Class Diagram

**개요**

- 시스템의 논리적인 구조(클래스)를 표현한다.

- 객체지향 개발에서 가장 공통적으로 많이 사용된다.

- class diagram : 시스탬 정적 설계도

- Active Class Diagram : 시스템의 프로세스도

**특징**

- 시스템의 요구사항에 표현된 작업에 대한 책임을 분할한다.

- 모델은 점점 증가되므로 관련된 클래스들끼리 패키지화 한다.

- 클래스를 너무 작게 쪼개거나 기능을 너무 많이 포함하면 복잡해지므로 적절히 구현해야 한다.

**Class**

- 객체의 속성(Attribute)와 행동(Operation)을 포함한다.

- 모든 Class는 유일한 이름을 갖는다.

- 단순명(Simple Name) : class 이름만 표현한다.

- 경로명(Path Name) : class 의 패키지명(namespace)를 포함하여 표현한다.



왼쪽이 단순명이고 오른쪽이 경로를 포함한 이름이다.

경로명의 경우 언어에 따라 표현을 달리하면 된다.

예를 들어

java : java.lang.String

c++ : std::String

c# : System.String

추상클래스의 경우 클래스명을 *이텔릭체*로 표기하면 된다.



**속성(Attribute)**

- 의미있는 명사형으로 표현

- 문법

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| <<stereotype>> | visibility | name | : | type | [multiplicity] | = | value |
|  | 접근제어 | 속성명 |  | 속성타입 | 배열 개수 |  | 기본값 |

**행동(Operation, method)**

- 의미있는 동사형 표현

- class 가 행할 수 있는 동작을 기술한다.

- 문법

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| <<stereotype>> | visibility | name | ( param1 : type [multiplictiy] = value, param2... ) | : | returntype |
|  | 접근제어 | 행동 | 인자( 위의 속성과 비슷하다.) |  | 반환 타입 |

- 추상 행동(메소드)의 경우는 *이텔릭체*로 표현한다.

**속성, 행동 공통 사항**

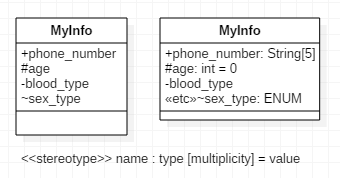
- static(정적) 속성(attribute)는 밑줄로 표현한다.

- 접근제어 표현법

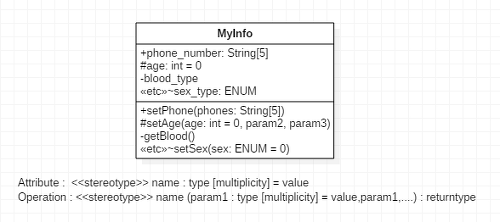
|  |  |
| --- | --- |
| visibility(접근제어) | |
| + | public |
| - | private |
| # | protection |
| ~ | package |

- stereotype : "<< contents >>" 이런식으로 표현한다.

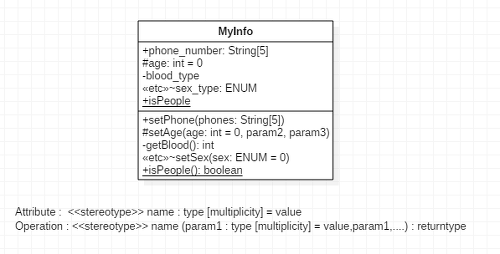
새로운 어휘를 표현하는 방법이다. UML의 모델요소는 한정되기 때문에 추가적인 의미, 목적, 프로그래밍 언어에 따른 UML에서 표현 못하는 요소를 추가할수 있다.



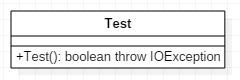
Class 표현 예1



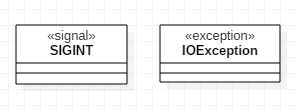
Class 표현 예2



class 표현 예3



추가적으로 Exception을 던지는 행동(메소드)의 경우는 throw Exception과 같은 표현을 통해 던지는 예외를 표현할 수 있다.



stereotype의 경우는 위의 예처럼 UML에서 지원하지 않는 signal이나 exception 혹은 추가적인 표현을 할 때 유용하다.

**Nested Class(중첩 클래스)**

- 클래스내부에 선언된 또다른 클래스를 말한다.



inner class 는 outer class 내부에 존재하므로 namespace까지 표현한 경로명으로 outer.inner이다.

**Active Class**

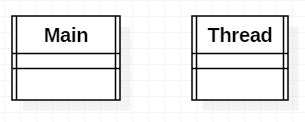
- 독자적인 제어흐름을 갖고 프로그램의 실행을 주도하는 클래스이다.(예를 들어 Main Class, Thread class)

- 반대의 클래스를 Passive Class라고 한다.

- Thread를 소유하거나 주소 공간을 소유한다.

- 프로그램 실행 혹은 코드 실행시 이를 시작하거나 흐름을 제어 할 수 있다.

- 변수를 조작하거나 프로그램의 행동을 변화시킬 수 있다.



기본 클래스 형태에 왼쪽,오른쪽 양 옆에 | | 로 표현한다.

**Interface(인터페이스)**

- Class 또는 Component 가 외부에 제공하는 기능(행동, 메소드)만을 목록화 한 것이다.

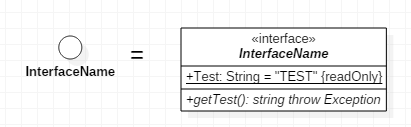
- Interface는 구조와 구현을 갖지 않는다.( 실제 동작하지 않는다. 이런 동작을 있다는 것을 알려줄 뿐이다.)

- Interface는 속성을 갖지 않고 구현된 동작(메소드)도 갖지 않는다.

- 오직 어떤 기능을 수행할 수 있다고 하는 기능 목록만 제공한다.

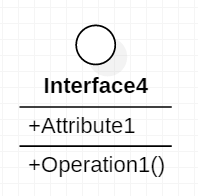
- 이 기능의 실제 구현은 ㅏㅎ위클래스에서 구현해야한다.

인터페이스는 속성을 갖지 않고 구현된 행동이 없이 행동이 선언만 된 추상클래스(abstract class)이다.



위의 Test란 변수는 상수를 표현한 것이다. interface는 변수는 갖지 못하지만 상수값(변수가 아니기 때문에)은 가질수 있다.

- interface 는 원형으로 표현하되 클래스로 표현도 가능하다. 클래스로 표현시 stereotype 인 <<Interface>>를 추가하여 해당 클래스가 interface라는 것을 알려주어야 한다.



interface 표현을 원형으로 하고 이런식으로 표현도 가능하다.

**Relation Ship**

**개요**

- 모델 요소간의 논리적 또는 물리적인 연결을 표현한다.

- 여러 객체간 관계를 의미한다.

**종류**

- Dependency (의존)

- Association (연관)

   - Aggregation (집합 연관)

   - Composition (복합 연관)

-  Generalization (일반화)

- Realization (실체화)

**1. Dependency Relationship (의존관계)**

- 한 클래스의 변화가 이것을 사용하는 다른 클래스에 영향을 미치는 관계(역은 성립하지 않는다.)

- using(사용) 관계를 나타낸다.

- 구현시 의존 클래스를 사용하는 클래스는 이를 찹조하고 있는 인스턴스 변수를 유지하지 않는다.(의존 클래스를 잠시 사용할 뿐이다.)

- 표현은 점선에 화살표이다. --------->

적용 예시

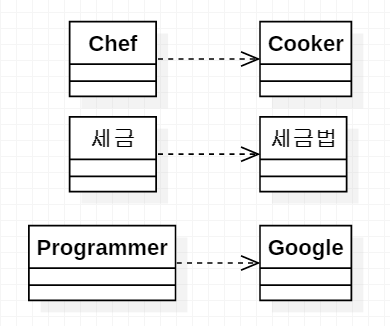
- 한 클래스의 메소드가 다른 클래스를 인자로 받아 메소드내에서 사용

- 한 클래스의 메소드 내에서 다른 클래스를 생성하고 생성된 클래스의 메소드를 사용한다.( 메소드 내에서만 사용하고 메소드 반환시 의존 클래스 인스턴스도 함께 사라진다. -> 즉 로컬변수로만 사용 )

- 한 클래스가 의존 클래스의 객체를 반환 ( 의존 클래스를 생성 반환만 한다. 인스턴스를 유지하지 않는다.)

- 의존 클래스의 정적 메소드 사용

**- 중요**) 즉 의존 클래스의 인스턴스 변수를 유지하지 않는다.



예를 들어 요리사는 요리도구에 의존한다. 요리도구가 바뀌면 요리방법도 바뀐다. 또는 세금의 경우 세금법에 의존한다. 세금법이 바뀌면 세금 계산 방법도 달라진다.

**2. Generalization (일반화 관계)**

- 여러 클래스가 가진 공통적인 특징을 추출하여 공통적인 클래스로 일반화시키는 것을 의미한다.

- 'is a' 관계 (예를 들어 인간은 동물이다.)

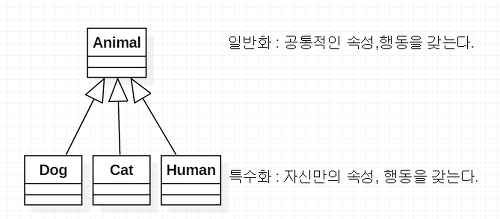
- 객체 지향 언어에서의 상속관계

  = 자식은 부모의 속성과 행동을 공유한다.

  = 자식은 부모가 갖지 않는 속성과 행동을 추가할 수 있다.

  = 부모가 사용되는 곳은 어떤 자식으로든 대체 가능하다. 역은 성립하지 않는다.

## 자세한 상속관계에 대한 지식은 객체지향언어의 상속편을 보아라.



객체지향언어에서는 animal을 부모클래스, dog, cat, human을 자식클래스라고 한다.

자식 클래스는 부모클래스의 모든 속성과 행동을 상속한다.

위의 그림을 보면

- 개, 고양이, 인간의 공통적인 속성,행동인 나이, 숨쉬다 등의 공통적인 부분으로 동물이란 클래스로 일반화했다.

- 인간의 경우 동물이란 공통직인 것에 "자동차를 운전한다."라는 행동이 추가 될 수 있다. 이를 특수화라고 한다.

- 동물의 공통적인 행동인 숨쉬다라는 행동을 개, 고양이, 인간 어떤 것으로도 대체해도 숨쉬다라는 행위를 할 수 있지만 인간의 행동중 하나인 "자동차를 운전한다"는 인간외 다른 동물로 대체 불가능하다.( 부모가 사용되는 곳은 어떤 자식으로든 대체가능, 그러나 역은 불가능)

**3. Association ( 연관 관계 )**

- 클래스로부터 생성된 인스턴스들 간의 관계를 표현한다.

  = 인스턴스란 클래스를 실제로 사용할 수 있는 하나의 개체로 만드는 것이다.

  = 프로그래밍 언어에서는 new 연산자를 이용하여 클래스에 메모리를 할당하고 프로그램내에서 사용할 수 있는 형태로 만드는 작업이다.

- Dependyncy와 Generalization 관계는 단순히 클래스들간의 관계를 나타낸다. 이 관계들은 인스턴스를 만들고 인스턴스에 대한 참조를 유지하지 않는다는 것이 특징이다.

- Association 관계는 Classifier 로부터 생성된 인스턴스 사이의 관계를 나타낸다. 생성된 인스턴스에 대한 참조(인스턴스 변수)를 유지한다는 것이 특징이다.

- 상대방의 인스턴스를 가리키는 attribute를 가진다.

- 참조할 수 있는 attribute는 UML 상에서 표현하지 않는다. 표현하고자 할 때는 role name을 이용한다.

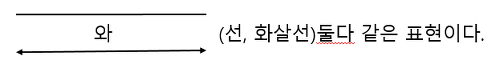
방향 종류

- Bidirectional Association (양방향) : 서로간 상대의 인스턴스를 가리키는 attribute(인스턴스 변수)를 갖는다.

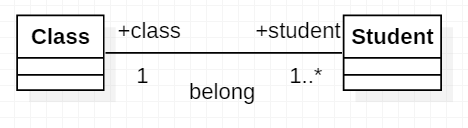
- Unidirectional Association (단방향) : 한쪽 클래스만이 다른 상대의 인스턴스를 가리키는 attribute(인스턴스 변수)를 갖는다.

classifier : interface, class, component와 같이 인스턴스화 될 수 있는 요소

표현방법



Association 예)



위의 예제는 학생과 교실과의 관계이다. 학생은 하나의 교실에 포함되고 교실은 한명 이상의 학생을 포함한다. 그래서 class 쪽은 1, Student쪽은 1..\* 이다. 이를 Multiplicity라 한다. +student와 +class 는 서로를 가르키는 인스턴스 변수를 말한다. 이 때 +는 public 접근제어를 뜻한다. 이는 위 클래스 다이어그램에서 언급했다.

Class(교실) 클래스 내부에는 Student클래스를 참조하는 student란 인스턴스 변수가 존재하고, Student(학생) 클래스 내부에는 class 란 인스턴스 변수가 존재한다.

#multiplicity와 인스턴스 변수 이름의 위치를 잘 써야한다. 위치를 반대로 쓸때가 있다.

|  |  |
| --- | --- |
| **Multiplicity** | |
| **1** | **1개** |
| **0..1** | **0또는 1** |
| **\*** | **다수(0포함)** |
| **1..\*** | **0을 제외한 다수. 최소 하나의 관계는 있어야 한다.** |
| **n..m** | **최소 n개 최대 m개의 관계** |

**Aggregation ( 집합 연관 관계 )**

- association 관계 중 특별한 관계를 나타낸다.

- 전체(whole)와 부분(part)을 나타내는 모델 요소이다.

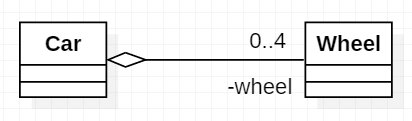
- 'has a' 관계이다.

- 전체와 부분은 서로 독립적인 관계이다. 즉 생명주기가 동일하지 않다.

- 부분(part) 인스턴스는 다른 클래스와 공유 될 수 있다.(생명 주기가 다르므로)

- 전체(whole)에 속이 빈 다이아몬드이다.

- 단방향 연관 관계이다.(whole은 part을 알지만 part는 whole을 모른다.)



차(Car)는 4개의 바퀴(Wheel)를 갖는다. 차를 폐차한다고 바퀴까지 없애지는 않는다. 바퀴는 바퀴 독립적으로 다른 차에 장착할수 있다.

**Composition (복합 연관 관계)**

- 전체(Whole)와 부분(Part)을 나타내는 모델요소

- 전체를 나타내는 클래스와 이를 이루고 있는 부분 클래스 관계

- Aggregation 과는 다르게 강한 연관 관계이다.

- 전체와 부분이 동시에 생성되고 동시 소멸된다. 즉 생명주기가 동일하다.

- 이 관계에서는 전체가 생성될때 전체가 부분을 생성시키는 형태이다. 부분은 독립적으로 혼자 생성되지 않는다.

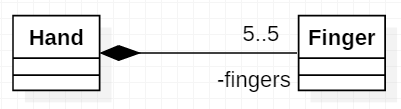
- 부분(Part)는 공유되지 않는다. 왜냐면 전체와 부분이 생명주기가 같다. 공유되면 전체가 살라질 때 부분도 사라져야 하는데 공유된다면 문제가 발생하기 때문이다.

예) 자바의 경우 인스턴스 참조 갯수를 보고 갈비지 콜렉터가 인스턴스를 해제할때 공유되면 해제되지 못한다.

C++의 경우에도 공유된 인스턴스가 다른 쪽에서 해제되면 공유된 인스턴스를 사용하는 다른 곳에서 잘못된 연산을 하게된다.

- 전체(whole)에 속이 꽉찬 다이아몬드이다.

- 단방향 관계이다.



손은 손가락 5개를 갖는다. 손이 사라지면 손가락도 사라지는 것이다.

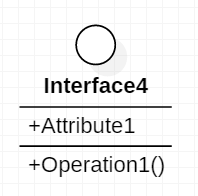
*근데 의문이 들수 있다. 손이 사라져도 손가락 따로 다른 사람 손에 이식 시킬수 있지 않나? 그리고 손이 꼭 손가락 5개를 가져야 하나? 이건 프로그램을 설계하는 사람이 어떻게 설계하느냐에 따르는 것이다. 필자는 손은 꼭 손가락 5개, 그리고 손이 사라지면 손가락도 사라진다라고 설계했기때문에 위와 같은 관계가 나온것이다.*

즉 관계라는 것은 설계에 따르는 것이다.

**Realization (실체화, 실현화)**

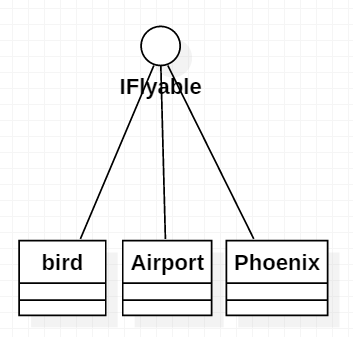
- 인터페이스와 실제 구현된 클래스간의 관계

- 인터페이스 UML 표현



- 전통적으로 인터페이스 명에는 접두에 'I'가 붙거나 접미에 able이 붙는다.

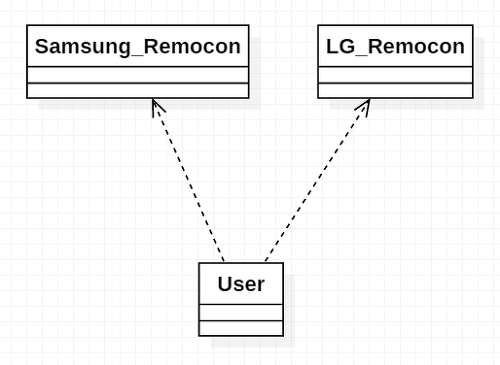
- 전통적으로 JAVA ~able, C# I~



인터페이스 확장

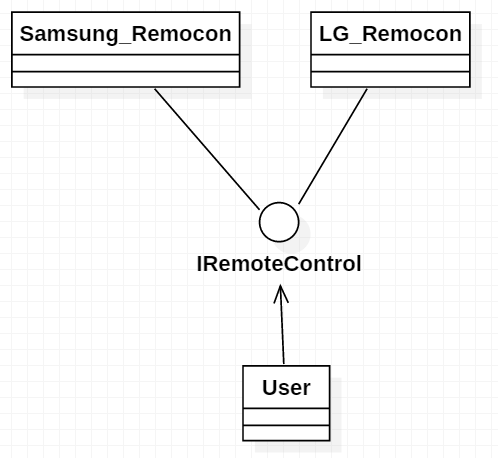
- 인터페이스는 컴포넌스(클르새ㅡ) 간의 결합력을 느슨하게 한다.(loose coupling)

- 인터페이스는 프로그램의 수정없이 쉽게 소프트웨어를 확장 할 수 있다.



위 그림은 인터페이스 없이 User라는 클래스가 Lg, Samsung 리모콘 클래스를 직접 사용하는 것이다. 프로그램 코드도 Samsung 이냐 Lg냐에 따라 코드를 따로 작성해야 한다. 소니 리모콘 추가시에 소니 리모콘 클래스 뿐 아니라 User가 이를 조작하는 코드가 추가되어야한다.

리모콘의 공통 기능을 인터페이스화 시키면 프로그램 코드가 확연히 줄어들고 또한 Samsung, Lg가 아닌 소니, 샤오미등 다른 리모콘 클래스를 만들어야 할때도 쉽게 이를 만들수 있다.



위 그림과 같이 User 클래스는 IRemoteControl이란 인터페이스 인스턴스를 통해 Samsung, Lg 리모콘의 기능을 동일하게 사용할 수 있다. 만약 소니 리모콘을 만들어야 한다고 해도 IRemoteControl를 실체화 시킨 클래스를 그냥 만들기만 하면 되고 User 클래스의 변경없이 소니 리모콘을 사용할 수 있다.

이 부분에 대해서는 코드가 있었으면 훨씬 이해가 쉬웠을 텐데 코드는 나중에 시간이 있다면 추가하겠다..( 거의 추가 안할것 같다.;;)

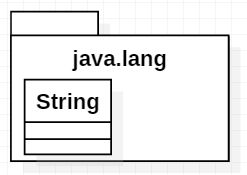
**Package**

개요

- UML 모델 요소를 조직화 하고 계층화 한다.

- Package는 각각의 Name space를 갖는다.

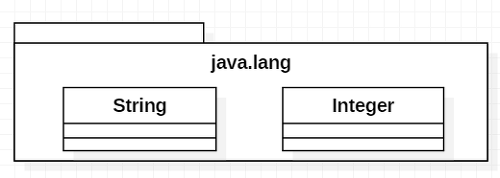
  = Package만 다르면 동일한 이름의 컴포넌트(클래스)를 가질 수 있다.



package는 위와 같이 그린다.



위와 같이 Name space가 같은 두 개의 클래스를 아래와 같이 Package로 묶어서 표현할 수 있다.



**클래스 다이어그램 주의점**

- 일반화는 'is a' 관계만 사용한다.

- 일반화관계를 균형있게 유지한다.(세분화 주의)

- 가능하면 interaction이 교차되지 않도록 주의한다.

- 이해하기 쉬울 정도의 관계만 표현한다. (너무 자세한 관계표현은 복잡해 진다.)

- 관련된 클래스는 가까이 배치한다.