C#으로 배우는 적응형 코드

Chapter7. 리스코프 치환 원칙

이찬헌

리스코프 치환 원칙의 이해

리스코프 치환 원칙(LSP, Liskov Substitution Principle)은 안정적으로 사용할 수 있는 **상속 구조**를 표현하기위한 가이드 라인의 집합

개방/폐쇄 원칙과 단일 책임 원칙을 더욱 강력하게 준수하는 밑거름이 된다.

리스코프 치환 원칙의 이해

형식적인 정의

S가 T의 서브타입이면 T타입의 객체는 프로그램의 실행에 문제를 일으키지 않고 S타입의 객체로 치환(대체)이 가능해야 한다. - 바바라 리스코프

LSP와 연관된 코드의 요소

1. 기반타입(Base Type)
2. 서브타입(Sub Type) - 클라이언트는 어떤 서브타입을 사용하는지 알필요 없다
3. 문맥 (Context) - 클라이언트, 서브타입이 상호작용하지 않는다면 LSP를 준수한다 말할수 없다.

리스코프 치환 원칙의 이해

LSP 규칙

계약 규칙

1. SubType에 더 강력한 사전 조건을 정의할 수 없다.
2. SubType에 더 완화된 사후 조건을 정의할 수 없다.
3. SuperType의 불변식은 서브타입에서도 반드지 유지 되어야 한다.

가변성 규칙

1. SubType의 메서드 인수는 반 공변성을 가져야 한다.
2. SubType의 리턴 타입은 공변성을 가져야 한다.
3. SubType은 SuperType이 발생시키는 예외와 다른 예외를 발생시켜서는 안 된다.

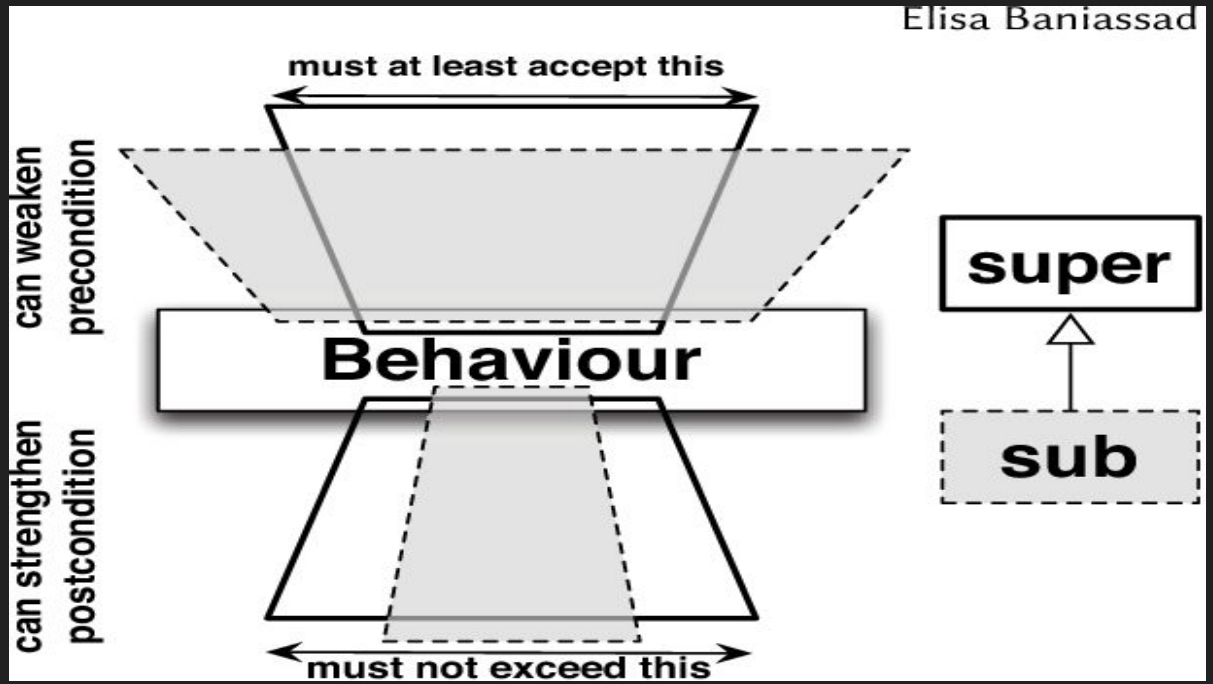
사전 조건

- 메서드가 안정적이고 오류 없이 실행되기 위해 필요한 모든 조건을 정의한 것
- 사전 조건에 의해 검사한 모든 상태는 반드시 클라언트가 공개적으로 접근이 가능해야 한다.
    - 호출 실패를 유발한 사전 조건에 대해 가능한 많은 정보를 제공하는 것이 중요하다.
- private 속성은 대상이 될 수 없으며, 메서드 매개변수와 public 속성들만이 사전 조건에 포함될 수 있다.

사후 조건

- 메서드 호출이 완료된 후에도 객체가 유효한 상태로 남아 있는지 여부를 검사하는 것.
- 사후 조건과 동일한 방식으로 구현할 수 있으며 대신 메서드를 종료하기 전에 작성한다.
- 사후조건을 완화할 수 없는 이슈는 새로운 서브클래스를 사용하는 클라이언트에서 문제를 유발할 수 있기 때문.

사전 조건, 사후 조건


Elisa Baniassad
must at least accept this
can weaken precondition
Behaviour
can strengthen postcondition
must not exceed this
super
sub

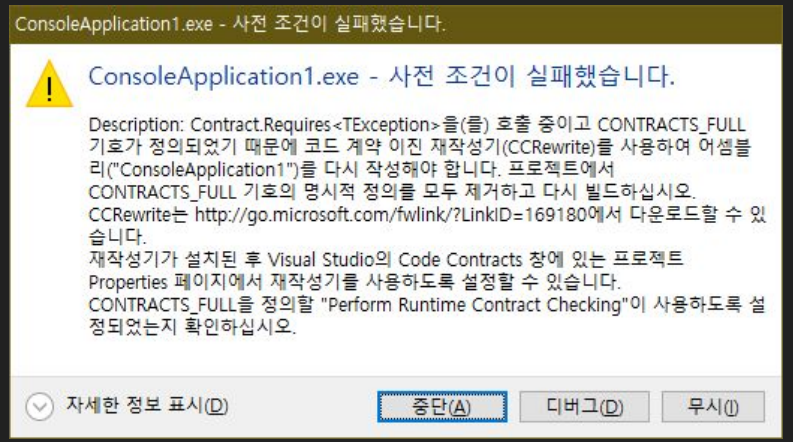
계약

불변 데이터

- 객체의 생명 주기 동안 참 인 상태로 유지되는 명제이다.
    - 즉 객체를 생성하고 나면 true 이며, 그 이후 객체가 범위를 벗어나기 전까지는 계속 true 상태를 유지한다
- 슈퍼타입의 불변데이터는 서브타입에서도 불변이여야 한다.
- 유효하지 않은 값이 설정되는 것을 방지할 수 있다.

코드 계약

- 코드 계약은 닷넷 프레임워크 4.0 이전에는 별도의 라이브러리 였지만 이제는 닷넷 프레임워크에 통합 되었다.
- 정적 계약 검증 기능을 통해 실행하지 않고도 계약의 위반 여부를 검증할 수 있다.
- 코드 계약에 대한 설정을 해야 하는데 안됨...
- VS2017에서 지원 안 한다능... (방법은 존재함)



코드 계약 - 사전 조건

- System.Diagnostics.Contracts 네임스페이스를 참조하여 사용하며 static 클래스인 Contract 클래스가 대부분의 기능을 제공한다.
- Contract.Requires 으로 사전조건을 정의하며 Boolean으로 평가되는 조건식을 받는다.
- Contract.Requires<> 제네릭 타입을 이용하여 원하는 예외를 발생시킬 수 있다.

코드 계약 - 사후 조건

- 사전 조건과 유사한 방법으로 설정할 수 있다.
- Contract.Ensures 으로 사후조건을 정의한다.
- 주의할 점은 Contract.Ensures 메서드를 호출한 다음 작성할 수 있는 구문은 return 구문뿐이다. (다른 구문이 존재한다면 그로 인해 상태 값이 다시 수정될 수 있다)
- Contract.Result 메서드를 이용하여 실제 리턴값을 액세스 할 수 있다. (실제로 메서드가 리턴된 이후의 최종 결과를 리턴한다)

코드 계약 - 불변 데이터

- 코드계약을 이용하면 클래스의 불변 데이터에 대한 정보를 선언적으로 정의할 수 있는 private 메서드를 작성할 수 있다.
- Contract.Invariant 메서드를 사용하여 불변데이터를 검증 한다. 필요한 만큼 호출 가능하기에 불변데이터 마다 호출하는 것이 좋다.
- ContractInvariantMethodAttribute 를 지정하면 다른 메서드들의 호출 전후로 지정 메서드를 호출해서 불변 데이터를 검사할 수 있다.

코드 계약 - 인터페이스 계약

- Contract.Requires , Ensures, Invariant 등을 이용해 자연스럽게 구현할 수 있지만 코드가 더 지저분해지는 경향이 있다.
- 검사하려는 인터페이스를 분리하고 ContractClassAttribute, ContractClassForAttribute 이용해서 계약조건을 명시하는 클래스를 만들수 있다.
- 한번만 작성하면 해당 인터페이스를 구현하는 모든 클래스에 적용가능하다.

코드 계약- 마지막 강조

- 코드계약 검사가 실패한다고 해서 클라이언트가 이 예외를 캐치해서는 안 된다.
- 예외를 캐치했다는 클라이언트가 해당 예외상황을 복구할 수 없다는 것을 의미.
- 가장 이상적인건 출시 전에 수정되는 것
- 수정이 안되었다면 잠재적으로 애플리케이션이 올바른 상태가 아니기에 애플리케이션이 실패하도록 하는 것이 바람직. (웹의 경우 오류페이지)

공변성과 반 공변성

공변 : X를 Y로 바꾸어 사용할 수 있는 경우, C<T>가 C<X>를 C<Y>로 바꾸는 것이 가능하다면 공변이다.

- 자신과 자식으로만 형변환. out 키워드로 지정. 리턴 타입 관련.

반공변 : X를 Y로 바꾸어 사용할 수 있는 경우, C<T>가 C<Y>를 C<X>로 바꾸는 것이 가능하다면 반공변이다.

- 자신과 부모로만 형변환. in 키워드로 지정. 매개변수 타입 관련.

※ 참고

C# 4.0 이전에는 제네릭 타입은 가변성을 지원하지 않았다. 제네릭 타입은 모두 불변이었으며 타입 매개변수가 다른 경우 대체가 일절 불가능했다.

C# 4.0이 나오면서 비로소 공변과 반공변을 지원하도록 in과 out 키워드가 추가 되었으며 인터페이스와 델리게이트에서 사용가능하다.

공변성

- 공변성의 일반적인 활용 예
- 제네릭에서 리턴타입에만 공변을 지원한다.

```
//User는 Entity의 SubType
public delegate T Covariant<out T>();

public static Entity SampleEntity()
{
    return new Entity();
}

public static User SampleUser()
{
     return new User ();
}

public void Test()
{

    Covariant<Entity> entity = SampleEntity;
    Covariant<User > user = SampleUser;

    entity = user;
    entity ();
}
```

공변성

잘못된 예제

```
public class EntityRepository
  {
      public virtual Entity GetByID(Guid id)
      {
          return new Entity();
      }
  }
  public class UserRepository : EntityRepository
  {
      public override User GetByID(Guid id)
      {
          return new User();
      }
  }
```

```
public interface IEntityRepository<TEntity> where TEntity : Entity
  {
      TEntity GetByID(Guid id);
  }

  // . . .
public class UserRepository : IEntityRepository<User>
  {
      public User GetByID(Guid id)
      {
          return new User();
      }
  }
```

기반 클래스를 제네릭 타입으로 정의해서 공변성을 확보하고 서브클래스가 리턴 타입을 재정의할 수 있도록 구현한 예제

반 공변성

- 반 공변성은 공변의 개념과 유사하나 리턴타입이 아닌 매개변수로 사용되는 타입의 처리와 관련이 있다

- 클래스 계층에서는 상속관계가 역전된다는 것을 예상할 수 있다.

```
public delegate T Covariant<in T>();

public static void SampleEntity(Entity entity)
{
}

public static void SampleUser(User user)
{
}

public void Test()
{

    Covariant<Entity> entity = SampleEntity;
    Covariant<User > user = SampleUser;

    user = entity ;
    user (new User);
}
```

반 공변성

```
public interface IEqualityComparer<in TEntity> where
TEntity : Entity
   {
      bool Equals(TEntity left, TEntity right);
   }

   // . . .
   public class EntityEqualityComparer :
IEqualityComparer<Entity>
   {
      public bool Equals(Entity left, Entity right)
      {
         return left.ID == right.ID;
      }
   }
```

```
[Test]
public void UserCanBeComparedWithEntityComparer()
{
   SubtypeCovariance.IEqualityComparer<User> entityComparer =
new EntityEqualityComparer();
   var user1 = new User();
   var user2 = new User();
   entityComparer.Equals(user1, user2).Should().BeFalse();
}
```

반 공변성은 클래스 계층 구조를 뒤집어 구체화된 비교 클래스가 필요한 곳에 보다 일반화된 비교 클래스를 사용할 수 있게 한다.

불변성

- 앞에서 설명한 데이터 불변성과는 다른 개념이다
- 타입이 가변적이지 않다면 클래스 계층내의 어떤 타입도 사용할 수 없다.
    - ex) IDictionary<,> 같은 타입만 사용가능
- C#은 메서드 매개변수와 리턴타입에 대해서는 불변성을 유지한다. (상속)
- 오로지 제네릭을 사용할 때에만 타입에 기반해서 가변성을 가미할 수 있다.

리스코프 타입 시스템 규칙

- LSP원칙은 가변성과 직접적으로 관련이 있는 두 가지 규칙을 정의하고 있다.
    - 서브타입의 메서드 인수는 반 공변성을 지원해야 한다.
    - 서브타입의 리턴 타입은 반드시 공변성을 지원해야 한다.
    - 새로운 예외를 발생시키지 않는다.- (가변성과 무관한 독자적인 규칙)

리스코프 타입 시스템 규칙

- 새로운 예외를 발생 시키지 않는다.
    - 클라이언트는 실제로 구현한 클래스에 대해서 어떤 정보도 알아서는 안된다.
    - 만일 새로운 예외가 원래의 예외에서 파생된 것이 아니라 새롭게 정의된 것이라면 클라이언트가 직접 참조해야만 캐치 및 처리가 가능하다.
    - 또한 새로운 예외 타입을 생성할 때마다 클라이언트도 함께 수정해야 한다.

마치며

- LSP 원칙은 클래스 계층 구조의 각 서브클래스들이 사전 조건을 강화하거나 사후 조건을 완화할 수 없도록 규정하고 있다.
- LSP는 서브타입의 가변성에 대한 규칙도 제안하고 있다.
  - 리턴 타입 공변, 매개변수 타입 반공변, 새로운 예외 타입 추가 금지
- LSP를 위반하는 코드는 모두 기술 부채로 취급하여야 하며, 가장 우선적으로 처리해야 한다.