3

상속

상속은 객체지향적 프로그래밍의 꽃이라 불린다. 이장에서 필자는 상속에 대한 설명과 함께 다형성에 대하여 다루고자 한다.

상속이란

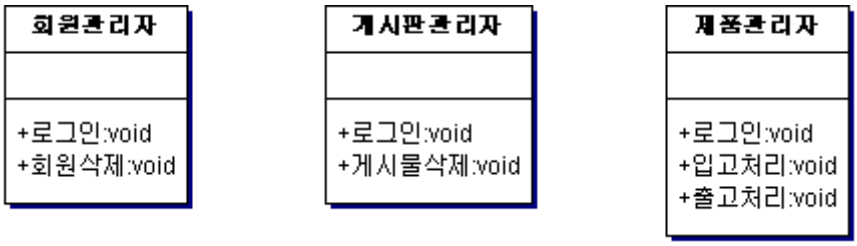
상속이란 여러 클래스가 가지고 있는 공통점을 모아서 표본 클래스를 만들어 내고, 해당 표본 클래스를 그대로 복사(유전)해 와서 중복되는 코드를 여러 번 구현하지 않고 재사용할 수 있는 클래스의 핵심기술이다.

이때 표본이 되는 클래스를 상위 클래스 또는 부모 클래스라 부르고, 표본 클래스에서 유전 받아서 새로 생성한 클래스를 하위 클래스 또는 자식 클래스라고 부른다.

자식 클래스는 상속받은 이후 새로운 메소드나 속성을 자유롭게 추가할 수 있다. 또한, 상속받아온 메소드를 새롭게 재정의할 수도 있다.

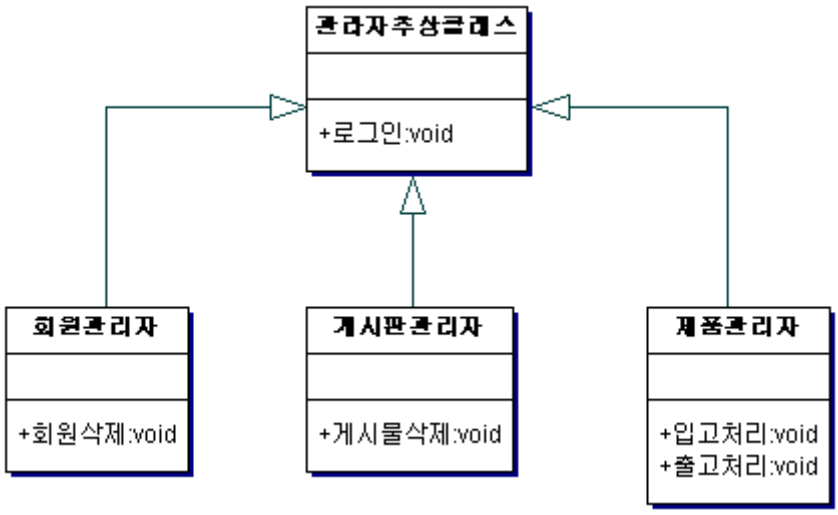
클래스의 정규화

어떤 시스템을 개발하고 구축하기 위하여 기능분석과 구조분석을 통하여 클래스의 종류에 대한 정의가 1차 종료되었다고 가정하겠다. [그림 1]은 발견된 관리자 클래스들이다.



[그림 1] 관리자 클래스

정규화의 기본원칙은 중복요소를 제거 하는데 있다. 여기서는 "로그인"이라는 메소드가 중복되고 있으므로 [그림 2]처럼 로그인 메소드를 상위 클래스를 생성한 후, 상위 클래스로부터 상속을 받아서 클래스 다이어그램을 구성해보겠다.



[그림 2] 클래스의 정규화

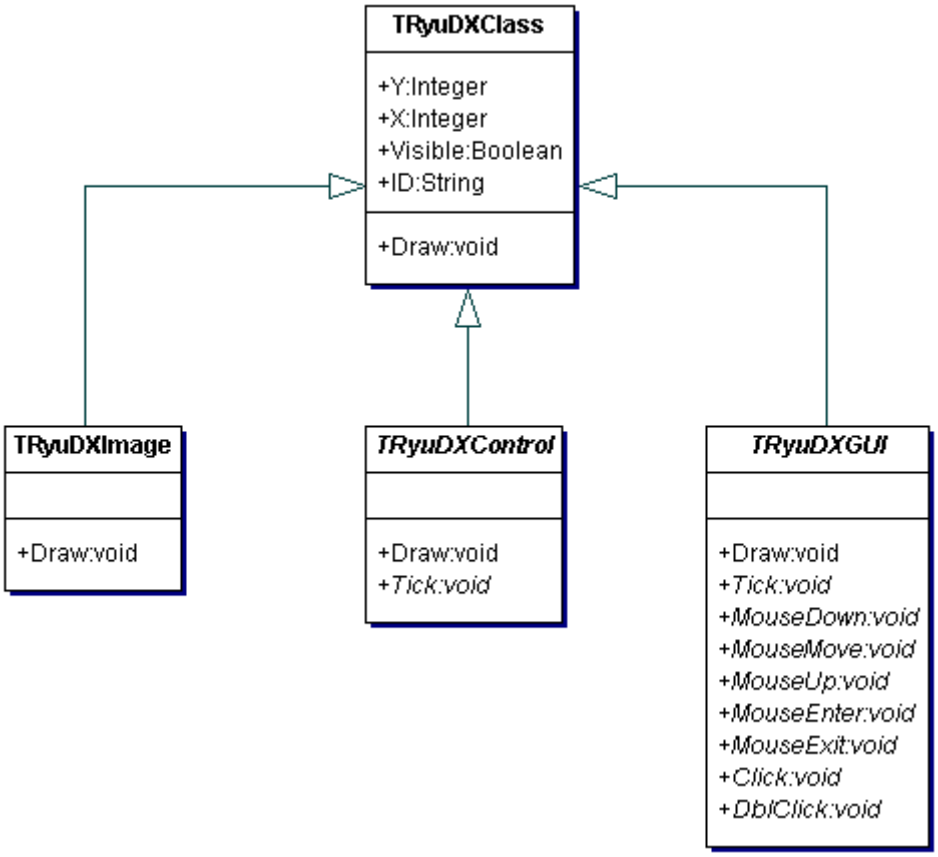
[그림 2]를 통해서 우리가 알 수 있는 것은 “관리자추상클래스”에서 정의된 로그인 메소드를 “회원관리자”, “게시판관리자” 그리고 “제품관리자” 등에서는 정의하지 않아도 사용할 수 있다는 것이다. 이로써 같은 동작을 하는 메소드를 여러 곳에서 구현할 필요가 없다. 또한 로그인 메소드에 대한 요구사항이 변경되더라도 각각의 클래스 마다 수정할 필요가 없다. 단순히 “관리자추상클래스”에서 로그인 메소드를 변경하면 하위 클래스에 그대로 적용될 것이기 때문이다.

돌연변이 처리 - Virtual and Override

인간은 부모로부터 대부분의 유전자를 물려받지만(상속) 또한 상당한 양의 유전자는 정자와 난자가 만나 수정하는 순간부터 많이 변질된다. 엄밀한 의미에서는 우리 모두가 돌연변이 인 셈이다. 그러니 가끔 이런 말을 들을 때 우리는 당연하게 생각해야 한다.

"너는 누구를 닮아서 그러니?"

이제 클래스로 넘어와서 생각한다면 우리는 부모로부터 상속받아온 메소드에 추가적인 작업하거나 다소 변형을 가해 야할 필요성을 느낄 때가 있다. 이때 우리는 Override라는 예약어를 통해 기존의 메소드를 상속받아오면서 변형하여 사용할 수가 있다.



[그림 2] Direct-X 게임 컨트롤 Class Diagram

[그림 2]는 필자가 보드게임을 만들 때 구현한 클래스들 중 일부를 표시한 것이다. Draw 메소드는 TRyuDXClass에서부터 상속을 받아서 하위 클래스들이 사용하도록 되어있다. TRyuDXClass.Draw는 virtual로 선언된 메소드이며 화면에 자기 자신을 그리는 역할을 담당하

고 있다.

TRyuDXImage의 경우에는 이 메소드를 상속만 받으면 자기 자신을 화면에 표시할 수 있기 때문에 추가작업이 필요 없는 경우이다. 하지만, TRyuDXControl과 TRyuDXGUI의 경우에는 상황이 다르다. 자기 자신을 그리는 것 이외에도 프로세스가 진행될 때, 그때 그때 상황에 맞춰서 내부에 구현된 동작을 할 필요가 있다. 예를 들면 미사일 같은 경우에는 시간이 흐를수록 미사일은 앞으로 전진해야 한다.

이를 처리하기 위해서 Direct-X 화면이 그려지는 매 프레임마다 Tick이라는 함수를 호출하고 Tick은 주기적으로 실행해야 할 코드를 실행하면서 마치 모든 클래스들이 병렬 처리되는 것처럼 구현하도록 한 것이다.

여하튼 여기서 여러분들이 주목해야 할 것은 그 로직 자체가 아니고, Draw 메소드는 TRyuDXImage와 TRyuDXControl이 서로 다르다는 것이다. 정확하게 TRyuDXControl은 부모에게서 상속받은 메소드를 변형(돌연변이 발생)할 필요가 있다는 것이다.

```
 1 : unit RyuDXClasses;
 2 :
 3 : interface
 4 :
 5 : type
 6 :   TRyuDXClass = class(TObject)
 7 :   private
 8 :   public
 9 :     Procedure Draw; virtual;
10 :   end;
11 :
12 :   TRyuDXImage = class(TRyuDXClass)
13 :   private
14 :   public
15 :   end;
16 :
17 :   TRyuDXControl = class(TRyuDXClass)
18 :   private
19 :   public
20 :     Procedure Draw; override;
```

```
21 :    Procedure Tick; virtual; abstract;
22 :  end;
23 :
24 : implementation
25 :
26 : { TRyuDXClass }
27 :
28 : procedure TRyuDXClass.Draw;
29 : begin
30 :   { TODO : 자기 자신을 그리기 }
31 : end;
32 :
33 : { TRyuDXControl }
34 :
35 : procedure TRyuDXControl.Draw;
36 : begin
37 :   inherited;
38 :
39 :   Tick;
40 : end;
41 :
42 : end.
```

37: 라인에서 상속받았던 Draw를 inherited 예약어를 통해 실행하게 된다.

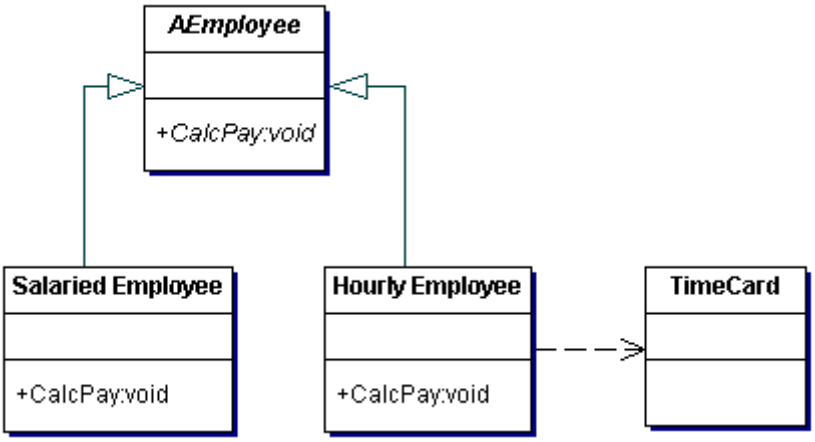
39: 라인에서 상속받은 메소드를 호출한 이후에 Tick 메소드를 실행하도록 되어 있다. 즉, 기존의 메소드를 상속받고 작업을 추가했다는 뜻이다. 만약 38: 라인이 지워진다면 기존의 메소드는 무시하고 전혀 새로운 동작을 하게 되는 것이다.

이처럼 override는 상속받은 메소드를 변형하고자 할 때 사용하게 된다. 이때 상속받을 상위 클래스의 메소드는 virtual 또는 dynamic으로 선언되어 있어야 한다.

TRyuDXImage의 경우에는 Draw 메소드를 전혀 구현해주지 않았다는 점도 유의하여야 한다. 이미 설명한 것처럼 클래스 정규화를 통하여 공통요소를 상위 클래스에서 선언한 후 재사용하고 있다.

그때 그때 달라요 - 다형성

다형성의 경우는 방금 전 설명한 돌연변이 클래스에 연장선으로 생각하면 된다. 다만, 여기서는 대체로 유사한 메소드지만 조금 들린 것을 상속받아오면서 적용하여 해결하고자 하는 것이 목적이 아니다. 다형성을 활용하는 최대의 목적은 **요구사항이 변하더라도 소스코드 변경의 쇼크는 최소화 하겠다**는 것이다.



[그림 3] 월급계산을 위한 클래스들

다형성은 같은 함수 이름을 호출해도 전혀 다른 동작을 하게 만드는 것을 뜻한다. 예를 들어 [그림 3]의 경우 AEmployee라는 상위클래스에는 CalcPay()라는 추상 메소드가 있다. 이를 상속받은 "Salaried Employee"는 월급을 받는 정직원이라 생각하고, "Hourly Employee"는 시급으로 받는 직원이라고 생각하겠다.

월급을 받는 직원의 경우에는 매달 정해진 월급만큼 지급하면 되지만, 시급제 직원의 경우에는 근무한 시간이 담긴 TimeCard를 참조해야만 지급될 임금을 계산할 수 있다. 만약 다형성을 사용하지 않고 프로그램을 짠다면 다음과 같은 방법이 될 것이다. 아래 제시하는 코드는 개념적인 설명일 뿐, 실제 동작하는 소스는 아니다.

```
1 : procedure TForm1.Button1Click(Sender: TObject);
2 : var
3 :   Loop : Integer;
4 : begin
5 :   For Loop:= 0 to Employees.Count-1 do
6 :     If Emplyoees.Persons[Loop].SalaryType = Monthly then
7 :       Emplyoees.Persons[Loop].MonthlyCalc
```

```
 8 :    Else
 9 :      Emplyoees.Persons[Loop].HourlyCalc;
10 : end;
```

이것을 [그림 3]의 클래스 구성도처럼 구축된 경우, 즉 다형성을 사용하는 경우를 살펴보면 다음과 같다.

```
1 : procedure TForm1.Button1Click(Sender: TObject);
2 : var
3 :  Loop : Integer;
4 : begin
5 :  For Loop:= 0 to Employees.Count-1 do Emplyoees.Persons[Loop].CalcPay;
6 : end;
```

CalcPay라는 동일한 메소드를 호출하였지만, Emplyoees.Persons[Loop]이 "Salaried Employee"로 선언된 오브젝트냐 또는 "Hourly Employee"로 선언된 오브젝트냐에 따라서 각자의 규칙에 맞는 서로 다른 계산을 하게 된다.

위 두 소스를 비교해보면 다형성을 쓰지 않았을 경우에는 또 다른 직원의 종류가 생겼을 때 "if"문의 조건을 추가하는 변경이 필요하다. 또한, 변경해야 하는 곳이 여러 곳에 분포되어 있다면 그 수정은 상당히 어려울 것이다.

하지만, 아래 다형성을 사용한 소스의 경우에는 AEmloyee 객체에서 또 다른 객체를 상속받아서 사원의 목록을 관리하는 Employees 객체에 생성된 사원의 객체를 삽입해주기만 된다. 소스코드의 변화는 전혀 없다.

이처럼 다형성을 효과적으로 사용하면 변화가 예측되는 곳에 변화에 대한 융통성을 극대화하여 다양한 변화가 발생하더라도 소스코드를 고쳐야 하는 가능성을 획기적으로 줄일 수 있는 것이다.