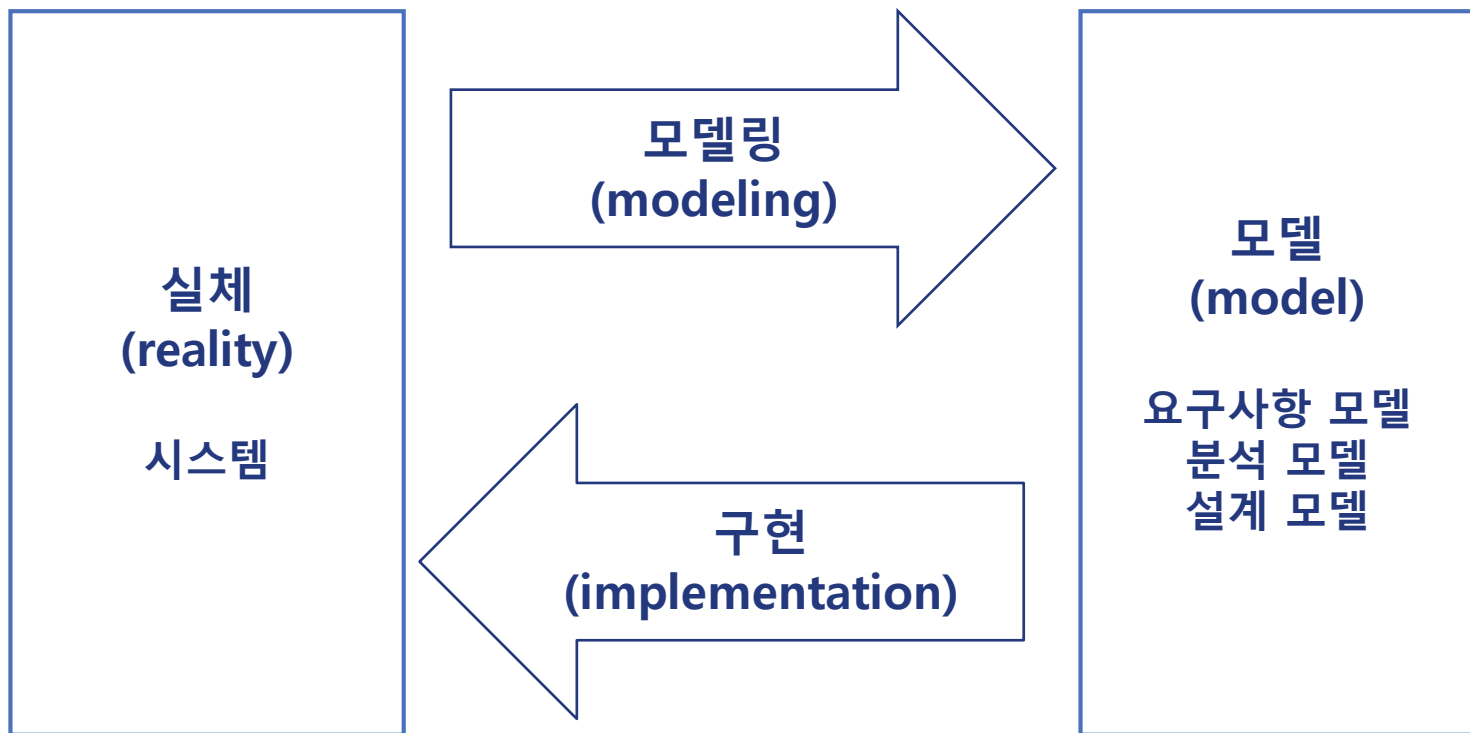


# 0. UML

**U**nified **M**odeling **L**anguage



# 모델링과 모델



# UML 배경

## ❖ 1980년대

- 객체 지향 개념이 산업계에 도래하기 시작
  - 많은 사람들이 객체지향을 위한 도식적인 설계 언어를 생각하기 시작

## ❖ 1988년 ~ 1992년

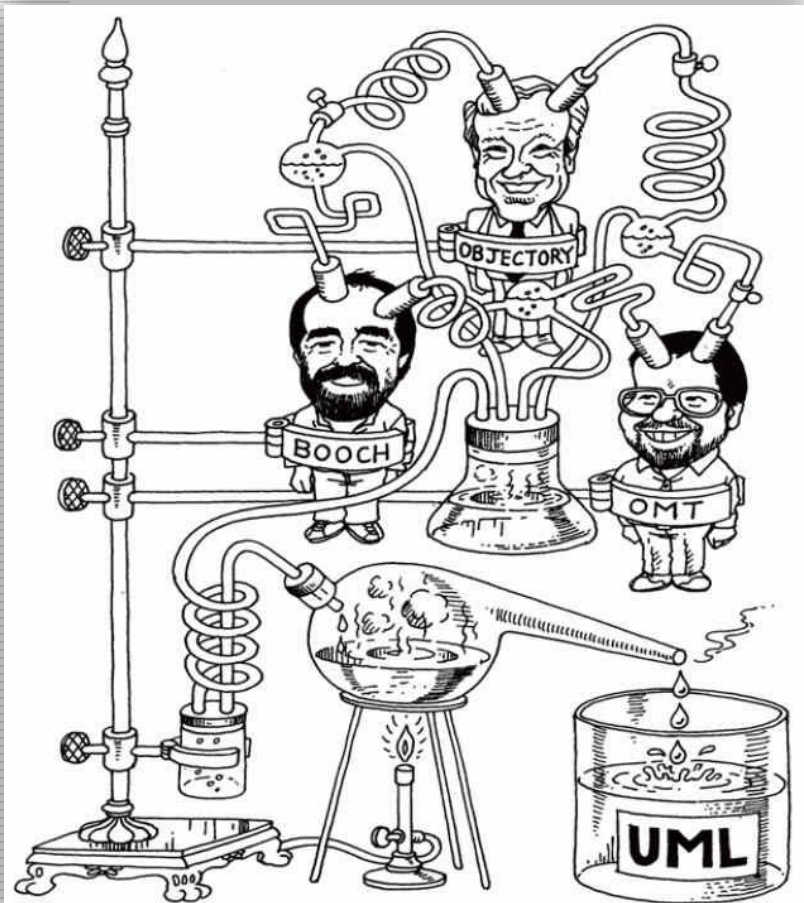
- 같은 개념이 다른 표기법으로 사용되어 혼란을 가중 시킴

방법론	Shlaer & Mellor 방법	Rumbaugh OMT 방법	Booch 방법	Jacobson OOSE 방법
특징	<ul style="list-style-type: none"> <li>• 실시간 시스템 의식</li> <li>• 치밀한 동적 모델링 지원</li> </ul>	<ul style="list-style-type: none"> <li>• 분석, 설계, 프로그래밍 적용</li> <li>• 설계 기법이 부족</li> </ul>	<ul style="list-style-type: none"> <li>• Ada용의 방법론이었던 Booch법을 확장</li> <li>• Hood, OOSD 등 Ada용의 영향이 강함</li> <li>• 분석을 수행하기에는 부족</li> </ul>	<ul style="list-style-type: none"> <li>• Use Case를 사용하여 시스템에 대한 요구사항을 사용자관점에서 효과적인 모델링 작업 가능</li> <li>• 분석, 설계의 세부사항이 부족</li> </ul>
UML적용	동적 모델링 기법	분석 기법	설계 기법	Use Case 기법

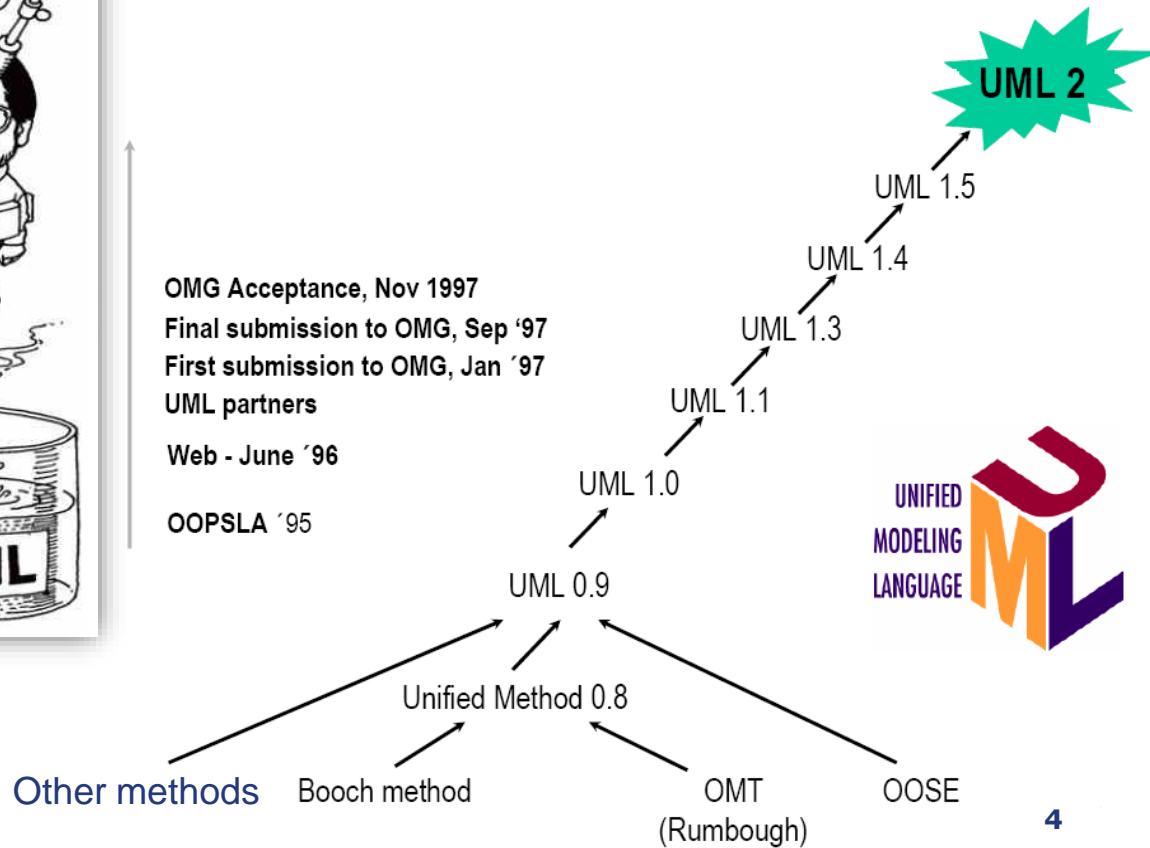


객체지향모델링의 산업계 표준  
UML

# UML 역사



OMG Acceptance, Nov 1997  
 Final submission to OMG, Sep '97  
 First submission to OMG, Jan '97  
 UML partners  
 Web - June '96  
 OOPSLA '95

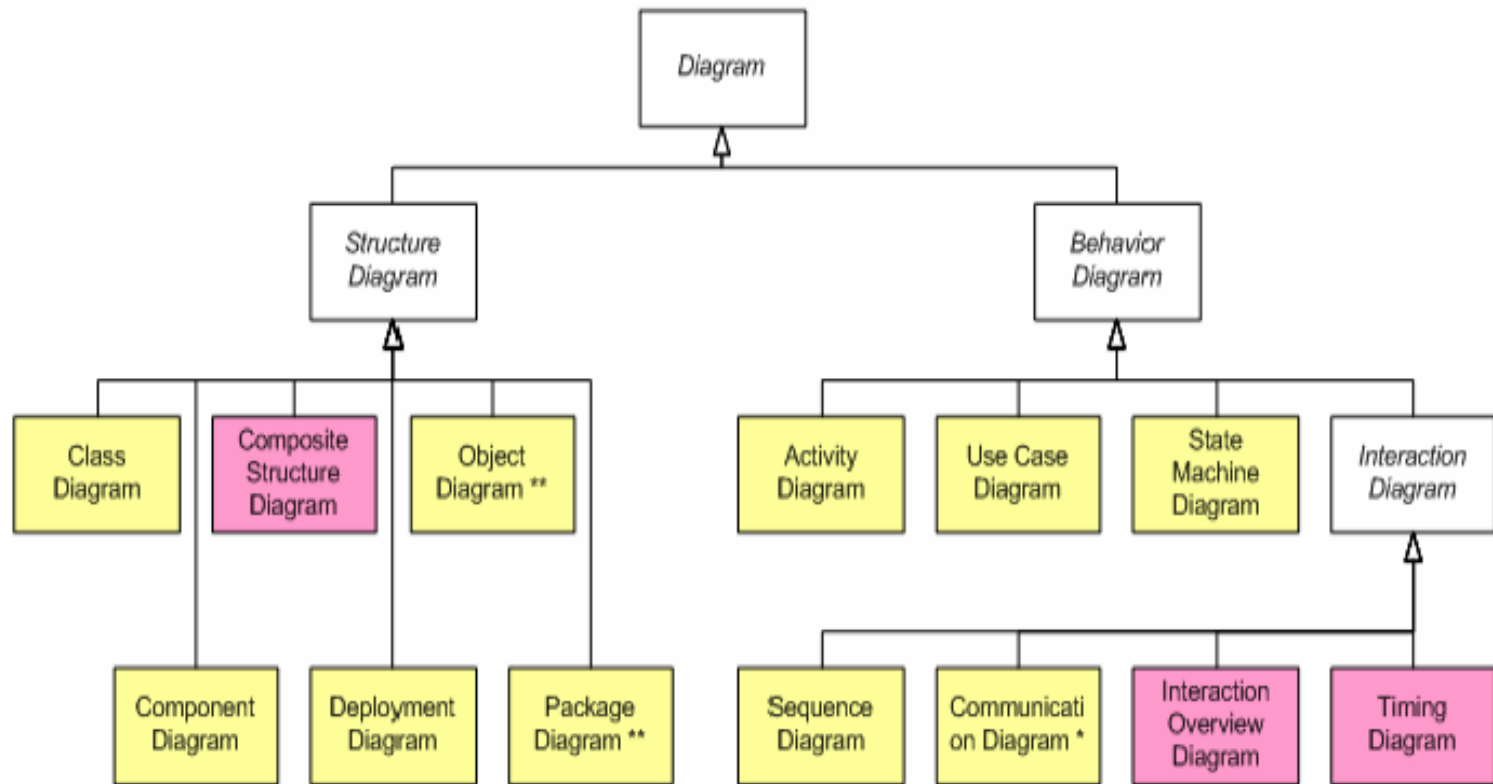


# UML 개요

## ❖ UML(Unified Modeling Language)

- 소프트웨어 청사진을 작성하는 표준 언어
- 객체지향 시스템의 개발 중에 생기는 산출물들을 가시화하고, 명세화하며, 시스템 구축 및 문서화하는 데 사용되는 언어
  - 가시화 언어
    - 소프트웨어의 개념모델을 가시적인 그래픽 형태로 작성하여 참여자들의 오류 없고 원활한 의사소통이 이루어지게 하는 언어
  - 명세화 언어
    - 소프트웨어의 각 개발과정에서 필요한 모델을 정확하고 완전하게 명세할 수 있게 하는 언어
  - 구축 언어
    - 다양한 객체지향 프로그래밍 언어로 변환 가능
  - 문서화 언어
    - 여러 개발자들 간의 통제, 평가 및 의사소통을 위한 문서화를 위한 언어

# UML 2.0 Diagram Taxonomy



\* UML 1 collaboration diagram

\*\* Unofficially in UML 1

New to UML 2

# 1. Use Case Diagram

- ❖ 시스템이 제공하는 기능 및 그와 관련한 외부요소를 표시
- ❖ Use case는 시스템의 기능적인 요구사항을 발굴하는데 사용되며, 시스템의 사용자와 시스템 사이의 정형적인 상호작용을 기술
  - 사용자의 입장에서 (시스템 외부에서) 보여지는 시스템의 기능이 **무엇** 인지를 기술
  - But, 그러한 기능이 **어떻게** 구현되는지는 기술하지 않음

# 1. Use Case Diagram

## ❖ 액터

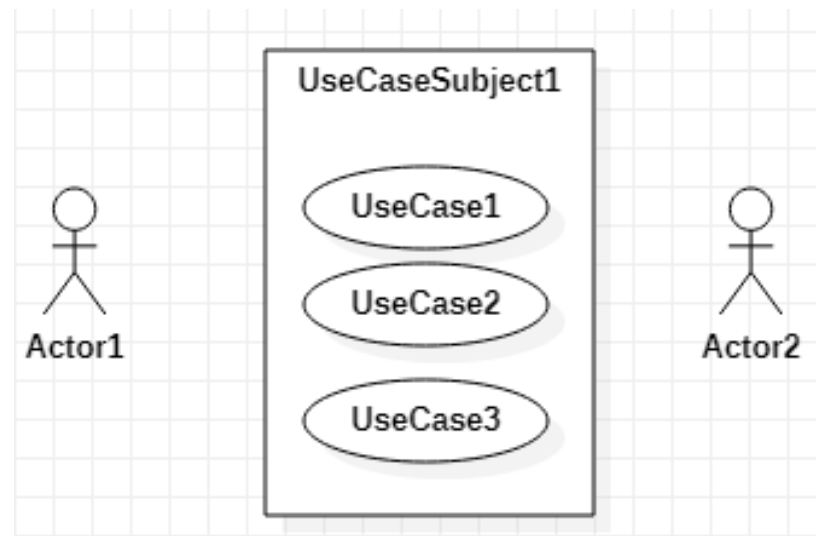
- 어떤 시스템을 중심으로 보았을 때, 거기와 관련된 모든 외부 요소를 표현
- 표기법 : 원과 선을 조합하여 그린 사람모양으로 표현

## ❖ 유스케이스

- 시스템 외부에서 본 시스템의 기능을 표시
- 표기법 : 타원으로 표시

## ❖ 시스템 경계

- 시스템화 대상 범위를 나타냄
  - 안쪽이 시스템화 대상 범위,  
바깥쪽이 시스템화 대상 범위  
이외가 됨
- 표기법 : 유스케이스나 액터를  
둘러싼 틀로써 표기





# 1. Use Case Diagram

## ❖ 유스케이스와 액터 사이의 관계

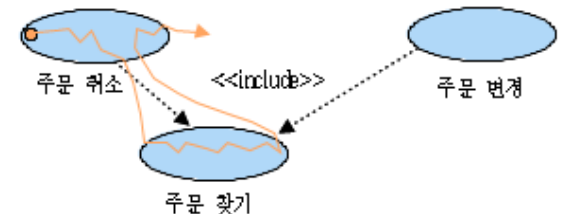
- 관련된 액터와 유스케이스를 연결
- 표기법 : 액터와 유스케이스 사이에 실선을 그어 표시

## ❖ 유스케이스 사이의 관계

### ■ 포함 관계 : <<include>>

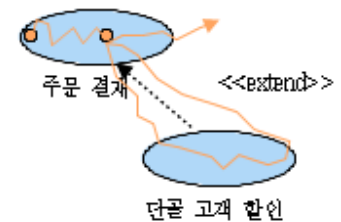
- 다른 유스케이스를 호출
- 공통되는 유스케이스를 별도로 정의
- 표기법 : 포함하는 쪽에서 포함되는 쪽으로

점선 화살표를 그리고, 스테레오타입 <<include>>를 함께 표시



### ■ 확장 관계 : <<extend>>

- 존재하는 유스케이스의 동작을 조건적으로 확장
- 이벤트의 추가나 예외적인 케이스
- 표기법 : 확장기능의 유스케이스에서 확장대상이 되는  
베이스 유스케이스 쪽으로 점선 화살표를 그리고,  
스테레오타입 <<extend>>를 함께 표시

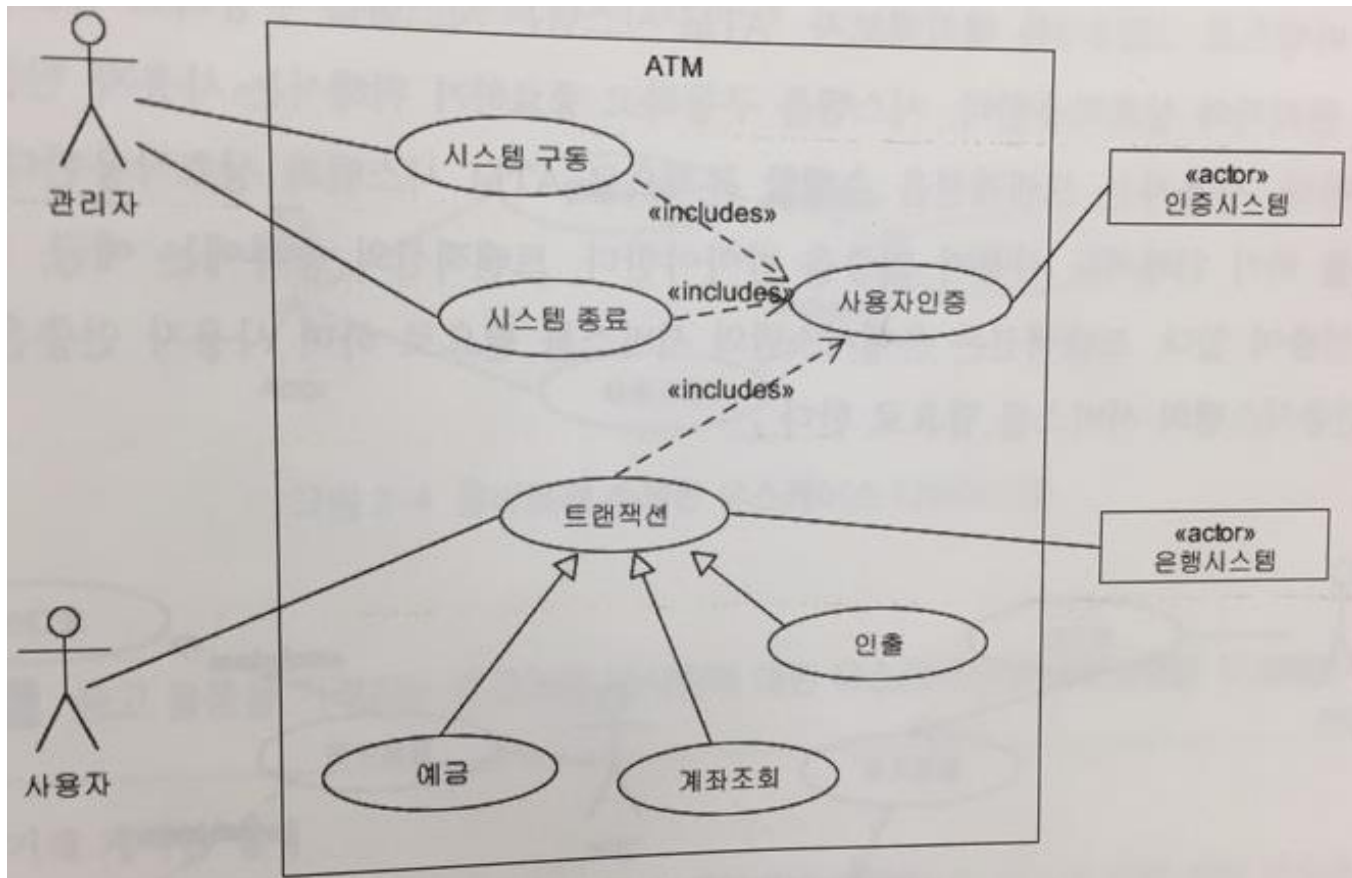


## ❖ 액터와 액터, 유스케이스와 유스케이스 사이의 관계

### ■ 일반화 관계

- 시스템 기능 사이에 추상화 혹은 구체화 관계가 존재함을 나타냄
- 표기법 : 구체적인 유스케이스로부터 추상적인 유스케이스 쪽으로 머리모양이 하얀색 삼각형으로 된 실선화살표를 그어 표시





# 1. Use Case Diagram

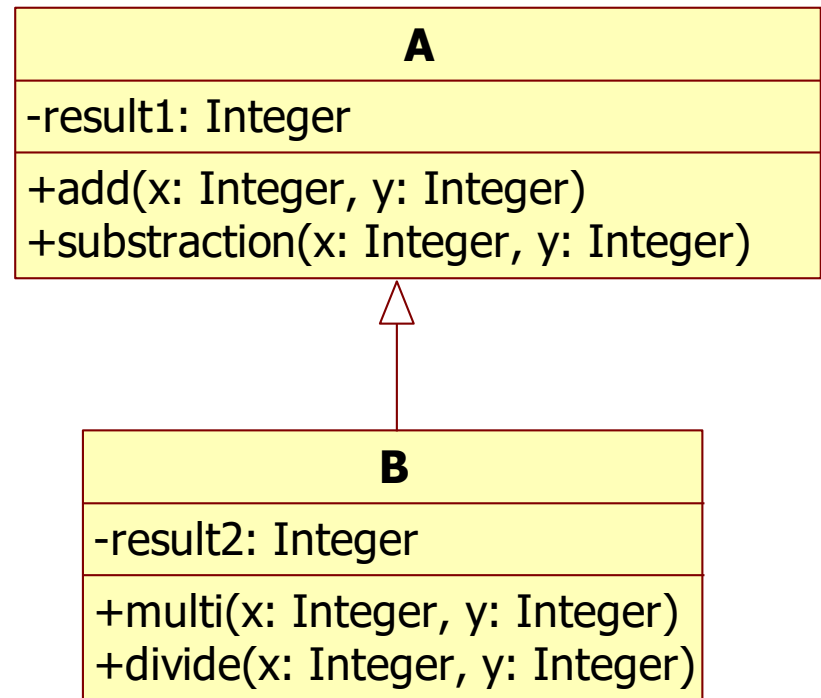


## 2. Class Diagram

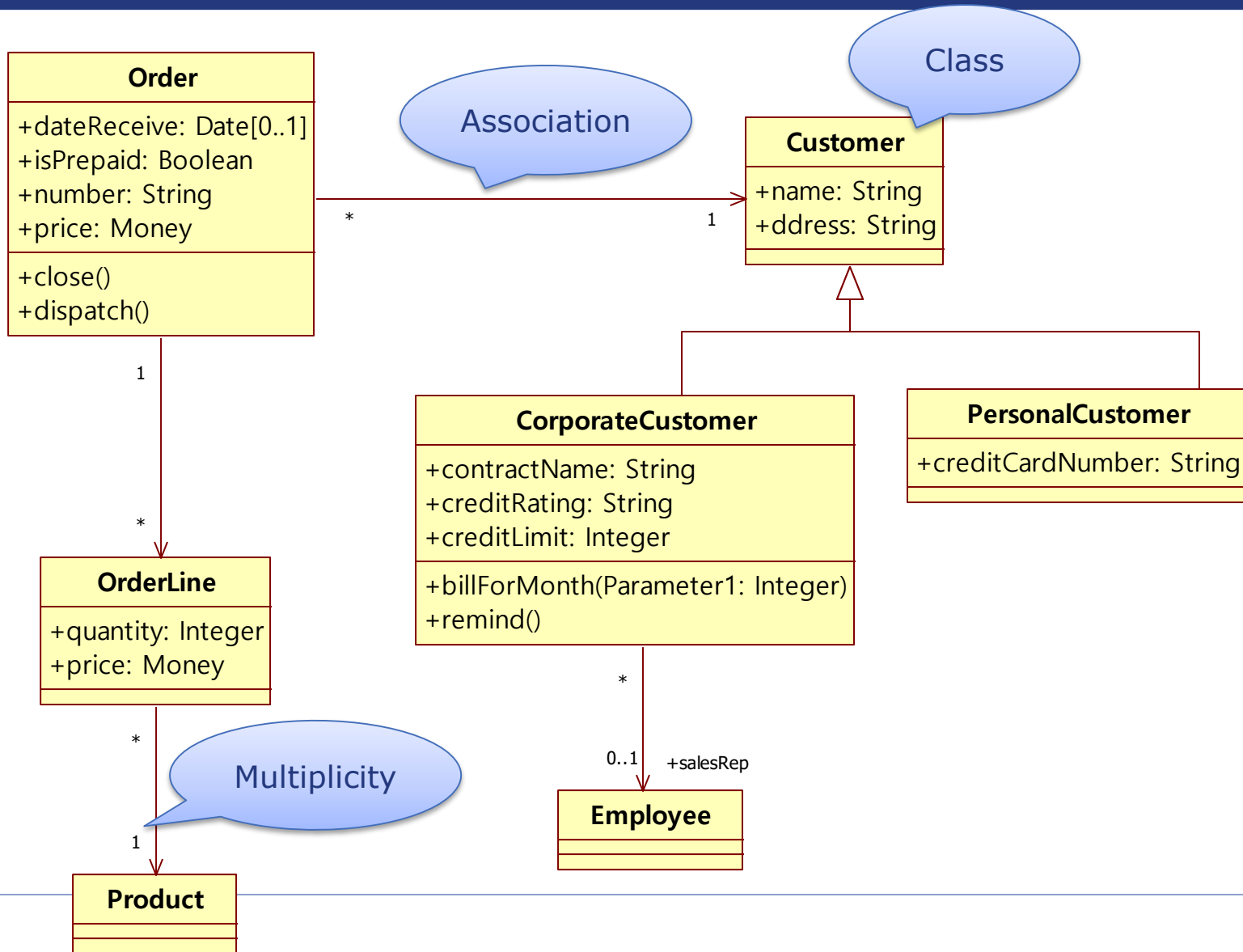
- ❖ UML diagram들 중 객체 지향 시스템의 모델링에 가장 많이 사용되는 diagram
- ❖ class, interface, collaboration과 그들간의 relationship를 기술
  - 시스템에 사용되는 객체의 형태(Class)와 그들 사이에 존재하는 여러 가지 정적인 관계를 기술
  - 클래스의 속성(property)과 연산(operation)을 보여주며, 클래스에 속하는 객체들이 연결되는 방식에 적용되어야 하는 제약사항들을 기술
- ❖ UML Code 생성은 기본적으로 class diagram에 기반
  - Modeling Tool에 따라 Java, C++, C#로 구현

## 2. Class Diagram

```
1. class A
2. {
3.     private int result1;
4.     public int add(int x,int y)
5.     {
6.         
7.     }
8.     public int subtraction(int x,int y)
9.     {
10.        
11.    }
12. }
13.
14.
15.
16. class B extends A
17. {
18.     private int result2;
19.     public int multi(int x,int y)
20.     {
21.         
22.     }
23.     public int divide(int x,int y)
24.     {
25.         
26.     }
27. }
28.
29. }
```



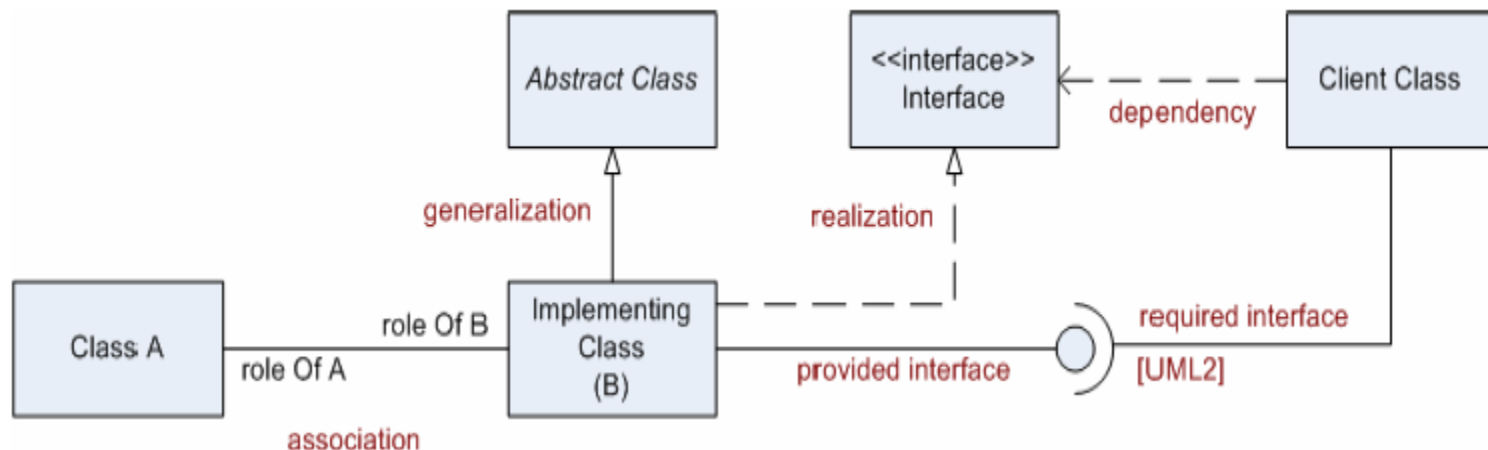
## 2. Class Diagram



# 클래스 간의 관계

## ❖ 클래스 간의 관계

- 연관(association)
  - 집합(aggregation)과 복합(composition)
- 의존(dependency)
- 일반화(generalization)
- 실체화(realization)



# 클래스 간의 관계

## ❖ 연관(Association)

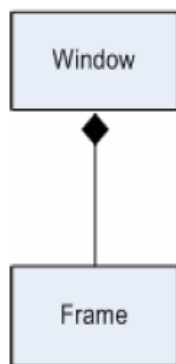
- 서로 알고 지내는 정도의 관계로, 하나의 클래스가 또 다른 클래스를 인지하고 있음을 의미
- 두 클래스는 서로 메시지를 주고받으며 이용하는 관계
- 연관의 표기
  - 두 클래스(source 클래스와 target 클래스)를 실선으로 연결
  - 양방향으로 연결되는 경우 화살표를 표시하지 않음
  - 역할(or 특성의 이름)은 multiplicity와 함께 연관관계의 끝부분에 기술



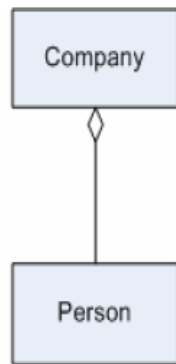
연관 관계의 예 : 사장과 직원 관계

# 클래스 간의 관계

- 집합(Aggregation)과 복합(Composition)은 연관(Association)의 특수한 경우
- Aggregation은 전체/부분(part-of 관계)의 관계
  - 하나의 클래스가 다른 클래스들로 구성되어 있음을 표현
  - 속이 빈 마름모로 표현
  - 마름모가 붙은 클래스가 그렇지 않은 클래스로 구성되어 있음
- Composition은 더 강력한 형태의 Aggregation
  - 속이 찬 마름모로 표현
  - 부분에 해당하는 객체는 전체의 객체에만 포함되어야 함
  - 부분에 해당하는 객체는 전체에 해당하는 객체와 함께 생성하고 소멸되는 것으로 기대됨



Composition



Aggregation

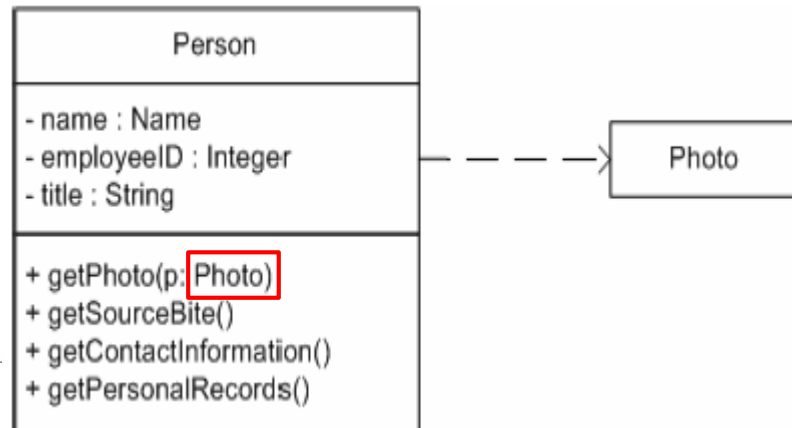
- ▶ Window는 Frame으로 구성되어 있다. (Composition)
- ▶ Company는 Person으로 구성되어 있다. (Aggregation)



# 클래스 간의 관계

## ❖ 의존(Dependency)

- 하나의 클래스가 다른 클래스를 작업의 인자(argument)로 사용하는 경우를 나타냄
- 이 경우 전자가 후자에 의존한다고 하고, 점선 화살표로 의존의 방향을 나타냄
  - Person 클래스의 getPhoto() 작업은 Photo 클래스의 객체를 인자로 받음
- 하나의 클래스(Person)가 다른 클래스(Photo)에 의존하고 있는 경우
  - Photo 클래스의 명세에 변화가 있는 경우 Person 클래스에 영향을 미칠 수 있음(But, 반대 방향으로서는 영향을 미치는 것은 아님)



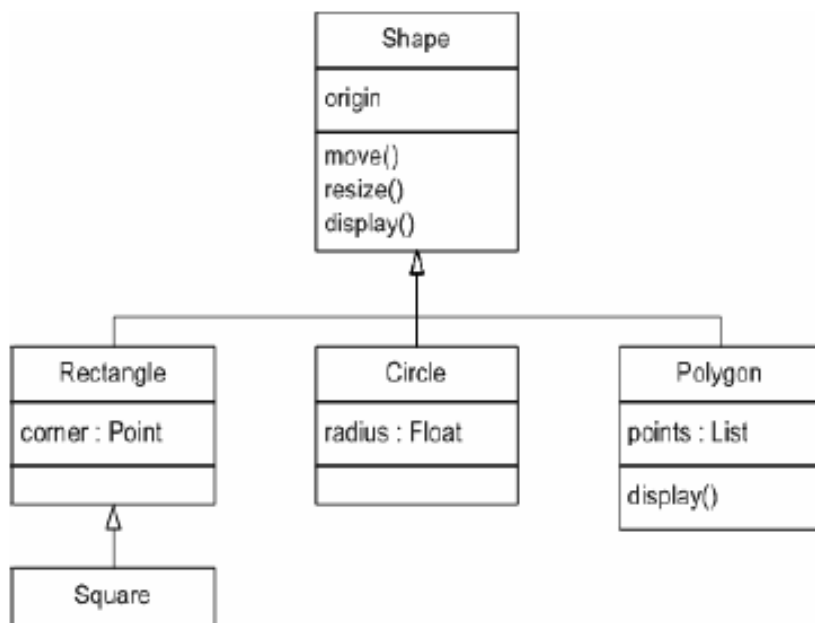
# 클래스 간의 관계

## ❖ 일반화(Generalization) : is-a-kind-of relationship

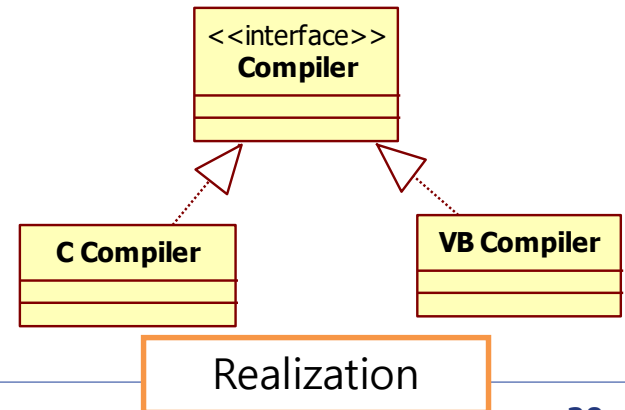
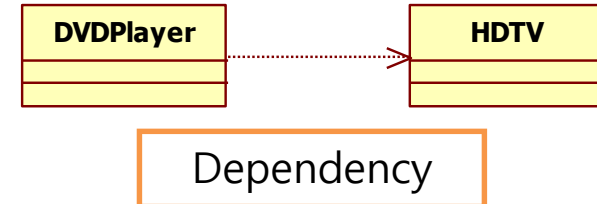
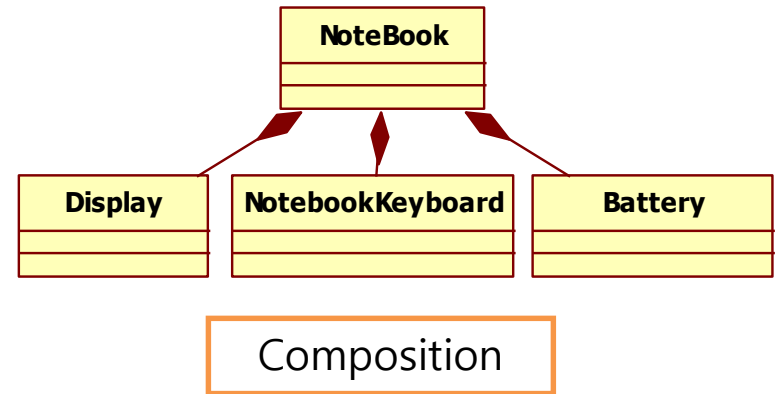
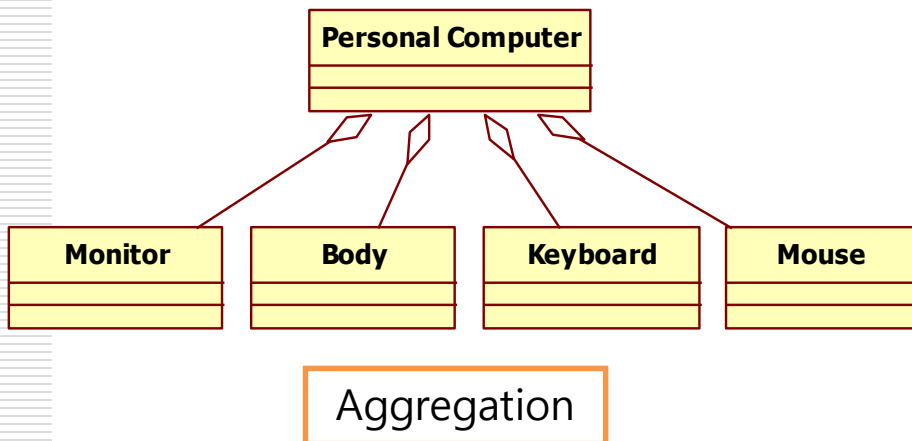
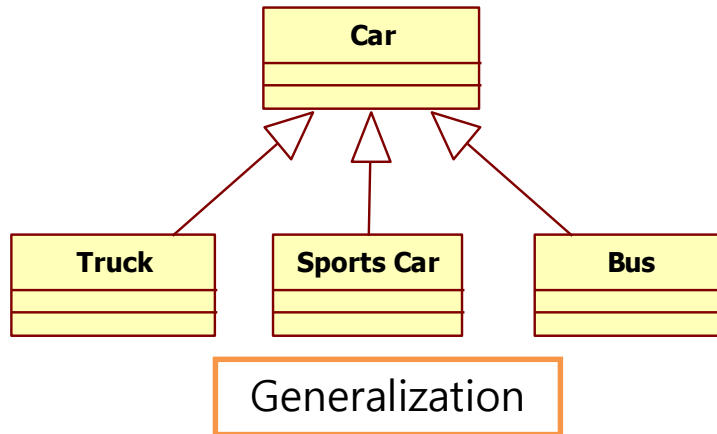
- 일반적인 클래스(슈퍼 클래스 or 부모 클래스)와 더 구체적인 종류의 클래스(서브 클래스 or 자식 클래스) 사이의 관계
  - 자식의 객체는 부모 객체가 나타나는 곳이면 어디서든 사용될 수 있으나, 반대로는 안됨
  - 자식 클래스의 작업으로 부모 클래스와 동일한 형태의 signature를 가지는 작업은 부모 클래스의 작업을 대체하게 됨(override)
    - 예) Polygon 클래스에서 새로 정의된 display() 작업은 Shape 클래스의 display()를 override 함
- 일반화는 인터페이스가 인터페이스를 상속받는 경우를 포함
  - 단, 클래스가 인터페이스(interface)를 구현하는 경우에는 **실체화(Realization)**이라고 하며, 실선 대신 점선으로 표현

# 클래스 간의 관계

- 일반화는 여러 단계로도 가능
  - Square 클래스는 Shape 클래스를 상속받은 Rectangle 클래스를 상속받음
  - But, 일반적으로 상속의 단계가 많아지는 것은 권장하지 않음
  - 아무리 복잡한 시스템이라도 최대 6~7 단계의 상속으로 모든 것을 표현할 수 있는 경우가 많음

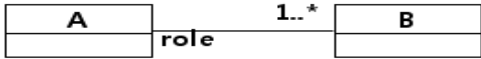
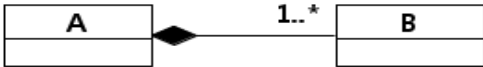
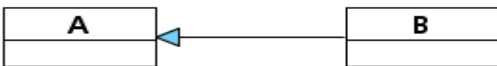
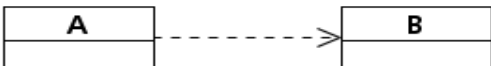
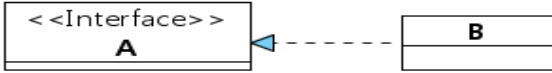
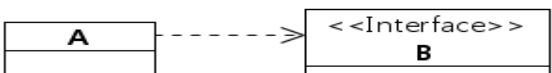


# 클래스 간의 관계



# 클래스 간의 관계

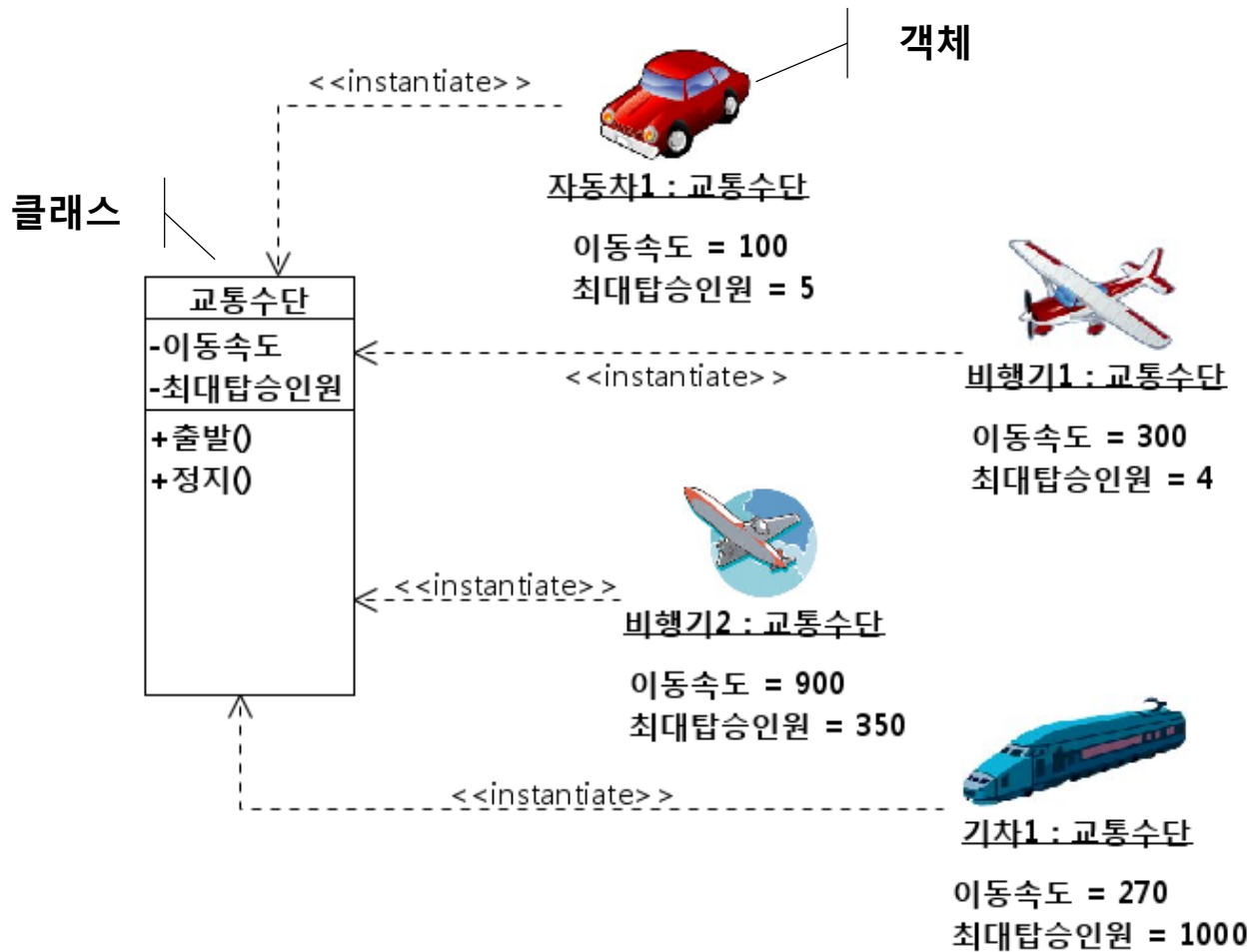
## ❖ Class Diagram의 관계 요약

관계	표기법	의미
연관 관계		클래스 A와 클래스 B는 연관 관계를 가지고 있다.
복합 관계		클래스 B는 클래스 A의 부분이다.
일반화 관계		클래스 B는 클래스 A의 하위 클래스이다.
의존 관계		클래스 A는 클래스 B에 의존한다.
인터페이스 실현 관계		클래스 B는 인터페이스 A를 실현한다.
인터페이스 의존 관계		클래스 A는 인터페이스 B에 의존한다.

# 3. Object Diagram

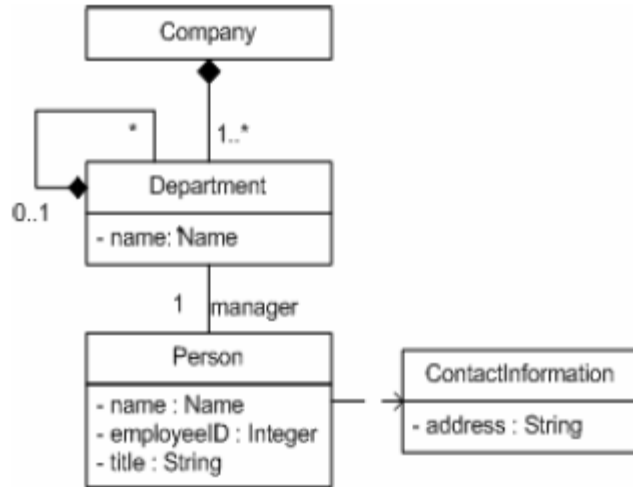
- ❖ 특정 시점의 시스템에 포함된 객체들의 모습(snapshot)을 기술
  - 클래스가 아니라 클래스의 instance(객체)들의 관계를 기술
  - 클래스 다이어그램
    - 모델링 대상이 가질 수 있는 모든 상황에 대응하는 다이어그램
  - 객체 다이어그램
    - 모델링 대상의 어느 한 순간에 대응하는 다이어그램
- ❖ 특히 객체의 구성이 복잡하게 얽혀 있는 경우 그들 사이의 관계를 보여주는 데 유용하게 사용할 수 있음
  - Class diagram으로 기술된 시스템의 구성이 이해하기 어려운 경우에 object diagram을 여러 개 작성함으로써 이해를 도울 수 있음
- ❖ 클래스를 기술하는 경우와 동일한 형식으로 기술
  - 객체 이름(d2) 과 해당 클래스의 이름(Department)을 함께 사용하고, 밑줄을 그어 표시 => d2 : Department
    - 객체 이름은 생략할 수 있음(Anonymous object) => : ContactInformation
  - 각 특성(property)이 가지는 현재 값들을 기술 => name = "Sales"
  - 작업(operation)은 표시하지 않음

### 3. Object Diagram

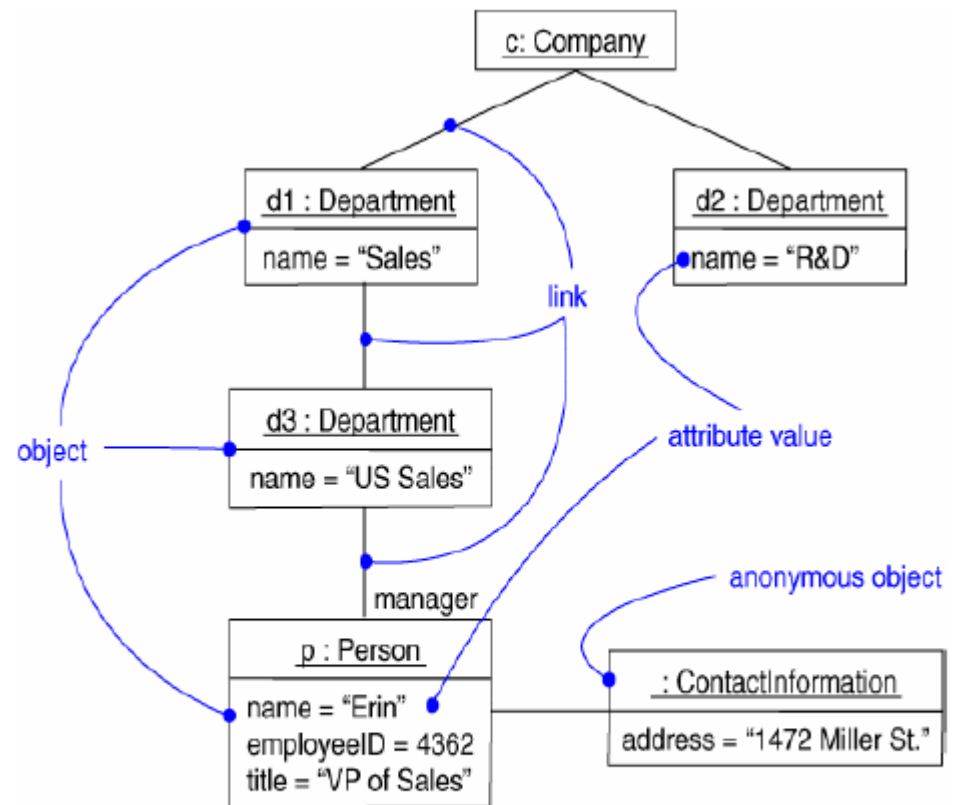


### 3. Object Diagram

❖ 왼쪽의 class diagram을 토대로 기술한 object diagram의 예



**a class diagram**



**a corresponding object diagram**



## 4. Sequence Diagram

- ❖ Communication diagram, Interaction Overview diagram, Timing diagram과 함께 Interaction diagram에 속함
- ❖ 시스템의 동적인 행위를 기술
- ❖ 객체들 사이의 메시지 교환(혹은 method 호출)을 시간의 순서에 따라 기술
- ❖ 클래스의 instance를 participant로 하여 수직의 축(lifeline)으로 나타내고, 이들 간의 메시지의 교환을 화살표로 표기
- ❖ 동일한 lifeline에서는 윗부분에 표시된 event가 시간적으로 우선
- ❖ 시스템 실행 시 생성되고 소멸되는 객체를 표기

## 4. Sequence Diagram

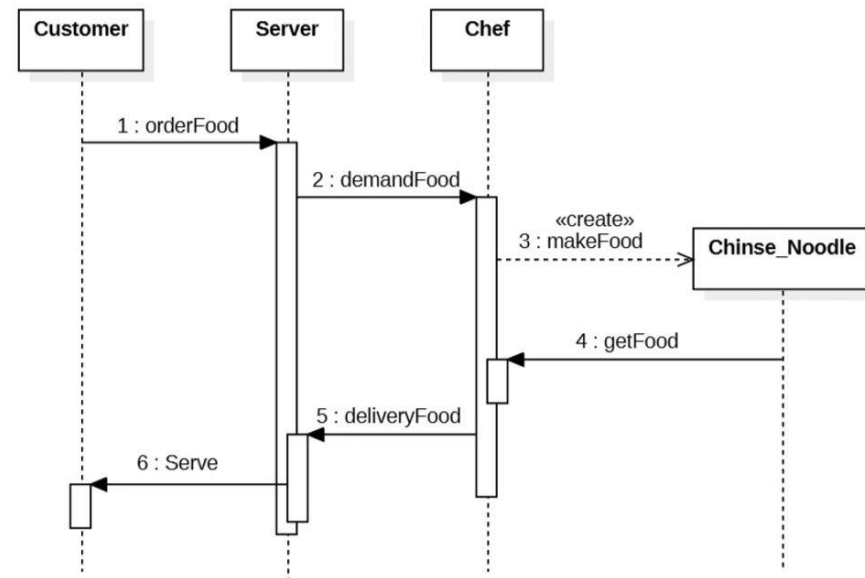
- ❖ 하나의 use case 내에 포함된 여러 객체들의 행동을 살펴보고 싶을 때 사용
- ❖ 객체들 사이의 협업 과정을 살펴보는데 좋은 diagram
- ❖ 만약, 하나의 객체가 여러 개의 use case들에서 어떻게 행동하는가를 살펴보려면 **State Machine diagram**을 이용
- ❖ 만약, 여러 갈래에 걸친 여러 use case들의 가로지르는 행위들을 살펴보려면 **Activity diagram**을 이용
- ❖ sequence diagram과 communication diagram은 의미상으로 동일한 정보를 기술
  - sequence diagram이 시간적 순서를 강조
  - communication diagram은 객체간의 연결을 강조

## 4. Sequence Diagram

### ❖ 식당 음식 주문

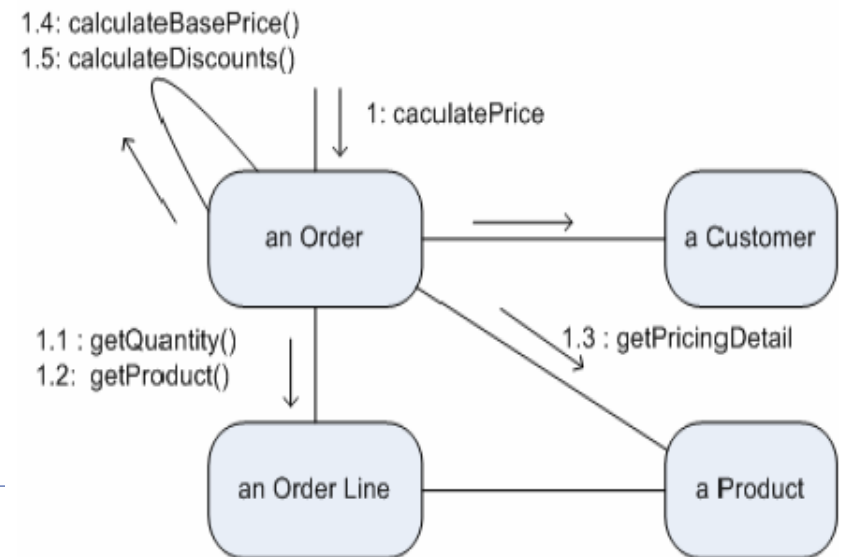
- 고객(Customer), 종업원(Server), 요리사(Chef)

- ① 고객이 종업원에게 음식을 주문 (orderFood)
- ② 종업원은 요리사에게 주문 받은 음식의 조리를 요청 (demandFood).
- ③ 요리사는 짜장면을 조리 (makeFood, getFood)
- ④ 요리사가 만든 요리를 종업원에게 전달 (deliveryFood)
- ⑤ 종업원이 음식을 고객에게 전달 (Serve)



# 5. Communication Diagram

- ❖ UML 1.x의 collaboration diagram
- ❖ Interaction diagram의 일종
- ❖ 객체와 객체들의 관계로 구성된 상호작용을 보여주며, 객체들 사이에 주고받는 메시지를 표현할 때 사용
- ❖ Sequence diagram과는 달리 참가 객체들을 자유롭게 배열할 수 있어, 각 객체들이 어떻게 연결되어 있는지를 알기 쉽게 보여줌
  - 메시지의 순서는 번호를 사용

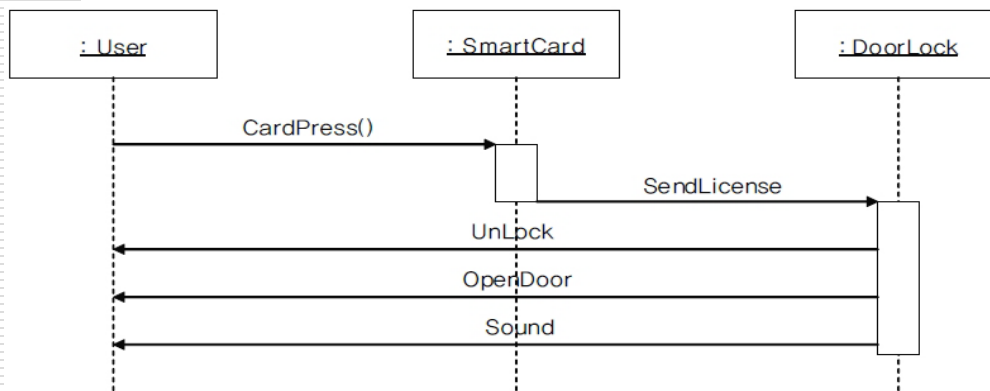


# Sequence Diagram vs. Communication Diagram

## ❖ Communication Diagram과 Sequence Diagram간의 자동변환 가능

### ❖ Sequence Diagram

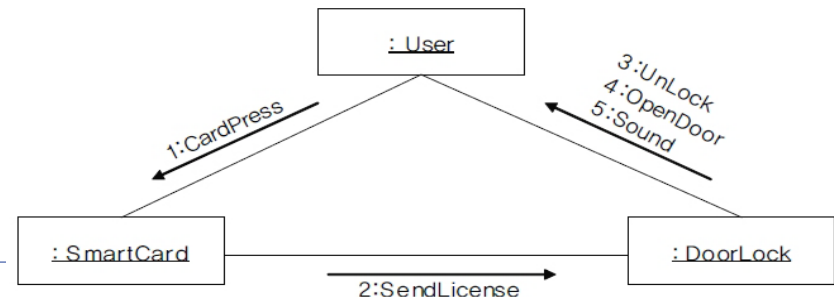
- 메시지의 상,하 배치 위치가 메시지의 전송 순서를 정의하므로 객체들간에 주고받는 메시지의 순서를 파악할 때 매우 효과적
- 메시지를 주고받는 객체들간의 관계는 표현되지 않음



(b) 순차 다이어그램

### ❖ Communication Diagram

- 객체들 간의 관계가 링크로 표현되므로 메시지를 전달하고 수신하는 객체들간의 관계를 쉽게 파악할 수 있음
- 객체가 통신 다이어그램의 임의의 위치에 배치될 수 있기 때문에 메시지의 위치에 따른 전송 순서를 가정할 수 없음 --> 각 메시지에 대하여 반드시 메시지 번호를 지정함으로써 메시지간의 시간적인 순서를 표현해야 함

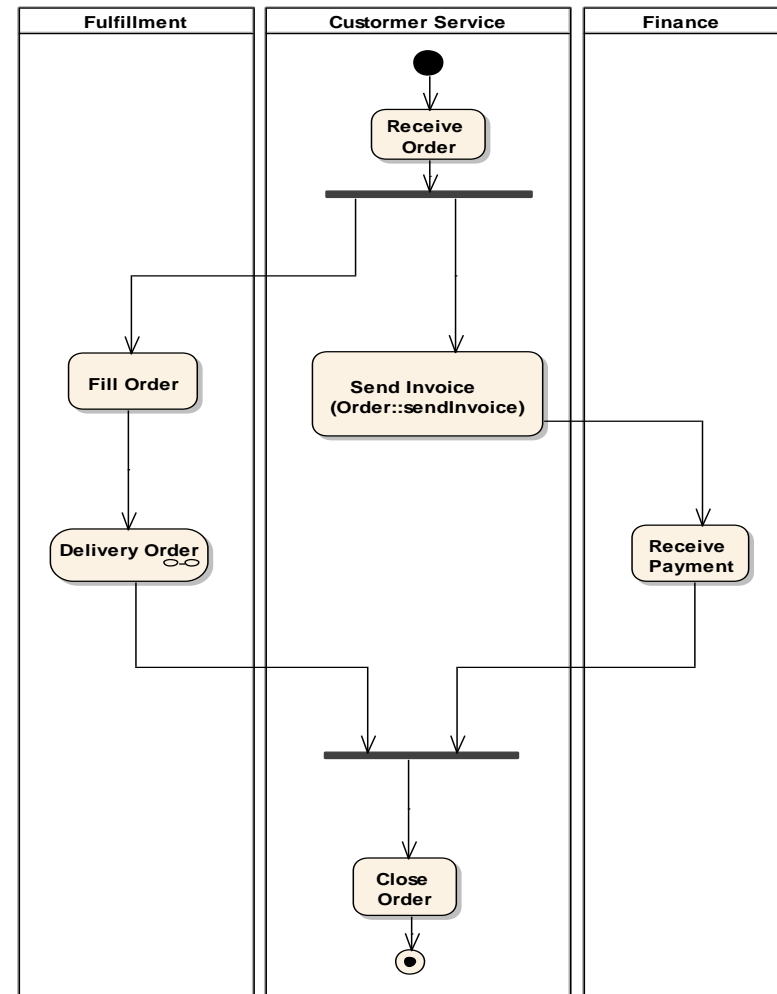
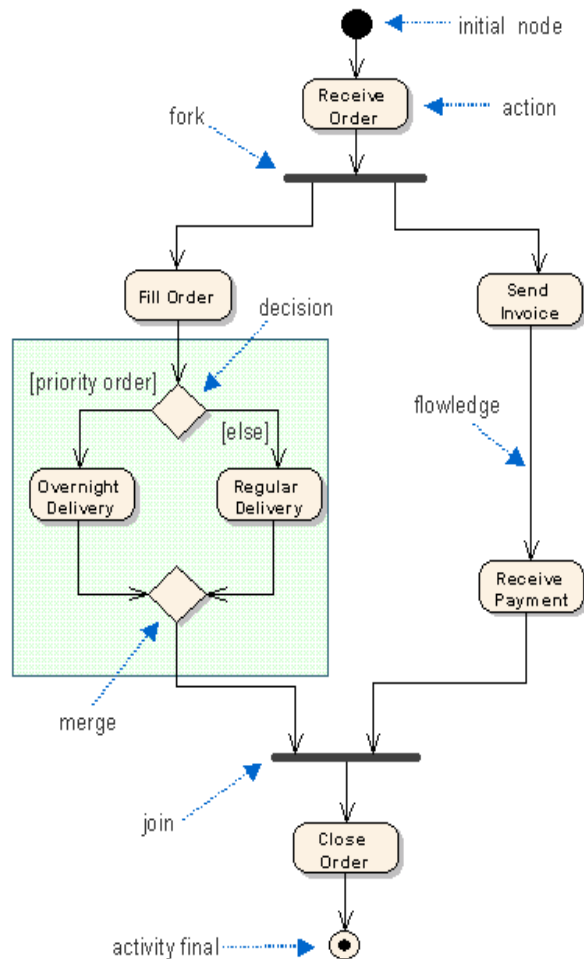


(a) 통신 다이어그램

## 6. Activity Diagram

- ❖ 절차적 로직(procedural logic), business process, 작업 흐름을 기술하는데 사용되는 기법
  - 작업(activity)의 순서를 기술
  - 작업간의 제어 흐름을 강조하여 기술
  - 본질적으로는 시간에 흐름에 따라 발생하는 작업들을 강조하는 flow chart이나, 병렬적인 행위를 지원한다는 점에서 차이
- ❖ UML 1.x 에서는 state machine diagram의 특별한 경우였으나, UML 2에서 크게 확장됨
- ❖ activity diagram에서 하나의 작업이 처리되면 그 다음 작업으로 자동적으로 옮겨지며, 작업 상태의 시작과 종료는 항상 존재해야 함

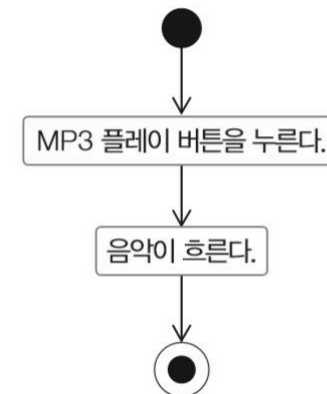
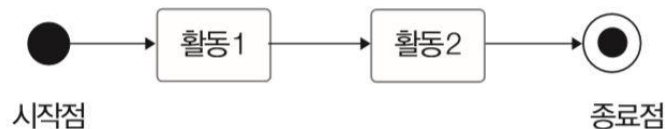
# 6. Activity Diagram



# 6. Activity Diagram

## ❖ Activity Diagram 구성요소

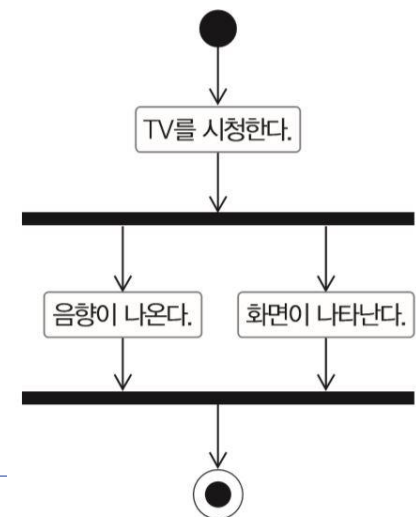
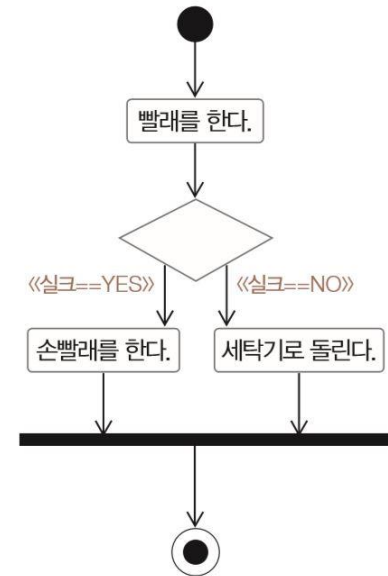
- 개시 상태(시작점) : 활동의 시작을 의미
  - 검은색 동그라미로 표현
- 활동 상태(액티비티, 액션 상태) : 어떠한 일들의 처리와 실행을 의미
  - 모서리가 둥근 사각형으로 표현
- 종료 상태(종료점) : 처리의 종료를 의미
  - 이중 동그라미로 표현
- 전이(이동) : 활동 상태에서 활동 상태로의 이동을 의미
  - 화살표 실선으로 표현





## 6. Activity Diagram

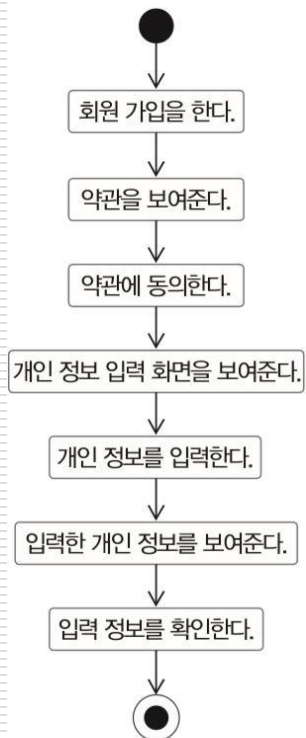
- 분기
  - 활동 흐름이 2가지 이상으로 나뉘는 것
  - 어떤 조건이냐에 따라 처리 경로가 결정
  - 흰색 마름모꼴로 표현
- 동기바(동기화 막대, Synchronization Bar)
  - 활동 다이어그램에서는 한 가지 활동만 수행하지 않고 병행해서 수행하는 경우가 있음
    - > 이럴 경우 동기바를 사용
  - 동기바는 동시 처리의 시작과 종료를 나타냄
    - 동시 처리의 시작을 포크(fork)
    - 동시 처리를 동기하여 결합하는 것을 조인(join)
  - 굵은 직선으로 표현



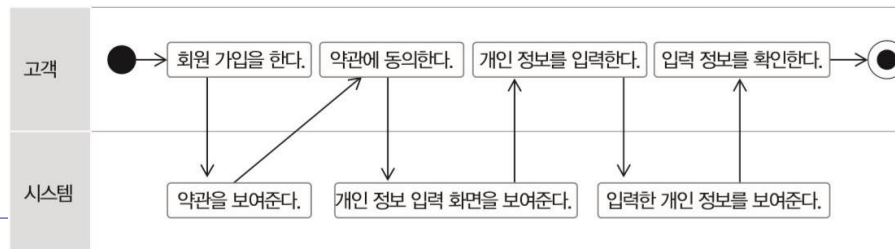
# 6. Activity Diagram

## ■ 파티션(구획면, partitions)

- Activity diagram에서 가로 혹은 세로 방향으로 그려지는 영역
- 각 활동 상태의 주체를 나타냄
- 2개 이상의 사각형으로 표시하며 이름을 기술
- Activity diagram은 어떠한 액션을 수행하는지 알 수 있지만 누가 수행하는지는 모호
  - 프로그래밍 관점 : 어느 클래스가 어떤 액션을 수행하는지 그 책임을 알 수 없음
  - 비즈니스 프로세스 모델링 관점: 각 액션을 책임지는 사람이나 부서를 알 수 없음
- 해결책
  - 각 액션의 책임 클래스 또는 사람을 표기
  - 파티션을 사용하여 책임성 부여



(a) 구획면이 없는 회원 가입 과정

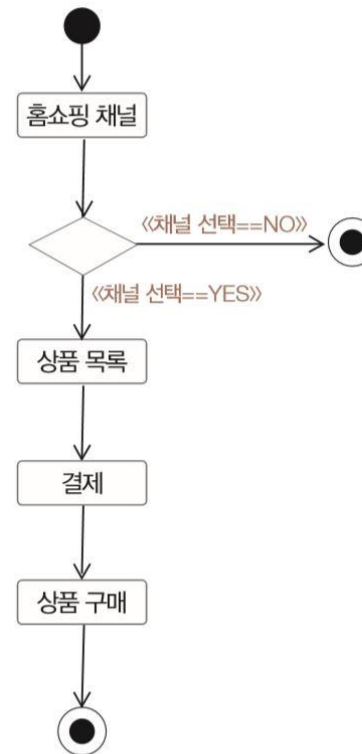


(b) 구획면으로 표현한 회원 가입 과정

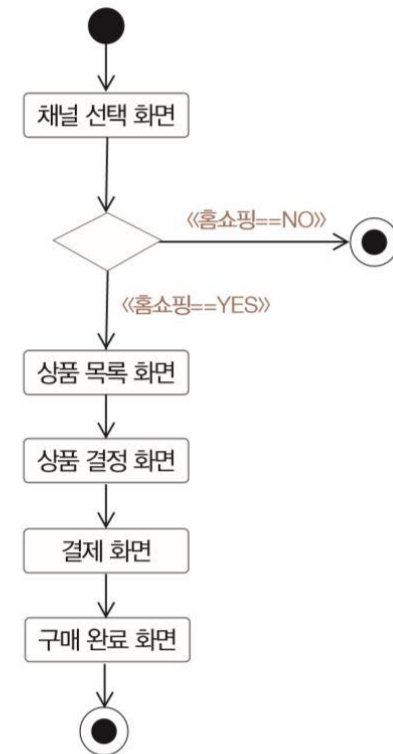
## 6. Activity Diagram

### ❖ Activity Diagram 용도

- 주로 유스케이스 수준 이상의 비즈니스 프로세스를 표현하고 분석 단계에서 유스케이스 내부에 대한 구체적인 흐름을 나타내기 위해 사용
- 설계 단계에서 클래스 내부 오퍼레이션에 대한 알고리즘이나 구체적인 로직을 표현하기 위해 사용
- interaction diagram인 sequence diagram이나 communication diagram에서 나타내기 어려운 상황을 표현할 수 있음 → 업무 흐름을 분석하거나 화면 흐름을 표현할 때 유용
- 업무 흐름을 표현할 때 가장 효과적으로 사용할 수 있음



(a) 홈쇼핑 업무 흐름 분석



(b) 홈쇼핑 화면 흐름 표현

## 7. State Machine Diagram

- ❖ 시스템의 행위를 기술하기 위해 전통적으로 사용해 오던 다이어그램
- ❖ 객체의 상태가 이벤트 발생 혹은 시간의 경과에 의해 어떻게 변화하는지를 나타냄
- ❖ 객체 지향 접근법에서는 하나의 클래스를 대상으로 작성하며, 하나의 객체의 lifetime 행위를 기술하는데 사용
- ❖ 상태의 변화는 전이(transition)을 따라 이루어지는데, 전이의 label에는 해당 전이가 활성화되는 조건이 기술
- ❖ 여러 Use case에 걸친 하나의 객체의 행위를 기술하는데 사용
  - 여러 객체가 협업하는 경우를 기술하는 데는 적합하지 않음
- ❖ State machine diagram을 사용하는 경우라도 모든 클래스에 대해서 기술하려고 말아야 함
  - 흥미로운 행위를 보이는 클래스를 더 잘 이해하기 위한 경우에 국한하여 사용

# 7. State Machine Diagram

## ❖ 상태 다이어그램의 표현

### ■ 상태

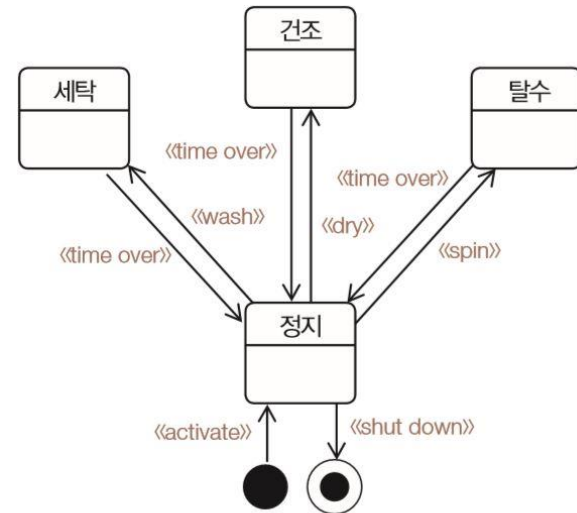
- 객체가 존재할 수 있는 조건 중 하나
- 모서리가 둥근 사각형으로 표시하고 안쪽 상단에 상태 이름을 기술
- 객체가 가질 수 있는 가능한 모든 경우가 상태로 파악되어야 하는 것이 중요함
  - 즉, 객체는 파악된 상태 외의 상태는 가질 수 없음
- 특정 순간에는 오직 한 가지 상태만 존재할 수 있음

### ■ 이벤트와 전이

- 전이(Transition): 객체의 상태가 다른 상태로 변경되는 것, 실선으로 표기
- 이벤트(event): 객체의 전이를 유발하는 자극, 전이 위에 이름 표기



(a) 세탁기 객체가 가질 수 있는 상태



(b) 세탁기 객체의 상태 다이어그램

# 7. State Machine Diagram

## ❖ 상태 다이어그램의 용도

### ■ 객체의 상태 변화를 상세히 분석

- 상태 다이어그램은 객체 하나를 대상으로 생성-소멸 기간 중 다양하게 가질 수 있는 상태를 분석하기 위해 작성됨
- 상태 다이어그램은 객체의 동적 상태 변화를 정의하고 분석하는 목적으로 사용

### ■ 이벤트에 의한 객체의 반응을 분석

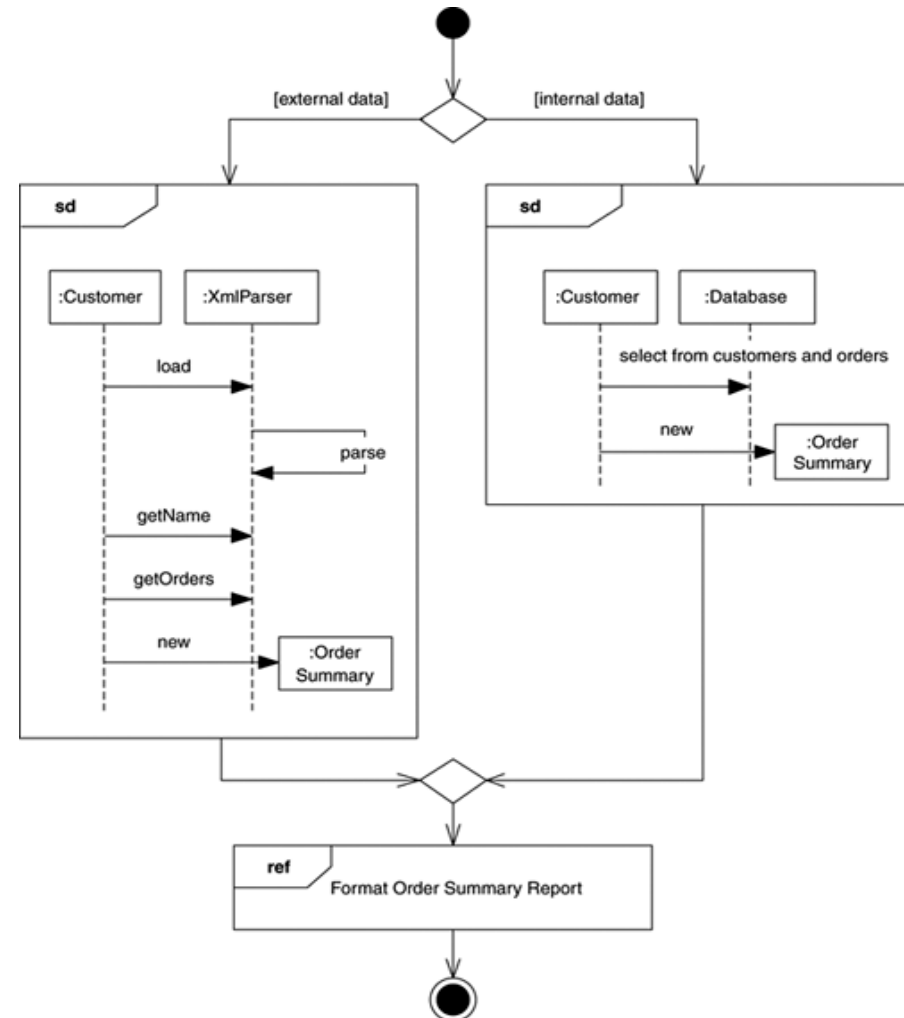
- 상태 다이어그램은 객체 상태 변화를 유발하는 이벤트를 정의하고 분석하기 위해 작성됨
- 객체 상태는 이벤트에 의해 변화되는데, 이처럼 객체의 상태 변화를 유발하는 이벤트를 식별·정의함

### ■ 객체의 속성이나 동작을 검증

- 상태 다이어그램은 객체의 속성과 동작을 검증하기 위해 작성됨
- 상태 다이어그램에서 분석 대상인 객체의 상태는 속성 값으로 정의되고, 이벤트는 대부분 객체의 동작으로 정의 → 클래스 다이어그램에서 정의된 클래스의 속성과 동작의 적합성을 검증할 수 있음

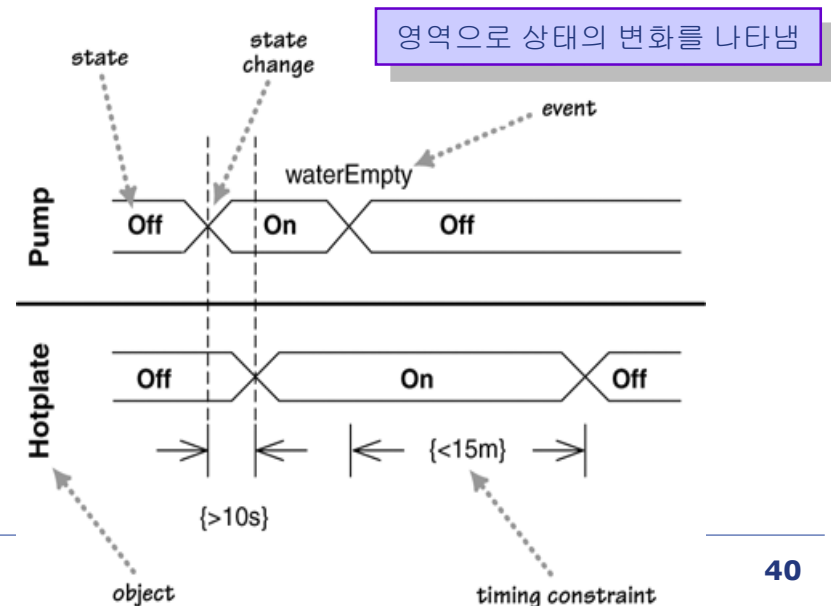
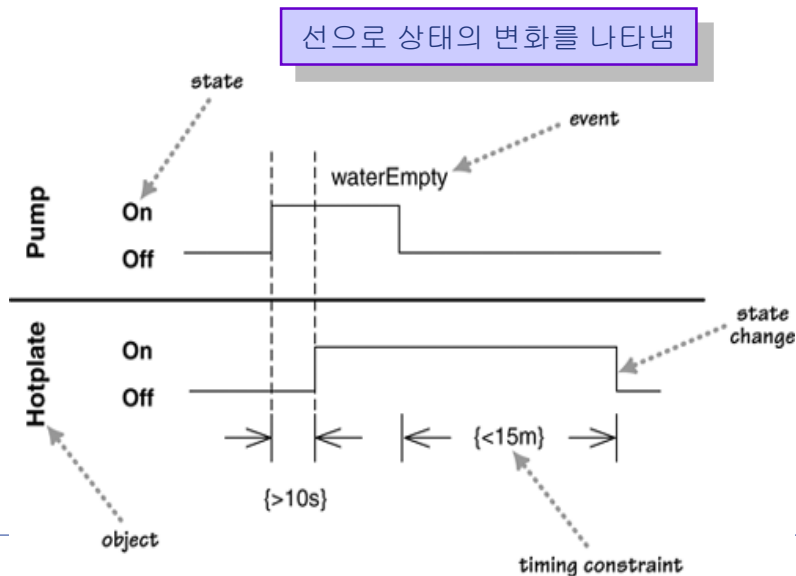
# 8. Interaction Overview Diagram

- ❖ activity diagram과 sequence diagram을 접목시킨 것
  - Activity diagram에서 activity 대신 작은 sequence diagram을 기술한 것
  - Sequence diagram을 제어 흐름을 보여주기 위해 activity diagram의 표기법에 따라 여러 개로 분할한 것
- ❖ UML 2에서 새로 추가됨



# 9. Timing Diagram

- ❖ Interaction diagram의 표현 형태 중 하나로서 객체들의 시간적 제약을 나타내는 다이어그램
  - 시간 제약의 기술에 초점을 맞춤
  - 하나의 객체 또는 여러 객체를 한꺼번에 기술할 수 있음
  - 서로 다른 객체 사이에 시간 제약 조건이 있는 경우 유용하게 사용
- ❖ UML 2에서 새로 추가됨
- ❖ timing diagram은 아래와 같은 2가지 형식으로 표현 가능





# 10. Component Diagram

## ❖ 컴포넌트 개념

- UML에서의 컴포넌트
  - 가상의 모델을 실제로 구현하여 나타내는 요소
  - 객체 지향의 원리에 따라 기능과 관련 데이터를 하나의 단위로 처리
- CBD(Component Based Development) 관점의 컴포넌트
  - 인터페이스에 의해 기능이 정의된, 독립적으로 개발·배포·조립이 가능한 시스템의 구성 단위
  - 컴포넌트 예: J2EE 플랫폼의 JAR 파일과 닷넷 플랫폼의 DLL 파일 등

# 10. Component Diagram

- ❖ Component Diagram은 소프트웨어를 물리적으로 어떻게 구현할 것 인지를 정의하고 모델링하는 다이어그램
  - 구체화된 class diagram의 완료 후 class들 간의 기능적인 연관성을 고려하여 결합력이 강한 class들을 하나의 component로 묶는 것
  - 대형 system 개발 시 복잡한 system을 이해하기 쉽고 변경하기 쉬운 작은 하위 system으로 분할, 새로운 단위의 하위 system → "component"
  - system의 component 혹은 package와 그들의 관계에 대한 종속 관계를 보여주는 diagram
  - 각 component는 또 다른 component들과 interface를 통해 연관
- ❖ 컴포넌트와 클래스의 구별에 대해 논쟁이 있어왔음
  - UML 1의 경우 컴포넌트를 위한 고유의 심볼이 있지만,
  - UML 2에 와서는 클래스의 기술에 사용되는 사각형을 사용하여 기술하고 유사한 심볼을 붙여 컴포넌트임을 표시
    - 이러한 심볼 대신에 "<<component>>"를 키워드로 부기해도 됨
- ❖ 하나의 컴포넌트는 Composite Structure Diagram으로 상세하게 표현할 수 있음

# 10. Component Diagram

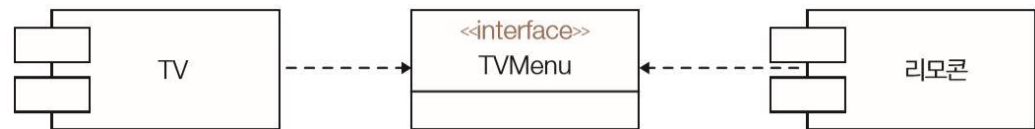
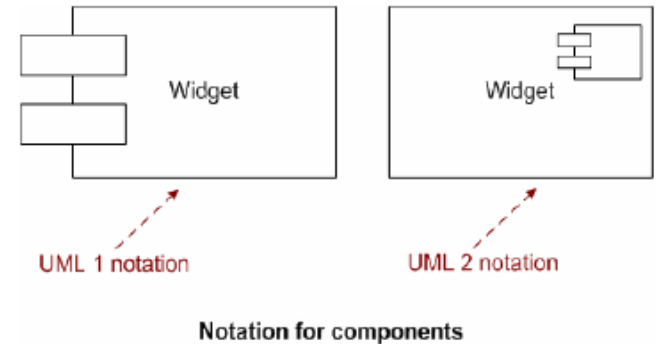
## ❖ 컴포넌트 다이어그램의 표현

### ■ 컴포넌트

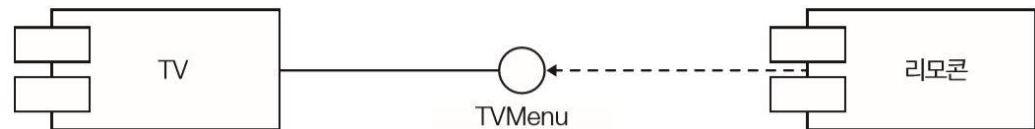
- 탭이 달린 직사각형으로 표현
- 모든 컴포넌트에는 반드시 이름이 필요

### ■ 인터페이스 : 두가지 방법으로 표기

- 컴포넌트와 인터페이스의 의존 관계로 표현
- 실제로 동작하는 컴포넌트에 인터페이스를 적용하여 표현 (인터페이스 실체화)



(a) 컴포넌트와 인터페이스의 실체화와 의존 관계 표현



(b) 인터페이스 실체화 표현

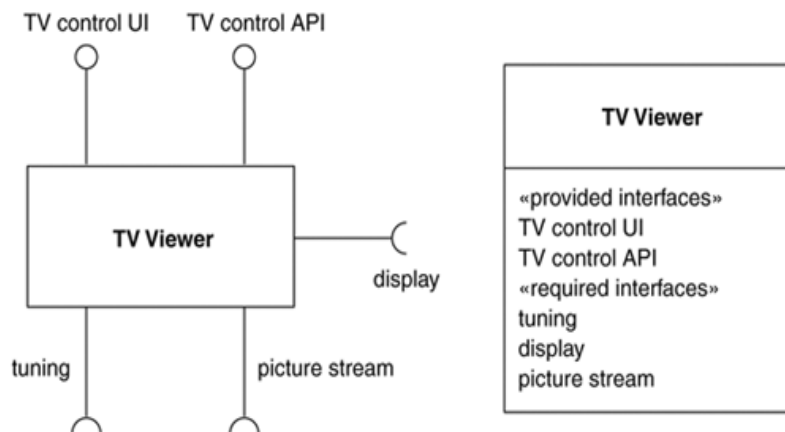
# 10. Component Diagram

## ❖ 컴포넌트와 클래스

- 컴포넌트와 클래스의 공통점
  - 둘 다 이름이 있고 정해진 인터페이스를 실현할 수 있음
  - 의존성이 있고 일반화가 가능
  - 연관 관계와 교류에 참여할 수 있고 중첩이 가능하며 인스턴스를 가질 수 있음
- 컴포넌트와 클래스의 차이점
  - 클래스는 논리적인 추상화이지만 컴포넌트는 물리적인 요소
  - 컴포넌트는 클래스나 통신과 같은 서로 다른 논리적 요소들을 물리적으로 패키지화한 것
  - 클래스는 속성과 오퍼레이션을 직접 가질 수 있지만, 컴포넌트는 자신의 인터페이스를 통해 접근할 수 있는 오퍼레이션들만 가짐

# 11. Composite Structure Diagram

- ❖ UML 2에서 새로 추가됨
- ❖ 하나의 클래스를 내부 구조로 분해할 수 있게 함
  - 복잡한 객체를 분할하여 여러 부분으로 나누어 표현할 수 있음
  - 각 구성요소들과 그 요소들이 어떻게 분리/연결되는지를 표현
  - 복잡한 객체를 부분들로 분해



# 11. Composite Structure Diagram

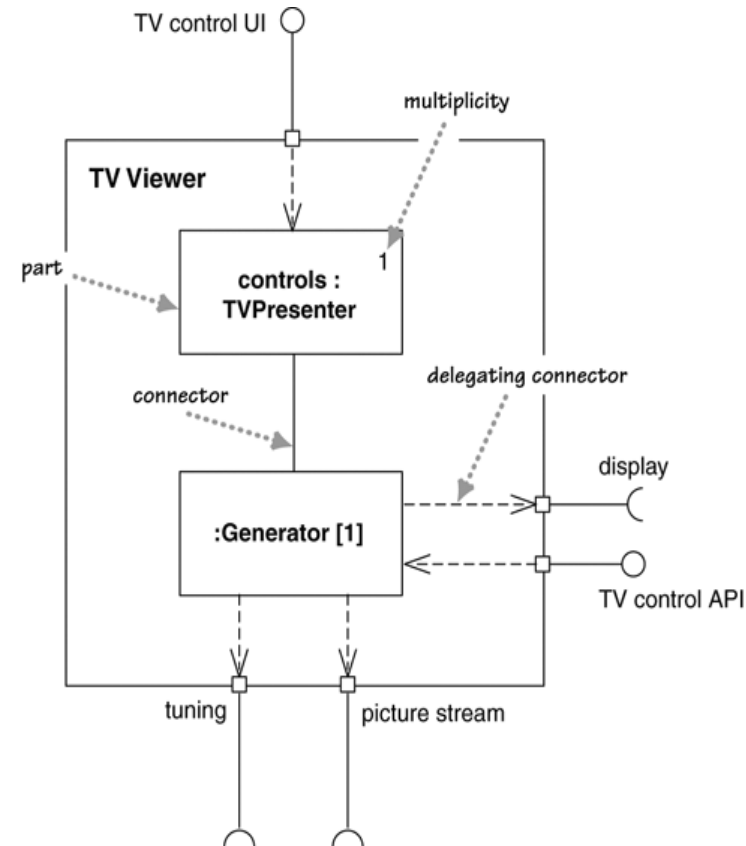
## ❖ 그림

- 내부적으로 두 부분으로 분석되는 방법과 어느 부분들이 각각 인터페이스를 요청하고 지원하는지 보여줌
- 각 부분들은 name: class 형태로 이름 붙여짐
  - 밑줄을 그어 표기 하지 않고, 굵게 표기

## ❖ Package와 Composite Structure의 구별

- Packages : compile-time grouping
- Composite Structure : run-time grouping

- ## ❖ 하나의 컴포넌트가 어떻게 내부적으로 분할되는가를 표현하는데 사용되므로, Component diagram과 공통적으로 사용하는 표기법이 많음(Component diagram에 주로 나타남)

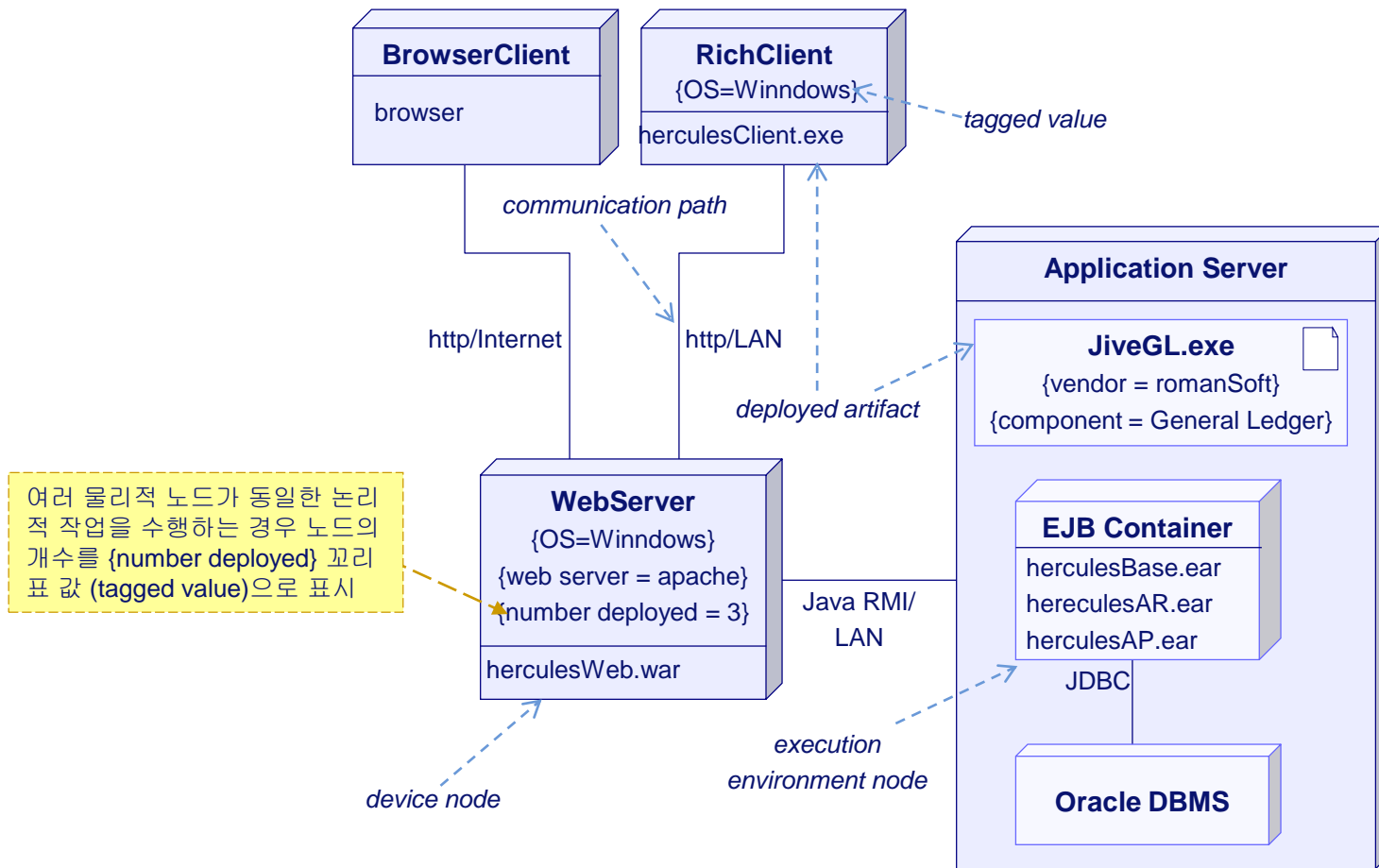


컴포넌트의 내부 뷰

## 12. Deployment Diagram

- ❖ 시스템의 물리적인 배치(layout)를 기술
- ❖ 어떤 소프트웨어 부분이 어떤 하드웨어에서 실행되는가를 표현
- ❖ 네트워크, 하드웨어 또는 소프트웨어들을 실행 파일 수준의 컴포넌트와 함께 표현한 것
  
- ❖ Node와 communication path로 구성
  - Node는 소프트웨어를 운용할 수 있는 것을 나타내며,
    - device node : 하드웨어 혹은 시스템에 연결된 하드웨어의 간단한 부분
    - Execution environment node : 그 자신이 다른 소프트웨어를 운용할 수 있는 소프트웨어를 나타냄
  - Node는 artifact를 포함
    - 주로 file을 의미 - 실행 파일, 데이터 파일, 설정 파일, 문서 파일 등
    - Node에 artifact를 나열하는 것은 그것들이 그 node 에 배치됨을 나타냄
  - Communication path는
    - node를 연결하며, node 간에 어떻게 통신하는지를 나타냄
    - 사용되는 프로토콜의 이름을 붙이기도 함

# 12. Deployment Diagram





# 12. Deployment Diagram

## ❖ 배치 다이어그램의 목적

- 다른 다이어그램과 달리 하드웨어 자원들을 명시적으로 정의하는 용도로 작성
  - 다른 다이어그램과 달리 하드웨어 자원들을 명시적으로 정의하는 용도로 작성
- 배치 다이어그램은 실행 컴포넌트를 어떤 분산 하드웨어 자원에 배치하여 원하는 성능과 효율을 낼지 정의하기 위해 작성
- 배치 다이어그램은 어떤 하드웨어 자원 간에 연결이 있는지, 그 연결은 어떠한 성능을 지닌 연결인지 정의

# 13. Package Diagram

- ❖ 복잡한 시스템을 이해하는 방법은 추상 개념들을 하나의 그룹으로 만들어 봄
  - => UML에서는 이러한 추상 개념들이 모인 하나의 그룹을 패키지라고 함
    - Grouping의 구성체로 임의의 UML 요소를 취하여 더 상위 level 단위로 모으기 위한 grouping 요소
    - Class 모임을 구조화하기 위해 가장 많이 사용
- ❖ 규모가 큰 시스템의 경우, 시스템을 구성하는 주요 요소들 간의 의존성을 나타내는데 package diagram을 사용하여 이해를 도울 수 있음
- ❖ package diagram은 일반적인 프로그램 구조와 일치

