Software Engineering

Lecture 09: 설계 (Part 2)
UML 모델링 – 정적 모델링

Professor: Jung Uk Kim

ju.kim@khu.ac.kr

Computer Science and Engineering, Kyung Hee University

Visual AI Lab.

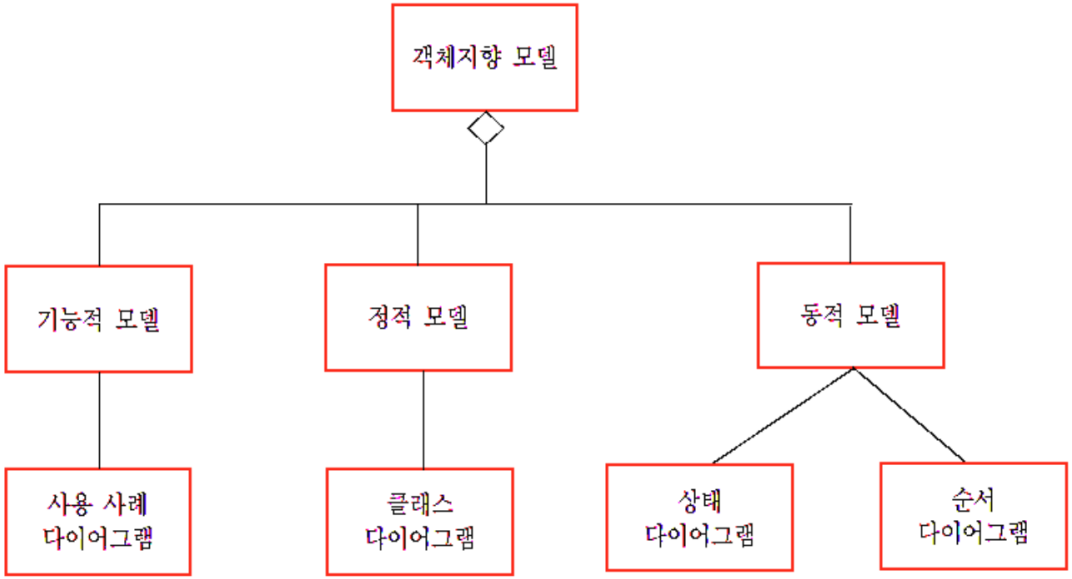
KYUNG HEE UNIVERSITY

UML 모델링

- UML (Unified Modeling Language)
  - 객체 지향 프로그램 설계를 표현하기 위해 사용하는 표기법
  - **설계단계 뿐만 아니라** 요구분석, 구현 단계에서도 사용가능
  - 개발자간 의사소통을 원활하게 이루어지게 하기 위하여 표준화한 모델링 언어 (실제 언어는 X)
  - 목적: 시스템을 가기화, 명세화, 문서화

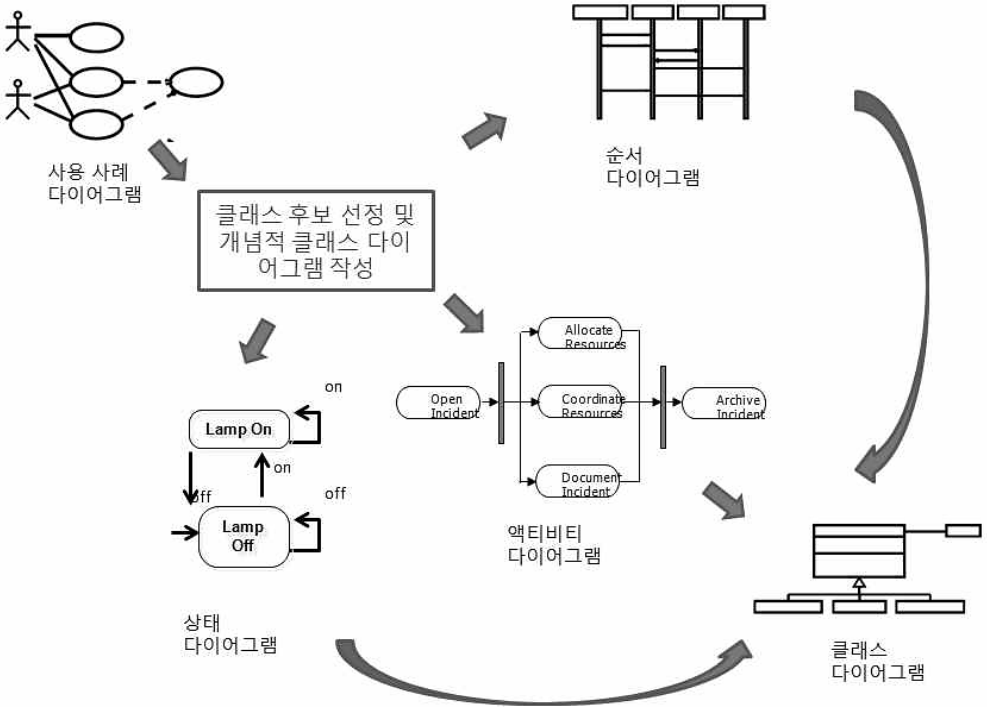
UML 모델링

- UML 모델링
  - 기능적 관점 (사용자 측면) - 주로 요구분석 단계
  - 정적 관점- 시간적인 개념 X, 시스템의 구조
  - 동적 관점 - 시간적인 개념 O, 시스템의 내부 동작



UML 모델링

- UML 모델링 과정 (유스케이스 다이어그램이 작성되었을 때)
  - (1) 요구를 유스케이스로 정리하고 **유스케이스 다이어그램** 작성
  - (2) 클래스 후보를 찾아내고 개념적인 객체 모형을 작성
  - (3) 유스케이스를 기초하여 **순서 다이어그램** 작성
  - (4) 클래스의 속성, 오퍼레이션, 클래스 사이의 관계를 찾아 객체 모형 완성
  - (5) **상태 다이어그램**이나 **액티비티 다이어그램**을 추가하여 UML 모델 완성

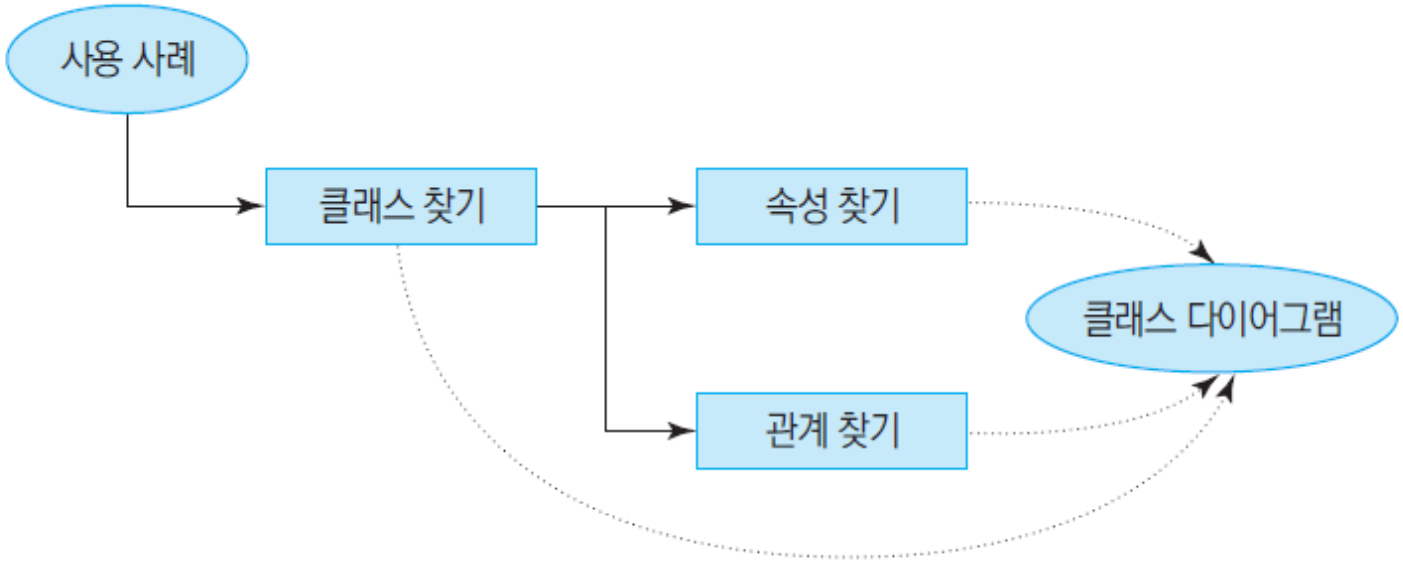


UML 모델링

- 객체지향 모델링
  - **정적 모델링**
    - 시스템의 구조와 정적인 관계를 표현하는 데 사용
    - 클래스, 객체, 데이터 구조와 이것들의 관계
    - 소프트웨어 요소에서 변하지 않는 논리적인 구조를 보여줌
    - 대표: **클래스 다이어그램 등**
  - **동적 모델링**
    - 객체들 간의 상호작용과 흐름을 시간순으로 표현
    - 예시: **상태 다이어그램, 순서 다이어그램, 액티비티 다이어그램 등**
- 절차지향 모델링(추가) - UML 모델링은 아님
  - 시스템의 처리 과정을 자료의 흐름에 중점을 두어 기술
    - 예시: **자료 흐름도 (DFD)**

정적 모델링

- 클래스 다이어그램
  - 객체지향 시스템에 존재하는 클래스, 클래스 안의 메소드, 서로 협력하거나 상속하는 클래스 사이의 **연결 관계를 나타내는 그림**
  - 요구를 만족하는 구조를 **클래스 관점에서 표현**
  - 과정
    - 클래스 및 속성 찾기
    - 관계 찾기
    - 클래스 다이어그램 그리기



클래스 다이어그램

- 클래스 이름, 속성, 오퍼레이션/메서드
  - 클래스 이름
  - 속성
    - Visibility: + public
      # protected
      - private
      ~ package (디폴트)
      / derived
    - Underline: 정적변수
  - 메서드
    - 메서드 이름(name: type): 리턴 타입

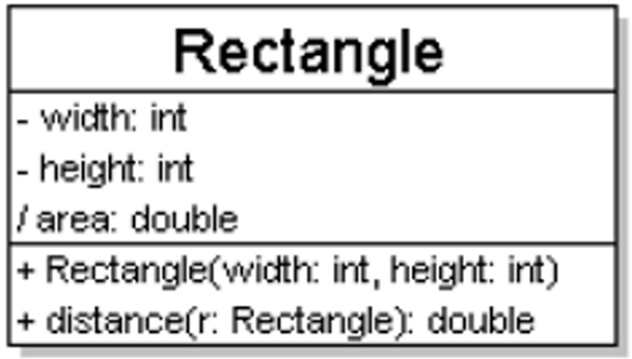
    참고) 리턴 타입이 void: 리턴 타입 생략

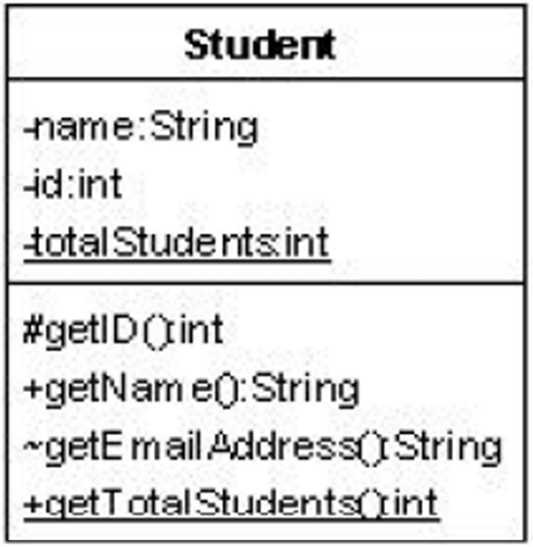
/: 다른 속성(width, height) 에 의해 유추될 수 있는 것

클래스

속성

오퍼레이션/메서드





(리마인드) 설계의 원리 - (4) 정보은닉

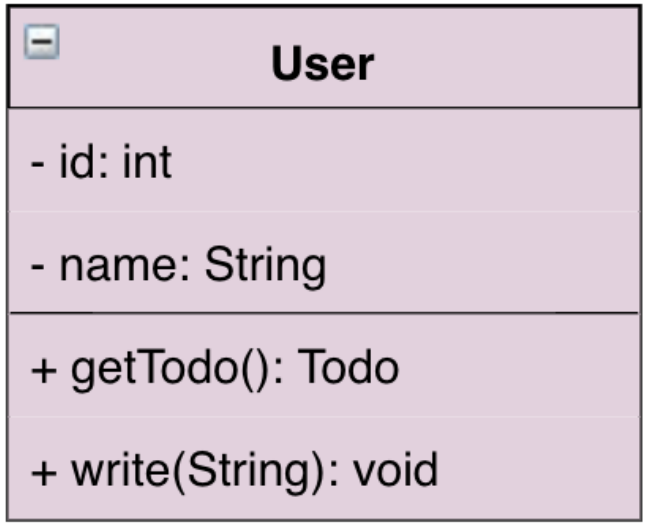
- **정보은닉 표기법 (Java, C++, UML)**
  - **공개(+, public):** 같은 시스템에 있는 모든 클래스가 접근할 수 있음
  - **은닉(-, private):** 같은 시스템 내의 다른 클래스가 직접 접근할 수 없고, 해당 클래스의 메서드를 통해서만 접근할 수 있음
  - **부분공개(#, protected):** 다른 클래스가 접근할 수 없고, 해당 클래스의 메서드와 클래스를 상속받은 하위 클래스만 접근 가능

	UML	C++	Java
공개	+	public	public
은닉	-	private	private
부분 공개	#	protected	protected

향후 UML 학습 예정

클래스 다이어그램

- 클래스 이름, 속성, 오퍼레이션/메서드 (예시)

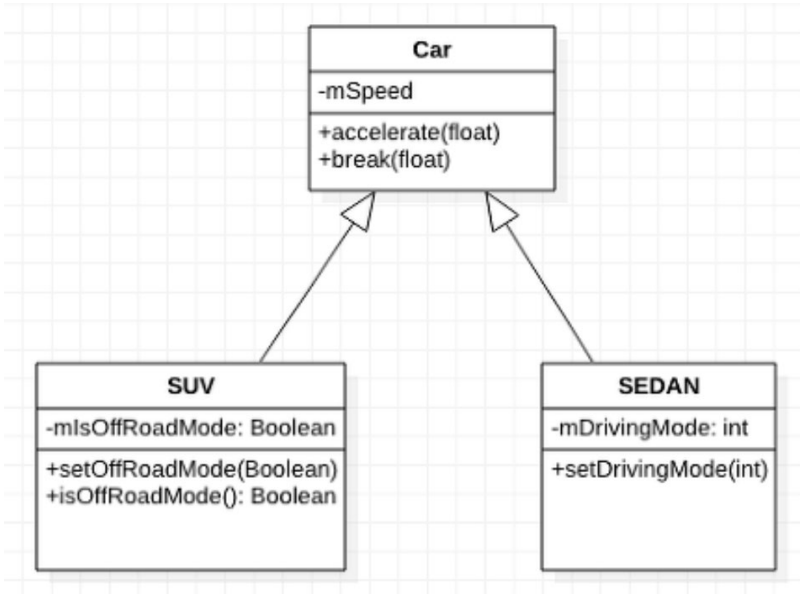


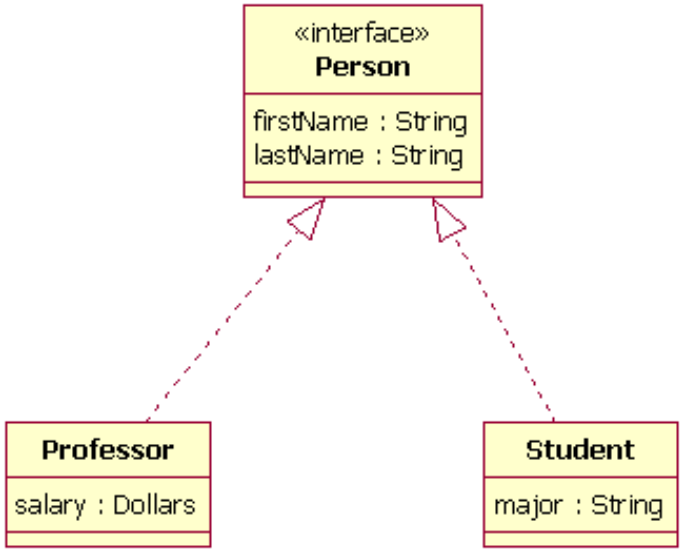

```java
public class User {
    private int id;
    private String name;

    public Todo getTodo() {
        // 투두리스트를 확인한다.
        return null;
    }

    public void write(String text) {
        // 투두리스트에 글을 적는다.
    }
}
```

실제 구현

클래스 사이의 관계

- **상속**
  - 일반화(generalization): 상속 관계
    - 한 클래스가 다른 클래스의 포함하는 상위 개념인 경우
    - 부모를 향한 화살표로 표시되는 하향 계층 관계
    - 클래스: 실선
    - 인터페이스: 점선

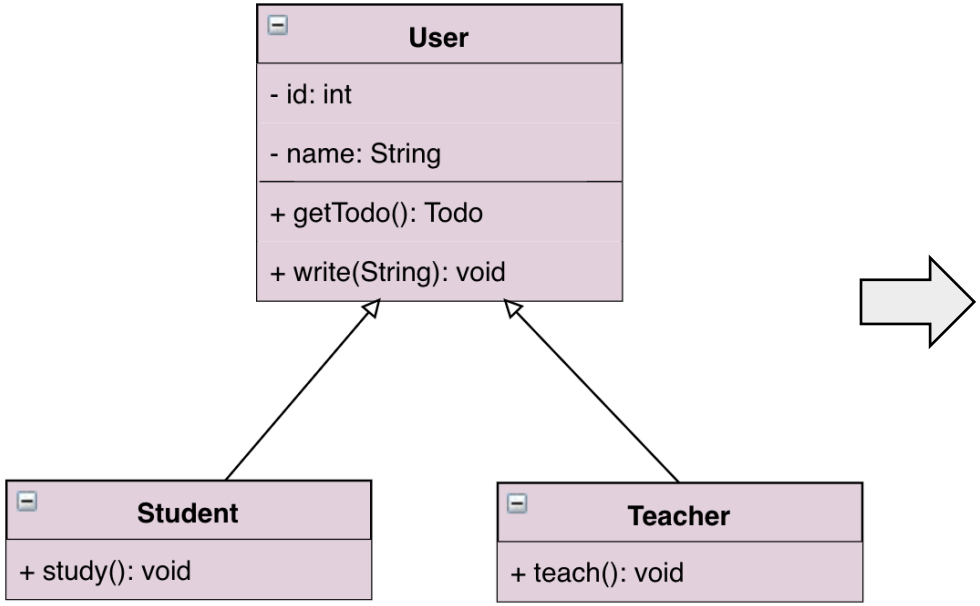




부모 클래스: 추상적인 개념, 삼각형 표시가 있는 쪽 (자식 클래스의 공통적인 속성을 제공)
자식 클래스: 추상적인 개념을 물려받은 구체적인 개념

클래스 사이의 관계

- **상속 (예시)**




```java
public class User {
    private int id;
    private String name;

    public Todo getTodo() {
        // 투두리스트를 확인한다.
        return null;
    }

    public void write(String text) {
        // 투두리스트에 글을 적는다.
    }
}
```

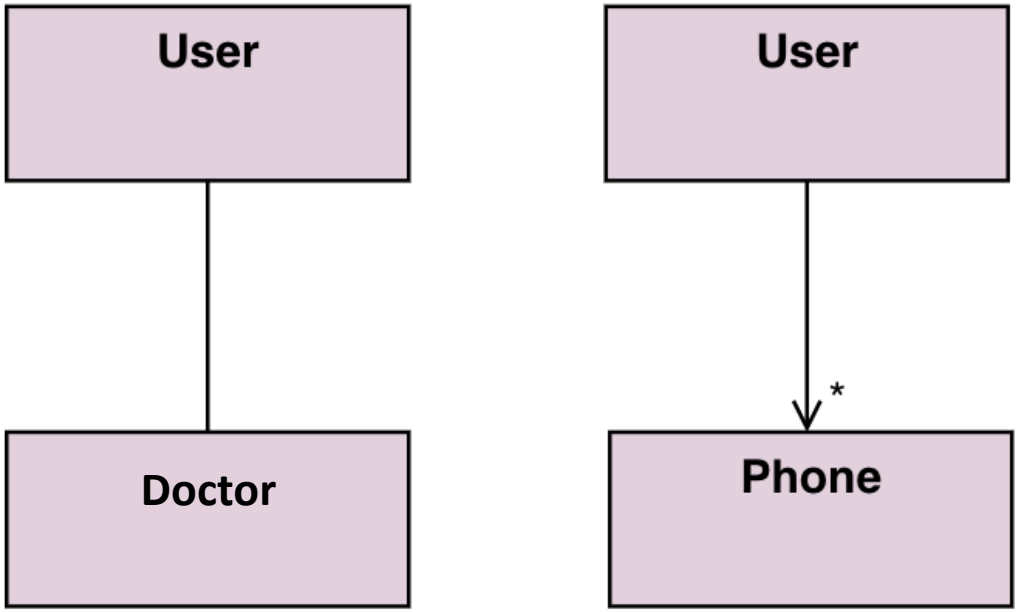
```java
public class Student extends User {
    public void study() {
        // 공부한다.
    }
}
```

```java
public class Teacher extends User {
    public void teach() {
        // 가르친다.
    }
}
```

실제 구현

클래스 사이의 관계

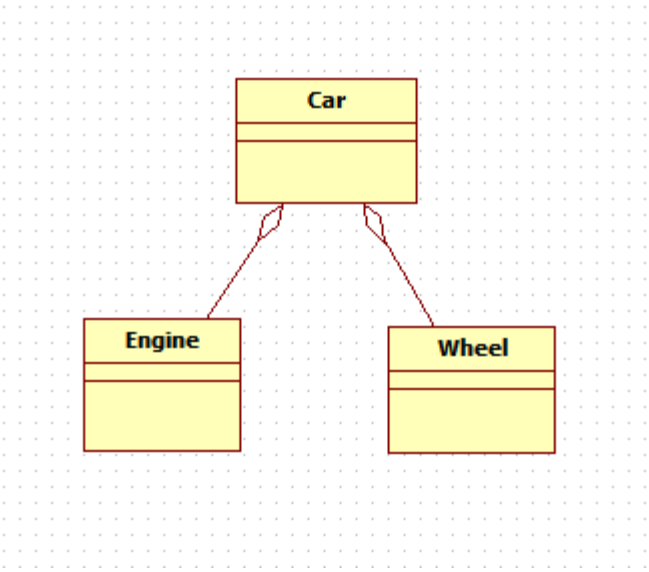
- **연관 관계 (Association)**
  - 클래스들이 **개념상 서로 연결**이 됨
  - 표시 - 양방향: 실선, 단방향: 화살표
  - 단방향(화살표)
    - 한쪽이 다른 쪽을 참조 (e.g., 사람과 핸드폰)
  - 양방향(실선)
    - 서로가 참조 (e.g., 상담의사와 환자)



클래스 사이의 관계

- **집합 관계 (Aggregation)**
  - 연관 관계의 한 종류 (관점에 따라 다르다 vs 같다 논란)
  - 연관 관계를 조금 더 특수하게 나타냄 (전체와 부분의 관계)
  - 전체 객체의 라이프 타임과 부분 객체의 라이프 타임은 **독립적**

  → 전체 객체가 사라져도 **부분 객체는 남아있음**



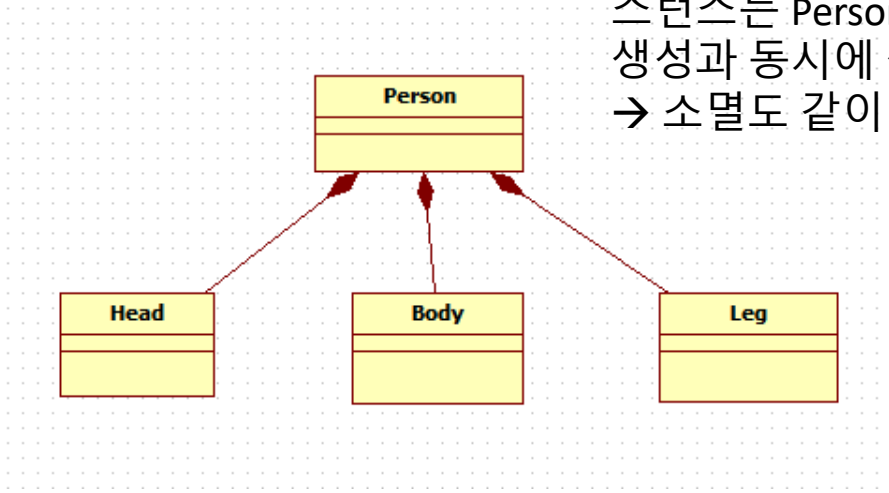

```
public class A
{
        E e;
        public A(E para)
        {
            e=para;
            System.out.println("Aggregation");
        }
}

public class E
{


}
```

E클래스는 기존에 존재하던 인스턴스가 A의 생성자에 파라미터로 들어옴
→ E의 인스턴스는 A의 인스턴스와 상관없이 독립적으로 존재

클래스 사이의 관계

- **합성 관계 (Composition)**
  - 클래스들 사이에 전체 또는 부분 같은 관계를 나타냄
  - 전체 객체의 라이프 타임과 부분 객체의 라이프 타임은 **의존적**

  → 전체 객체가 사라지면 **부분 객체도 사라짐**



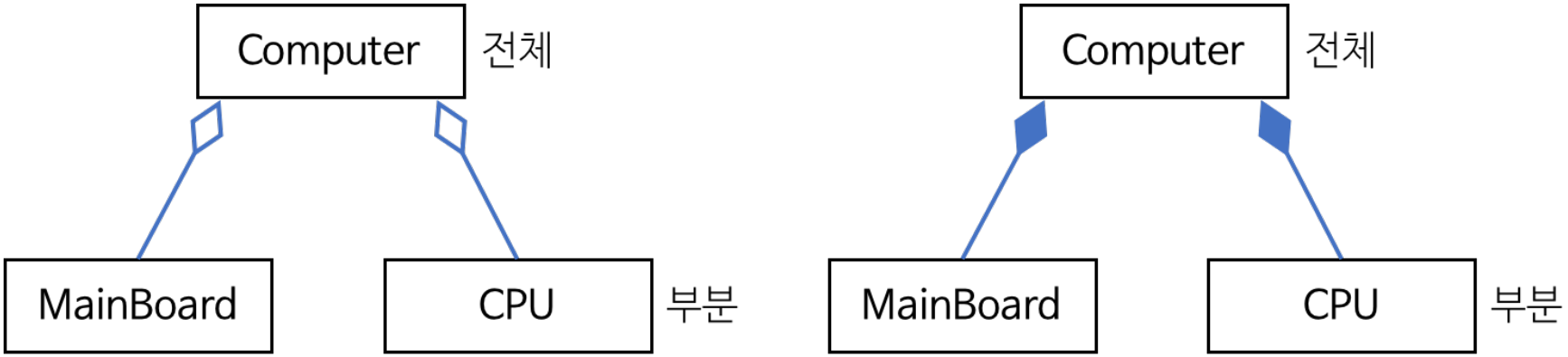
Head, Body, Leg의 인스턴스는 Person의 생성과 동시에 생성

→ 소멸도 같이

```
public class Person {

    private Head head;
    private Body body;
    private List<Leg> legs;

    public Person() {
        super();
        this.head = new Head();
        this.body = new Body();
        this.legs = Arrays.asList(new Leg(), new Leg());
    }

    public void growUp() {

        head.growUp(1);
        body.growUp(2);
        legs.stream()
                .forEach(leg -> leg.growUp(5));
    }

    public void checkLengths() {

        System.out.println(head);
        System.out.println(body);
        legs.stream()
                .forEach(System.out::println);
    }
}
```

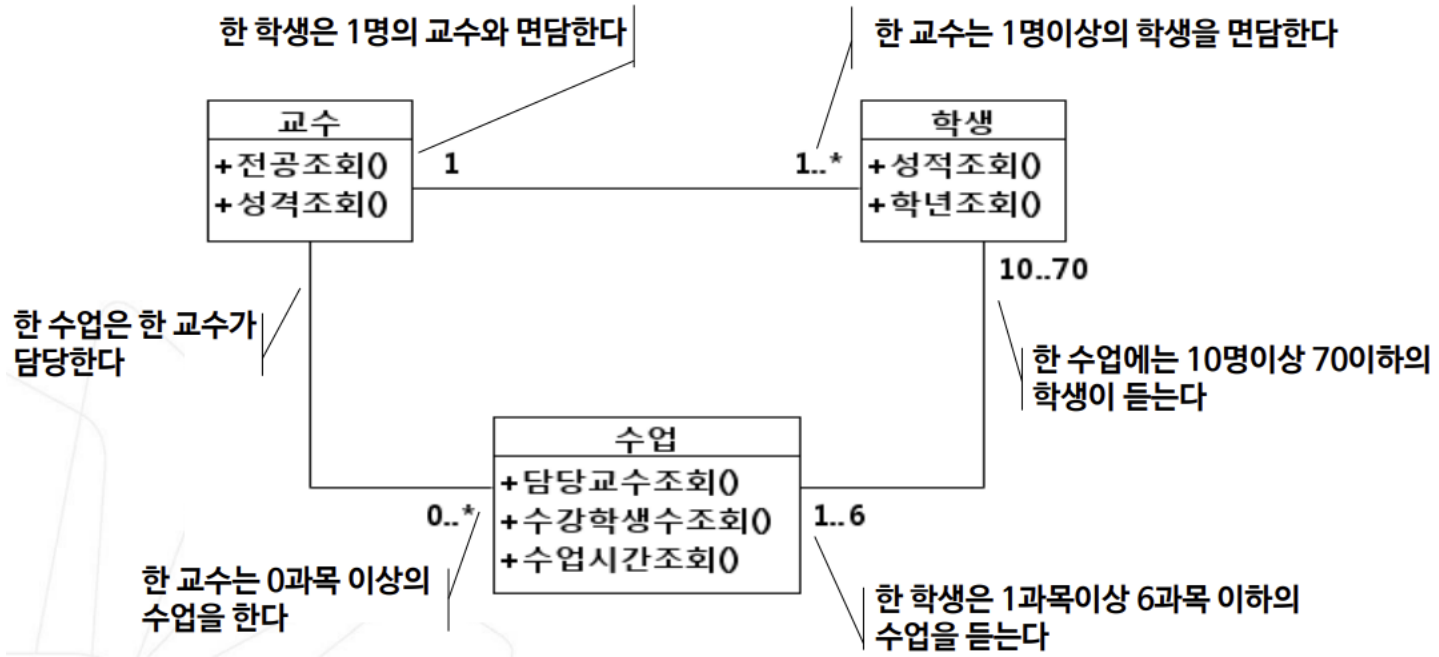
클래스 사이의 관계

- **집합관계 vs. 합성 관계**
  - 라이프 타임 차이 (독립적(집합) vs. 의존적(합성))

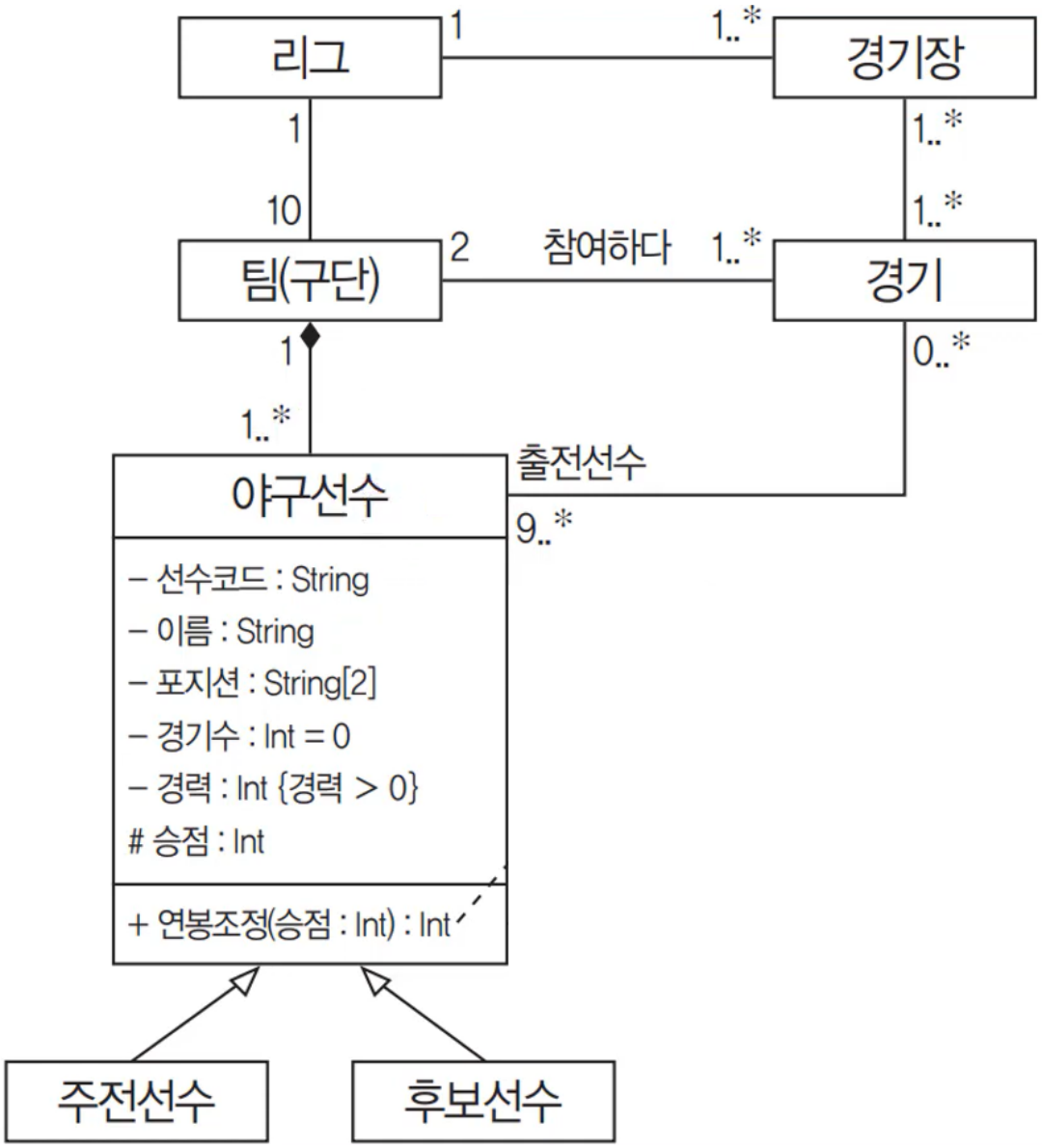


관계의 숫자 표현

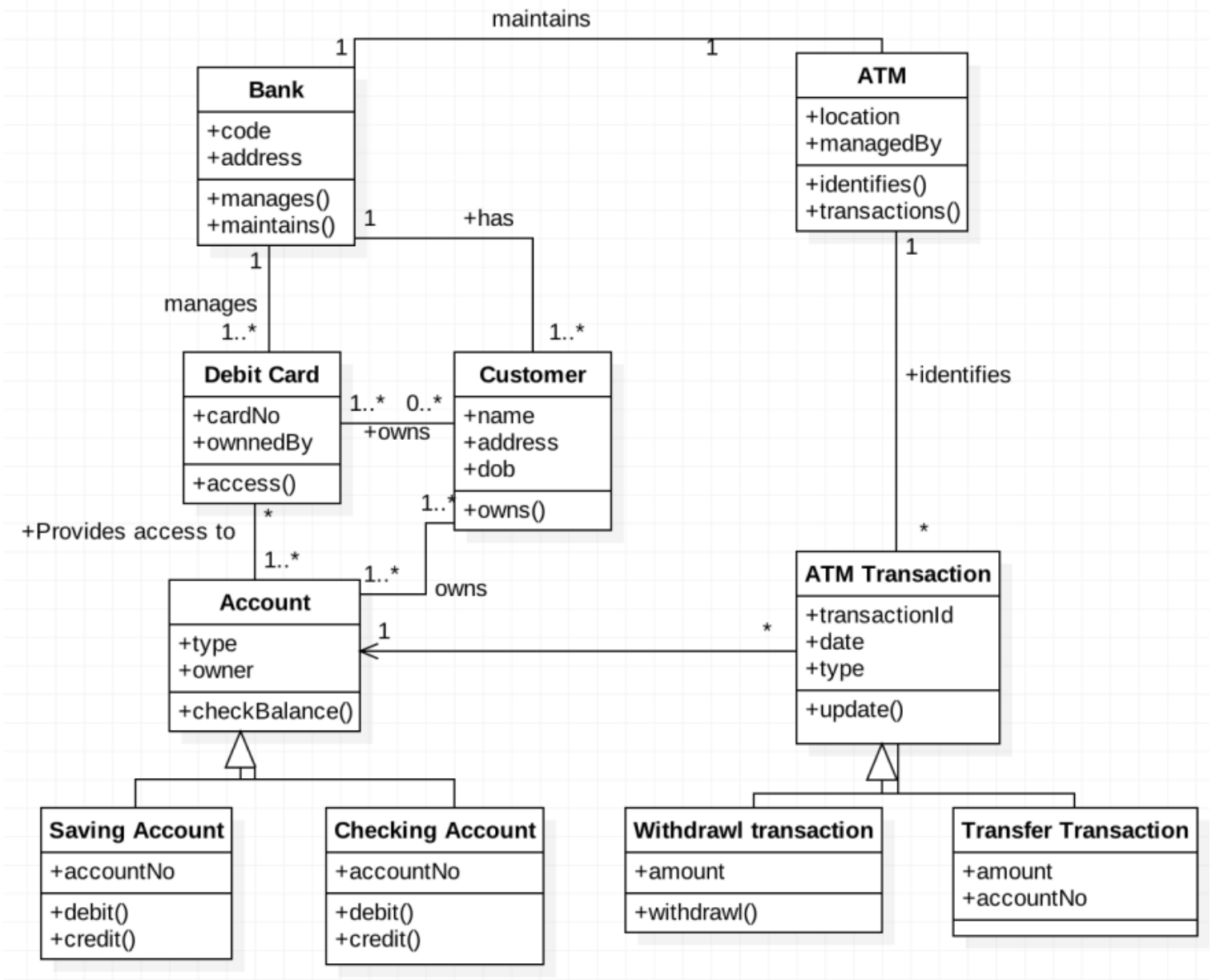
- 숫자 표현 — 참조 가능한 인스턴스의 수
  - 1 : 1개
  - 0..1 : 0 또는 1개
  - * : 0 ~ n개
  - 1..* : 1 ~ n개
  - n..m : n ~ m개



클래스 다이어그램 (예시1)

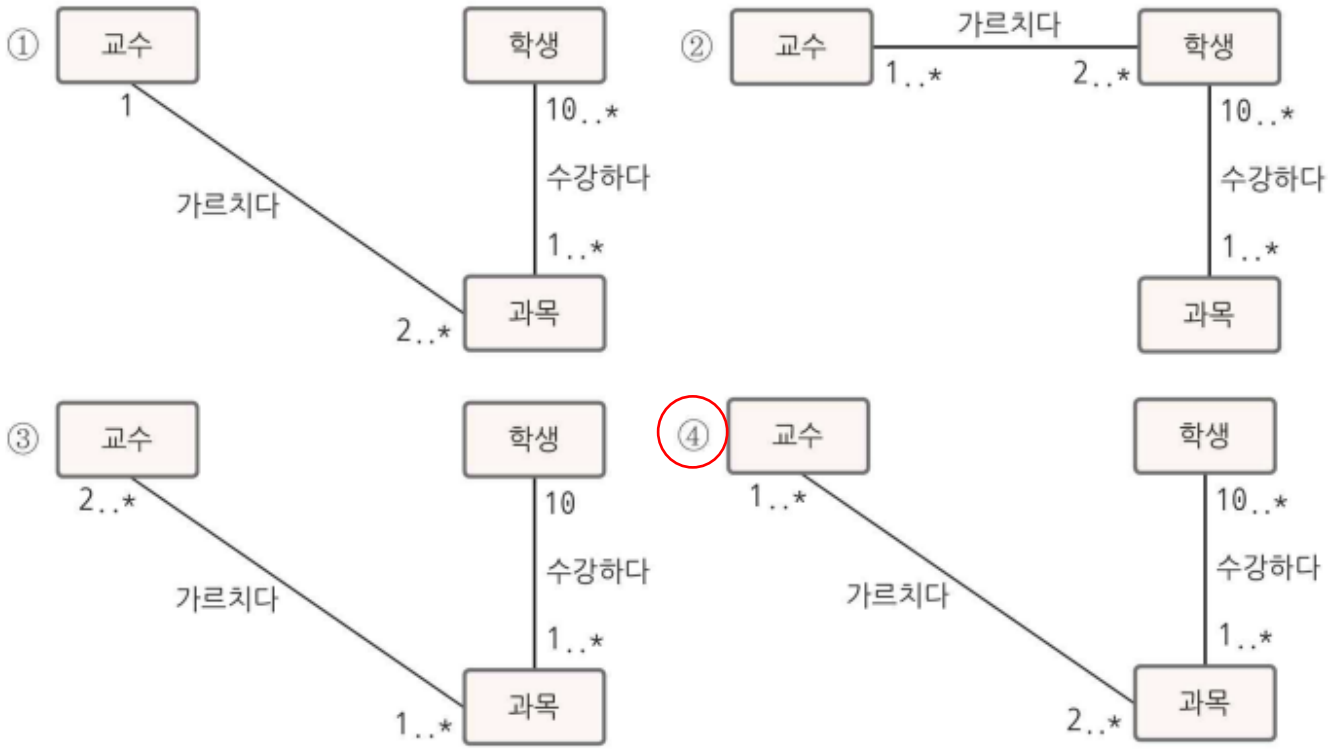


클래스 다이어그램 (예시 2)



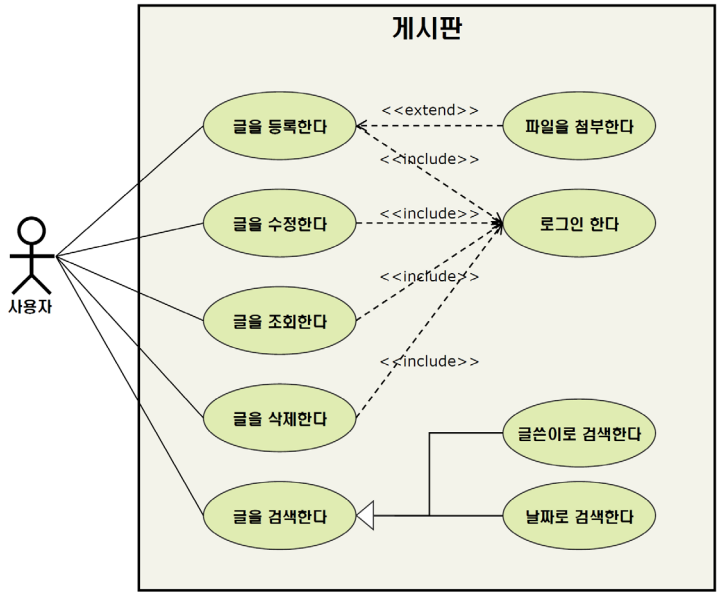
클래스 다이어그램 (예시 3)

'교수'는 적어도 두 '과목' 이상을 가르쳐야 한다.
'과목'은 1명 이상의 '교수'가 가르쳐야 한다.
'과목'은 10명 이상의 '학생'들이 수강해야 한다.
'학생'은 한 '과목' 이상을 수강해야 한다.

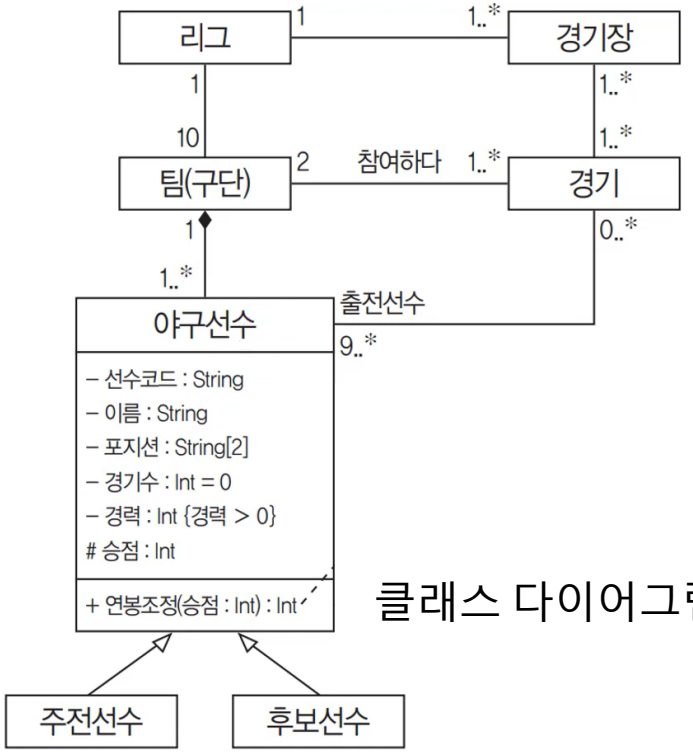


클래스 다이어그램

- 클래스 다이어그램 (장점)
  - 유스케이스 다이어그램 보다 설계관점에서 **더욱 구체적임**
  - 설계에서 클래스 다이어그램을 사용하는 가장 큰 장점은 클래스 간 인터페이스를 빠르게 명확하게 알 수 있음
  - 클래스, 인터페이스, 관계 등이 표시가 되어있어 향후 개발 단계에서 중요한 도구로 사용됨



유스케이스 다이어그램



클래스 다이어그램

클래스 다이어그램

- 클래스 다이어그램 (단점)
  - 너무 상세한 내용을 기입하면 구현단계에서 할 일이 미리 이루어지는 오류를 범함
  - 이는 실제 구현 단계에서 커다란 위험의 요소가 내재

  Why? → 설계에 한 내용을 실제 구현 해야 하기 때문에

Questions?