ShapeManager {
    private IShape shape;

    public ShapeManager(IShape shape) {
        this.shape = shape;
    }
}
```

Lợi ích của DI, IoC và DI

- **Tăng tính modular hóa:** Các module trở nên độc lập và dễ thay thế.
- **Nâng cao khả năng tái sử dụng:** Các module có thể được sử dụng trong nhiều ngữ cảnh khác nhau.
- **Dễ dàng testing:** Viết các unit test trở nên đơn giản hơn.
- **Cải thiện khả năng bảo trì:** Code trở nên dễ hiểu và dễ bảo trì hơn.

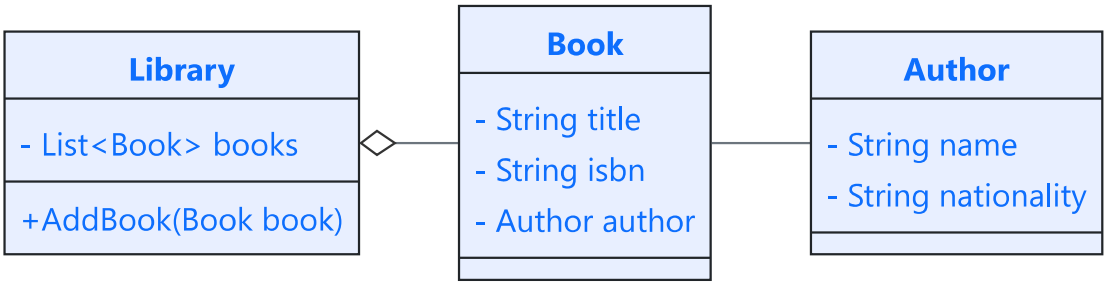
Các framework hỗ trợ DI

- **Unity (C#):** Một framework DI nhẹ nhàng và linh hoạt cho .NET.
- **Spring (Java):** Một trong những framework DI phổ biến nhất, cung cấp đầy đủ các tính năng để quản lý các dependency.
- **Dagger (Android):** Một framework DI được thiết kế đặc biệt cho Android, cung cấp khả năng compile-time dependency injection.

Áp dụng

Giả sử chúng ta đang phát triển một ứng dụng quản lý thư viện. Trong ứng dụng này, chúng ta có các class:

- `Book`: Đại diện cho một cuốn sách.
- `Author`: Đại diện cho một tác giả.
- `Library`: Đại diện cho một thư viện.



Cách làm truyền thống:

```
public class Library
{
    private List<Book> books = new List<Book>();

    public void AddBook(Book book)
    {
        books.Add(book);
    }
    // ...
}

public class Book
{
    private Author author;

    public Book(Author author)
    {
        this.author = author;
    }
    // ...
}

public class Author
{
    // ...
}
```

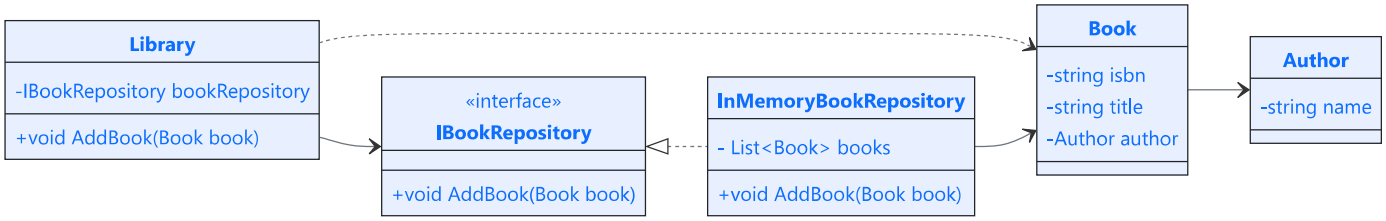
Trong ví dụ trên, class `Library` phụ thuộc trực tiếp vào class `Book`. Điều này có nghĩa là nếu chúng ta muốn thay đổi cách lưu trữ thông tin về sách, chúng ta phải sửa đổi cả class `Library` và `Book`.

Áp dụng Dependency Inversion:

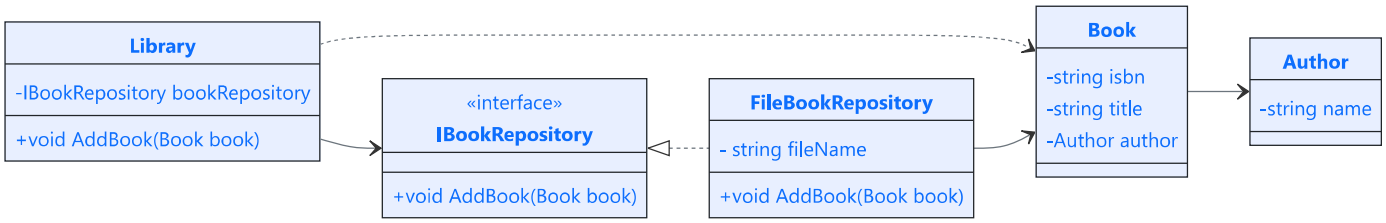
```
public interface IBookRepository
{
    void AddBook(Book book);
}

public class Library
```

```
{
    private IBookRepository bookRepository;

    public Library(IBookRepository bookRepository)
    {
        this.bookRepository = bookRepository;
    }

    public void AddBook(Book book)
    {
        bookRepository.AddBook(book);
    }
}
```

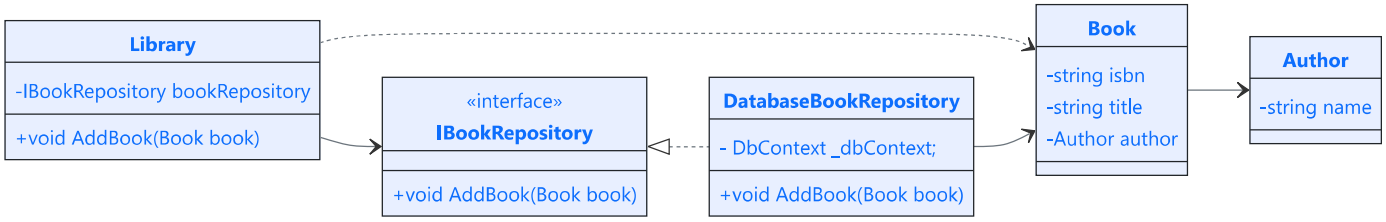
- Cài đặt bằng danh sách bộ nhớ trong



- Cài đặt bằng tập tin



- Cài đặt bằng cơ sở dữ liệu



Bằng cách sử dụng interface `IBookRepository`, chúng ta đã tách rời sự phụ thuộc giữa `Library` và cách thức lưu trữ sách. Giờ đây, chúng ta có thể dễ dàng thay đổi cách lưu trữ sách bằng cách triển khai một interface khác mà không cần sửa đổi class `Library`.

Kết luận

Dependency Inversion, Inversion of Control và Dependency Injection là những khái niệm quan trọng trong thiết kế phần mềm hiện đại. Việc hiểu rõ và áp dụng các nguyên tắc này sẽ giúp bạn viết ra những ứng dụng phần mềm chất lượng cao, dễ bảo trì và mở rộng.