Báo cáo về Design Pattern

I. Design Pattern là gì, có mấy loại?

- ***Design Pattern*** *là một giải pháp chung để giải quyết các vấn đề phổ biến khi thiết kế phần mềm trong OOP.*

Khi gặp phải một vấn đề thì mỗi lập trình viên sẽ có một cách giải khác nhau và không chắc chắn được những cách giải đó đã tối ưu hay chưa, Design Pattern ra đời để giả quyết vấn đề này, tạo ra tiếng nói chung, dễ làm việc cùng nhau, dễ bảo trì code hơn, tái sử dụng code và không mất nhiều thời gian.

- ***Design Pattern có 23 mẫu*** *được chia thành 3 nhóm chính là: Creational Pattern, Structural Pattern, Behavioral Pattern.*
    - **Creational Pattern:** các giải pháp liên quan đến khởi tạo object - *cách thức tạo object một cách linh hoạt và có khả năng tái sử dụng cao* (Factory Method, Abstract Factory, Builder, Prototype, Singleton,...).
    - **Structural Pattern:** các giải pháp liên quan đến thiết lập kết cấu, liên hệ giữa các đối tượng - *định nghĩa cách thức để kết hợp các objects theo một cấu trúc nhất định* (Adapter, Bridge, Composite, Decorator, ...).
    - **Behavioral Pattern:** các giải pháp liên quan đến các hành vi của đối tượng và giao tiếp giữa các đối tượng với nhau - *mô tả các hành vi mà các object giao tiếp và phân chia nhiệm vụ.* (Interpreter, Template, Method, Chain of Responsibility, Command, Iterator,...).

II. Một số Design Pattern đã tìm hiểu:

- ***Singleton (thuộc nhóm* Creational Pattern):** *Khi sử dụng singleton ta sẽ giới hạn duy nhất một instance của class tồn tại ở bất kì thời điểm hoặc vị trí nào và cho phép quyền truy cập là global.*
    - ***Tại sao phải dùng Singleton?***
        - Những tài nguyên được chia sẻ như DB để truy suất, file vật lý được sử dụng chung. Những tài nguyên này nên chỉ có 1 instance tồn tại để sử dụng và theo dõi trạng thái của nó trong suốt quá trình sử dụng.
        - Hoặc những kiểu object không cần thiết phải tạo instance mới mỗi lần sử dụng như logging object, hãy tưởng tượng chuyện gì xảy với mỗi dòng log chúng ta lại tạo mới một logging object.
        - Ví dụ 1:

```
-   public class Singleton {
-       // 1. Set instance là static
-       private static Singleton singletonInstance = new Singleton();
-       // 2. Constructor set về private
-       private Singleton() {
-       }
-       // 3. Getter set là static method
-       public static Singleton getSingletonInstance() {
-                return singletonInstance;
-       }
-       // 4. Loại bỏ setter
-       /*
-        * public void setSingletonInstance(Singleton singletonInstance) {
-        * this.singletonInstance = singletonInstance; }
-        */
-   }
```

- ***Các biến thể của singleton:***
  - ***Lazy Initialization:*** *Có thể thấy với cách implement như ở ví dụ 1, singletonInstance luôn được khởi tạo ngay cả khi nó chưa được sử dụng (hay còn gọi là Eager Singleton). Nếu chúng ta muốn tối ưu vùng nhớ, khi cần thì mới khởi tạo thì phải sửa lại code như thế nào? -> Lazy initilization*
    - *Ví dụ 2:*

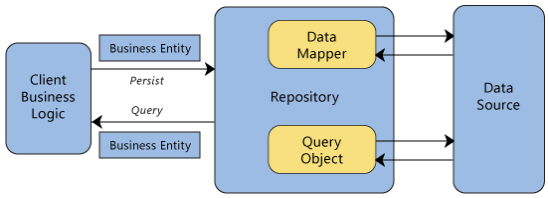
```
-   public class LazyInitializationSingleton {
-       // Không khởi tạo instance ở đây
-       private static LazyInitializationSingleton lazyInitializationSingletonInstance;
-       private LazyInitializationSingleton() {
-       }
-       public static LazyInitializationSingleton getLazyInitializationSingleton() {
-                if (Objects.isNull(lazyInitializationSingletonInstance)) {
-                // Khởi tạo instance chỉ khi instance này được request lần đầu
-                         lazyInitializationSingletonInstance = new LazyInitializationSingleton();
-                }
-                return lazyInitializationSingletonInstance;
-       }
-   }
```

    - Vấn đề của cách implement này là trong trường hợp nhiều thread cùng gọi hàm getLazyInitializationSingleton() và đều thấy rằng instance chưa được khởi tạo. Điều này dẫn đến việc có khả năng các thead sẽ cùng gọi hàm khởi tạo một new instance, đồng nghĩa với việc nó đã phá vỡ tiêu chí của singleton về việc chỉ có một instance duy nhất ở mọi thời điểm.
  - ***Thread-safe Singleton:*** Để giải quyết vấn đề của lazy *initialization singleton*, chúng ta sẽ tạo một object 'padlock' và sử dụng từ khóa lock. Việc này sẽ lock tất cả các thread không cho thread đó gọi method Singleton() và tạo instance cho đến khi thread đang gọi thực thi xong.
    - Ví dụ 3:

```
-   public sealed class Singleton
-   {
-       Singleton() {}
-       private static readonly object padlock = new object();
-       private static Singleton instance = null;
-       public static Singleton Instance
-       {
-           get
-           {
-               if (instance == null)
-               {
```

```
-                   lock (padlock)  //trường hợp khi nhiều thread gọi method này cùng một lúc khi
    chưa tạo instance nếu không dùng lock sẽ dẫn đến việc nhiều instance được tạo ra và phá vỡ quy
    tắc của Singleton.
-                   {
-                       if (instance == null)
-                       {
-                           instance = new Singleton();
-                       }
-                   }
-               }
-               return instance;
-           }
-       }
-   }
```

- ***Repository (thuộc nhóm Behavioral Pattern):*** *Nằm giữa Data stores và Business Logic, cho phép chúng ta lấy records từ datasets, và những bản ghi đó như một object collection.*

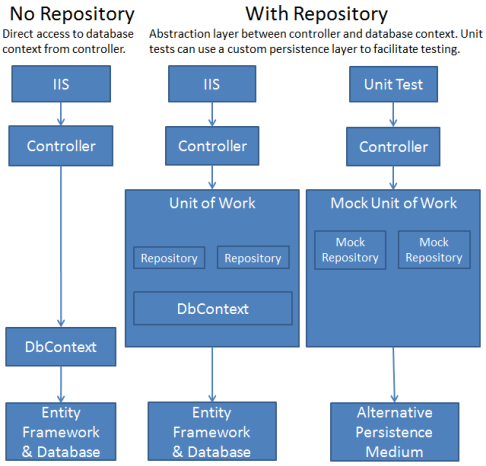


  - *Tại sao phải sử dụng Repository?*
    - *Test dễ dàng hơn: từ việc tạo ra một interface rồi implement vào object, lúc này sẽ dễ kiểm tra lỗi hơn khi các object không phụ thuộc vào nhau.*
    - *Dễ dàng thay đổi data stores mà không cần phải thay đổi api: Do respository nằm ở giữa* Business Logic và data stores đóng vai trò kết nối giữa tầng Business và Model.
    - *Một nơi duy nhất để thay đổi quyền truy cập dữ liệu cũng như xử lý dữ liệu:*

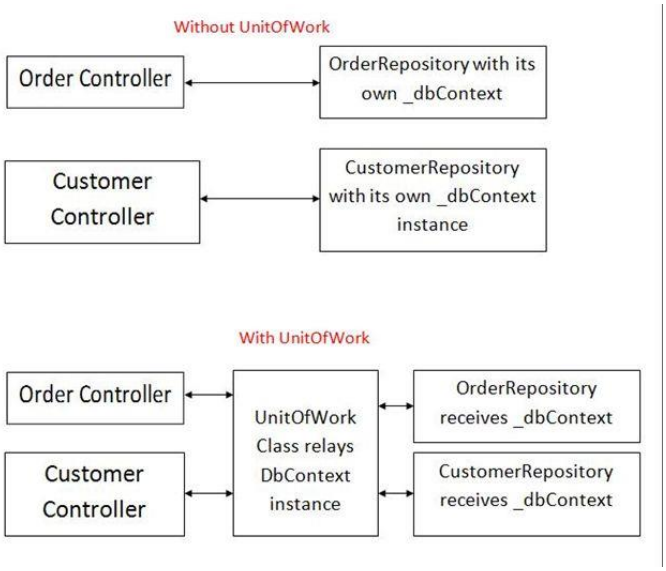
      Thông thường các phần truy xuất database sẽ nằm rải rác trong code, khi muốn thực hiện một thao tác lên database thì phải tìm trong code cũng như các thuộc tính trong bảng để xử lý gây lãng phí thời gian và công sức.
  - *Sử dụng Repository*
    - Chúng ta sẽ có một lớp *trừu tượng ngay trên tầng Cơ sở dữ liệu (database), bởi vậy thay vì việc Controller tương tác trực tiếp với Model, Controller sẽ làm việc với lớp *trừu tượng đã được đóng gói với các thao tác trong Model.
    - Nó giúp bạn dễ dàng hơn trong việc thay đổi sử dụng các loại ORM khác nhau, hoặc các kỹ thuật ở tầng Database khác nhau. Bạn có thể thay đổi sử dụng MongoDB thay vì MySQL hay tương tự như vậy mà không sợ ảnh hưởng quá nhiều đến việc xử lý logic ở Controller. Việc chúng ta cần làm chỉ là thực hiện thay đổi ở các lớp *trừu tượng thay vì phải đi tìm ở tất cả các Controller để thay đổi thao tác phù hợp với những thay đổi phía Model.

      Tài liệu tìm hiểu: https://codewithshadman.com/repository-pattern-csharp/

- ***Unit Of Work trong Repository Pattern:*** Unit Of Work được sử dụng để đảm bảo nhiều hành động như insert, update, delete...được thực thi trong cùng một transaction thống nhất. Nói đơn giản hơn, nghĩa là khi một hành động của người dùng tác động vào hệ thống, tất cả các hành động như insert, update, delete...phải thực hiện xong thì mới gọi là một transaction thành công. Gói tất cả các hành động đơn lẻ vào một transaction để đảm bảo tính toàn vẹn dữ liệu.



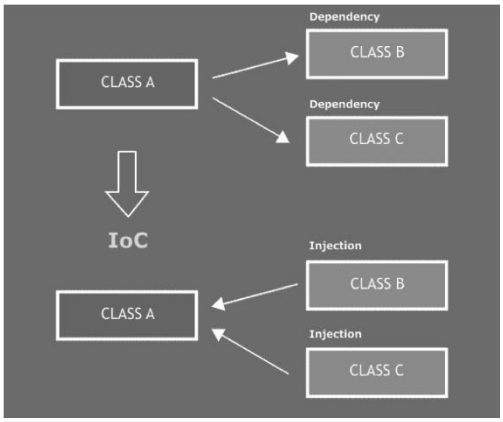
  - Ví dụ về Customer Controller và Order Controller sử dụng cùng một transaction duy nhất.



  - Database sẽ toàn vẹn vì Customer và Order đều sử dụng chung một đối tượng của lớp DbContext. Khi một trong hai bị lỗi thì dữ liệu cả 2 sẽ không bị lưu lại.
  - Trường hợp ứng dụng: ví dụ về một trang web bán hàng khi Customer lần đầu truy cập vào và chọn mua hàng thì phải vừa tạo Customer vừa tạo một Order của Customer đó, nếu Order xảy ra trước khi tạo Customer thì databse sẽ lỗi vì không biết được Order là của Customer nào.

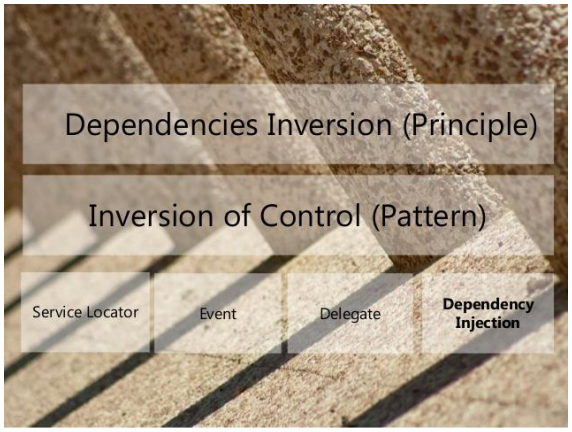
    Tài liệu tìm hiểu: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

- ***Inversion of Control:*** Là một nguyên lý thiết kế mà trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.



- Hình bên là Class A ở hai trường hợp, áp dụng IoC và không. Lớp này nó làm việc phụ thuộc vào hai lớp Class B và Class C (B và C gọi là các dependency)
  - Ở mô hình không IoC, Class A khi cần chủ động tạo ra đối tượng lớp Class B và Class C (nó nắm quyền khởi tạo, điều khiển)
  - Với mô hình IoC thì class A không tự khởi tạo cũng không chịu trách nhiệm quản lý Class B, Class C. Nó nhận được hai dependency này từ bên ngoài thông qua một cơ chế nào đó (như bằng setter, bằng tham số hàm tạo lớp A, bằng gán thuộc tính ...)

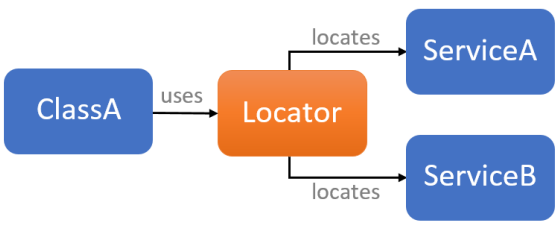
- Triển khai nguyên lý IoC thường thực hiện bởi các Framework theo từng loại ngôn ngữ lập trình, các mô hình lập trình (pattern) triển khai từ IoC:



  - Qua hình trên ta thấy được Inversion of Control: là một design pattern tuân thủ theo nguyên lý thiết kế Dependency Inversion.
  - Có nhiều cách để hiện thực Inversion of Control: Service Locator, Event, Delegate, Dependency Injection (DI), ...).

  Tài liệu tìm hiểu: https://toidicodedao.com/2015/11/03/dependency-injection-va-inversion-of-control-phan-1-dinh-nghia/

- **Service Locator:** một mô hình khá phổ biến trong C#.
  - **Sử dụng:** mô hình này có nghĩa là cần có cơ chế sao cho tất các các tiến trình (các đối tượng, dịch vụ ...) được đưa vào quản lý bởi một trung tâm được gọi là Service Locator, từ trung tâm này các lớp muốn sử dụng dịch vụ nào thì Locator sẽ cung cấp (khởi tạo nếu chưa, trả về dịch vụ cho đối tượng sử dụng).



        - **Lợi ích:** Đây là mô hình hoạt động đơn giản như một cơ chế liên kết với nhau lúc thực thi.
            - Cho phép code được thêm vào lúc chạy mà không cần biên dịch lại, hay khởi động lại một tiến trình nào đó.
            - Ứng dụng tự tối ưu vào lúc chạy bằng cách thêm bớt các nội dung từ trung tâm.
            - Có thể chia nhỏ thư viện lớn, ứng dụng lớn và liên kết chúng lại với nhau thông qua trung tâm đăng ký này.
        - **Hạn chế:**
            - Mọi thứ bên trong Service Locator là hộp đen tách biệt với phần còn lại của hệ thống.
            - Điều này có thể dẫn đến khó phát hiện lỗi, có thể dẫn đến không ổn định.
            - Trung tâm đăng ký là duy nhất, điều này có thể dẫn tới tình trạng nghẽn với các ứng dụng nhiều tiến trình.
            - Có thể dẫn tới các lỗ hổng vì cho phép mã được inject vào ứng dụng.

        Tài liệu tìm hiểu: https://xuanthulab.net/mo-hinh-lap-trinh-service-loacator.html

- ***Dependency Injection (DI - đảo ngược phụ thuộc):*** Là một hình thức cụ thể của Inverse of Control (Dependency Inverse) đã nói ở trên. DI thiết kế sao cho các dependency (phụ thuộc) của một đối tượng có thể được đưa vào, tiêm vào đối tượng đó (Injection) khi nó cần tới (khi đối tượng khởi tạo).
    - ***Khi sử dụng phải:***
        - Xây dựng các lớp (dịch vụ) có sự phụ thuộc nhau một cách lỏng lẻo, và dependency có thể tiêm vào đối tượng (injection) - thường qua phương thức khởi tạo constructor, property, setter.
        - Xây dựng được một thư viện có thể tự động tạo ra các đối tượng, các dependency tiêm vào đối tượng đó, thường là áp dụng kỹ thuật Reflection của C# (xem thêm lớp type): Thường là thư viện này quá phức tạp để tự phát triển nên có thể sử dụng các thư viện có sẵn như: Microsoft.Extensions.DependencyInjection hoặc thư viện bên thứ ba như Windsor, Unity Ninject ...
    - ***Ưu điểm:***
        - Làm giảm (khử) sự phụ thuộc giữa các object: Các object không giao tiếp trực tiếp với nhau mà thông qua Interface.
        - Dễ viết unit test: Dễ hiểu là khi ta có thể Inject các dependency vào trong một class thì ta cũng dễ dàng "tiêm" các mock object vào class (được test) đó.
        - Dễ dàng thấy quan hệ giữa các module (Vì các dependecy đều được inject vào constructor)
    - ***Nhược điểm:***
        - *Khó debug: Vì sửa dụng các interface nên có thể gặp khó khăn khi ta debug source code (bằng mắt) vì không biết implement nào thực sự được truyền vào.*
        - *Ẩn lỗi khi comlile: Vì Dependency Injection ẩn các dependency nên một số lỗi chỉ xảy ra khi chạy chương trình thay vì có thể phát hiện khi biên dịch chương trình.*
        - *Các object được khởi tạo toàn bộ ngay từ đầu, có thể làm giảm performance*
        - *Làm tăng độ phức tạp của code*
    - **Trường hợp không sử dụng DI:**

```
-   public class Horn {
-       public void Beep () => Console.WriteLine ("Beep - beep - beep ...");
-   }
-   public class Car {
-       public void Beep () {
```

```
-           // chức năng Beep xây dựng có định với Horn
-           // tự tạo đối tượng horn (new) và dùng nó
-           Horn horn = new Horn ();
-           horn.Beep ();
-       }
-   }
-   // Code viết như trên khá thông dụng và vẫn chạy tốt. Bấm còi xe vẫn kêu Beep, beep ...
-   var car = new Car();
-   car.Beep();          // Beep - beep - beep ...
```

Nhưng code trên có một vấn đề là tính linh hoạt khi sử dụng. Chức năng Beep() của Car nó tự tạo ra đối tượng Horn và sử dụng nó - làm cho Car gắn cứng vào Horn với cấu trúc khởi tạo hiện thời. Việc thay đổi Horn làm cho Car không còn dùng được nữa, nếu muốn Car hoạt động cần sửa lại code của Car.

- **Trường hợp có thể sử dụng DI:**

```
-   public class Horn {
-       public void Beep () => Console.WriteLine ("Beep - beep - beep ...");
-   }
-   public class Car {
-       // horn là một Dependecy của Car
-       Horn horn;
-       // dependency Horn được đưa vào Car qua hàm khởi tạo
-       public Car(Horn horn) => this.horn = horn;
-       public void Beep () {
-           // Sử dụng Dependecy đã được Inject
-           horn.Beep ();
-       }
-   }
-   //Khi sử dụng
-   Horn horn = new Horn();
-   var car = new Car(horn); // horn inject vào car
-   car.Beep(); // Beep - beep - beep ...
```

Code trên hoạt động tương tự trường hợp thứ nhất. Bằng cách khai bảo Horn là một biến thành viên trong Car, Car đã có một dependency là đối tượng lớp Horn, dependency này không phải do Car tạo ra, nó được bơm vào (cung cấp) thông qua phương thức khởi tạo của nó.

Kết quả gọi car.Beep(); có vẻ kết quả vẫn như trên, nhưng code mới này có một số lợi ích. Ví dụ, nếu sửa cập nhật lại Horn bằng cách sửa phương thức khởi tạo của nó, thì lớp Car không phải sửa gì.

Như vậy, viết code mà các dependency có thể đưa vào từ bên ngoài (chủ yếu qua phương thức khởi tạo), giúp cho các dịch vụ tương đối độc lập nhau. Nó là cơ sở để có thể dùng các Framework hỗ trợ DI (tự động phân tích tạo dịch vụ, dependency).

- **Các kiểu Dependency Injection:**
  - *Inject thông qua phương thức khởi tạo:* Cung cấp các Dependency cho đối tượng thông qua hàm khởi tạo (như đã thực hiện ở ví dụ trên) - tập trung vào cách này vì thư viện .NET hỗ trợ sẵn.
  - *Inject thông qua setter:* Các Dependency như là thuộc tính của lớp, sau đó inject bằng gán thuộc tính cho Depedency object.denpendency = obj;
  - *Inject thông qua các Interface:* Xây dựng Interface có chứa các phương thức Setter để thiết lập dependency, interface này sử dụng bởi các lớp triển khai, lớp triển khai phải định nghĩa các setter quy định trong interface.
  - ➔ *Trong ba kiểu Inject thì Inject qua phương thức khởi tạo rất phổ biến vì tính linh hoạt, mềm dẻo, dễ xây dựng thư viện DI...*

- **Xây dựng lại code ở phần trên bằng kỹ thuật *Inject thông qua phương thức khởi tạo*:**

```
-   public interface IHorn {
-       void Beep ();
-   }
-   public class Car {
-       IHorn horn;// IHorn (Interface) là một Dependecy của Car
-       public Car (IHorn horn) => this.horn = horn; // Inject từ hàm  tạo
-       public void Beep () => horn.Beep ();
-   }
-   public class HeavyHorn : IHorn
-   {
-       public void Beep() => Console.WriteLine("(kêu to lắm) BEEP BEEP BEEP ...");
-   }
-   public class LightHorn : IHorn
-   {
-       public void Beep() => Console.WriteLine("(kêu bé lắm) beep  bep  bep ...");
-   }
-   //Sử dụng
-   Car car1 = new Car(new HeavyHorn());
-   car1.Beep();                              // (kểu to lắm) BEEP BEEP BEEP ...
-   Car car2 = new Car(new LightHorn());
-   car2.Beep();
```

Để triển khai thực tế thì cần có một dịch vụ trung tâm gọi là DI Container, tại đó các lớp (dịch vụ) đăng ký vào, sau đó khi sư dụng dịch vụ nào nó tự động tạo ra dịch vụ đó, nếu dịch vụ đó cần dependency nào nó cũng tự tạo dependency và tự động bơm vào dịch vụ cho chung ta. Để tự xây dựng ra một DI Container rất phức tạp, nên ở đây ta không cố gắng xây dựng một DI Container riêng, thay vào đó ta sẽ sử dụng các thư viện hỗ trợ sẵn cho .NET.

- **DI Container:** Mục đích sử dụng DI, để tạo ra các đối tượng dịch vụ kéo theo là các Dependency của đối tượng đó. Để làm điều này ta cần sử dụng đến các thư viện, có rất nhiều thư viện DI - Container (cơ chứa chứa và quản lý các dependency) như: Windsor, Unity Ninject, DependencyInjection ...

  Tích hợp Package Microsoft.Extensions.DependencyInjection vào dự án:

```
-   dotnet add package Microsoft.Extensions.DependencyInjection
```

  Sử dụng namespace:

```
-   using  Microsoft.Extensions.DependencyInjection;
```

➔ *Từ đây các đối tượng lớp, các dependency ta gọi chúng là các dịch vụ (service)!*

Lớp ServiceCollection: Lớp triển khai giao diện IServiceCollection nó có chức năng quản lý các dịch vụ (đăng ký dịch vụ - tạo dịch vụ - tự động inject - và các dependency của đích vụ ...). ServiceCollection là trung tâm của kỹ thuật DI, nó là thành phần rất quan trọng trong ứng dụng ASP.NET

Lớp ServiceProvider: Lớp ServiceProvider cung cấp cơ chế để lấy ra (tạo và inject nếu cần) các dịch vụ đăng ký trong ServiceCollection. Đối tượng ServiceProvider được tạo ra bằng cách gọi phương thức BuildServiceProvider() của ServiceCollection

Tài liệu tìm hiểu:

https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

https://xuanthulab.net/dependency-injection-di-trong-c-voi-servicecollection.html

- **Áp dụng DI vào code**
    - **Dependency là gì?**
        - Dependency là những module cấp thấp, hoặc cái service gọi từ bên ngoài. Với cách code thông thường, các module cấp cao sẽ gọi các module cấp thấp. Module cấp cao sẽ phụ thuộc và module cấp thấp, điều đó tạo ra các dependency.
        - Vấn đề:

```
public class Cart
{
    public void Checkout(int orderId, int userId)
    {
        Database db = new Database();
        db.Save(orderId);

        Logger log = new Logger();
        log.LogInfo("Order has been checkout");

        EmailSender es = new EmailSender();
        es.SendEmail(userId);
    }
}
```

Cách viết code như trên sẽ khiến chúng ta gặp những vấn đề trong tương lai:

Rất khó test hàm Checkout này, vì nó dính dáng tới cả hai module Database và EmailSender.

Trong trường hợp ta muốn thay đổi module Database, EmailSender,… ta phải sửa toàn bộ các chỗ khởi tạo và gọi các module này. Việc làm này rất mất thời gian, dễ gây lỗi.

➔ Inversion of Control và Dependency Injection đã ra đời để giải quyết những vấn đề này.

Để các module không "kết dính" với nhau, chúng không được kết nối trực tiếp, mà phải thông qua interface. Đó cũng là nguyên lý cuối cùng trong SOLID.

1. *Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.*

2. *Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation)*

Ví dụ: ta viết lại đoạn code ở trên:

```
// Interface
public interface IDatabase
{
    void Save(int orderId);
}
public interface ILogger
{
    void LogInfo(string info);
}
public interface IEmailSender
{
    void SendEmail(int userId);
}
```

```
// Các Module implement các Interface
public class Logger : ILogger
{
    public void LogInfo(string info)
    {
        //...
    }
}
public class Database : IDatabase
{
    public void Save(int orderId)
    {
        //...
    }
}
public class EmailSender : IEmailSender
{
    public void SendEmail(int userId)
    {
        //...
    }
}
```

Lúc này hàm Checkout mới sẽ trông như sau:

```
public void Checkout(int orderId, int userId)
{
    // Nếu muốn thay đổi database, ta chỉ cần thay dòng code dưới
    // Các Module XMLDatabase, SQLDatabase phải implement IDatabase
    //IDatabase db = new XMLDatabase();
    //IDatebase db = new SQLDatabase();
    IDatabase db = new Database();
    db.Save(orderId);

    ILogger log = new Logger();
    log.LogInfo("Order has been checkout");

    IEmailSender es = new EmailSender();
    es.SendEmail(userId);
}
```

Để dễ quản lý, ta có thể bỏ tất cả những hàm khởi tạo module vào constructor của class Cart:

```
public class Cart
{
    private readonly IDatabase _db;
    private readonly ILogger _log;
    private readonly IEmailSender _es;

    public Cart()
    {
        _db = new Database();
        _log = new Logger();
        _es = new EmailSender();
    }

    public void Checkout(int orderId, int userId)
    {
        _db.Save(orderId);
        _log.LogInfo("Order has been checkout");
        _es.SendEmail(userId);
    }
}
```

Cách này thoạt nhìn khá khá ổn. Tuy nhiên, nếu có nhiều module khác cần dùng tới Logger, Database, ta lại phải khởi tạo các Module con ở constructor của module đó.

Dependency Injection giải quyết được vấn đề này. Các Module cấp thấp sẽ được inject (truyền vào) vào Module cấp cao thông qua Constructor hoặc thông qua Properties. Nói một cách đơn giản dễ hiểu về DI:

```
public Cart(IDatabase db, ILogger log, IEmailSender es)
{
        _db = db;
        _log = log;
        _es = es;
 }
//Dependency Injection một cách đơn giản nhất
Cart myCart = new Cart(new Database(),
                       new Logger(), new EmailSender());
//Khi cần thay đổi database, logger
myCart = new Cart(new XMLDatabase(),
                  new FakeLogger(), new FakeEmailSender());
```

```
//Với mỗi Interface, ta define một Module tương ứng
DIContainer.SetModule<IDatabase, Database>();
DIContainer.SetModule<ILogger, Logger>();
DIContainer.SetModule<IEmailSender, EmailSender>();

DIContainer.SetModule<Cart, Cart>();

//DI Container sẽ tự inject Database, Logger vào Cart
var myCart = DIContainer.GetModule();

//Khi cần thay đổi, ta chỉ cần sửa code define
DIContainer.SetModule<IDatabase, XMLDatabase>();
```

Sau khi áp dụng Dependency Injection, code bạn sẽ dài hơn, có vẻ "phức tạp" hơn và sẽ khó debug hơn. Đổi lại, code sẽ uyển chuyển, dễ thay đổi cũng như dễ test hơn.

Phần DI này em đã áp dụng vào phần bài test của OOP tại thư mục code.