Interface 라는 게 OOP에서 가장 중요한 개념인 class를 추상화 한 것이라 더욱 개념잡기가 힘듭니다.

일반적으로 interface라는 용어는 "상이한 시스템간의 계약 정보를 표현하는 것" 이라고 정의할 수 있습니다.

C#의 예약어이면서, OOP의 기본개념으로 자리잡고 있는 Interface는 형태상으로 몇 가지 특징이 있습니다.

1. 선언(Declaration)은 있고, 정의(Definition)은 없다. (Abstract class도 같은 의미)

2. 접근자는 쓸 수 없고, 모두 public 이다. (계약사항이 투명해야 하고, 명확해야 하므로)

3. 다중 상속이 가능하다.

특징은 더 있겠지만... 생각나는 게 이 정도이고...

Interface를 사용해야 하는 이유에 대해서 몇 가지 서술해보자.

1. 외부 시스템에 제공할 서비스 또는 객체 정보를 직접적인 class가 아닌 "선언" 만을 제공한다.
2. 다중 상속에 따른 class의 표현을 여러 가지로 가능케 한다.
3. 의도하지 않은 속성/메소드 공개를 막기 위해
4. 실제 object 의 class 정보를 제공하고 싶지 않을 때 ( 또는 명확히 하고 싶을 때)
5. OOP의 상속에 억매이지 않고, 구현할 수 있도록 개방성을 제공할 때...

**Interface 설계시의 주의점.**

**1. 속성, 메소드의 변화가능성이 많다면, 아예 사용하지 마라.**

interface에 변경이 일어난다면, 아주 위험한 상황이 올 수 있다. (OOP와 마찮가지로) 계약이 변경된다는 뜻은, 모든 구현이 변경되어야 함을 뜻하고, 이는 전체적인 Class 의 상속 체계에 위험을 줄 수 있다.

즉 Interface라는 것이 OOP의 Abstract Class와 같이 상속(엄밀히 구현)의 원형이 되는 것이므로, 원형이 변경되면 자손들도 변경이 일어나야 한다는 뜻인데, interface의 경우, 전혀 다른 제품으로의 확장도 지원할 수 있으므로, Interface 설계 시 아주 조심해야 한다.

**2. 너무 많은 속성, 메소드를 노출시키지 마라 (5개 이하) - 인터페이스의 성격이 명확해야 한다.**

너무 많은 정의가 있다는 것은, 계약이 다른 뜻으로 해석될 수도 있고, 명확한 의미를 이해하거나, 구현하는 데 방해가 될 수 있다. 보통의 경우는 5개 이하로 하고, 더 많아질 경우에는 Interface 상속으로 해결해야 한다.

예:

public interface IVertex

{

Guid Id {get;set;}

IList<ILink> InLinks { get; }

IList<ILink> OutLinks { get; }

}

public interface INamedVertex : IVertex

{

string Name { get; set;}

}

**3. Interface의 name은 xxx-able, xxx-er, xxx-tion 등 실제 복합 기능을 하는 class와는 달리 단품의 성격을 나타내도록 한다.**

Interface 명명시에는 몇가지 방식이 있는데, Interface가 표현하고자 하는 것에 따라 몇가지 형태로 사용할 수 있다.

-able (예: ISerializable, ICloneable 등) 동작할 수 있는 메소드가 있음을 나타낸다.

-er (예 : ISerializer, IComposer, ICompressor, IFormatter, IStringProvider 등) 동작 자체에 의미가 있는 경우

-tion 또는 Class 원형 이름 (예: IUserType, IGraph, IProcess 등) 객체의 가장 추상화 단계를 표현하는 Interface

인터페이스를 정의한 경우, 인터페이스를 구현하는 방식에는 크게 2가지로 나뉜다.

1. 인터페이스를 직접 구현
2. 클래스는 보통 Abstract class로 기본 동작을 표현하고, Abstract class를 상속 받아, Concrete class를 구현을 하는 방식

직접 구현하는 방식에는 ICompressor <- GZipCompressor, DeflateCompressor, SevenZipCompressor 등과 같이 Algorithm 자체가 상이하고, 추상화할 정보가 없을 경우이다.

이 방식은 Abstract class에 의한 상속의 사슬에 얽매이지 않기 때문에, 외부 개발자가 제공되는 Interface로 다양한 구현이 가능토록 한다.

Abstract class를 정의한 후, Concrete class로 상속하는 경우는 폐쇄적인 방식으로 유용하다. 즉 외부에는 interface로 노출하여 다양한 구현이 가능토록 제공은 하지만,

내부에서는 OOP 상속 및 다형성을 계속 가져가겠다는 의도인 것이다.

보통의 경우 2번이 1번의 모든 경우를 포함하기 때문에 2번 방식이 주를 이룬다. 패턴으로 말한다면 Abstract Factory Pattern이 2번 방식을 사용한다고 보면 된다.

마지막으로 인터페이스를 구현하지 않고, OOP 기능만을 이용하여 Abstract class, Concrete class 상속 체계만 정의하여 쓰는 경우가 있는데,

이는 인터페이스가 없으므로 Abstract class가 외부에 노출 시켜서 사용할 수 있다. 다만, 문제는 C#, Java의 경우 다중상속이 허용되지 않기 때문에,

다양한 Interface 다중상속 (결합) 또는 분리로 표현하고자 한다면, 문제가 될 수 있다.

다음번 글에서는 RCL, RealAdmin에서 사용한 Interface를 가지고, 실전에서 사용하는 방식을 설명하겠습니다.