Lưu trữ dữ liệu (2): interface, loosely coupling

Trong bài học này chúng ta sẽ xem xét sử dụng một công cụ đặc biệt hữu ích trong phát triển ứng dụng: interface. Chúng ta sẽ vận dụng Interface để giúp dễ dàng chuyển đổi giữa các cách thức lưu trữ dữ liệu đã biết (xml, binary, json).

> Để thực hiện bài thực hành này, bạn cần nắm bắt rõ ý nghĩa và cách sử dụng interface trong C#.

NỘI DUNG CỦA BÀI [Ẩn]

Thực hành: áp dụng interface cho các lớp data access

Bước 1. Xây dựng giao diện IDataAccess

Tạo file mã nguồn IDataAccess.cs trong thư mục DataServices (Add => New Item, chọn kiểu là Interface). Viết code cho IDataAccess như sau:

IDataAccess.cs

```
1.  using System.Collections.Generic;
2.
3.  namespace BookMan.ConsoleApp.DataServices
4.  {
5.      using Models;
6.      public interface IDataAccess
7.      {
8.          List<Book> Books { get; set; }
9.
10.         void Load();
11.         void SaveChanges();
12.     }
13. }
```

Ở bước này chúng ta xây dựng một interface IDataAccess chứa định nghĩa của một property (Books) và hai phương thức (Load, SaveChanges). Để ý rằng hai phương thức này chỉ có phần mô tả (specification/declaration) mà không có thân (implementation), gần giống như phương thức abstract.

Bước 2. Điều chỉnh lớp Repository

```
1.  public class Repository
2.      {
3.          protected readonly IDataAccess _context;
```

```
        public Repository(IDataAccess context)
        {
            _context = context;
            _context.Load();
        }
...
```

Ở bước này chúng ta thay thế lớp data access cụ thể bằng interface IDataAccess. Ở bài trước chúng ta đã xây dựng 3 lớp data access để sử dụng trong Repository. Ở đây thay vì sử dụng một trong ba lớp đó, chúng ta sử dụng kiểu IDataAccess mới xây dựng.

Bước 3. Điều chỉnh lớp BookController và ShellController

BookController | ShellController

```
internal class BookController : ControllerBase
{
    protected Repository Repository;
    public BookController(IDataAccess context)
    {
        Repository = new Repository(context);
    }
...
```

Ở bước này chúng ta điều chỉnh hàm tạo của BookController và ShellController để chúng nhận tham số đầu vào là một biến thuộc kiểu IDataAccess.

Bước điều chỉnh này giúp các controller không còn phụ thuộc trực tiếp vào một lớp data access cụ thể nào mà chuyển sang phụ thuộc vào interface.

Để ý rằng với bước này, chúng ta không còn cần phải xây dựng controller sau data access nữa (vì chúng không còn phụ thuộc nhau), dẫn đến các lớp data access và controller giờ có thể được phát triển độc lập.

Bước 4. Điều chỉnh các lớp BinaryDataAccess, XmlDataAccess, JsonDataAccess

BinaryDataAccess | XmlDataAccess | JsonDataAccess | SimpleDataAccess

```
using Models;
public class BinaryDataAccess : IDataAccess
{
    public List<Book> Books { get; set; } = new List<Book>();
    private readonly string _file = "data.dat";
...
```

Ở bước này chúng ta điều chỉnh để các lớp data access sẵn có "kế thừa" từ IDataAccess.

Cấu trúc cú pháp này nhìn tương tự như kế thừa giữa các class nhưng nó phản ánh một quan hệ khác giữa class và interface: quan hệ *thực thi* (implementation).

Một class thực thi một giao diện sẽ phải xây dựng (implement) tất cả các thành viên của interface. Hay nói cách khác, class này sẽ phải xây dựng tất cả các phương thức và thuộc tính theo quy định của interface.

Ở đây, interface IDataAccess quy định rằng các class thực thi nó sẽ phải xây dựng hai phương thức Load, SaveChanges(), và phải chứa thuộc tính Books. Các thành viên này

chúng ta đều đã xây dựng từ trước ở các lớp data access nên không cần thiết phải viết thêm nữa.

Bước 5. Điều chỉnh phương thức ConfigRouter

```
1.  private static void ConfigRouter()
2.  {
3.      IDataAccess context = new BinaryDataAccess();
4.
5.      BookController controller = new BookController(context);
6.      ShellController shell = new ShellController(context);
7.  ...
```

Ở bước này chúng ta khai báo một biến thuộc kiểu `IDataAccess` và gán nó cho một object của `BinaryDataAccess`.

Giống như trong quan hệ kế thừa, object của lớp con có thể gán cho biến thuộc kiểu cha, một biến thuộc kiểu interface cũng có thể nhận một object của bất kỳ class nào thực thi giao diện này. Chúng ta cũng thấy, một biến thuộc kiểu interface được sử dụng như biến của bất kỳ kiểu dữ liệu nào khác của .NET.

Đến đây chúng ta thấy rằng, việc chuyển đổi sang một lớp data access khác giờ rất dễ dàng: chỉ cần thay đổi ở một chỗ duy nhất trong phương thức `ConfigRouter`.

Qua bước này, tất cả controller không còn phụ thuộc vào data access nữa. Sự phụ thuộc này đẩy sang một class khác, Program. Khi controller và data access không còn phụ thuộc nhau, chúng ta có thể phát triển và test chúng một cách độc lập.

Cách giải quyết sự phụ thuộc giữa các class như vậy có tên gọi chung là *Inversion of Control* (nguyên lý IoC). Việc khai báo và sử dụng biến thuộc kiểu interface này còn đơn giản hơn rất nhiều và hoàn toàn tự động nếu chúng ta sử dụng một *Dependency Injection container* nào đó (như Unity, Ninject).

Vận dụng interface trong project

Ở bài trước chúng ta đã xây dựng 3 class để hỗ trợ lưu trữ dữ liệu ở các định dạng khác nhau. Có một điều rất dễ nhận thấy là khi muốn chuyển từ định dạng lưu trữ này sang dạng khác chúng ta phải thay đổi code ở hàng loạt class: `Repository`, `BookController`, `ShellController`, `Program` (phương thức `ConfigRouter`).

Lý do phải thay đổi code ở nhiều chỗ như vậy là vì `Repository`, `BookController` và `ShellController` đều phụ thuộc vào lớp data access. Sự phụ thuộc này thể hiện ở chỗ constructor của các lớp controller đòi hỏi object của một lớp data access cụ thể nào đó (`BinaryDataAccess`, `XmlDataAccess`, v.v.).

Loại quan hệ này giữa các class được gọi là *quan hệ chặt* (tight-coupling) giữa các class, trong đó các lớp controller *phụ thuộc* vào các lớp data access.

Quan hệ phụ thuộc chặt này đơn giản khi sử dụng nhưng có thể gây ra rắc rối, giống như tình huống của các lớp controller và các lớp data access, khi cần thay thế class bị phụ thuộc

(các lớp data access). Quan hệ phụ thuộc chặt yêu cầu các lớp phụ thuộc phải xây dựng sau, dẫn tới không thể phát triển song song các class. Quan hệ chặt cũng có thể gây khó khăn cho việc test các class độc lập (vì chúng phụ thuộc vào nhau).

Để có thể phát triển song song hoặc dễ dàng thay thế class này bằng class khác, người ta cần làm giảm sự phụ thuộc giữa các class, thay phụ thuộc chặt bằng *phụ thuộc lỏng* (loosely-coupling). Interface là công cụ thường được sử dụng để làm giảm sự phụ thuộc này.

Để thực hiện ý tưởng này, trong phần thực hành trên chúng ta xây dựng interface `IDataAccess`. Interface này đóng vai trò một bản hợp đồng với 3 "điều khoản": property Books, phương thức Load và SaveChanges.

Các lớp controller và lớp `Repository` bây giờ hoàn toàn chỉ sử dụng `IDataAccess` làm kiểu dữ liệu cho biến context mà không cần biết đến class cụ thể nào. Bất kỳ lớp data access nào muốn làm việc với controller bây giờ phải tuân thủ theo bản hợp đồng `IDataAccess` trên, nghĩa là phải xây dựng đủ 3 thành phần `Books`, `Load`, `SaveChanges` theo các điều khoản của hợp đồng.

Nếu một class nào thỏa mãn các điều kiện trên thì object của nó có thể truyền cho `Repository` và controller ở giai đoạn khởi tạo. Việc này giúp controller và `Repository` không phụ thuộc vào bất kỳ data access cụ thể nào và chúng ta có thể phát triển controller và Repository hoàn toàn độc lập với data access.

Như trong phần thực hành trên, object của một lớp data access cụ thể (ví dụ, `XmlDataAccess`) được khởi tạo ở lớp `Program` cùng với object của các controller. Đây là chỗ duy nhất thực hiện ghép nối data access với controller.

Việc tách rời khởi tạo object của controller và data access sang một bên thứ ba như vậy được gọi là *Inversion of Control* (IoC). Việc khởi tạo object của controller sử dụng object của một lớp data access cụ thể như vậy được gọi là *Dependency Injection* (DI). DI có thể được thực hiện một cách tự động nhờ sử dụng một DI container.

Kết luận

Trong bài này chúng ta đã xem xét một kỹ thuật đặc biệt quan trọng trong phát triển ứng dụng: sử dụng interface. Chúng ta cũng thấy, interface cho phép giảm bớt sự phụ thuộc giữa các class và tạo ra quan hệ lỏng giữa chúng. Quan hệ này cho phép các class có thể được xây dựng và test độc lập.

Chúng ta cũng đã vận dụng interface cho các lớp truy xuất dữ liệu (data access) để giúp giảm bớt sự phụ thuộc của Repository và các lớp controller và một lớp data access cụ thể. Từ đây chúng ta có thể dễ dàng thay đổi cách thức lưu trữ dữ liệu.

Trong bài tiếp theo chúng ta sẽ xem xét sự thay đổi cuối cùng về các thức truy xuất dữ liệu với thư viện LINQ.

+ Nếu bạn thấy site hữu ích, trước khi rời đi hãy **giúp đỡ** site bằng một hành động nhỏ để site có thể phát triển và phục vụ bạn tốt hơn.
+ Nếu bạn thấy bài viết hữu ích, hãy giúp **chia sẻ** tới mọi người.
+ Nếu có thắc mắc hoặc cần trao đổi thêm, mời bạn viết trong phần **thảo luận** cuối trang.
Cảm ơn bạn!