II. 객체지향 분석/설계 방법

넥스젠 테크놀러지(주)
객체기술 컨설팅 사업부

목차

II - 1. 요구사항 분석

- 요구사항 파악
- 유즈케이스 모델
- 유즈케이스 다이아그램
- 유즈케이스 명세서
- 유즈케이스 레포트
- 액티비티 다이아그램
- 용어해설
- 부가사항 기술서

목차

II - 2. 유즈케이스 분석

- 유즈케이스 실체화
- Object
- Object Interaction
- 클래스 추출
- 분석 클래스

II - 3. 아키텍쳐 설계

- Package
- Subsystem
- Design Pattern
- Layered Architecture
- Object Model vs Relational Tables
- Reverse Engineering Relational Tables
- Mapping Object to Relational Tables
- Object in jasmine OODB

II - 4. 클래스 설계

- 클래스
  - 오퍼레이션, 애트리뷰트
  - Utility, Abstract, Concrete
- 관계성
  - Multiplicity, Navigability,
  - 연관관계, 포함관계, 상속관계, 종속관계
- Statechart Diagram

목차

II - 5 컴포넌트 설계 및 배치

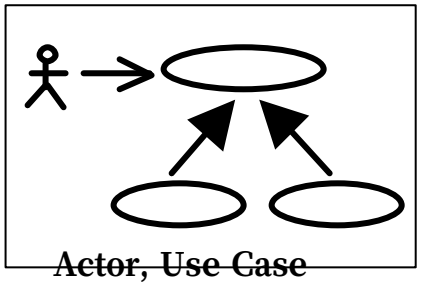
- Logical vs Physical Architecture
- Component and Interface
- Component Diagram
- Porcess Modeling
- Deployment Diagram

II-1. 요구사항 분석

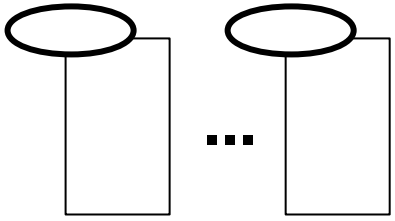
- 요구사항 파악
- 유즈케이스 모델
- 유즈케이스 다이아그램
- 유즈케이스 명세서
- 유즈케이스 레포트
- 액티비티 다이아그램
- 용어해설
- 부가사항 기술서

산출물

유즈 케이스 모델

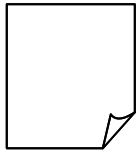


Actor, Use Case



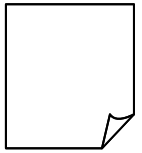
Use Case Reports

용어해설집



glossary

부가사항 기술서



Supplementary Specification

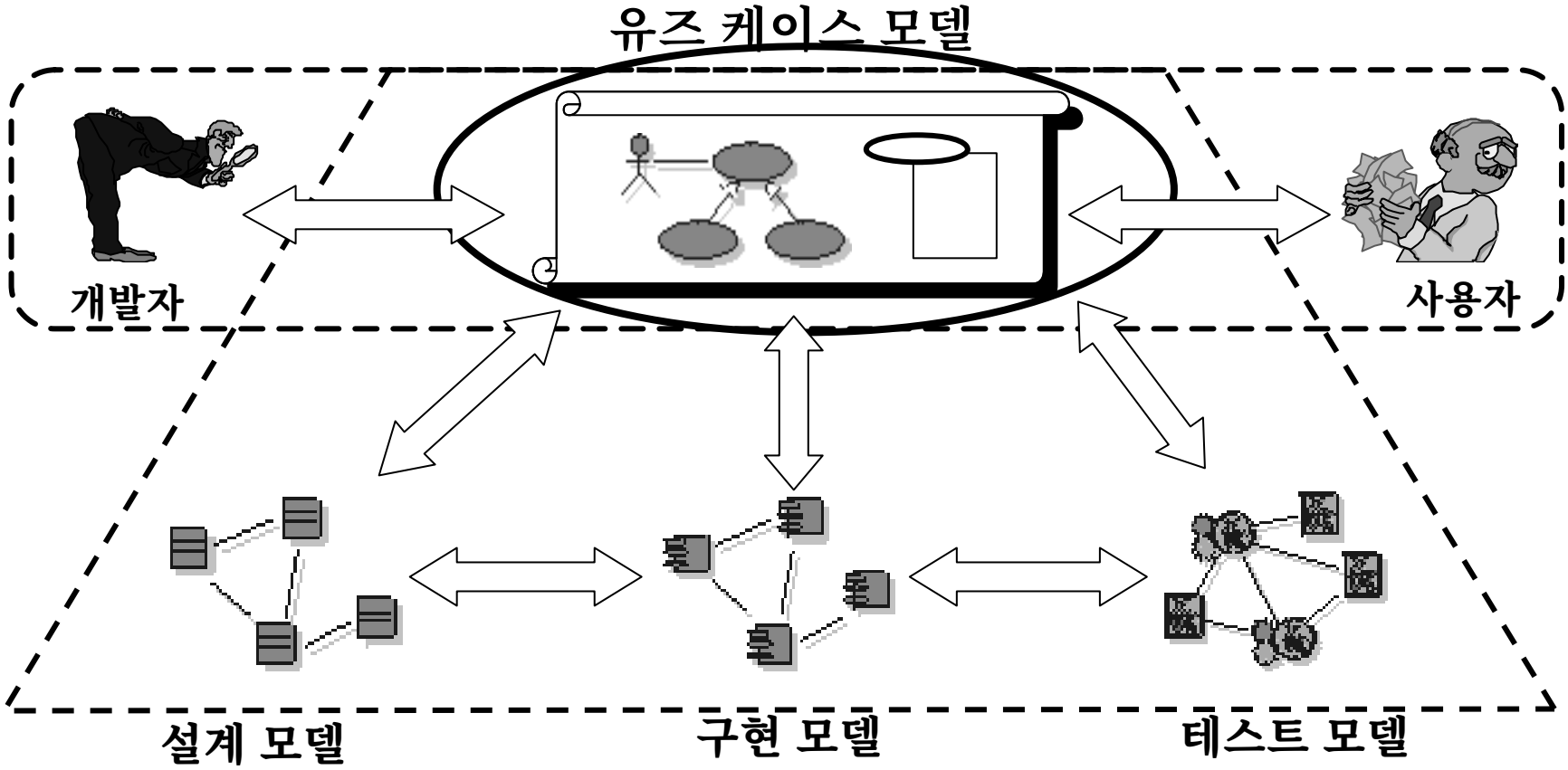
사용자 인터페이스 프로토 타입

요구사항 파악

- 요구사항 기술서 작성
  - 의의
    - 고객의 요구사항은 업무 시스템 개발에서 부터 테스트 전반에 걸쳐 중요
  - 도출방법
    - 최종 사용자가 현 시스템의 문제점 및 구축되어야 할 업무 시스템에 대한 요구사항 등을 작성
    - 시스템 분석설계자 사용자와의 면담, 설문, 기존 시스템 분석을 통해 작성
  - 필요사항
    - 구축되어야 할 업무시스템의 영역이 명확히 정의
    - 사용자의 검증

유즈 케이스 모델

- 분석, 설계, 구현, 테스트 등 모든 시스템 개발주기 동안 분석가, 개발자와 사용자간의 의사소통에 대한 기준 제공

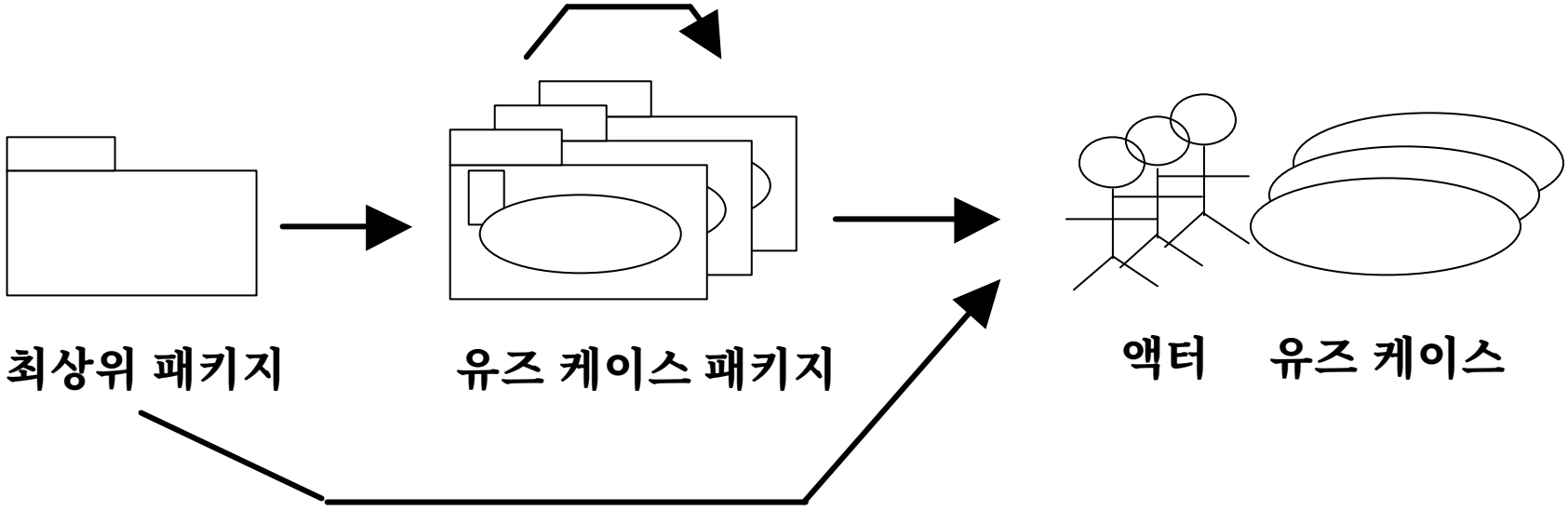


유즈 케이스 모델 (계속)

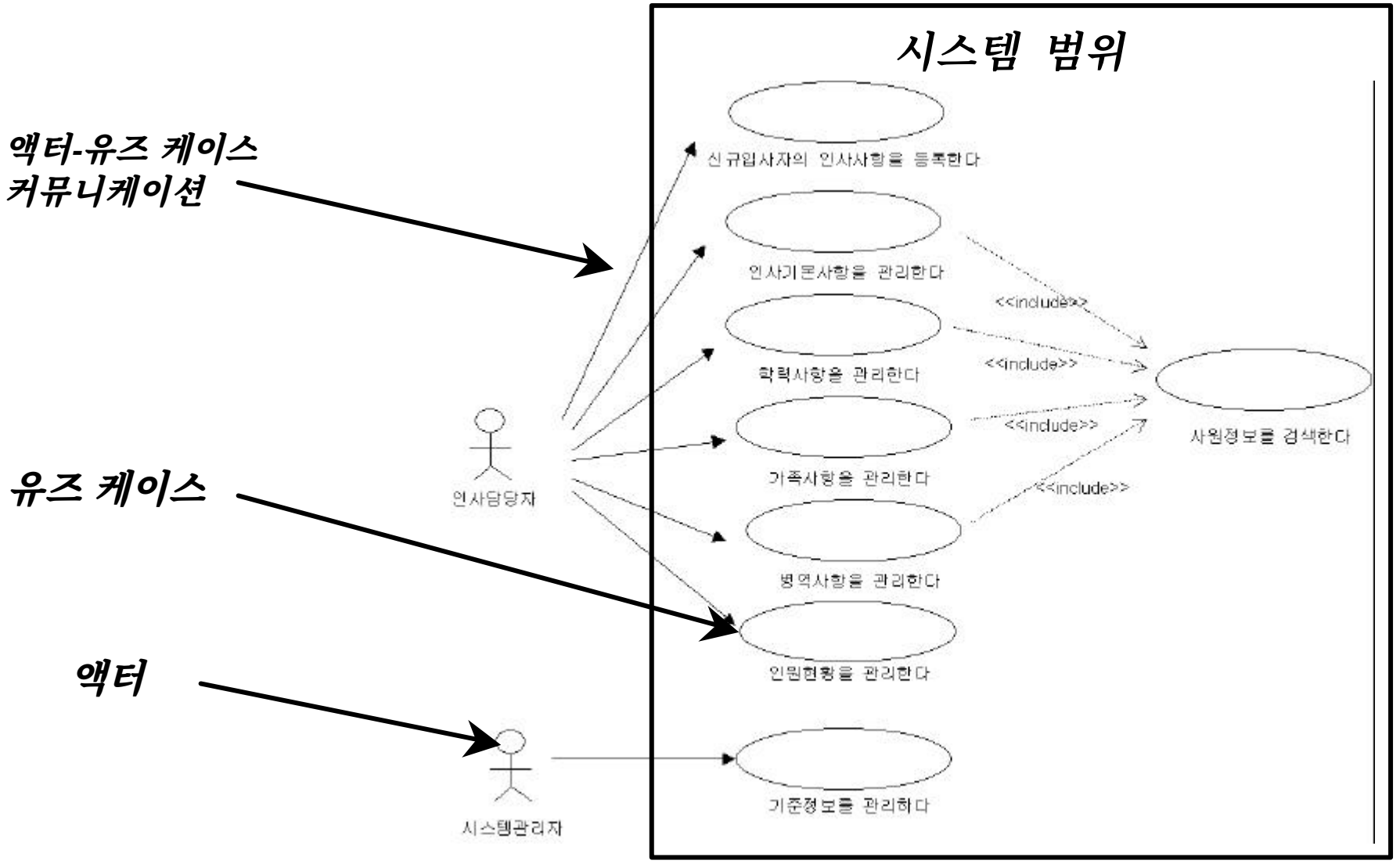
- 장점
  - 시스템 사용자, 업무 영역의 전문가와 시스템 개발자 간의 의사 소통 수단 제공
    - 요구사항에 대한 명확한 이해 가능
  - 다음의 사항을 파악하는데 용이
    - "누가(Who)" 시스템을 사용할 것인가 (시스템과의 상호작용) ?
    - 시스템은 사용자를 위해 "무엇(What)"을 해야 하는가 ?
    - 사용자와의 상호작용을 위해 시스템이 제공해야 할 "인터페이스 (Interface)"는 무엇인가 ?
  - 사용자의 모든 요구사항이 제대로 반영되었는지 검증(Verification) 하는 기회 제공

유즈 케이스 모델 (계속)

- 유즈 케이스 모델의 계층구조
  - 모델이 복잡하거나 여러 명의 분석가가 동시에 작업할 경우 업무적으로 연관된 유즈 케이스를 패키지로 분류 가능

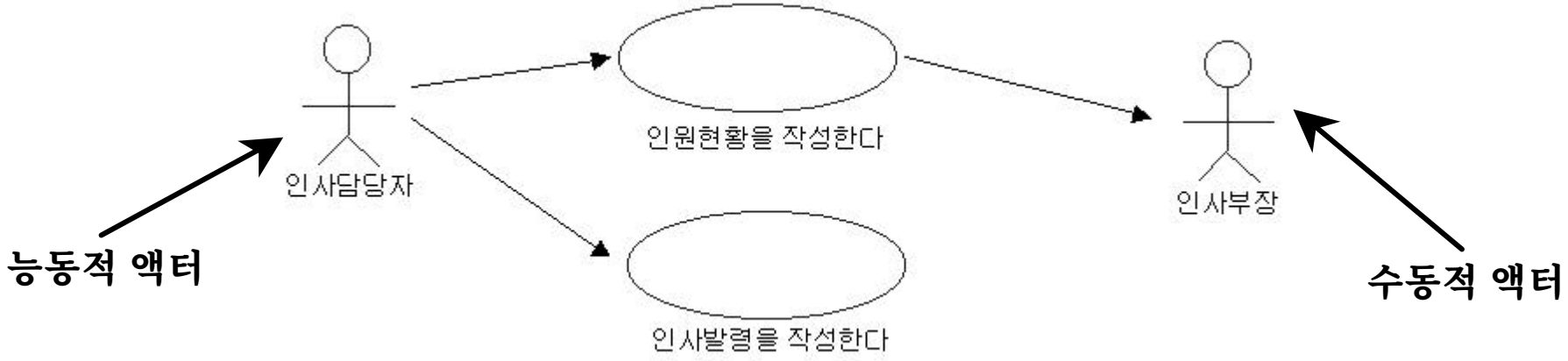


유즈 케이스 다이아그램


액터-유즈 케이스
커뮤니케이션
유즈 케이스
액터
시스템 범위
신규입사자의 인사사항을 등록한다
인사기본사항을 관리한다
학력사항을 관리한다
가족사항을 관리한다
병역사항을 관리한다
인원현황을 관리한다
기준정보를 관리하다
사원정보를 검색한다
<<include>>
<<include>>
<<include>>
<<include>>
인사담당자
시스템관리자

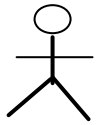
유즈 케이스 다이아그램 (계속)

- 유즈 케이스는 적어도 하나 이상의 액터와 관계를 가지며,
- 모든 액터는 적어도 하나의 유즈 케이스와 관계를 가진다.
  - 능동적 액터(Active Actor) : 유즈 케이스를 초기화시키는 액터
  - 수동적 액터(Passive Actor) : 유즈 케이스로부터 결과를 넘겨 받는 액터 , 인터페이스하는 외부 시스템



액터

표기법



액터 명 (명사)

Actors are external !!

정의

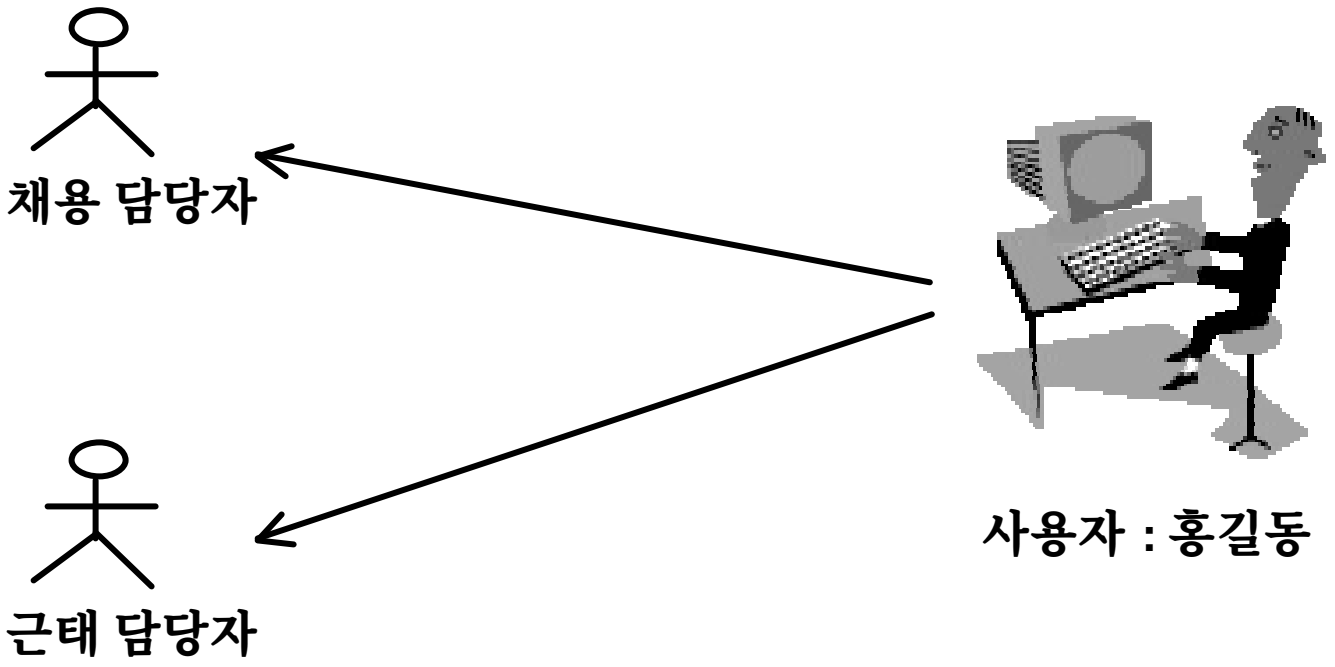
시스템과 상호작용 하는 것(Thing; 사람과 사물 모두 포함)

특징

- 시스템 사용자의 역할(Role)을 의미
- 반드시 시스템 외부에 존재
- 능동적으로 시스템과 정보 교환/제공 가능
- 시스템으로 부터 일방적인 정보 수신 가능(수동적 액터)
- 유즈 케이스를 동작시키는 요소
- 사람, 기계, 또는 다른 시스템이 될 수도 있음

액터 (계속)

- 액터는 시스템을 사용하는 사용자의 역할이다.



- 한 명의 시스템 사용자는 시스템에 대해 여러 개의 역할을 가질 수 있다.

액터 (계속)

액터를 찾기 위한 6가지 질문

- 업무 시스템의 주요 기능(main functionality)을 사용할 사람은 누구인가 ?
  (주 액터)
- 매일 고정적으로 하는 일(daily task)을 처리하기 위해
  업무 시스템의 지원을 필요로 하는 사람은 누구인가 ?
- 업무 시스템을 유지하고 관리해야 할 필요가 있는 사람은 누구인가 ?
  (보조 액터)
- 업무 시스템에서 필요한 하드웨어 장치는 무엇인가 ?
- 해당 시스템과 상호 작용할 필요가 있는 다른 시스템은 무엇인가 ?
- 시스템의 처리 결과에 관심이 있는 사람 또는 사물은 무엇인가 ?

액터 (계속)

기술 방법

액터 명

시스템과의 상호작용에서의 역할(Role) 중심으로 명명

간단한 설명

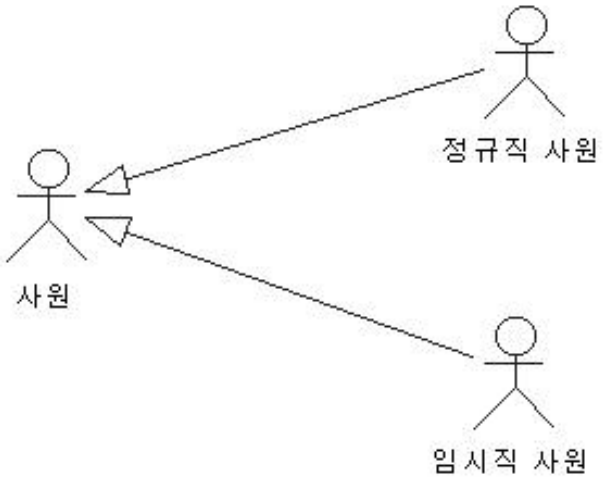
- 액터가 상징하는 것이 무엇[누구]인가 ?
- 이 액터는 왜 필요한가 ?
- 시스템에서 이 액터가 관심을 갖고 있는 것은 무엇인가 ?
- 너무 길게 쓰지 말고 간략히 작성할 것 (3~4 Line 이내)

예) 입사지원자

- 회사에 입사지원서를 접수시키고자 하는 사람들로서, 이를 위해 채용시스템을 사용해야 하며, 입사지원서를 입력하고 조회하는데 관심이 많음

액터 (계속)

- 일반화(Generalization) 관계
  - 어떤 액터가 다른 액터의 역할을 상속 받아 구체화될 경우, 액터 간의 관계도 일반화(Generalization) 관계로 표현



유즈 케이스

표기법

유즈 케이스 명 (간략한 서술형)

정의

- 액터가 해당 시스템을 통해 원하는 결과를 얻어내기까지 시스템 내부에서 이루어지는 이벤트들의 흐름

특징

- 항상 시스템이 제공하는 특정 기능과 연관된 액터에 의해 시작
- 액터와 시스템 간의 상호작용을 표현
- Use Case들의 총합은 시스템 사용시 가능한 모든 경우를 표현한 것임
- 단순한 기능의 열거가 아닌 이벤트들의 흐름
- 시작과 끝이 명확하며 액터에게 결과물을 제공해야 함

유즈 케이스 (계속)

주의 사항

- 액터의 관점에서 기술할 것
  - '입사를 지원한다'
- 전산용어를 피하고, 사용자에게 친숙한 용어 사용
  - '새로운 Parameter를 추가한다'(X); '새로운 응시분야를 추가한다'(O)
- 명사가 아닌 서술형 사용
  - '일일 결산 보고서'(X); '일일 결산 보고서를 출력한다'(O)

유즈 케이스 도출을 위한 정보원

- 시스템 사양, 요구사항 기술서
- 업무영역에 관련된 책자, 해당 분야 전문가[실무자]와의 인터뷰
- 기존 시스템 분석

유즈 케이스 (계속)

- 유즈 케이스를 찾기 위한 4가지 질문
  - 액터가 시스템에 요구하는 기능에는 어떤 것들이 있는가 ?
  - 액터는 시스템 내에 있는 특정 정보에 대해 조회, 생성, 삭제, 수정, 저장 해야 할 필요가 있는가 ?
  - 액터는 시스템 내부에서 발생하는 이벤트들에 대해 통보 받거나 시스템 외부에서 발생하는 특정 이벤트를 시스템에 통보해야 할 필요가 있으며, 이러한 이벤트들은 어떠한 기능(functionality)으로 표현할 것인가 ?
  - 시스템이 제공하는 새로운 기능들을 통해 액터가 매일 고정적으로 하는 일(daily work)들을 단순화하거나 더욱 효율적으로 처리할 수 있는가 ?

유즈 케이스 (계속)

✍ 유즈 케이스간의 관계 : 포함(Include)

✍ 목적

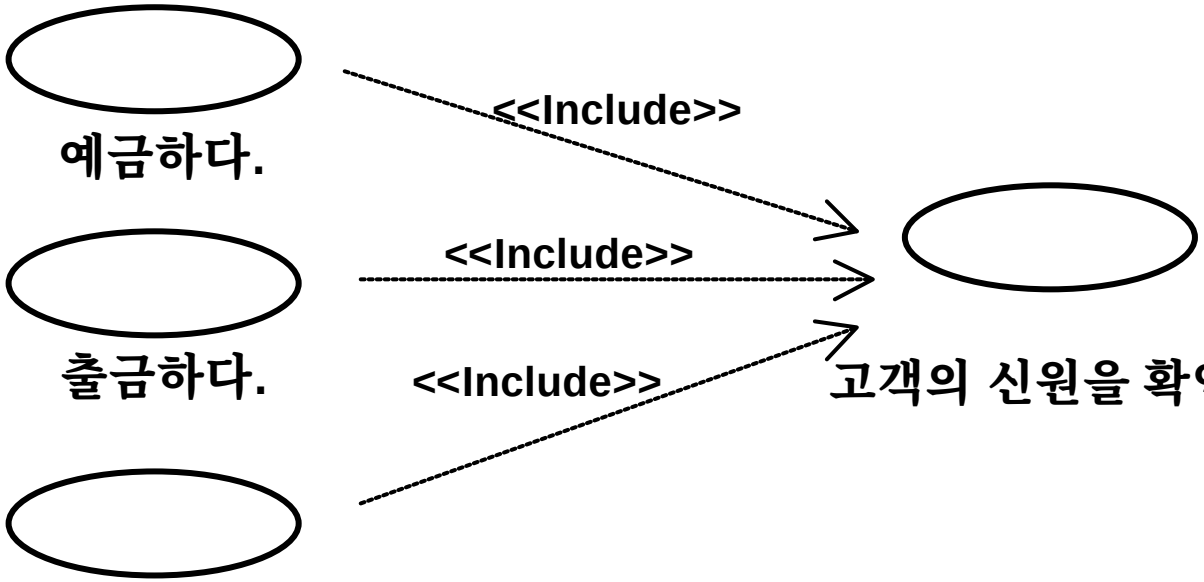
✍ 두개 이상의 유즈 케이스(Base Use Case)에서 공통된 이벤트 흐름이 존재할 때, 공통된 이벤트 흐름을 별도의 유즈 케이스(Inclusion Use Case)를 추출

기본 유즈 케이스 포함 유즈 케이스

✍포함 유즈 케이스는 Abstract!
✍어떤 액터와도 연관관계를 가질 수 없다.

유즈 케이스 (계속)

- 유즈 케이스간의 관계 : 포함(Include)
  - 예: ATM 시스템



- 유즈 케이스 “예금하다.”, “출금하다.”, “계좌이체하다.” 는 “고객의 신원을 확인한다.” 라는 유즈 케이스를 포함한다.

유즈 케이스 (계속)

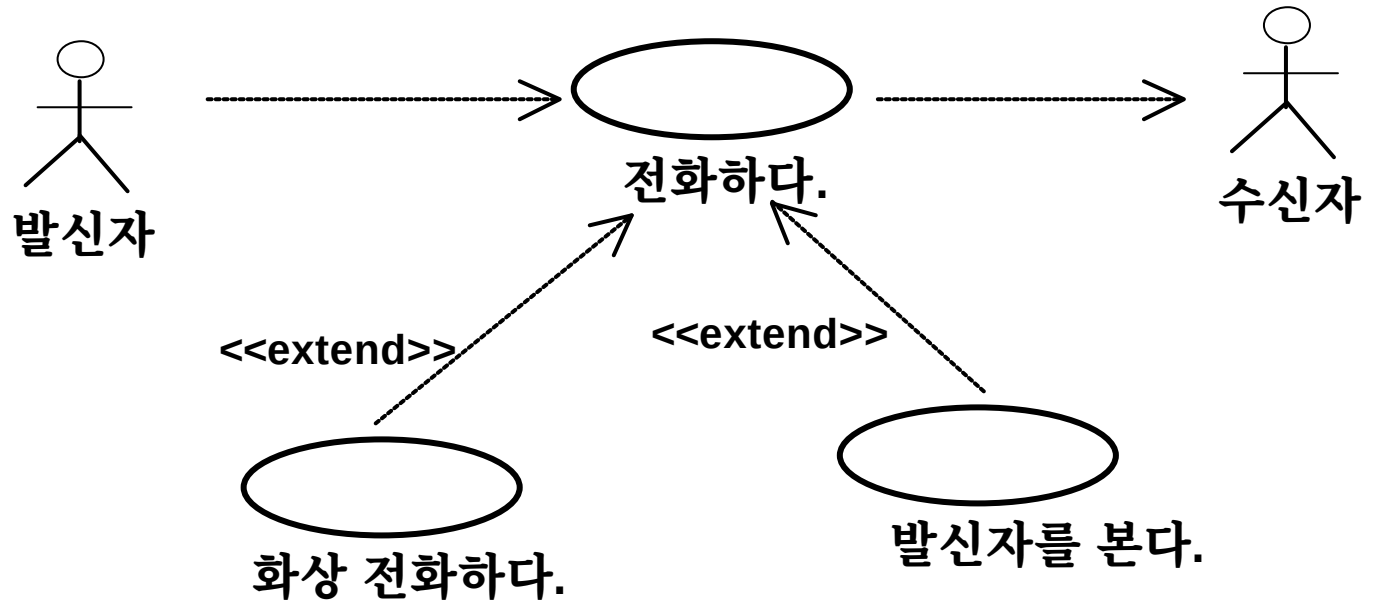
- 유즈 케이스간의 관계 : 확장(Extend)
  - 목적
    - 유즈 케이스의 일부분이 Optional 할 경우
    - 유즈 케이스의 일부분이 특정 조건에만 의해서 발생할 경우
    - 기존의 유즈 케이스 (Base Use Case)를 수정하지 않고 새로운 요구사항을 추가로 표현하고자 할 경우

기본 유즈 케이스 확장 유즈 케이스

- 확장 유즈 케이스의 수행은 기본 유즈 케이스의 속성 변경 가능 but, 반대의 경우는 불가

유즈 케이스 (계속)

- 유즈 케이스간의 관계 : 확장(Extend)
  - 예 : 전화 시스템



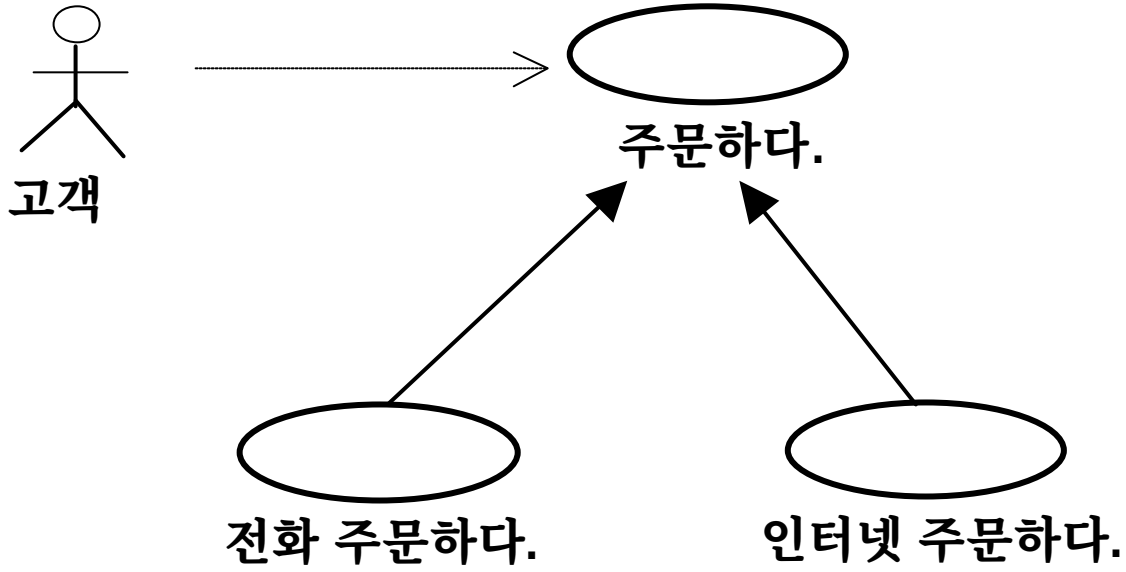
- "화상 전화하다.", "발신자를 본다"는 유즈 케이스 "전화하다" 에 대한 확장 유즈 케이스 이다.

유즈 케이스 (계속)

유즈 케이스간의 관계 : 일반화(Generalization)

목적

- 구체화된 유즈 케이스(Child Use Case)는 일반화된 유즈 케이스(Parent Use Case)의 기능과 의미를 상속 받고, 자신만의 기능을 추가하거나, 재정의하여 하나의 완전한 유즈 케이스를 만든다.



유즈 케이스 명세서

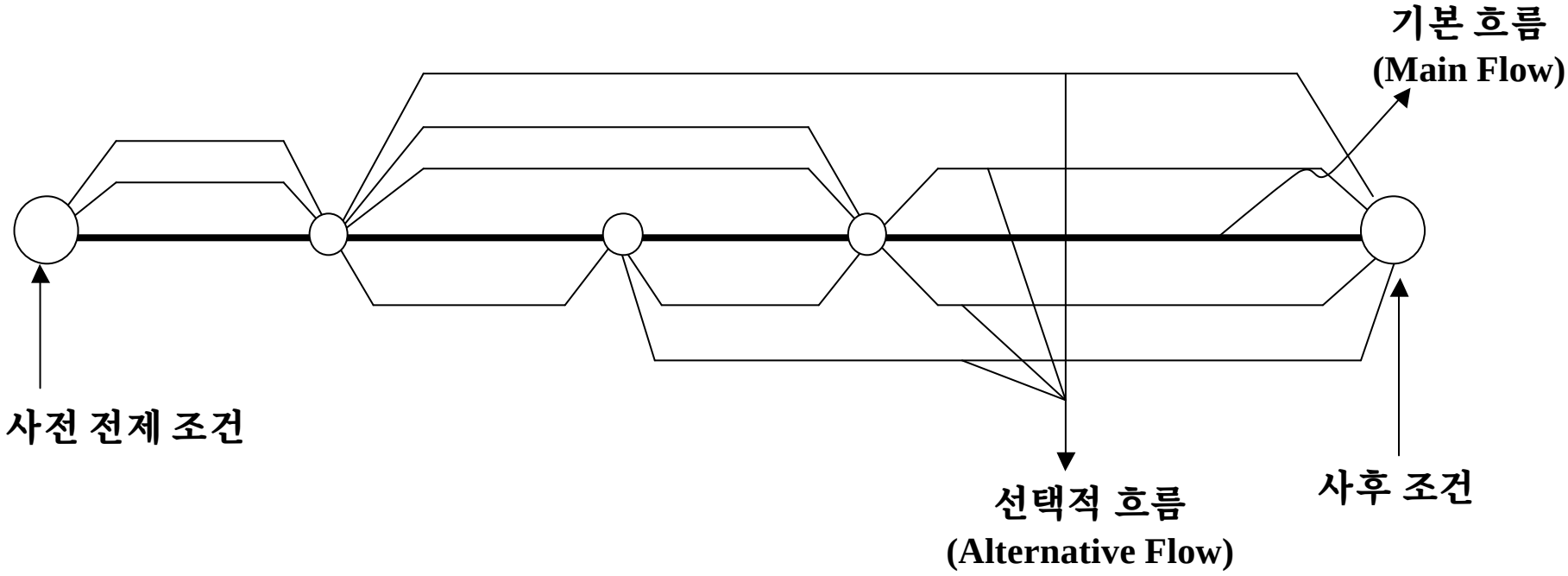
- 이벤트 흐름의 표현 방법
  - 유즈 케이스 레포트
    - 서술형으로 기술
    - 유즈 케이스에 대한 상세 내용을 체계적으로 정리 가능
  - 액티비티 다이아그램
    - 다이아그램 형태로 유즈 케이스의 이벤트 흐름을 표현
    - 간결하게 표현 가능

유즈 케이스 레포트

- '유즈 케이스 레포트' 기술 방법
  - 유즈 케이스 이름
    - 액터의 관점에서 서술형으로 명명
  - 간단한 설명
    - 액터가 무엇[누구]인가 ?
    - 액터가 시스템을 사용하는 목적은 무엇이며, 이를 위해 어떠한 이벤트들이 순서적으로 발생하는가 ?
    - 전체적인 이벤트의 흐름이 파악되도록 작성
    - 사용자에게 친숙한 용어를 사용
  - 이벤트 흐름
    - 'Basic flow of events(Normal Case)'와 'Alternative flow of events(Abnormal Case)'로 구성
  - 요구 사항
    - 유즈 케이스를 위해 필요한 요소(성능, 보안, 신뢰성 등)들을 기술

유즈 케이스 레포트 (계속)

✍ 유즈 케이스 이벤트 흐름



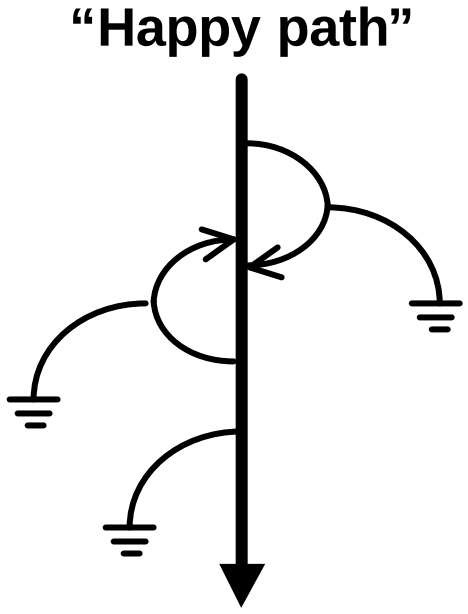
OTC 유즈 케이스 레포트 (계속)

✍ 유즈 케이스 이벤트 흐름 (계속)

- ✍ 해당 유즈 케이스에 속하는 이벤트들만 기술
  - ✍ 다른 유즈 케이스에 속한 이벤트들에 대해서는 무시
- ✍ "예를 들면", "기타", "정보" 등과 같은 용어는 쓰지 말 것
- ✍ 반드시 있어야 할 사항들
  - ✍ 해당 유즈 케이스가 언제, 어떻게 시작되고, 종료 되는가 ?
  - ✍ 해당 유즈 케이스가 언제 액터와 상호작용(Interaction) 하는가 ?
  - ✍ 액터와 유즈 케이스간에 어떤 정보가 교환 되는가 ?
  - ✍ 이벤트의 기본 흐름
  - ✍ 이벤트의 예외 흐름

유즈 케이스 레포트 (계속)

- 유즈 케이스 이벤트 흐름(계속)
  - 기본 흐름 : One Normal Case, “Happy Path”
  - 예외 흐름 : Several Cases, Exception Flows 포함



- 기본과 예외를 정확히 구분하기 보다는
  읽기 용이하고(Readability)과
  이해 하기 쉽게(Understandability)
  이벤트 흐름을 서술하는 것이 핵심!!

유즈 케이스 레포트 (계속)

- 유즈 케이스 문서를 읽는 사람들
  - 고객
  - 시스템 사용자
  - 시스템 개발자
  - 검토자
  - 시스템 분석/설계자
  - 시스템 테스터
  - 프로젝트 리더 및 관리자

유즈 케이스 레포트 (계속)

✍ 유즈 케이스 레포트 <사원의 인사기본사항을 관리한다.>

1. Brief Description

이 Use Case는 Use Case <사원 인사정보를 유지/관리한다.>에서 인사부 담당자가 인사기본사항을 선택함으로써 시작된다. 이 Use Case에 의해 등록된 인사기본 사항에 대한 수정, 삭제, 조회를 할 수 있다.

2. Main Flow of Events

이 Use Case는 Use Case <사원 인사정보를 유지/관리한다.>에서 인사부 담당자가 인사기본 사항을 선택함으로써 시작된다. 인사부 담당자는 인사기본 사항의 수정, 삭제, 조회, 종료의 작업을 선택하게 된다.

인사기본사항 수정	: 3-1 의 인사기본사항 수정 flow 가 수행된다.
인사기본사항 삭제	: 3-2 의 인사기본사항 삭제 flow 가 수행된다.
인사기본사항 조회	: 3-2 의 인사기본사항 조회 flow 가 수행된다.
종료	: Use Case 를 종료한다.

OTC 유즈 케이스 레포트 (계속)

✍ 유즈 케이스 레포트 <사원의 인사기본사항을 관리한다.> (계속)

3. Alternative Flow of Events

3-1. 인사 기본사항을 수정한다

시스템은 인사 기본사항을 수정하기 위해 수정할 사원번호를 입력할 수 있도록 입력란을 화면에 Display 한다. 시스템은 인사부 담당자가 입력한 사원번호의 인사기본사항을 화면에 Display 한다. (E-1). 인사부 담당자가 수정을 완료하고 수정된 내용을 데이터베이스에 갱신하도록 요청하면시스템은 미입력된 사항이 있는지를 Check 하고(E-2), 인사 데이터베이스에 수정이 완료되면(E-1) 시스템은 '갱신완료' 라는 메시지를 Display 한다. 인사부 담당자는 메시지를 확인하고 종료를 선택하여 Use Case를 종료한다.

3-2. 인사 기본사항을 삭제한다

시스템은 삭제할 사원번호를 입력할 수 있도록 입력란을 화면에 Display 한다. 시스템은 데이터베이스에 인사부 담당자가 입력한 사원번호에 해당되는 인사기본 사항에 대한 삭제요청을 하고 삭제가 완료되면, '삭제완료' 라는 메시지를 Display 한다. 인사부 담당자는 메시지를 확인하고 종료를 선택하여 Use Case를 종료한다.

3-3. 인사 기본사항을 조회한다

시스템은 조회할 사원번호를 입력할 수 있도록 입력란을 화면에 Display 한다. 시스템은 인사부 담당자가 입력한 사원번호에 해당되는 인사기본 사항을 화면에 Display 하고(E-1), '조회완료' 라는 메시지를 Display 한다. 인사부 담당자는 메시지를 확인하고 종료를 선택하여 Use Case를 종료한다.

유즈 케이스 레포트 (계속)

✍ 유즈 케이스 레포트 <사원의 인사기본사항을 관리한다.> (계속)

3-E Exception 처리

E-1 시스템은 'DB Error: Error #(오류번호와 내용)' 라는 메시지를 Display 하며, 인사부 담당자는 메시지를 확인하고 종료를 선택하여 Use Case를 종료한다.

E-2 시스템은 '미입력 사항이 있음. 확인 후 재입력 바람' 이라는 메시지를 Display 하며, 인사부 담당자는 메시지를 확인하고 입력사항을 모두 입력하고 Use Case를 계속 진행하거나 종료를 선택하여 Use Case를 종료할 수 있다.

4. Special Requirements

5. Preconditions

6. Postconditions

7. Pictures of the User Interface

액티비티 다이아그램

용도

시스템이 액터에게 제공해야 할 기능을 액티비티들의 순차적인 나열로 나타낸다.

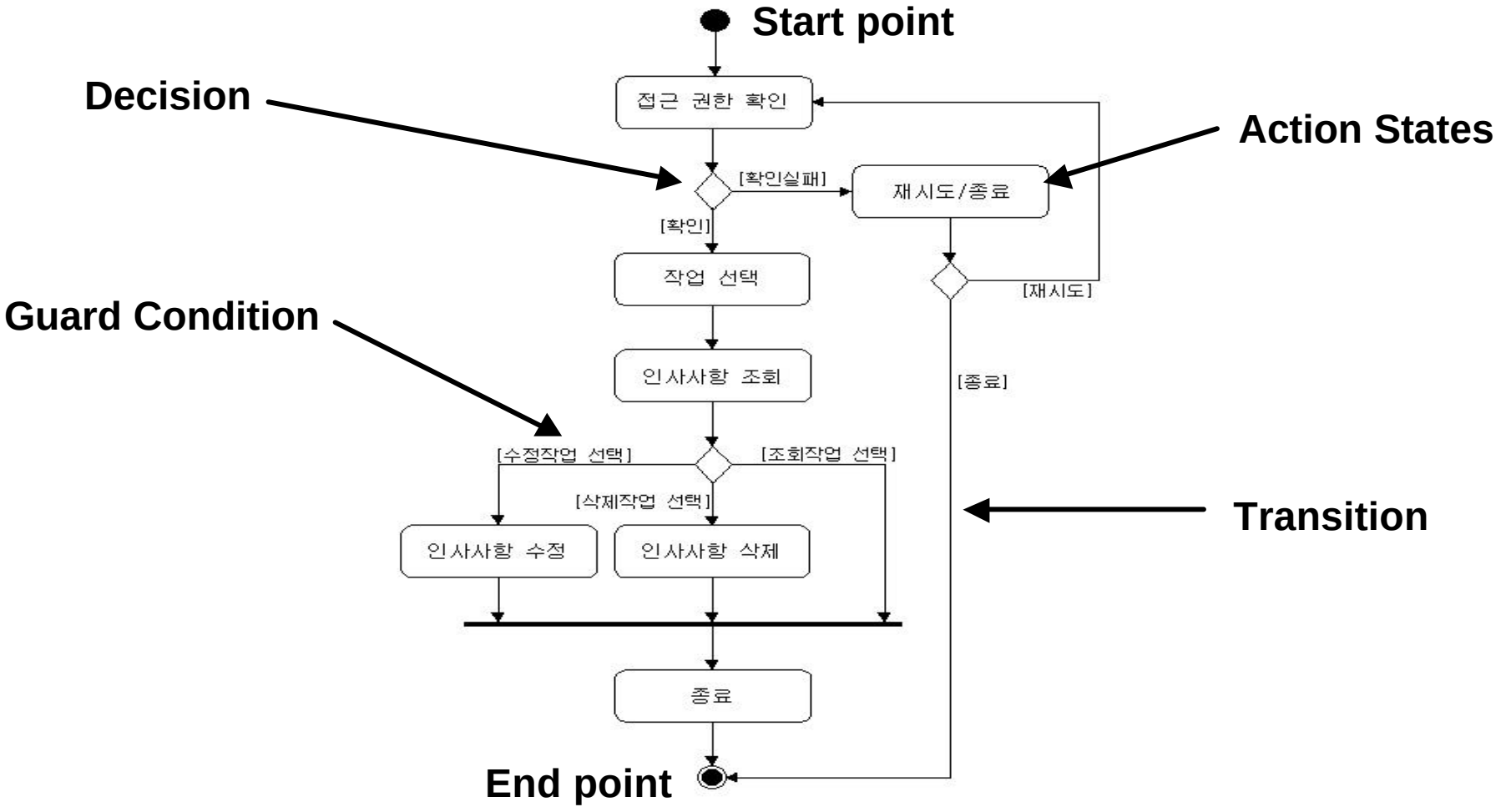
특징

액티비티 다이어그램은 스테이트챠트 다이어그램의 특별한 경우로서, 거의 모든 상태가 액티비티 상태이며 대부분의 상태전이는 이전 상태의 액션들이 완료함에 의해 수행된다.

구성요소

- Activity State : 액티비티 수행 또는 이벤트 흐름의 단계 표시
- Transition : 한 Activity State에서 다른 Activity State로의 전이 표시
- Decision : 조건 분기 표시
- Synchronization bar : 이벤트 흐름에서 Concurrent Thread들 간에 동기화 표시

액티비티 다이아그램 (계속)



✍"인사 기본사항을 유지/관리한다"의 이벤트 흐름을 액티비티 다이아그램으로 표현

OTC 용어 해설집

- 특징
  - 프로젝트에서 자주 사용하는 용어들을 정리한 것
  - 도메인 전문가와 시스템 개발자 간의 의사소통을 촉진

- 구성
  - 개요
  - 항목 및 설명

용어 해설집 (계속)

Glossary

1. 개요

사원 기본정보 관리 업무에서 사용되는 용어들을 정리하여 기술한다.

2. 항목

사원 : 년간 채용계획 혹은 업무상 필요에 의해 채용되어 근무하는 직원을 지칭

정규직 : 년간 인력소요계획에 의해 정기적인 입사전형을 통해 채용된 신입 혹은 경력사 원을 지칭

임시직 : 업무상 필요에 의해 1년 단위 혹은 1 년 이내의 적정 기간동안 채용된 사원

계약직 : 업무의 필요에 의해 1 년 단위로 채용된 임시직 사원

시간직 : 업무의 필요에 의해 시간제로 채용된 임시직 사원

인력소요계획 : 각 부서에서 업무의 규모와 사업계획을 기준으로 작성된 예상 소요인원 계획. 인력소요계획은 정규직 및 임시직 사원 채용을 위한 기준으로 사용된다.

적정인원 : 인사부에서 결정한 부서별 사업계획을 기준으로 산정된 필요 예상인원

조정인원 : 부서별로 작성되어 보고된 인력소요계획의 인원에 대해 인사부에서 조사된 적정인원과 비교해 산정된 채용규모

부가사항 기술서

- 특징
  - 구현단계의 제약사항까지도 기술 가능

- 구성
  - 기능성(Functionality)
  - 유용성(Usability)
  - 신뢰성(Reliability)
  - 성능(Performance)
  - 지원(Supportability)
  - 설계 제한사항(Design Constraints)

부가사항 기술서 (계속)

부가사항 기술서(예)

1. 유용성
2. 신뢰성
3. 성능

   이 시스템에 접속 가능한 사람의 숫자는 3,000명이다.

4. 지원
5. 보안

   이 시스템을 통해 개인의 인사정보를 수정하는 작업은 인사부 직원만 가능하다. 다만 인사부 직원을 제외한 다른 직원들은 자신의 개인정보에 대한 조회만 가능하다. 인사부 직원은 전체 사원에 대한 조회 및 수정이 가능하며, 팀장은 자신의 팀원의 정보에 대한 조회가 가능하다.

부가사항 기술서 (계속)

✍ 부가사항 기술서 (계속)

6. 설계 부가사항

1) 이 시스템은 MS DCOM을 이용하여 구현한다.

2) 이 시스템은 Oracle8의 객체 타입(Object Type)과 스토어드 프로시져(Stored Procedure)를 이용하여 구현한다.

3) 이 시스템에 사용될 운영체제는 NT-Server 4.0과 Windows 95를 사용한다.

4) 이 시스템은 데스크 탑 형태의 컴퓨터뿐 아니라 웹 환경에서도 동일한 사용자 인터페이스를 제공한다.

5) 트랜잭션은 MTS(Microsoft Transaction Server)를 사용하여 관리한다

II. 유즈케이스 분석

- 유즈케이스 실체화
- Object
- Object Interaction
- 클래스 추출
- 분석 클래스

목적

- 유즈 케이스 안에 서로 연관되어 작용하고 있는 객체들을 추출하고, 객체들간에 주고 받는 메시지를 정의
- 분석 클래스 도출(1차 레벨의 추상 클래스)

- 단계
  - 유즈 케이스 명세 상세화
  - 유즈 케이스내의 객체 상호작용 표현(Interaction Diagram)
  - 분석 클래스에 대한 속성 및 오퍼레이션 정의(Class Diagram-VOPC)
  - 분석 클래스들 간의 관계 설정 (Class Diagram-VOPC)
  - 분석 클래스 검증(Package Diagram & Class Diagram)

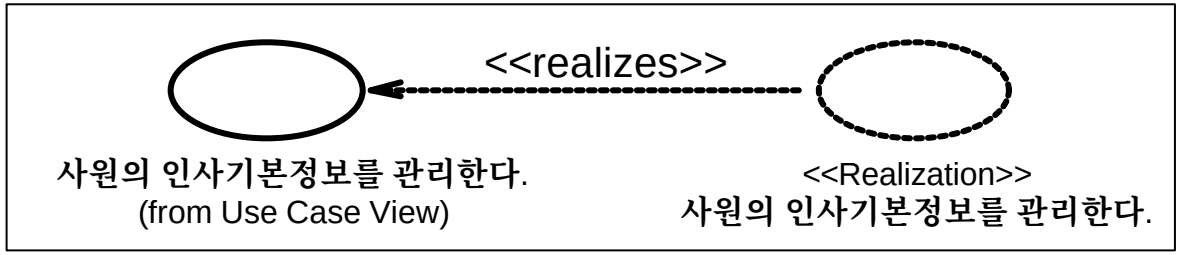
유즈 케이스 실체화

- 정의
  - 각각의 유즈 케이스 안에서 상호 작용하는 객체를 찾아내고 객체 간에 전달되는 메시지를 찾아 내는 유즈 케이스 분석(Use Case Analysis) 작업

- 목적
  - 유즈 케이스의 흐름을 다이아그램으로 표현
    : Sequence Diagram & Collaboration Diagram
  - 객체 추출
  - 객체간의 메시지 추출
    ? 1차 레벨의 추상 클래스 도출 및 오퍼레이션 정의

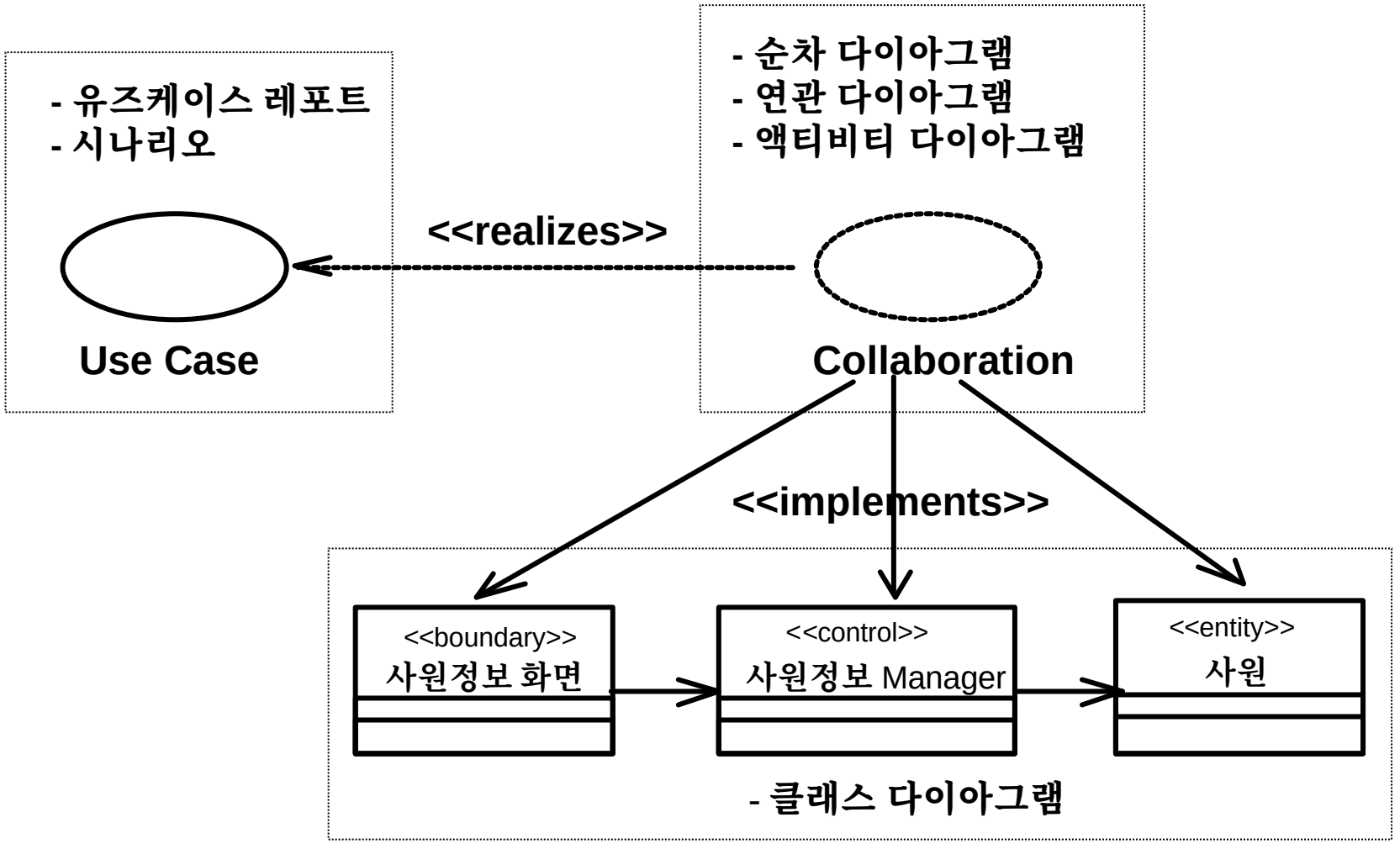
유즈 케이스 실체화 (계속)

- 표현 방법
  - Collaboration으로 표현
    - <<realization>> 스테레오 타입
    - 유즈 케이스와 종속관계로 설정하고 <<realizes>> 스테레오 타입을 정의한다.



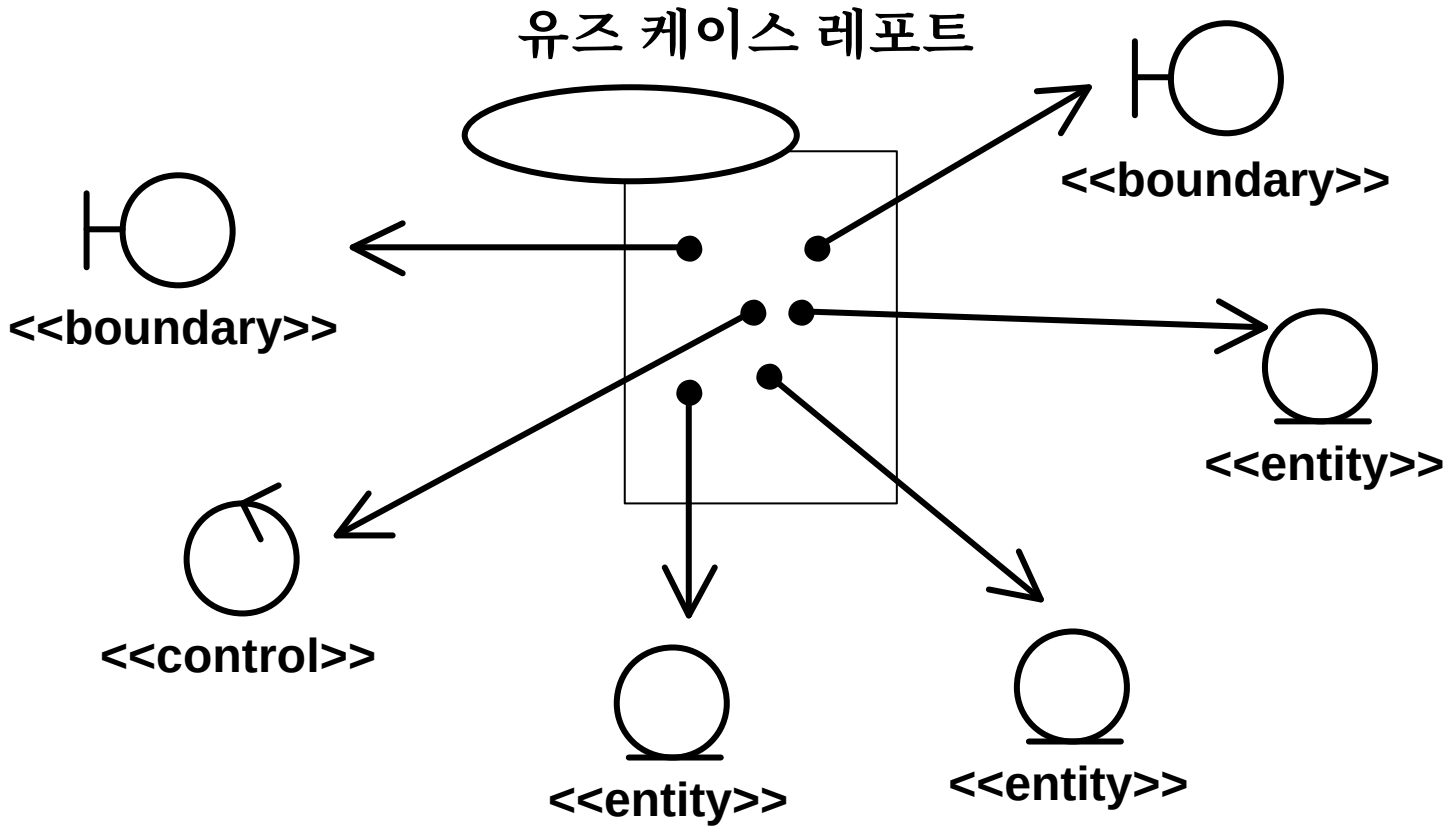
  - 동적인 측면
    - 인터랙션 다이아그램으로 표현(Sequence Diagram & Collaboration Diagram)
  - 정적인 측면
    - 클래스 다이아그램(VOPC:View of Participating Classes)으로 표현
    - 바운더리(Boundary), 콘트롤(Control), 엔터티(Entity) 클래스를 추출하고 클래스들간의 관계(Relationship) 정의

유즈 케이스 실체화 (계속)


- 유즈케이스 레포트
- 시나리오
Use Case
<<realizes>>
- 순차 다이아그램
- 연관 다이아그램
- 액티비티 다이아그램
Collaboration
<<implements>>
<<boundary>>
사원정보 화면
<<control>>
사원정보 Manager
<<entity>>
사원
- 클래스 다이아그램

유즈 케이스 실체화 (계속)

✍ 유즈 케이스 레포트에서 분석 클래스 도출



Object

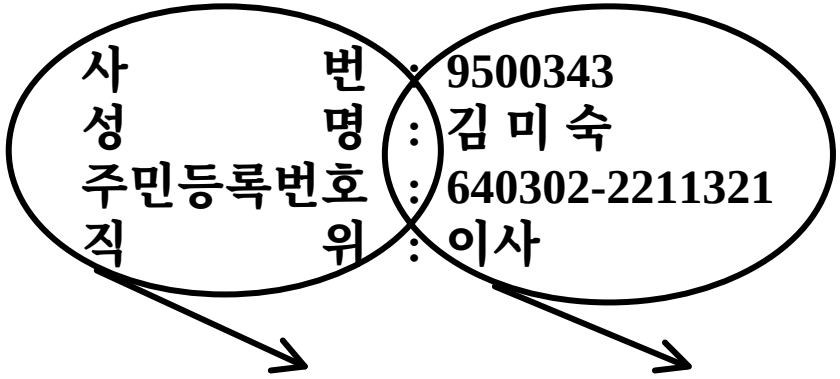
- 정의
  - 현실세계에서 존재하는 모든 것
  - 특정 업무 영역(Business Domain)에서 명확한 범위와 의미를 가지는 실체
    - 개념적 실체, 물리적 실체, 소프트웨어적 실체
  - 구현을 위한 실질적인 기반 제공
- 특징
  - 상태(State)
  - 행위(Behavior)
  - 식별자(Identity)를 갖는다.

Object (계속)

상태 (State)

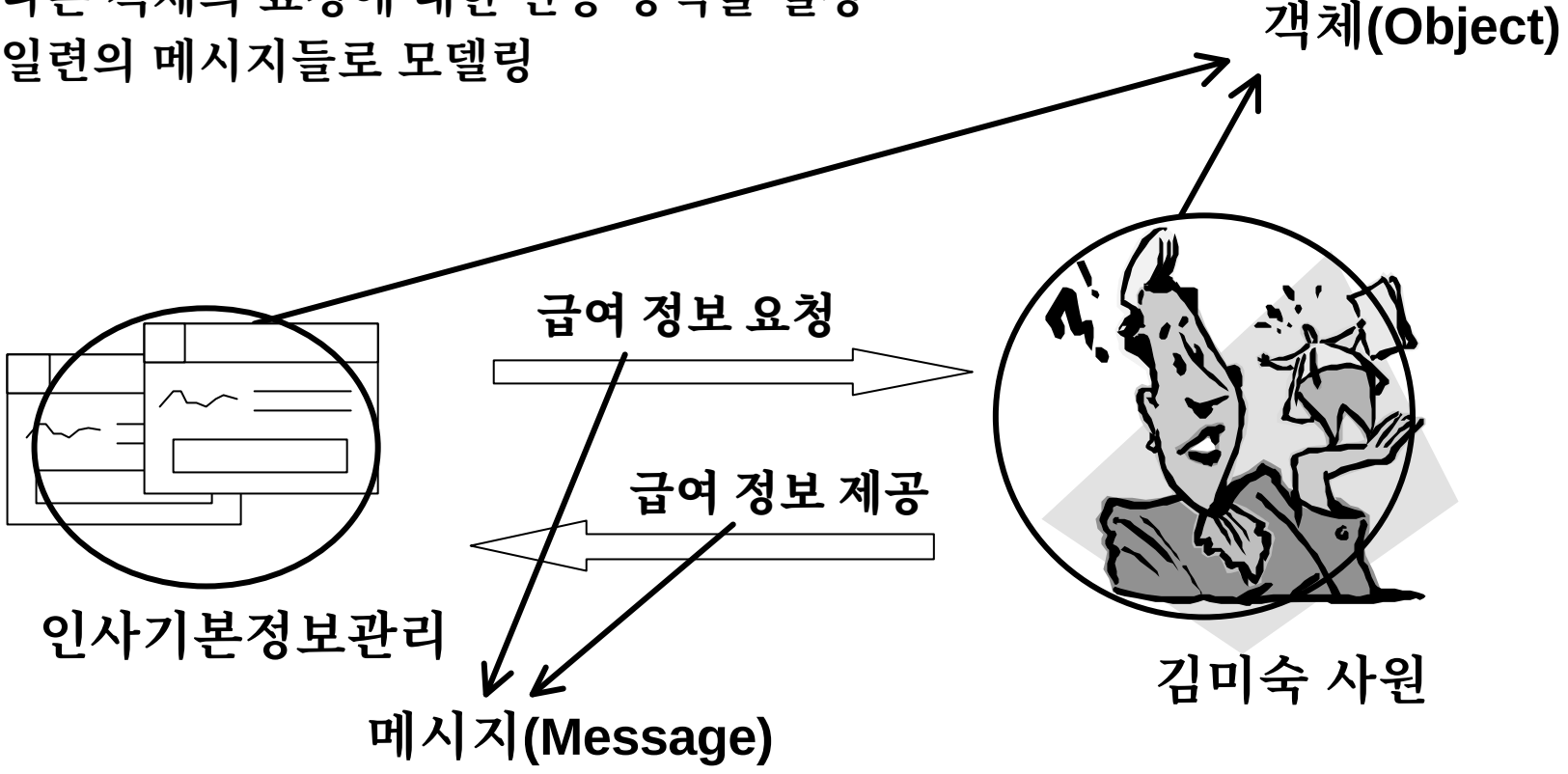
- 각 객체가 가지는 일련의 속성들이 가지는 구체적인 값
- 객체가 존재할 수 있는 조건(Condition) 중의 하나
- 시간에 따라 변화 가능

사원 객체



Object (계속)

- **행위** (Behavior)
  - 객체 자신의 행동 방식을 결정
  - 다른 객체의 요청에 대한 반응 방식을 결정
  - 일련의 메시지들로 모델링



Object (계속)

- 식별자(Identity)
  - 각각의 객체는 유일한 식별자를 가짐
    - 동일한 상태(State)를 지닌 여러 객체들이 존재하더라도 식별 가능

사번 : 9300402

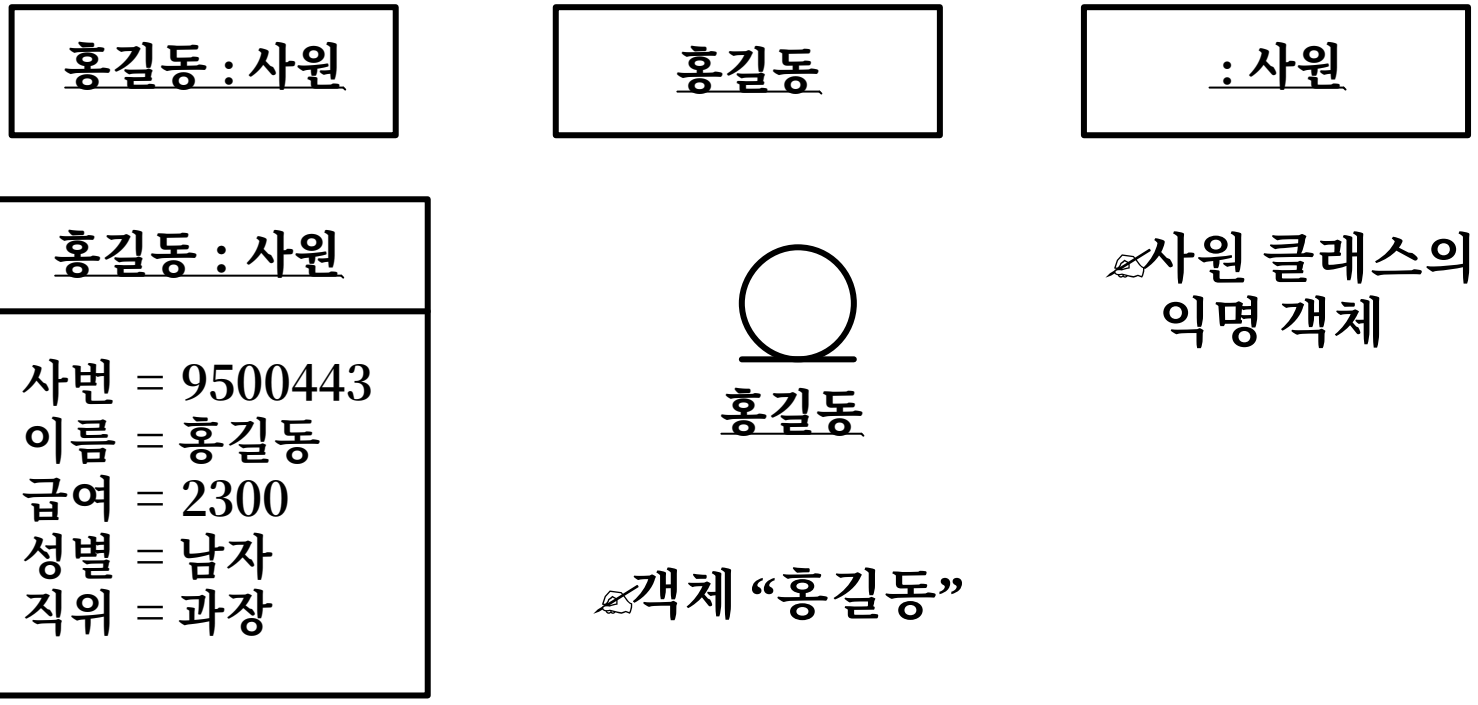
사번 : 9320402

사번 : 9400422

- 각각의 사원객체의 상태는 모두 같음
- 사원 객체의 식별자는 사번

Object (계속)

✍ 객체의 표현 방식



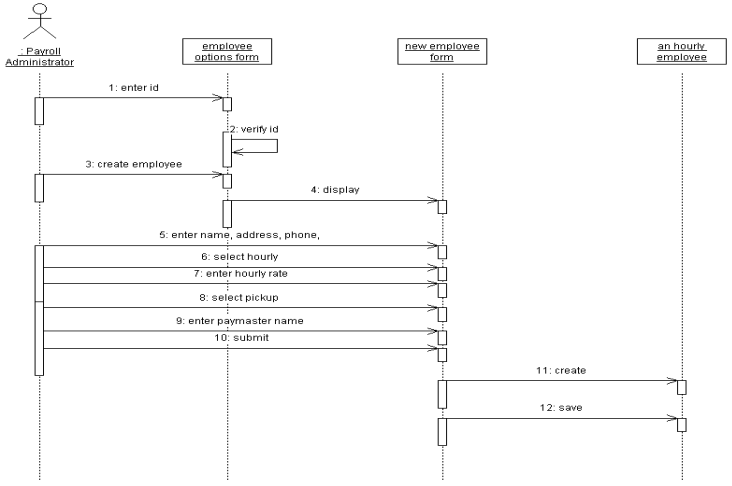
✍사원 클래스의 익명 객체

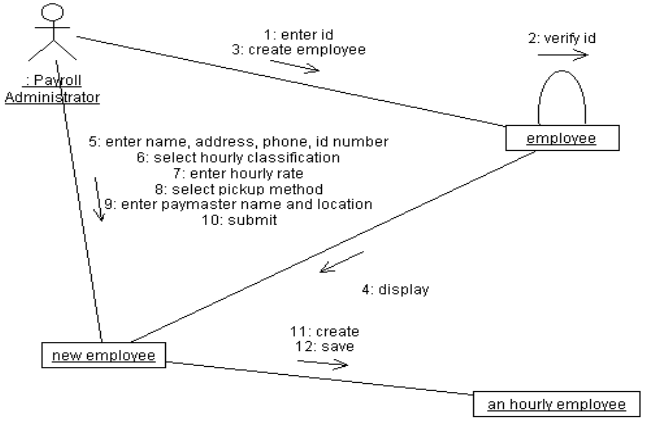
✍사원 클래스의 객체 “홍길동”

✍객체 “홍길동”

Object Interaction

- 객체간의 상호작용 : Interaction Diagram으로 표현
- Interaction Diagram
  - Sequence Diagram
  - Collaboration Diagram

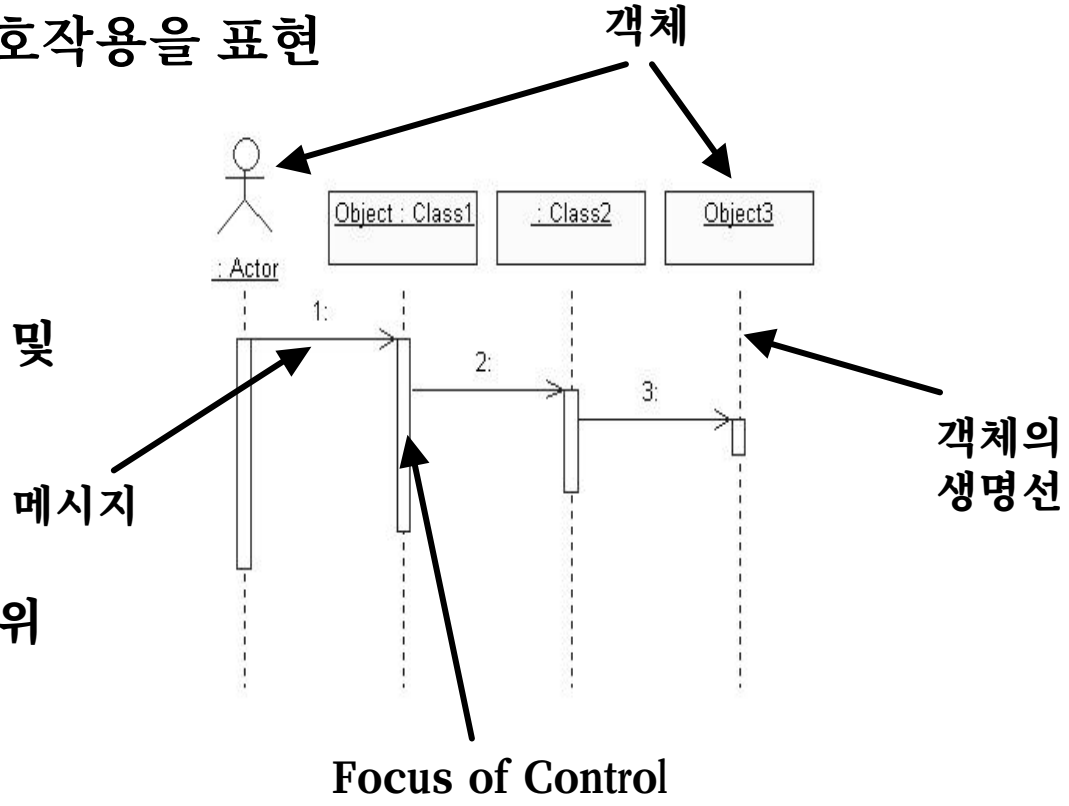




Object Interaction (계속)

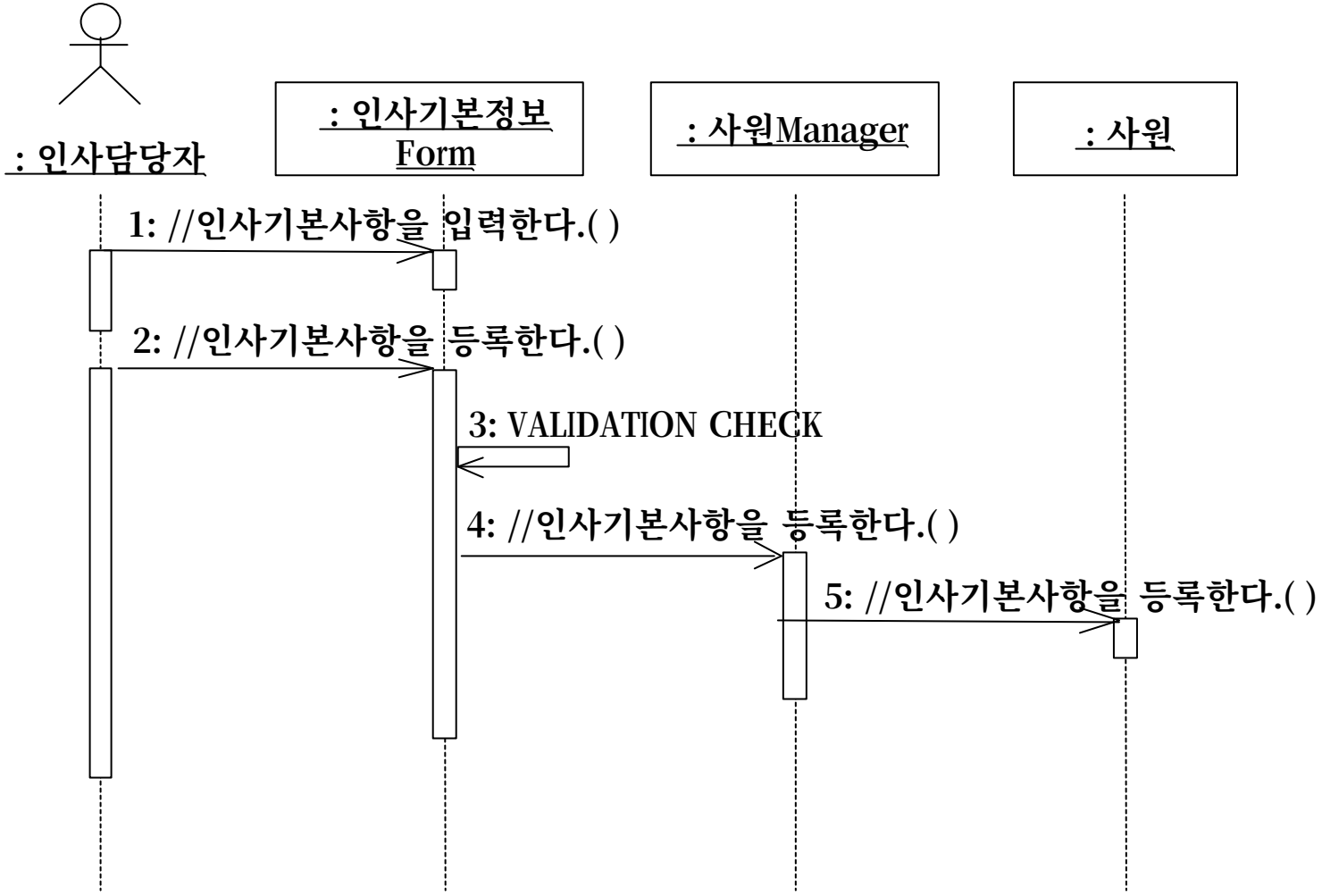
Sequence Diagram

- 유즈 케이스별로 작성
  - 하나의 유즈 케이스에서 다양한 이벤트 흐름별로 작성
- 시간의 경과에 따른 객체의 상호작용을 표현
- 포함되는 내용
  - 관련 객체들
  - 객체들 간에 교환되는 메시지 및 메시지들의 순서
  - 각 객체의 생명주기 (Life Line)
  - 메시지들의 트랜잭션 처리 단위 (Focus of Control)



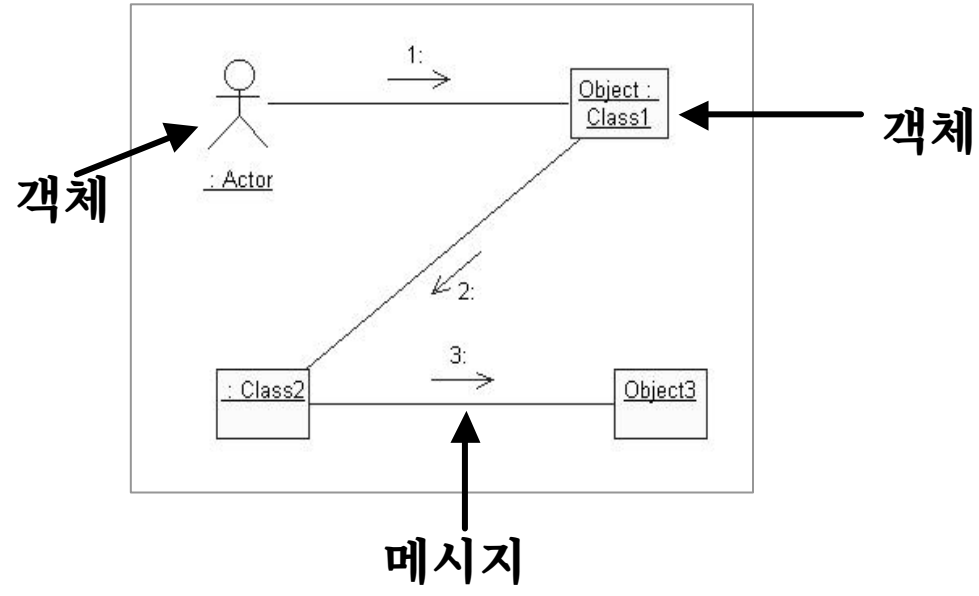
Object Interaction (계속)

Sequence Diagram의 예



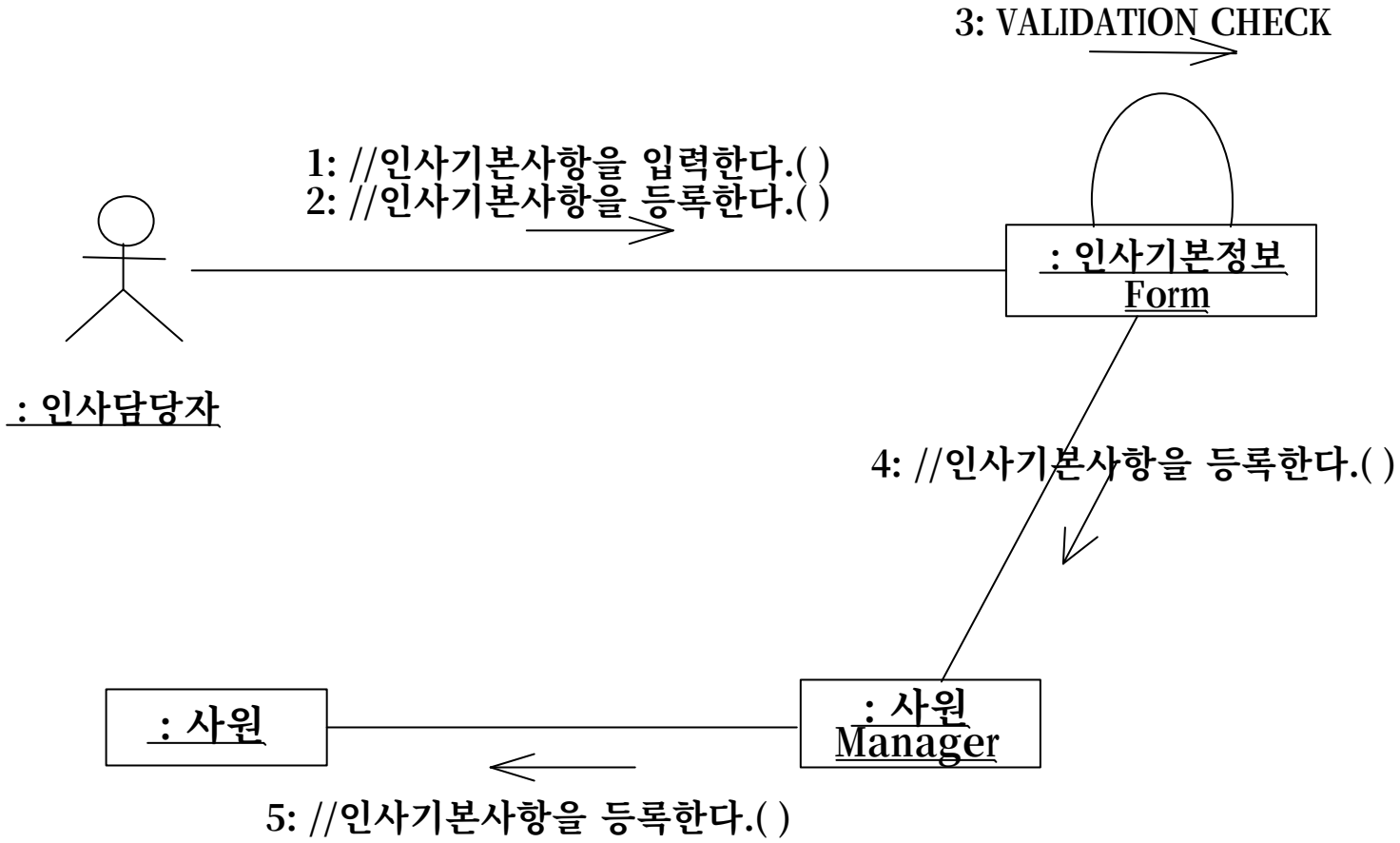
Object Interaction (계속)

- Collaboration Diagram
  - Use Case별로 작성
    - 하나의 유즈 케이스에서 다양한 이벤트 흐름별로 작성\
  - 객체들 간에 교환되는 메시지에 초점
  - 포함되는 내용
    - 관련 객체들
    - 객체들 간의 연결 (Link)
    - 객체들 간에 교환되는 메시지
    - 객체들 간의 데이터 흐름 (Optional)



Object Interaction (계속)

Collaboration Diagram의 예

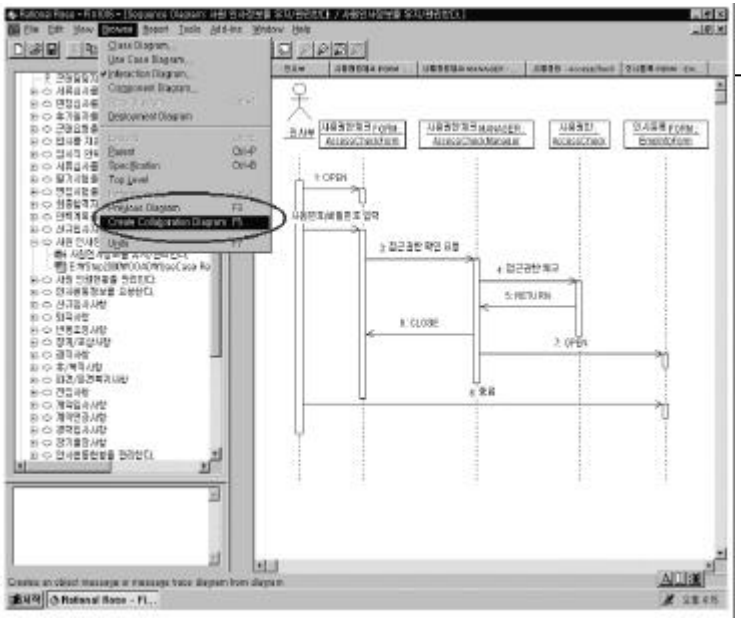


Object Interaction (계속)

Sequence Diagram vs Collaboration Diagram

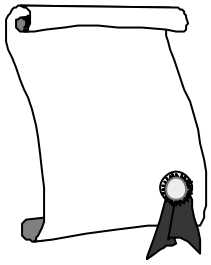
Sequence Diagram	Collaboration Diagram
메시지의 순서(Sequence)에 촛점	객체간의 상호작용과 메시지에 촛점
전체적인 흐름을 가시적으로 표현	객체간의 Collaboration 표현
복잡한 시나리오 표현에 적합	

- Rational Rose(CASE Tool)에서
  하나의 다이아그램만 작성하면
  나머지 다이아그램은 자동으로 생성

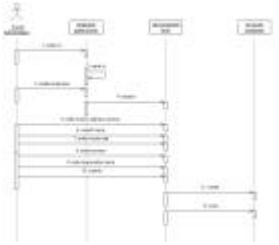


클래스 추출

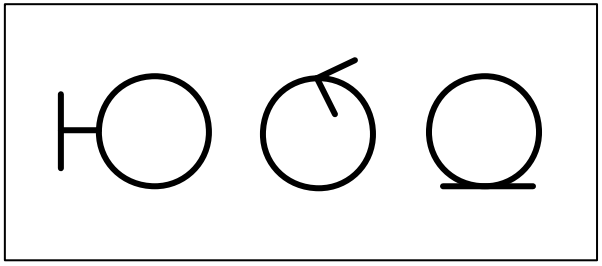
Use Case Analysis



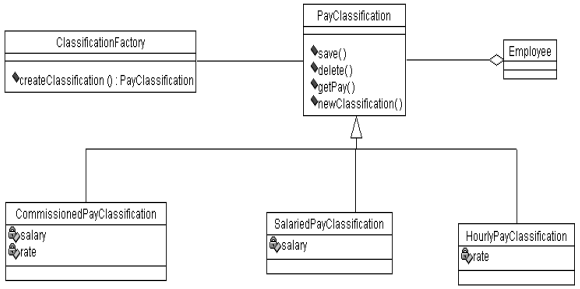
Use Case Report



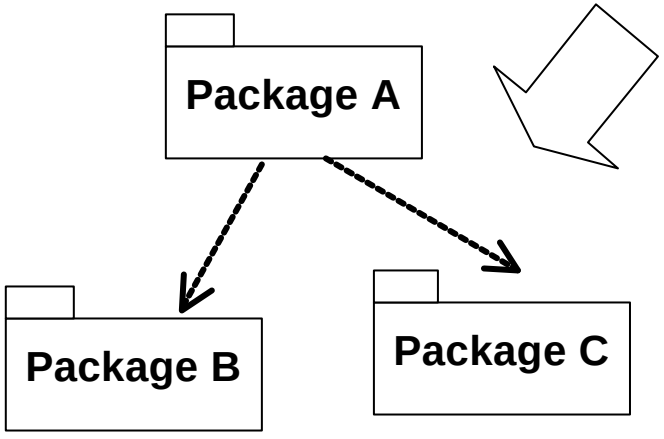
Object Interaction



Class 추출



Class Diagram



Package Diagram

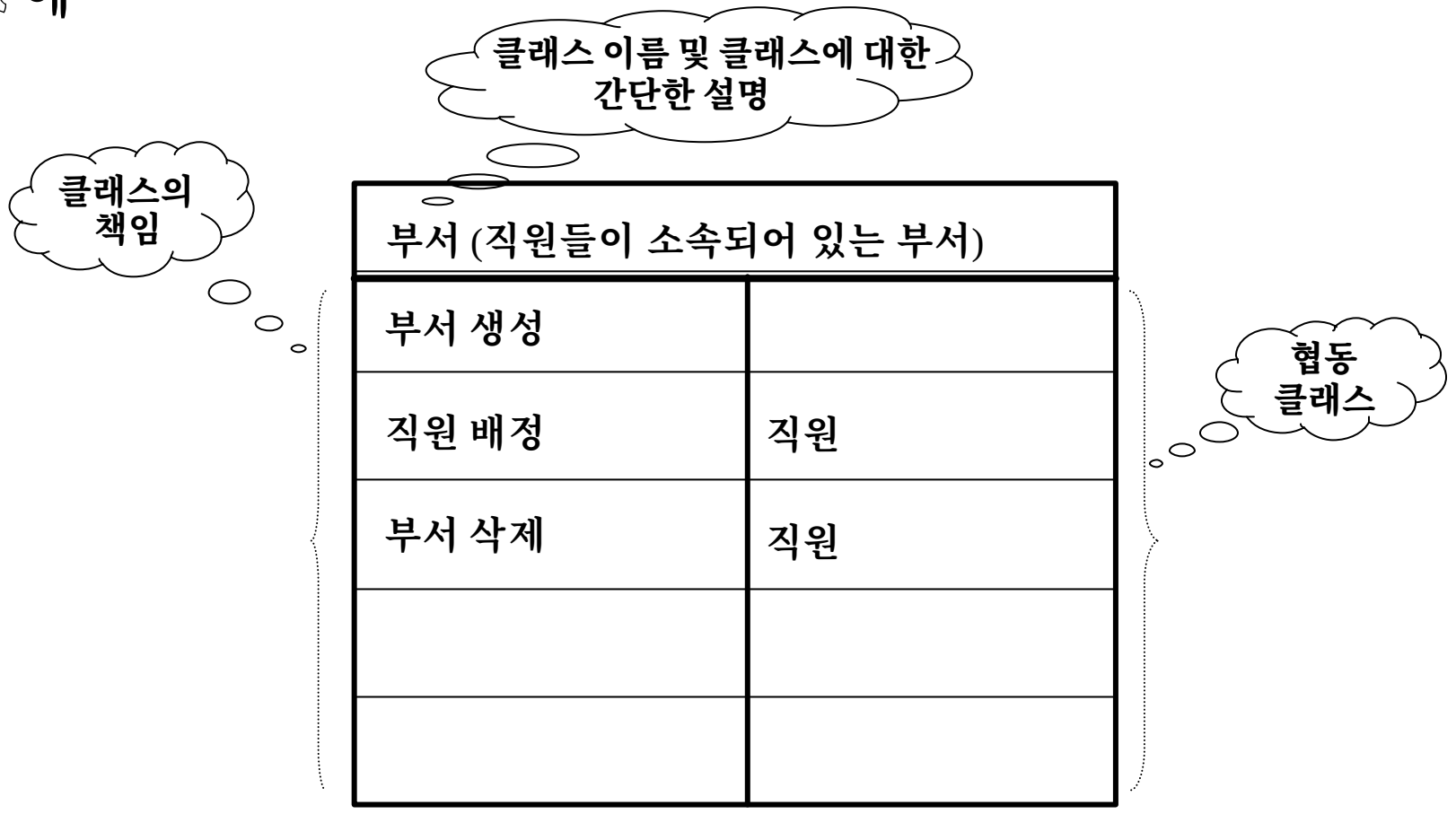
클래스 추출 (계속)

- **CRC(Class Responsibility Collaboration) Cards**
  - **찾은 객체들로 부터 클래스를 파악하기 위한 수단**
    - Ward Cunningham & Kent Beck, OOPSLA, 1989
  - 3’ * 5’ **크기의 카드**
    - **클래스 이름, 정의**
    - **클래스 책임**(responsibility)
      - Class **내부 정보**
      - **제공되는** Service**들**
  - **동일한 책임에 관련되어 있는 협력** Class (Collaborative Class)**를 찾기 위한 목적**
    - **특정 책임을 위해 필요한 서비스를 제공하는 클래스**

클래스 추출 (계속)

CRC Cards (계속)

예



클래스 추출 (계속)

CRC Cards (계속)

장점

- Scenario가 완전하게 되어감에 따라 협력(Collaboration) 유형(Pattern) 파악 가능
- Class간의 Generalization/Specialization/Aggregation 계층(Hierarchies) 파악 용이

주의사항

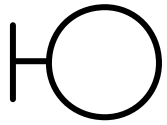
- 모든 Class의 이름은 함축적(meaningful) 이고, 업무영역에 적합한(domain-specific) 형태이어야 함
- 각 Class는 일련의 협력 Class(Collaborator) 를 가지도록 해야 함
- "반드시 있어야만 하는 필수적인(indispensable) Class"의 개념은 없음
- 요구사항의 변경은 Class에 의해 처리 가능
- 대부분의 Class들은 responsibility를 가짐
- 모든 Class들이 서로 협력(Collaboration)하는 경우는 없음

분석 클래스

- 정의
  - 분석 클래스는 시스템 내부에서 책임(Responsibility)과 행위(Behavior)를 가지는 것(Things)에 대한 개념적 모델(Conceptual Model)
- 타입
  - 바운더리 클래스
  - 콘트롤 클래스
  - 엔터티 클래스
- 추출 방법
  - 유즈 케이스 분석
  - CRC 카드 기법 적용 등

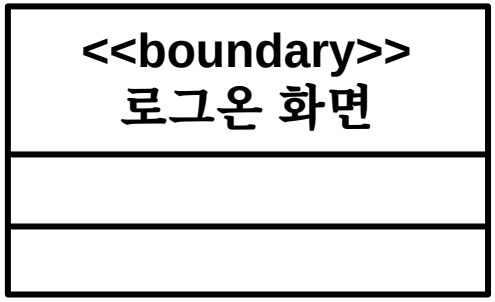
분석 클래스 (계속)

- 바운더리 클래스
  - 사용자 인터페이스 클래스
    - GUI Window
  - 시스템 인터페이스 클래스
    - 외부 시스템
  - 하드웨어 인터페이스 클래스
    - 프린터, 스캐너, ...



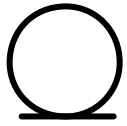
로그온 화면

- 바운더리 클래스 추출
  - Actor와 Scenario 비교 후 파악
    - 사용자 인터페이스
  - System-to-System Communication 파악
    - H/W, S/W 모두

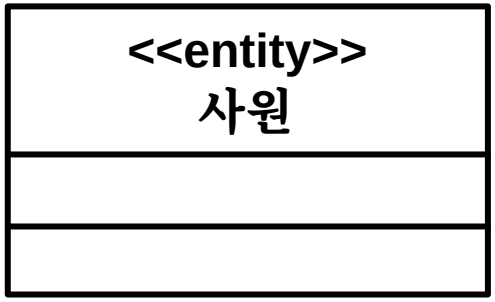


분석 클래스 (계속)

- 엔터티 클래스
  - 오랫동안 지속되는 정보를 저장하는 역할을 하는 클래스
  - 시스템의 Key Concepts



사원



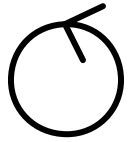
분석 클래스 (계속)

엔터티 클래스 추출

- 용어 해설집에 정의된 용어
- Scenario에서 명사(구) 중심으로 판별
- 동일한 객체에 대해 여러 가지 용어 사용 가능
- 하나의 용어가 여러 객체들을 함축하는 경우도 있음
- 동사 같은 명사, 명사 같은 동사에 주의
- 반드시 Filtering할 것
- 단어 자체의 의미보다 전체 문제영역에서 갖는 의미를 파악할 것
- 찾은 객체들이 유사한 구조(Structure)/행위(Behavior)를 가지는지 파악하여 Grouping

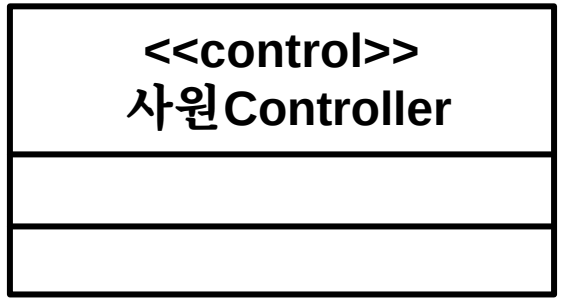
분석 클래스 (계속)

- 콘트롤 클래스
  - 다른 클래스를 제어하는 클래스
    - Logic Control(events간의 순서 등)
    - 트랜잭션 처리
    - 여러 개의 엔터티 클래스의 내용을 사용 또는 변경
    - 명확한 실체를 가지고 있지 않으므로 파악하기가 매우 난해



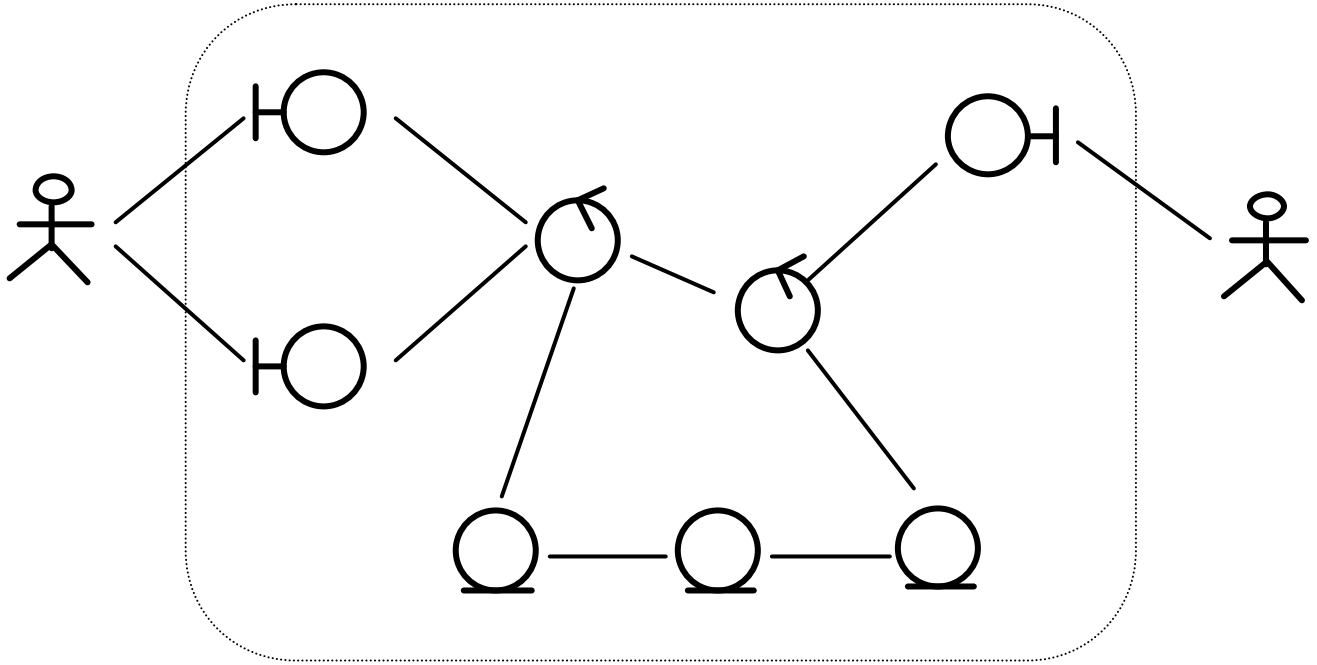
사원Controller

- 콘트롤 클래스 추출
  - Sequencing Information 파악
    - Entity/Boundary Class들의 responsibility와는 구별
  - 유즈 케이스 마다 하나 이상 존재
    - 이벤트의 흐름(Flow of Event)에 대한 responsibility
  - Use Case, Scenario가 추가될 때마다 없어지거나, 합쳐지거나, 분기될 수 있음



분석 클래스 (계속)

- 액터는 바운더리 클래스를 통해 시스템과 상호작용하며,
- 바운더리 클래스는 콘트롤 클래스를 통해 엔터티 클래스에 접근한다.



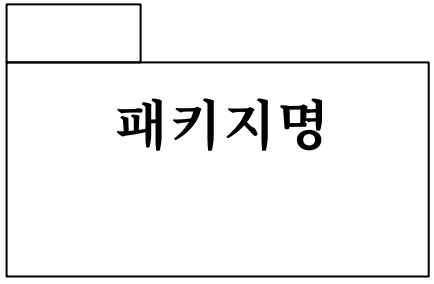
III . 아키텍쳐 설계

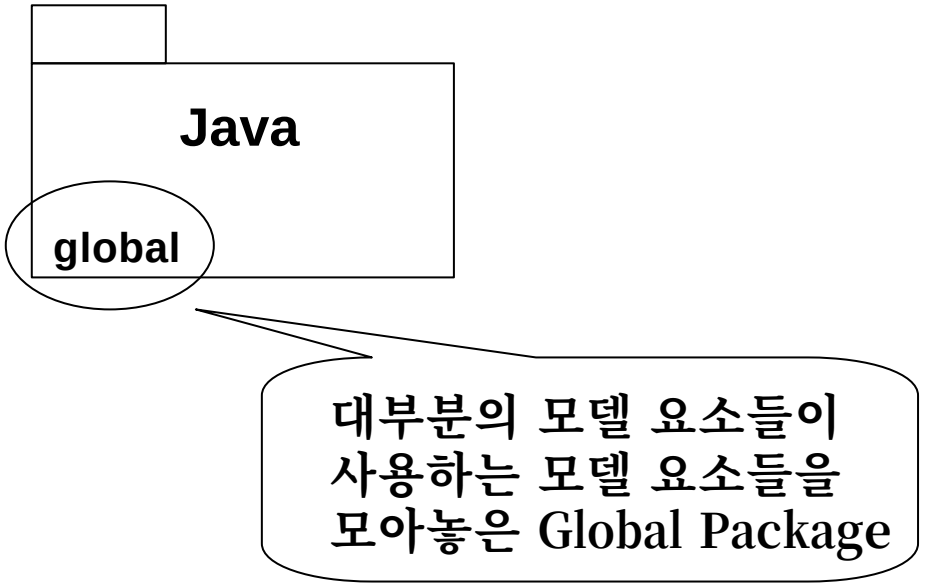
- Package
- Subsystem
- Design Pattern
- Layered Architecture
- Object Model vs Relational Tables
- Reverse Engineering Relational Tables
- Mapping Object to Relational Tables
- Object in jasmine OODB

Package

- 정의
  - 관련 있는 모델요소 들을 분류하기 위한 Mechanism
  - 패키지 안에 다른 패키지가 포함 가능

- 표기법

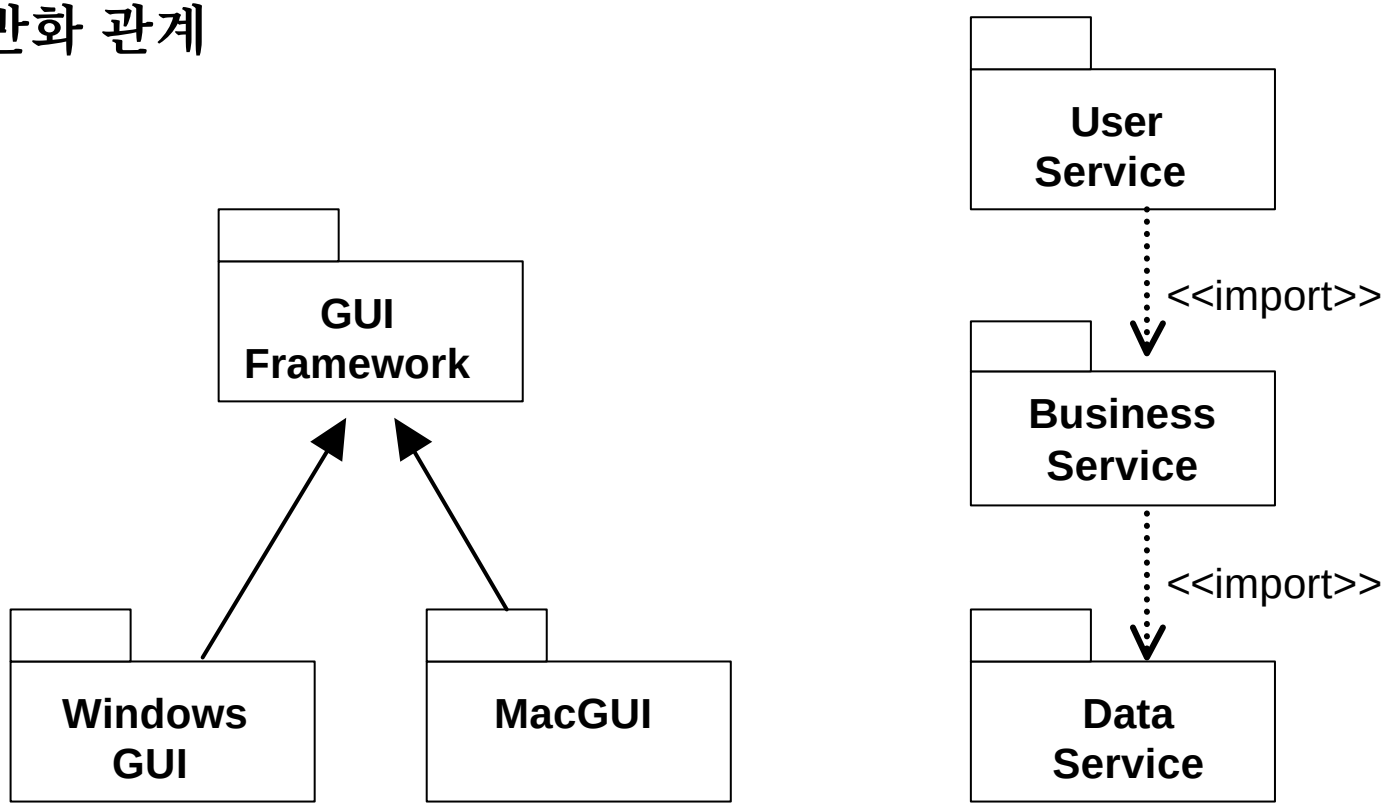




Package (계속)

Package Relationships

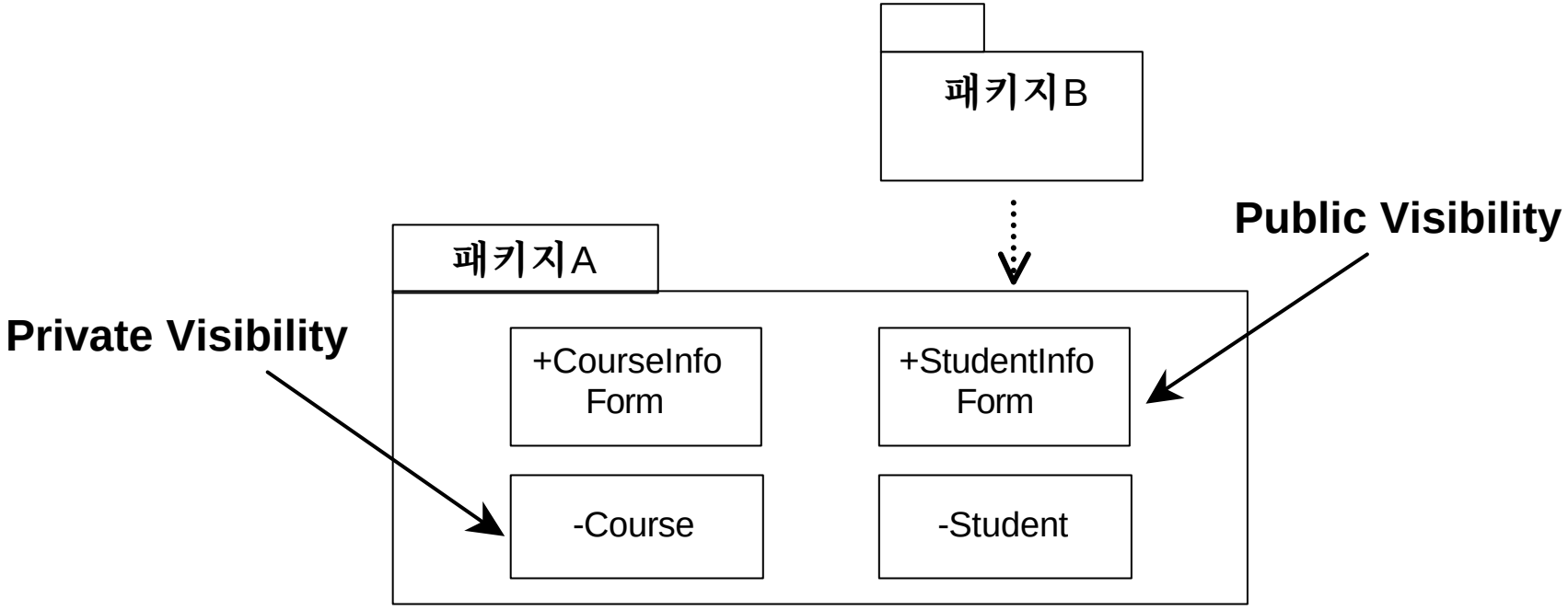
- 종속 관계 : <<import>>, <<access>> stereotype
- 일반화 관계



Package (계속)

Package Visibility

- Public : 다른 패키지의 모델요소에서 참조 가능
- Protected : 패키지의 모델 요소를 상속 받는 서브 패키지의 모델요소에 의해서만 참조 가능
- Private : 패키지 내 모델요소에 의해서만 참조 가능



- PackageB의 Element는 StudentInfoForm, CourseInfoForm은 Access 가능
  But, Course, Student는 직접 Access 불가

Subsystem

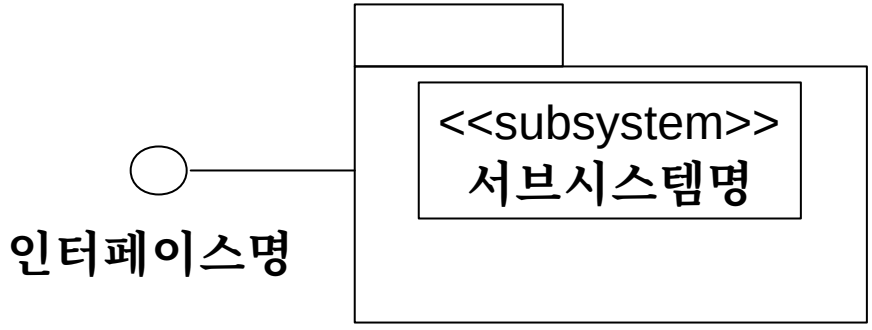
Subsystem

정의

- 물리적인 시스템 내의 행위적인 단위를 나타낸 것
- 인터페이스(Interface)와 오퍼레이션(Operation) 제공
- 재사용 컴포넌트, 외부 시스템, 공통 유틸리티 등을 표현

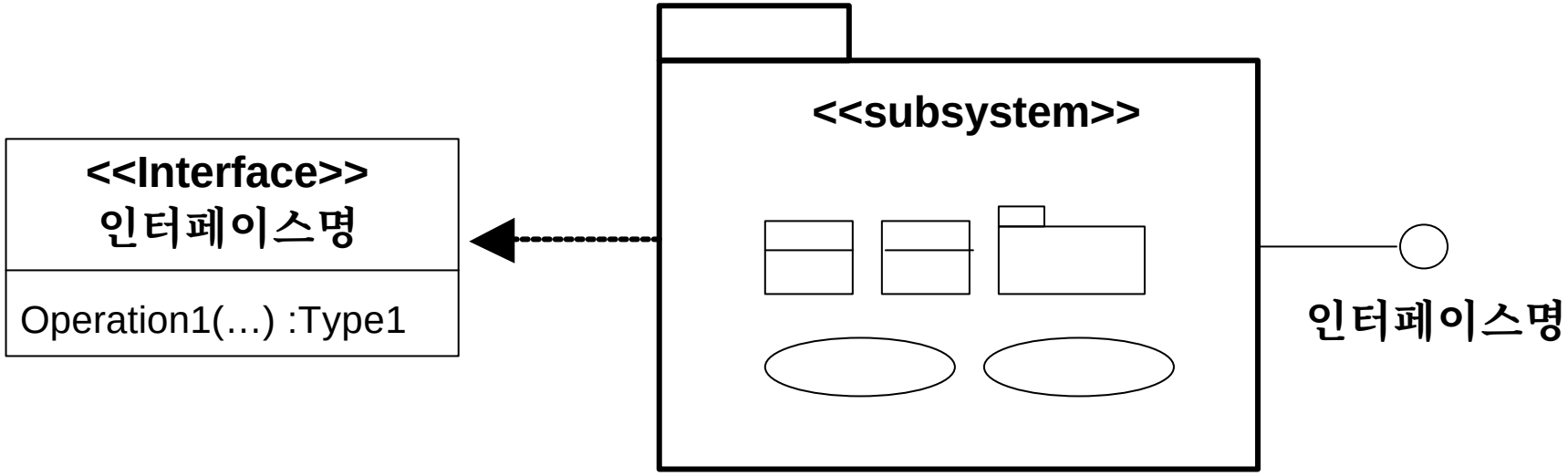
표기법

- 패키지에 <<subsystem>> 스테레오 타입을 정의



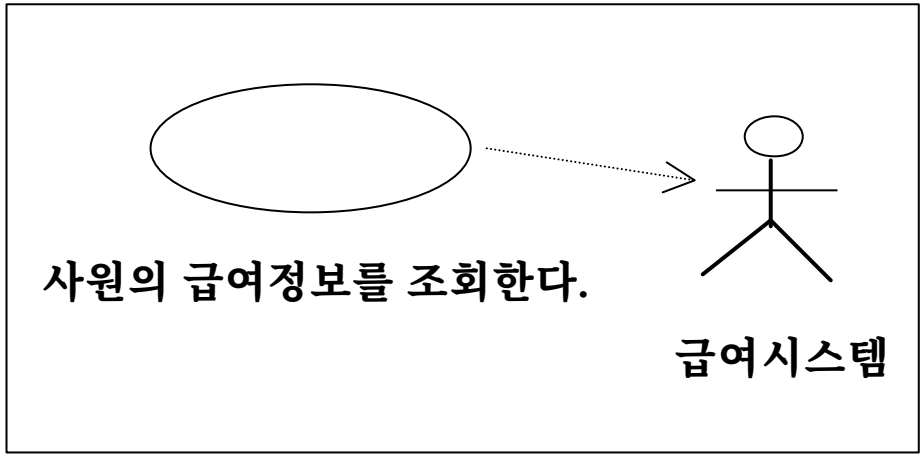
Subsystem (계속)

- **Subsystem 과 Interface**
  - 서브시스템은 적어도 하나 이상의 인터페이스를 갖음
  - 서브시스템과 인터페이스와의 관계 : 실체화(Realization)로 표현

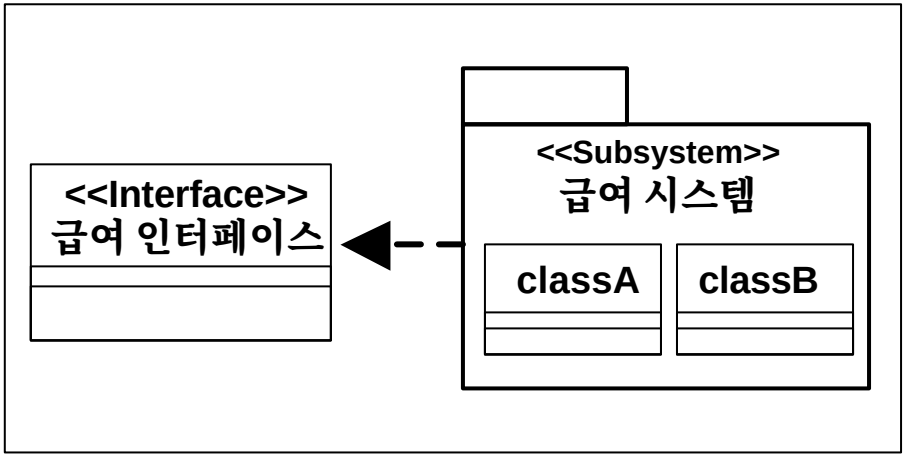


Subsystem (계속)

- **Subsystem 과 Actor**
  - 외부 시스템을 표현한 Passive Actor는 인터페이스를 가진 Subsystem으로 설계된다.



유즈케이스 모델

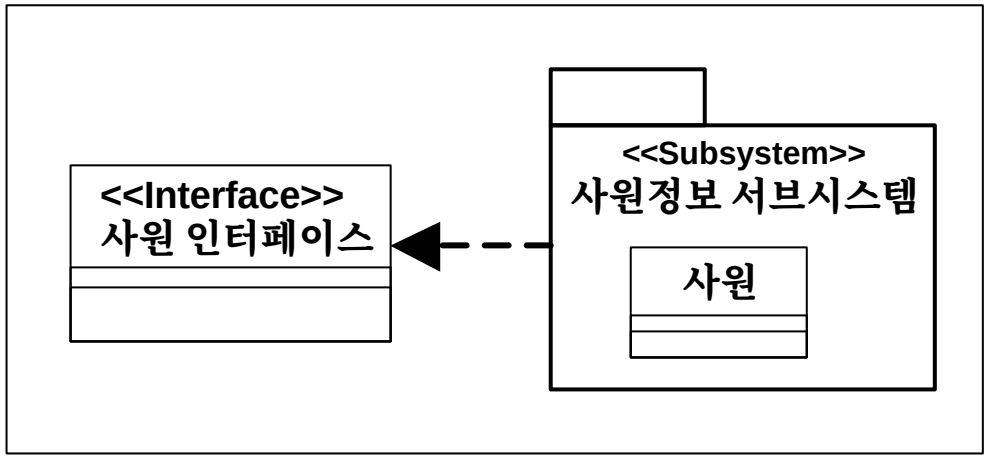


설계 모델

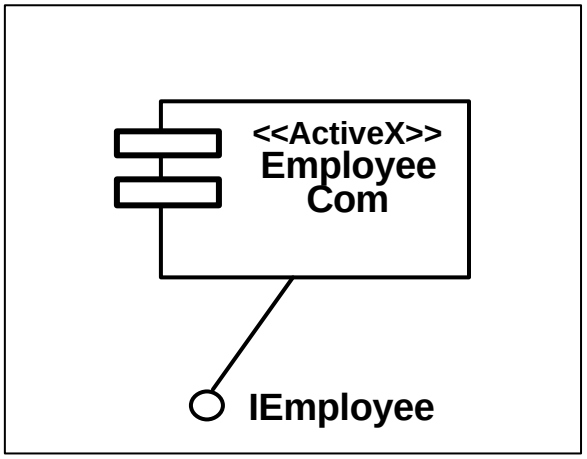
Subsystem (계속)

Subsystem 과 Component

- 설계단계의 Subsystem은 컴포넌트로 구현된다.



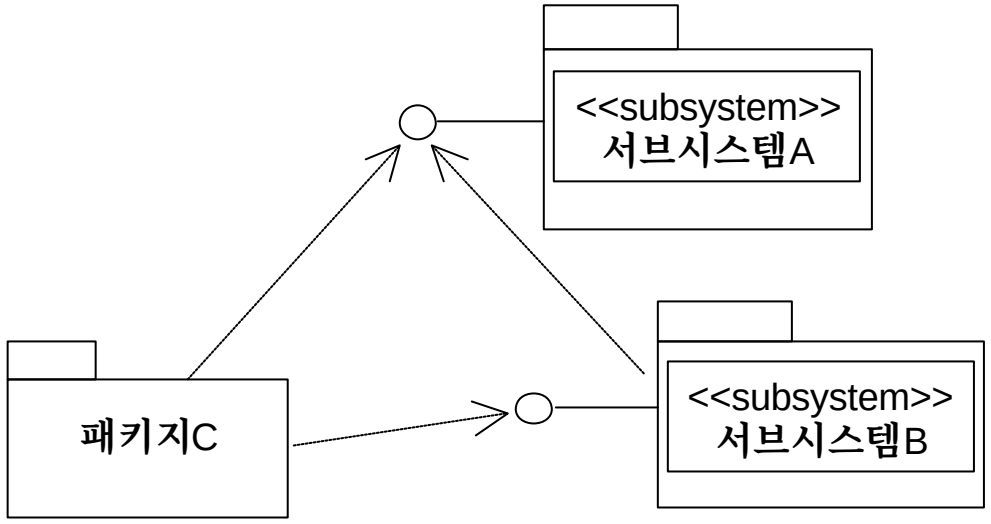
설계 모델



구현 모델

Subsystem (계속)

- **Package와 Subsystem**
  - Subsystem은 행위를 포함하고 인터페이스를 갖는다.
  - Package는 단순히 모델 요소들을 의미상으로 그룹화한 것이다.



Design Pattern

- **Design Pattern**
  - 정의
    - 특정 분야에서 일반적인 설계 문제를 해결하기 위해 커스터마이징된 여러 객체와 클래스들간의 의사소통관계(communicating)의 Description
    - 일반적인 문제에 대한 일반적이고 핵심적인 해결방안 제시
  - 성공적인 설계 및 아키텍쳐의 재사용을 용이하게 함
    - 이전 경험을 다른 문제에 대한 해결방법으로 사용

Design Pattern (계속)

Design Pattern (계속)

주요 구성요소

이름(Pattern Name)

- 한 두 단어를 이용해 문제, 해결방안, 결과를 표현할 수 있는 이름으로 명명

문제(Problem)

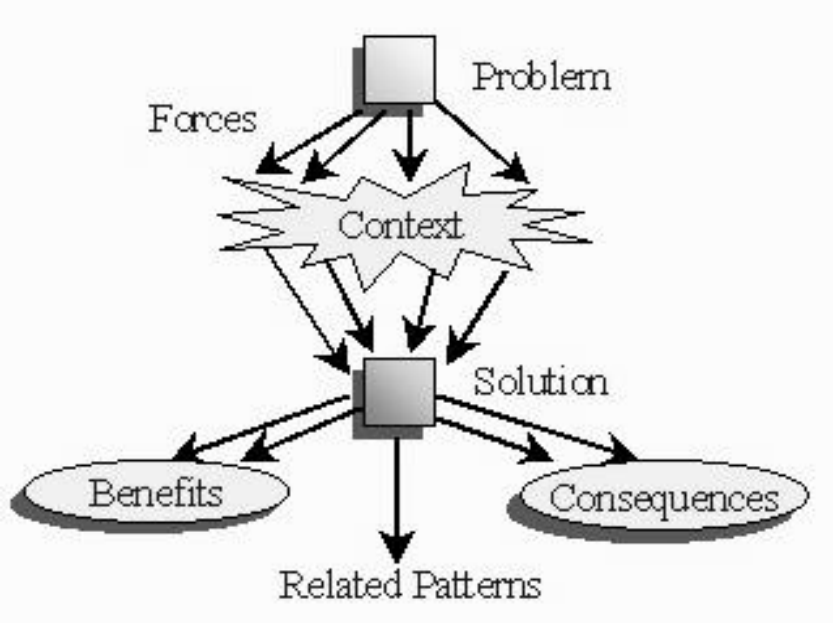
- 문제 및 상황을 설명

해결방안(Solution)

- Design Element
- Relationships
- Responsibility
- Collaboration

결과(Consequence)

- Pattern을 적용한 결과
- Trade - off



Design Pattern (계속)

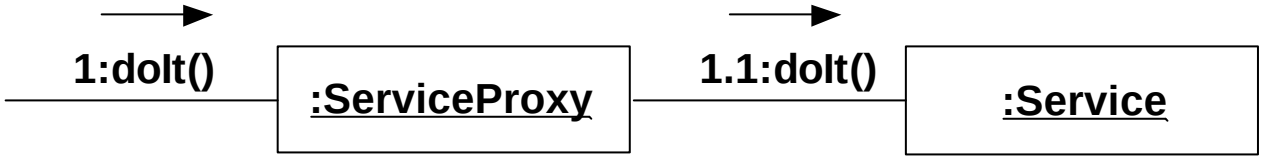
Design Pattern catalog

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Source : Design Pattern by Eric Gammar

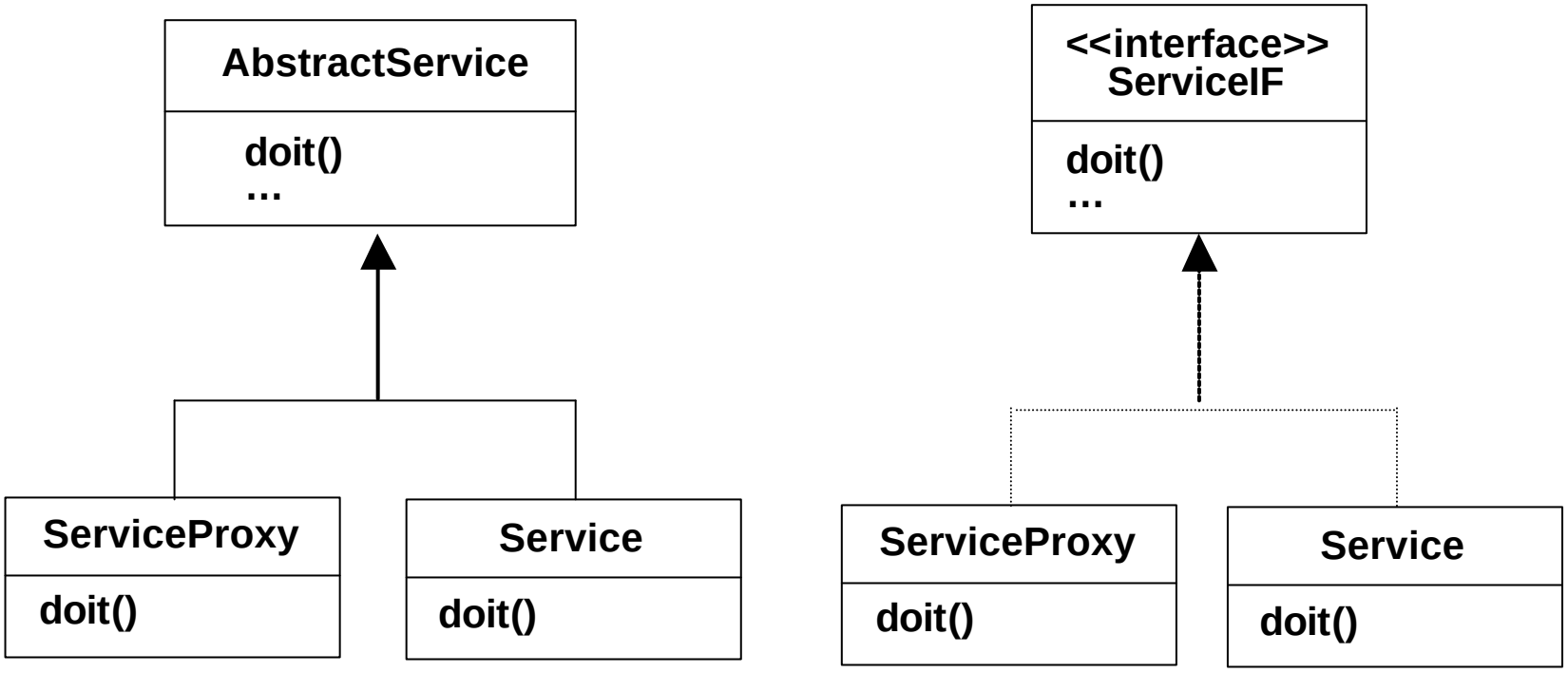
Design Pattern (계속)

- **Proxy**
  - 가장 일반적인 Pattern
  - Context
    - 클라이언트는 Proxy Object의 메소드를 호출
    - Proxy Object는 실제 서비스를 제공하는 Object의 메소드를 호출



Design Pattern (계속)

Proxy pattern Class Diagram



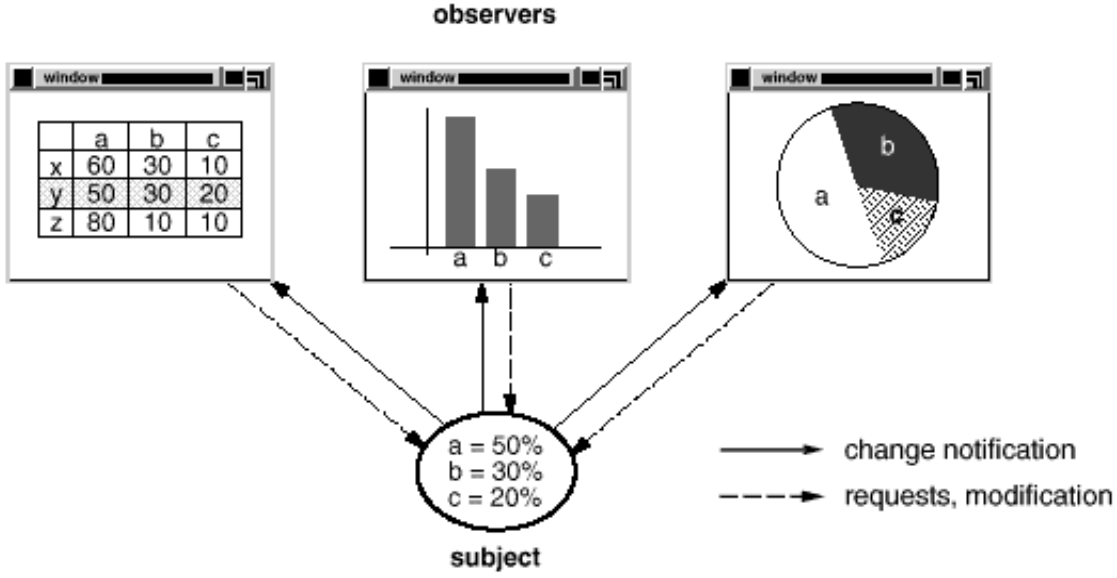
Design Pattern (계속)

Observer

동기

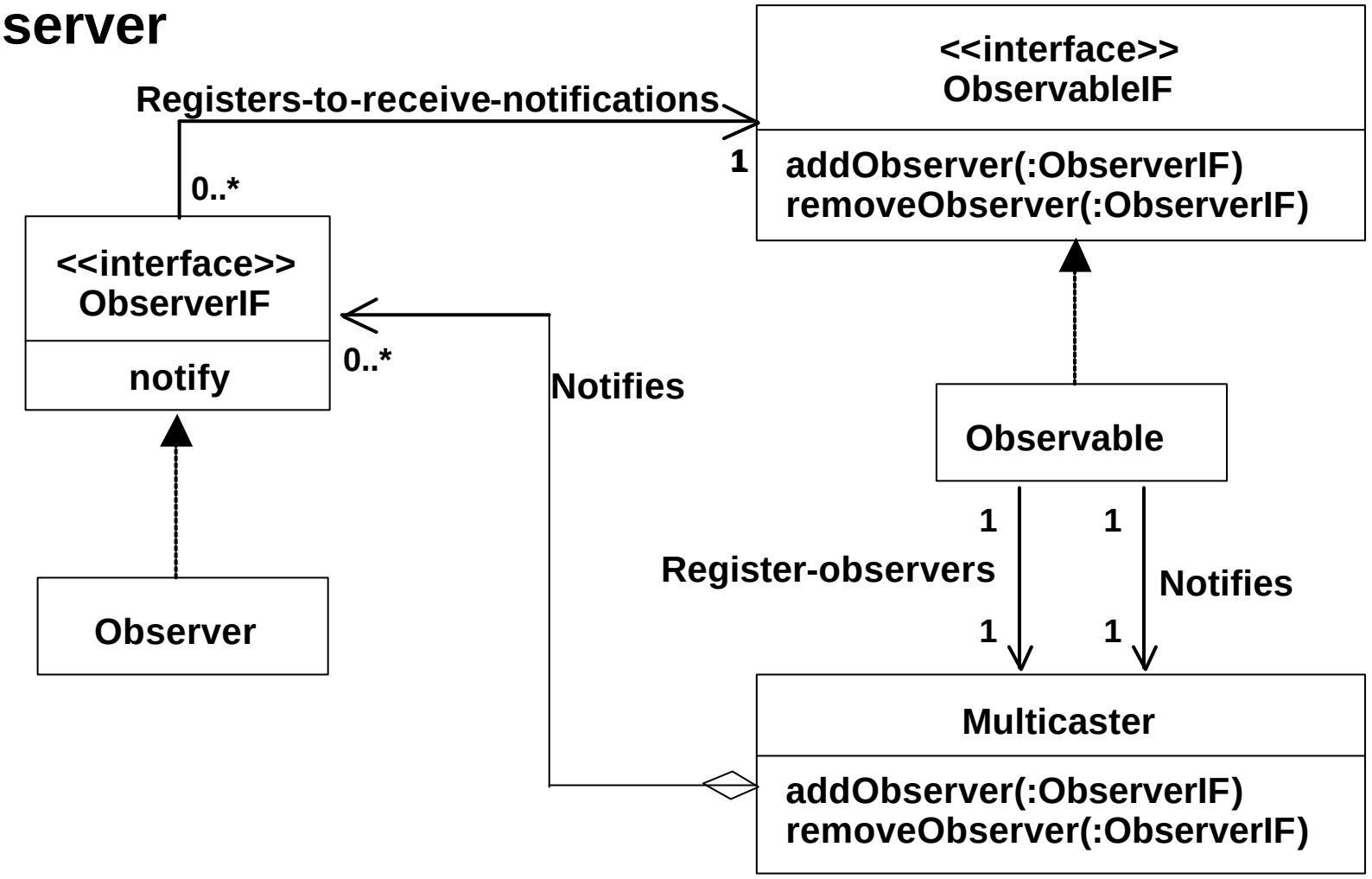
- 시스템을 협동관계가 있는 클래스들의 집합으로 분해함으로써 생기는 일반적인 부작용으로 인해 관련 있는 객체들간에 일관성을 유지해야 할 필요가 존재

publish-subscribe Relationship



Design Pattern (계속)

Observer



Design Pattern (계속)

- **Pluggable Reuse Mechanisms**
  - Inheritance vs Composition
  - Polymorphism and Delegation

Design Pattern (계속)

Inheritance vs Composition

- 객체지향 시스템에서 재사용을 위한 가장 일반적인 기법

	Class Inheritance	Object Composition
Visibility	- White-box Reuse	- Black-box Reuse - 잘 정의된 Interface가 필요
Advantage	- 프로그래밍 언어에 의해 바로 지원 - 재사용되는 구현 부분을 수정하기 용이	- don't break Encapsulation
Disadvantage	- Inheritance breaks Encapsulation - 구현 종속성이 강함 : Domain 에 따라 상위 클래스를 변경해야 할 경우 발생	- 객체의 구현은 인터페이스를 이용하기 때문에 구현 종속성이 상당히 적다.

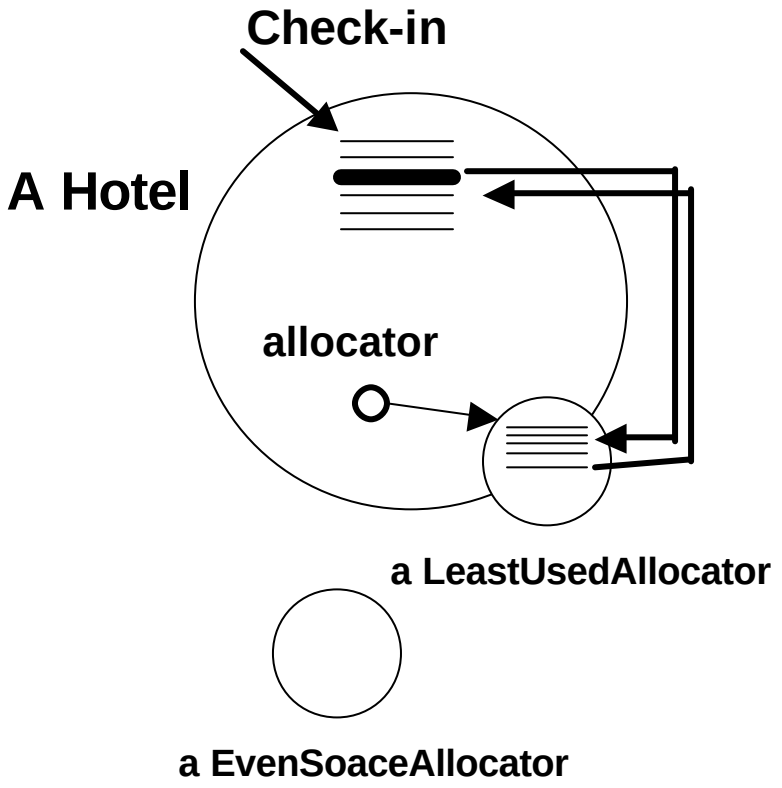
- Favor object composition over class inheritance

Design Pattern (계속)

Polymorphism and Delegation

```
Class Hotel {
    Allocator allocator;
    Public void checkInGuest (Guest g)
                {…allocator.doAllocation(g);…}
}

class Allocator {
    Room doAllocation (…); //returns a free room
}
class LeastUsedAllocator implements Allocator {
    Room doAllocation (…) {… code …}
}
class EvenSpaceAllocator implements Allocator {
    Room doAllocation (…) {… code …}
}
```



- Component Based and pure Object-Oriented approaches 에서는 Interface를 통한 Delegation–based Approach를 이용

Layered Architecture

✍ 야콥슨의 Layered Architecture

어플리케이션을 구성하는 다양한 어플리케이션 서브 시스템으로 구성

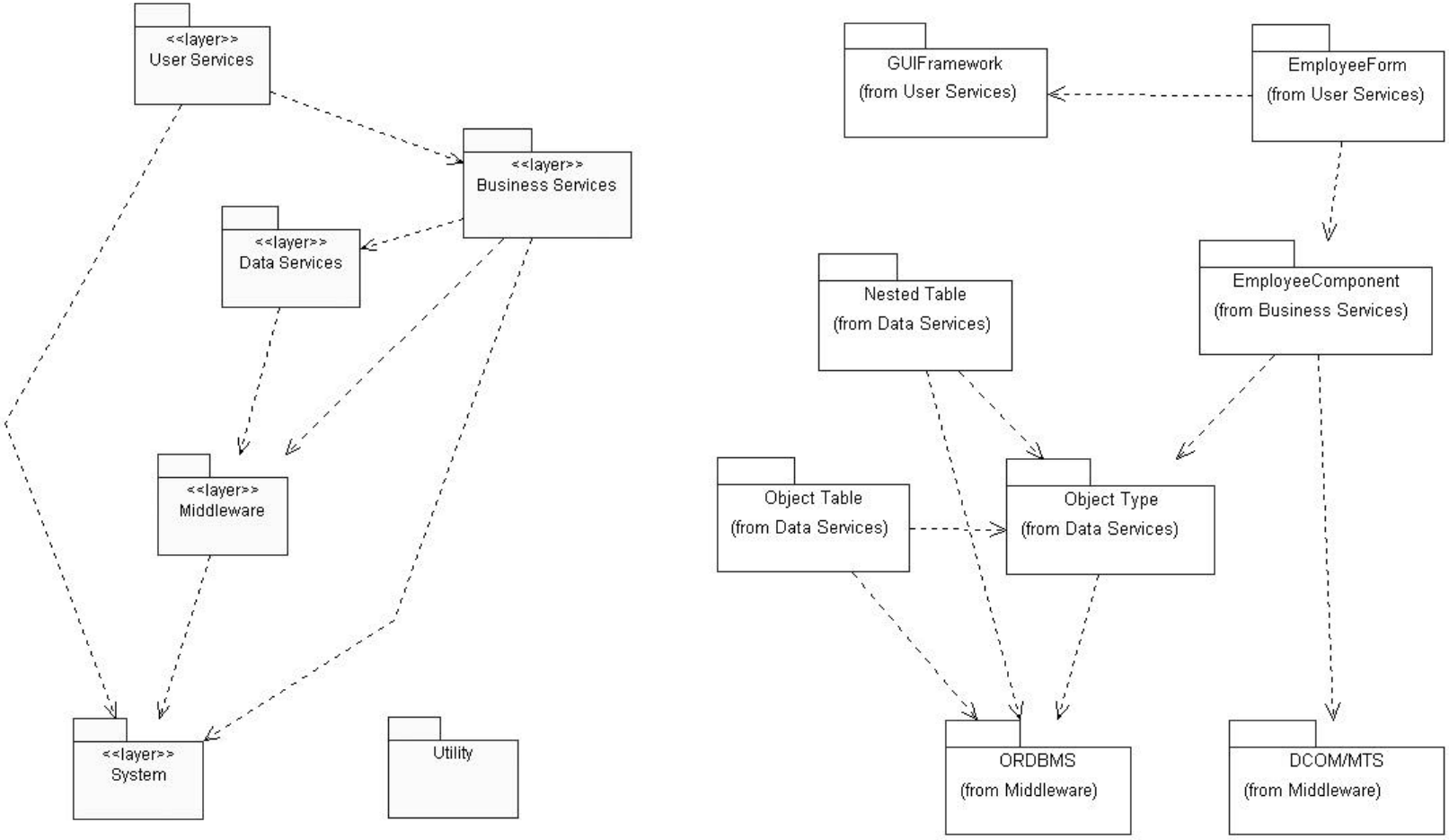
업무 영역에 구체화된 재사용 가능한 서브시스템

다양한 환경에서 분산 객체를위한 플랫폼 독립적인 서비스 제공

운영체제, 특정 하드웨어를 위한 인터페이스, 디바이스 드라이버 등

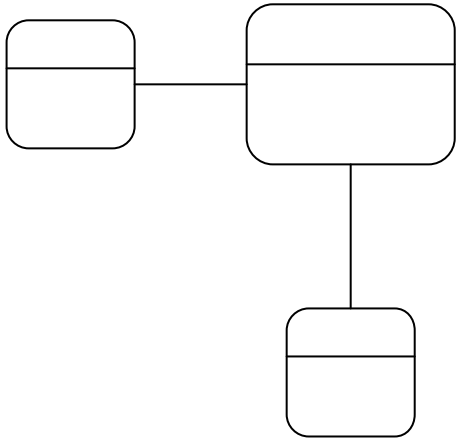
Layered Architecture (계속)

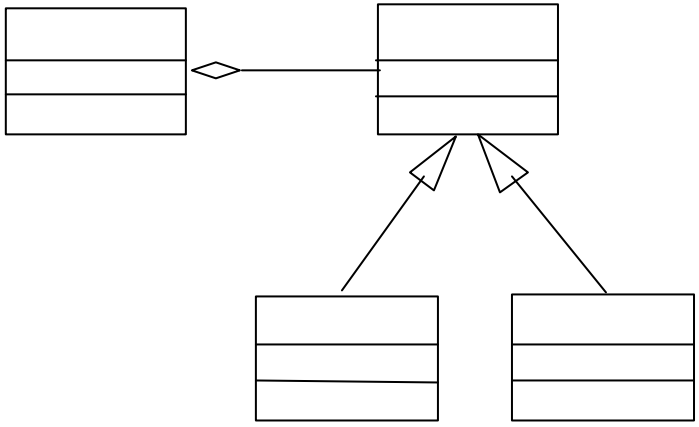
- Package의 stereotype으로 각 layer 를 표현



Object Model vs Relational Tables

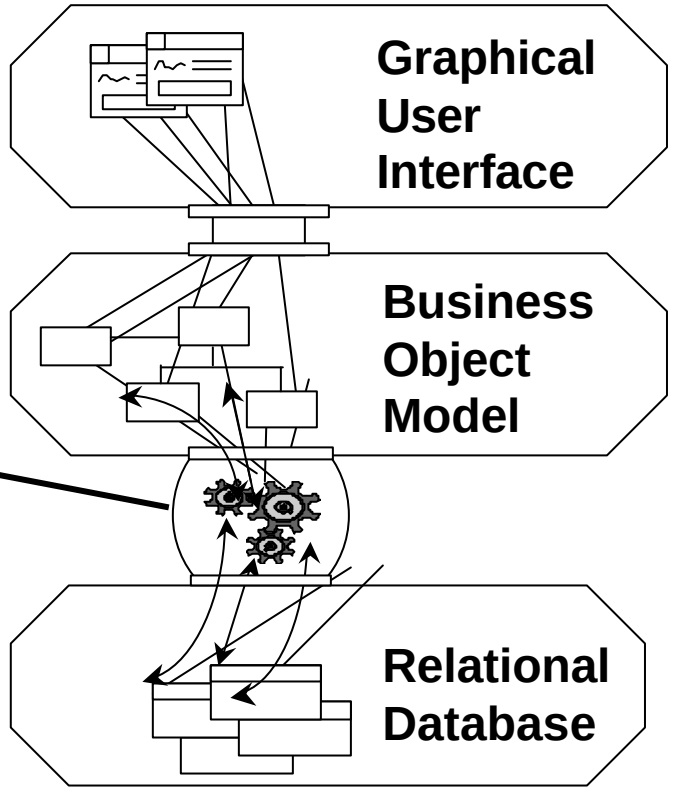
- **RDB Data Model**
  - Data 관점
  - Entity들과 Relationship으로 표현
- **Object Model**
  - 행위(behavior) 관점
  - Class들과 Relationship으로 표현





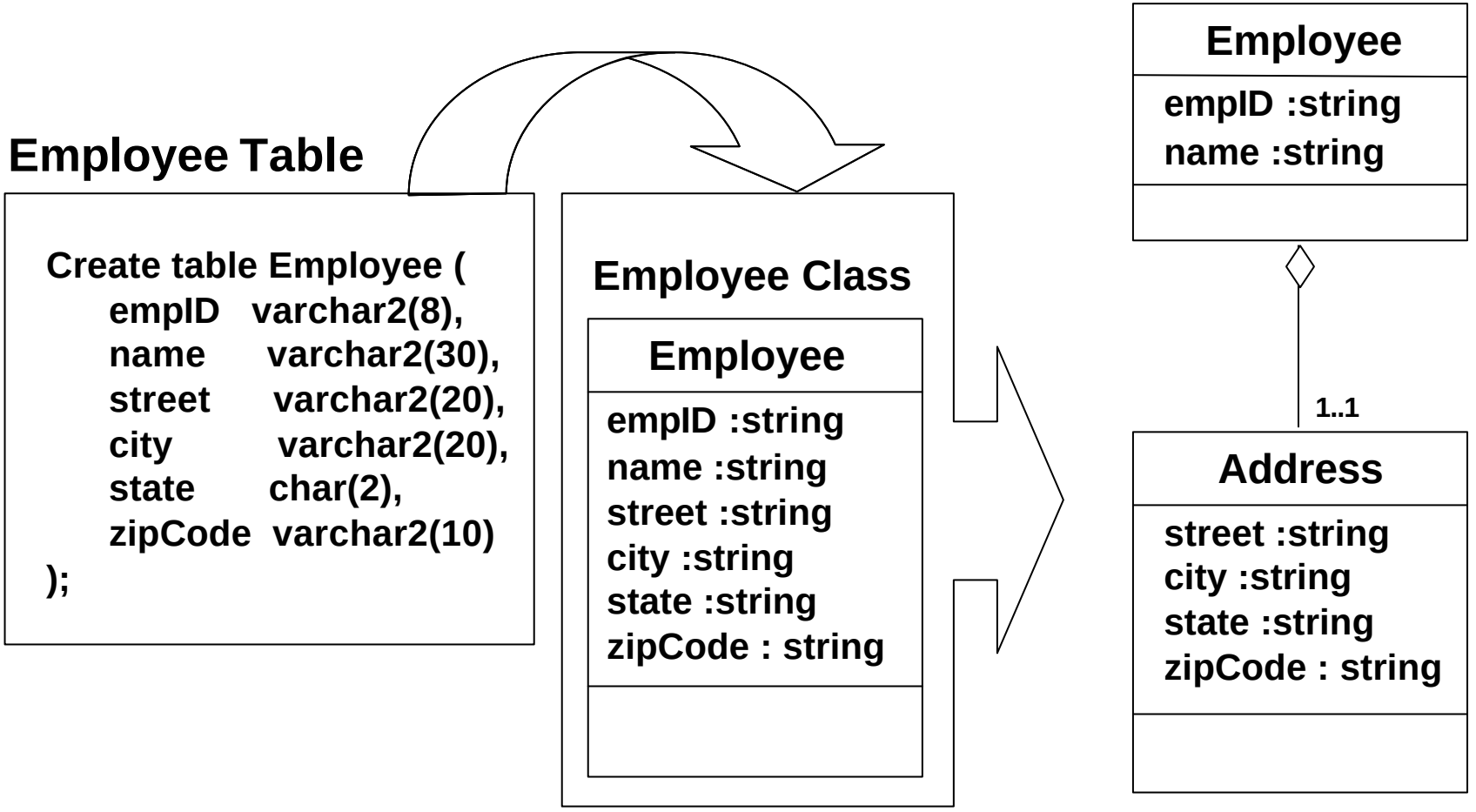
Object Model vs Relational Tables

- 완벽한 인터페이스
- Object to Relational Mapping
- 데이터 저장소의 캡슐화



Reverse Engineering Relational Tables

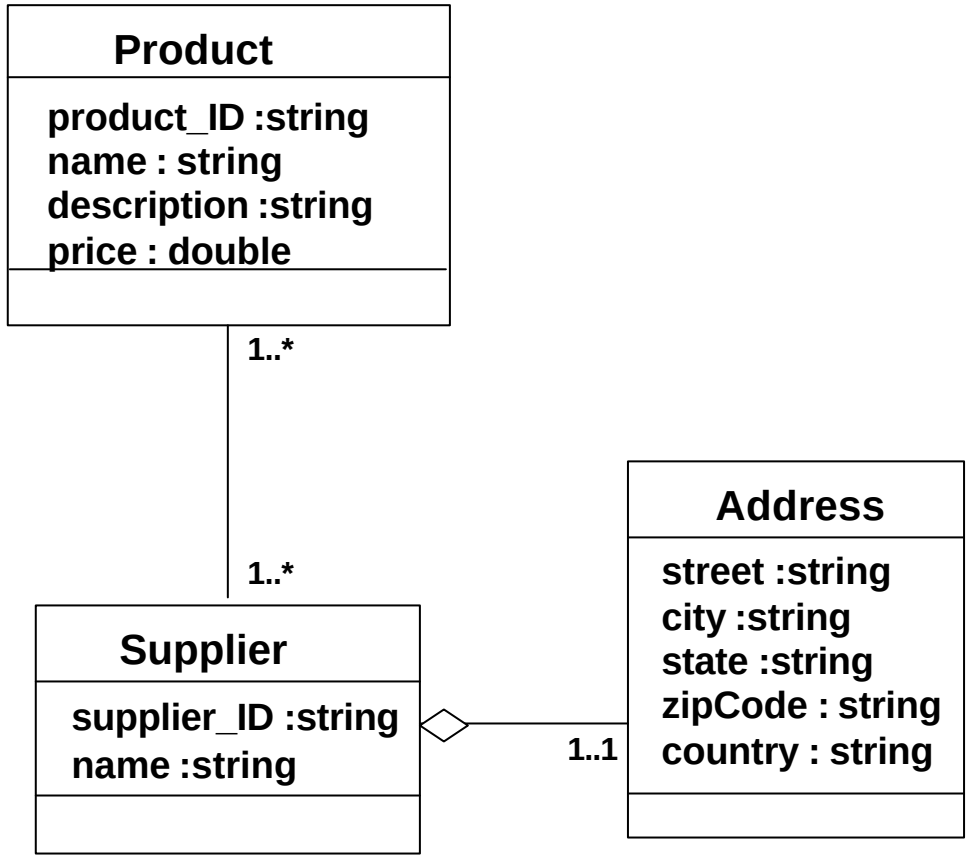
기본 Table의 Class Mapping



Reverse Engineering Relational Tables

Many-to-Many Relationships

```
Create table product (
     product_ID      varchar2(8),
     name            varchar2(30),
     description     varchar2(20),
     price           number
);
Create table supplier (
     supplier_ID     varchar2(8),
     name            varchar2(30),
     street          varchar2(20),
     city            varchar2(20),
     state           char(2),
     zipCode         varchar2(10),
     country         varchar2(20)
);
Create table product-supplier (
     product_ID      varchar2(8),
     supplier_ID     varchar2(8)
);
```



Mapping Object to Relational Tables

- **Mapping attributes to columns**
- **Mapping several classes tables**
  - Implementing inheritance in an RDB
  - Mapping several classes to a single table
- **Mapping Many-to-many relationships**

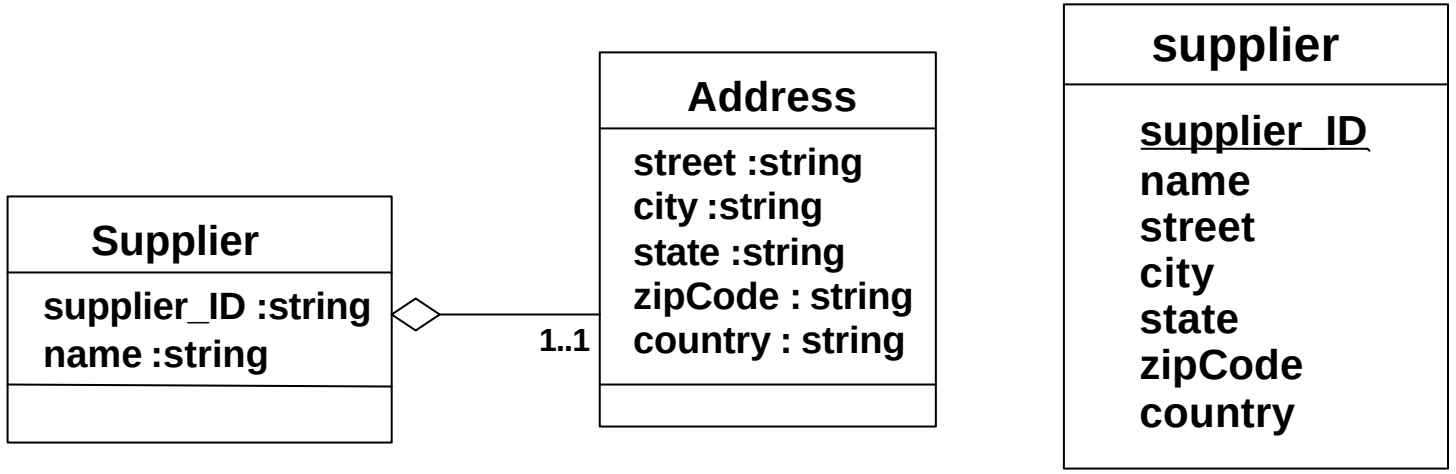
Mapping Object to Relational Tables (계속)

Mapping attributes to columns

- 클래스의 애트리뷰트는 zero or more 컬럼으로 Mapping!

- 모든 애트리뷰트가 persistent한 것은 아니다.
  : Derived attribute는 DB에 저장되지 않을 수도 있다.

- 하나의 클래스 애트리뷰트도 여러 개의 Column으로 Mapping

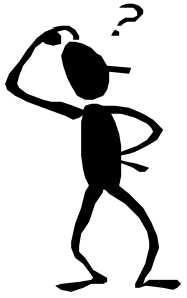


Mapping Object to Relational Tables (계속)

Mapping several classes tables

: Implementing inheritance in an RDB

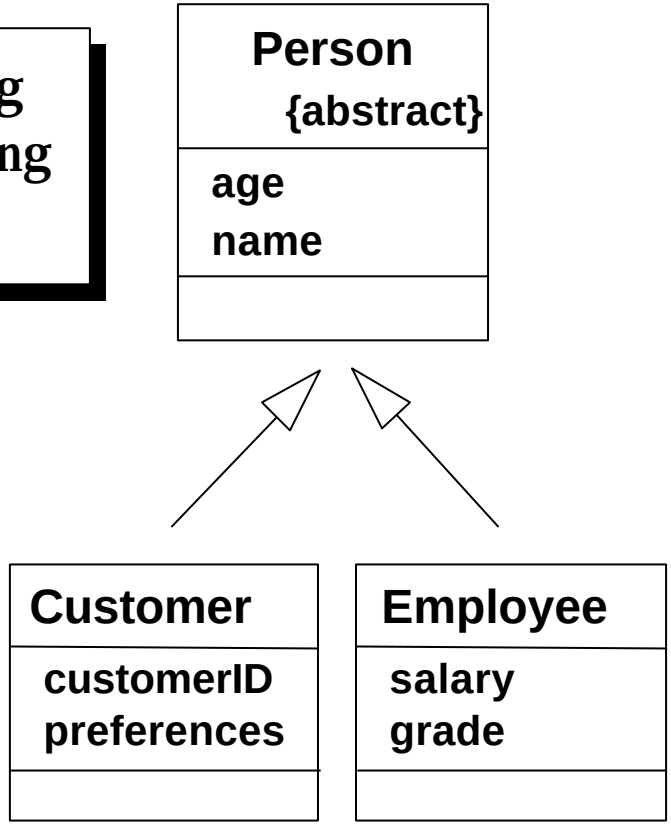
- 전체 클래스 hierarchy가 하나의 Table로 Mapping
- 실제(concrete) 클래스 당 하나의 Table로 Mapping
- 각 클래스 당 하나의 Table로 Mapping



Performance

vs

Storage Space

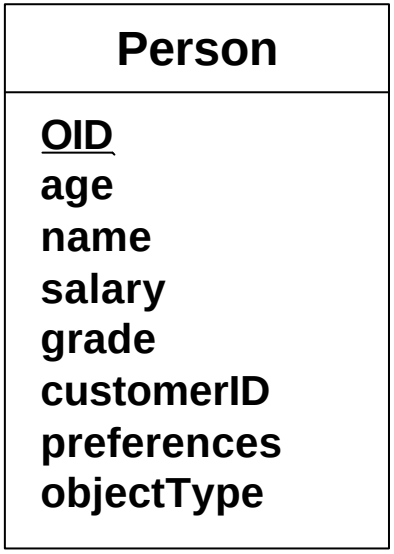


Mapping Object to Relational Tables (계속)

Implementing inheritance in an RDB

: 전체 클래스 hierarchy가 하나의 Table로 Mapping

- 장점
  - Simple
  - Polymorphism 지원
    - 역할이 바뀌거나 (고객< -> 사원)
      동시에 두개의 역할을 가질 경우
  - Ad hoc Reporting 이 편리
    - 하나의 Table만 Query
- 단점
  - Space 낭비



One table
Per Hierarchy

Mapping Object to Relational Tables (계속)

Implementing inheritance in an RDB

: 실제(concrete) 클래스당 하나의 Table로 Mapping

- 장점
  - Simple
  - Ad hoc Reporting 이 편리
    - 하나의 Table만 Query
- 단점
  - Super Class의 Attribute 추가/삭제시 여러 Table 수정
  - Multiple Role을 표현하기 어렵다.
  - Role이 바뀌면, 바뀐 테이블로 데이터를 복사한 뒤 새로운 OID를 부여해야 함

Customer
OID
age
name
customerID
preferences

Employee
OID
age
name
salary
grade

One table
Per Concrete Class

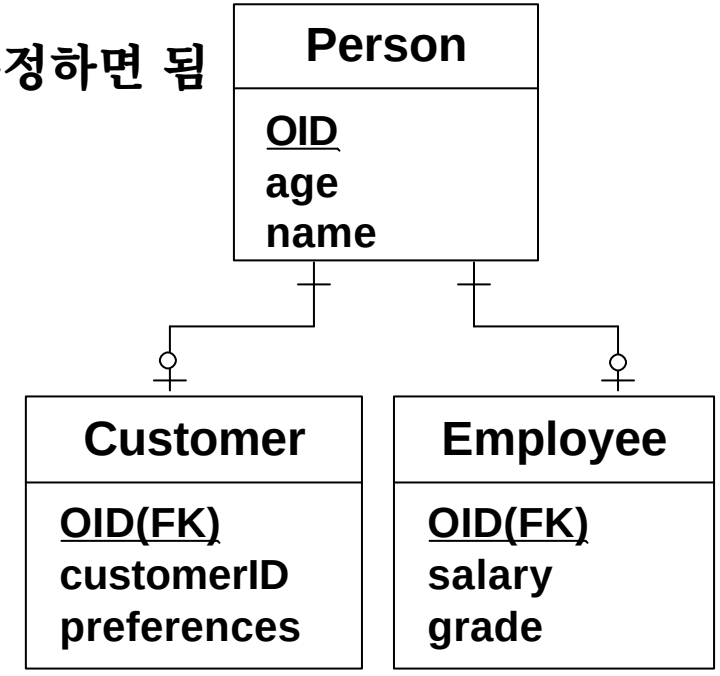
Mapping Object to Relational Tables (계속)

Implementing inheritance in an RDB

: 실제(concrete) 클래스당 하나의 Table로 Mapping

- 장점
  - OO Concept을 가장 많이 반영
  - Attribute 추가/삭제시 하나의 테이블만 수정하면 됨
- 단점
  - 테이블이 너무 많아짐
  - Read/Write 시간이 많이 걸림
    - Multiple Table access
  - Ad hoc Reporting 이 복잡

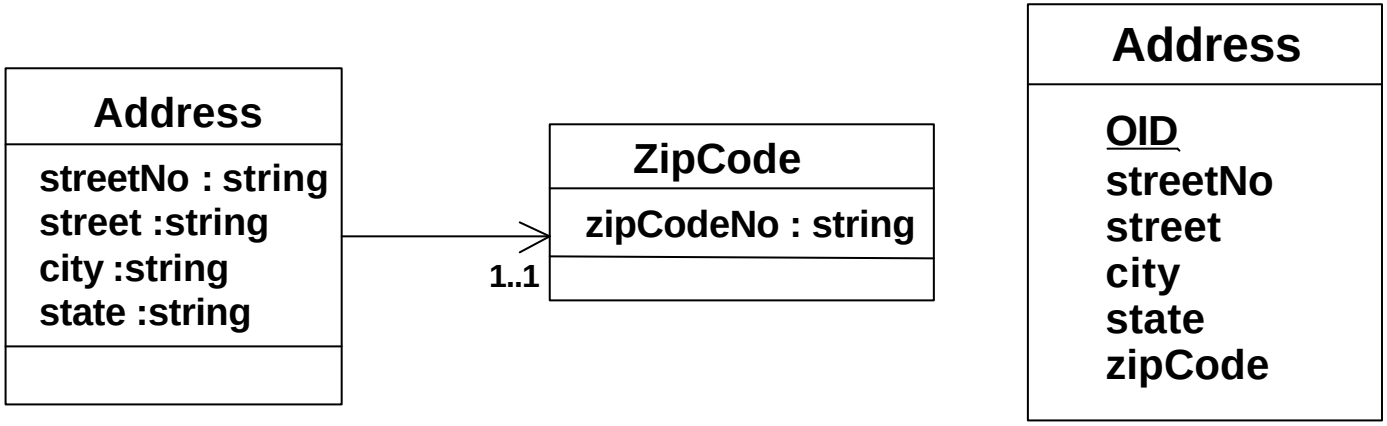
One table per Class



Mapping Object to Relational Tables (계속)

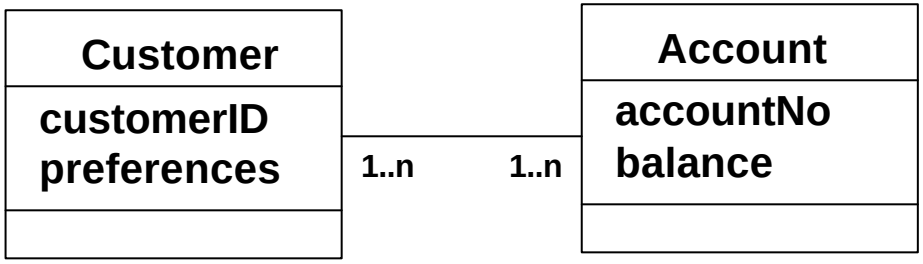
✍ Mapping several classes tables

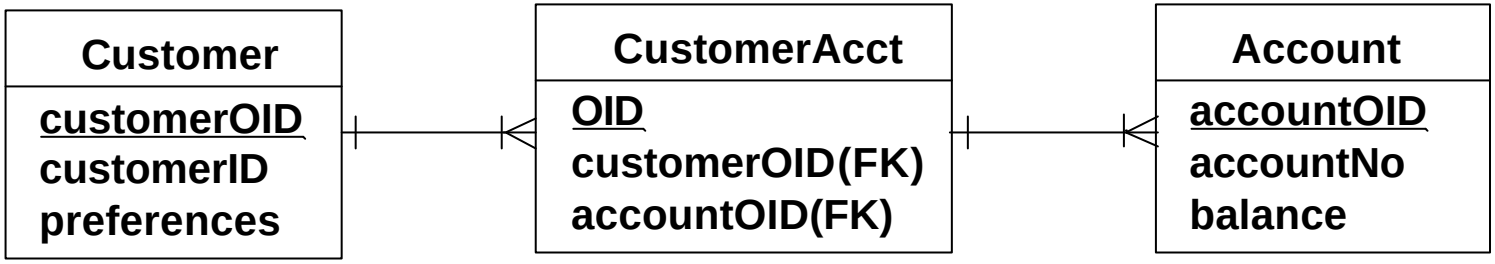
: Mapping Several Classes to One Table



Mapping Object to Relational Tables (계속)

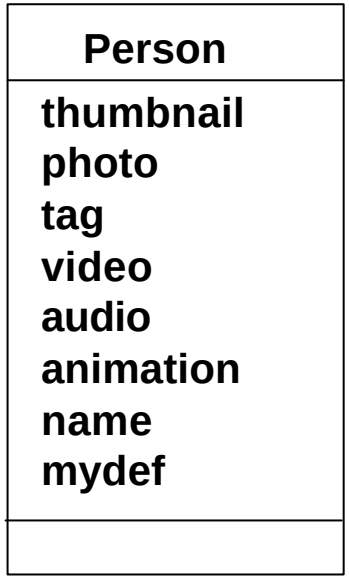
- Mapping Many-to-many relationships
  - Table 추가 생성





Object in Jasmine OODB (계속)

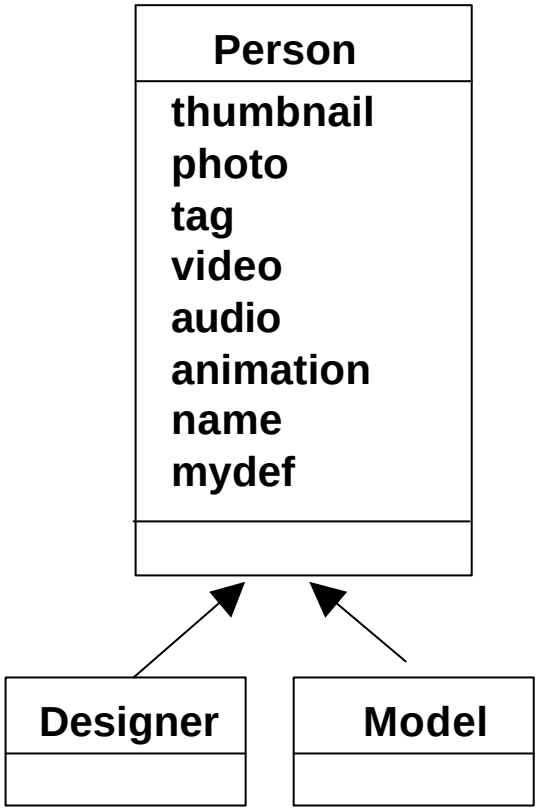
✍ Class 정의




```
DefineClass MyStore::Person
Description: “person abstract class”
{
Instance:
    mediaCF::Bitmap      thumbbail;
    mediaCF::Bitmap      photo;
    mediaCF::Bitmap      tag;
    mediaCF::Video       video;
    mediaCF::Audio       audio;
    mediaCF::Animation   animation;
    String               name;
    String               mydef;
}
```

Object in Jasmine OODB (계속)

Inheritance Relationships

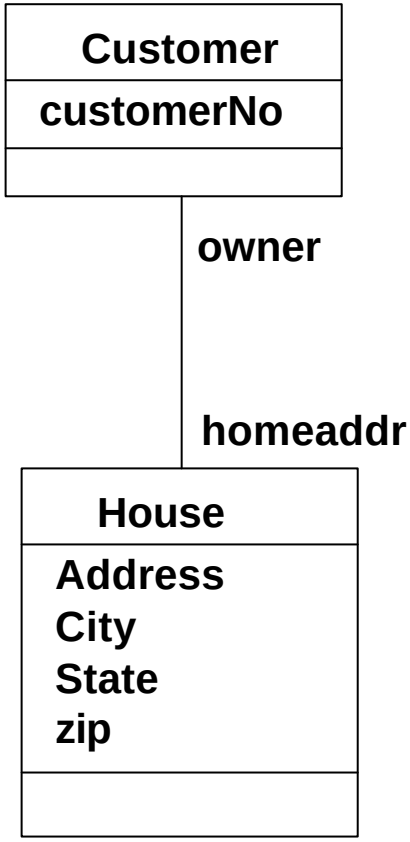



```
defineClass MyStore::Designer
super:Person
description: “Designer class”
{};

defineClass MyStore::Model
super:Person
description: “Model class”
{};
```

Object in Jasmine OODB (계속)

One-to-One Relationships

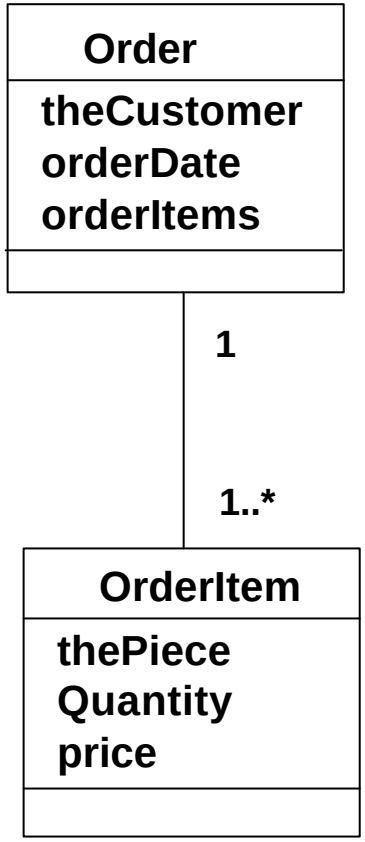



```
defineClass MyStore::Customer
super:Person
description: "고객정보"
{
Instance
    String customerNo;
    House homeaddr;
} ;

defineClass MyStore::House
description: "고객주소"
{
Instance
     Customer owner;
     String address ;
     String city;
     String state ;
     String zip;
} ;
```

Object in Jasmine OODB (계속)

One-to-many Relationships

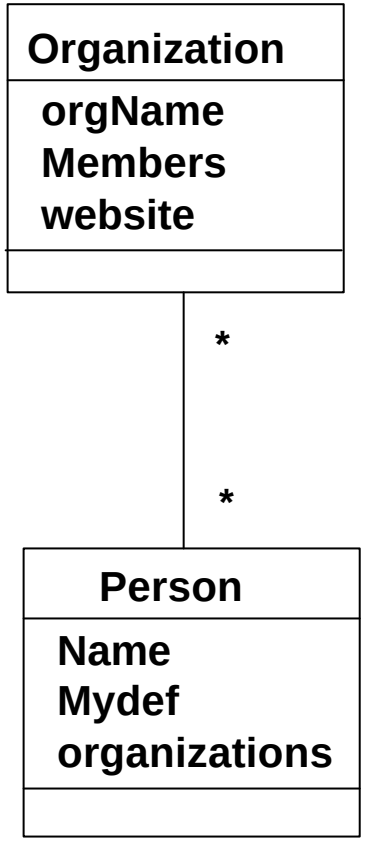



```
defineClass MyStore::Order
description: "주문정보"
{
Instance
    Customer theCustomer;
    String orderDate;
    OrderItem set orderitems;
} ;

defineClass MyStore::OrderItem
description: "주문 상세"
{
Instance
     Piece thePiece;
     Integer quantity ;
     Currenct price;
} ;
```

Object in Jasmine OODB (계속)

One-to-many Relationships




```
defineClass MyStore::Organization
description: "조직"
{
Instance
    String orgName;
    Person set members;
    String website;
} ;

defineClass MyStore::Person
description: "사람"
{
Instance
     String Name;
     String mydef ;
     Organization organizations;
} ;
```

Ⅳ. 클래스 설계

- 클래스
  - 오퍼레이션, 애트리뷰트
  - Utility, Abstract, Concrete
- 관계성
  - Multiplicity, Navigability,
  - 연관관계, 포함관계, 상속관계, 종속관계
- Statechart Diagram

클래스

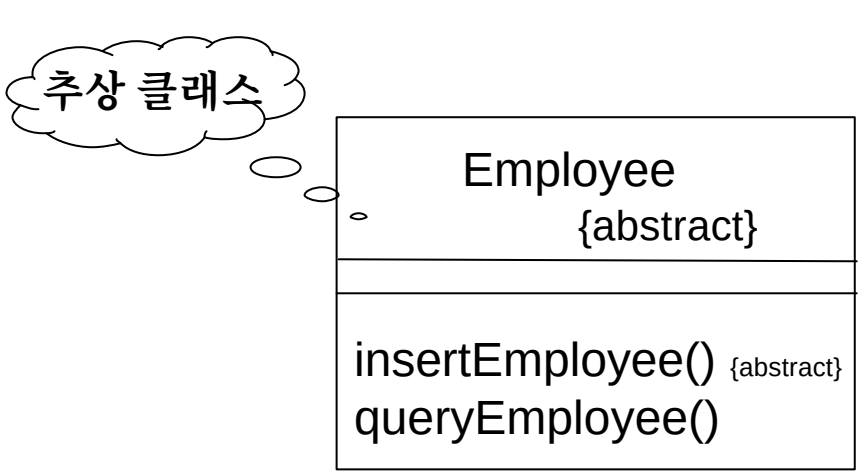
정의

- 클래스는 특정 관점에서 공통적인 속성, 행위, 의미를 가지는 객체 집합에 대한 명세(Description)
- 구현될 시스템내의 주요 개념(Concepts)
- 소프트웨어적, 하드웨어적, 개념적 실체 표현 가능

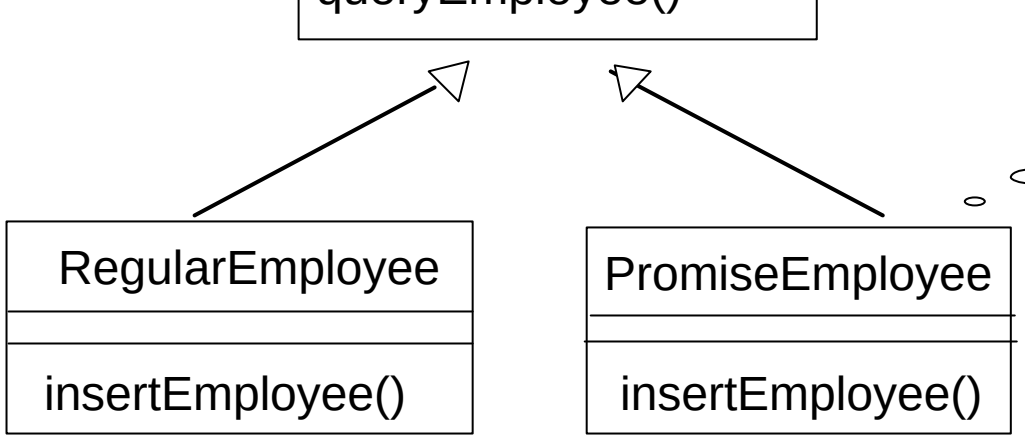
표기법

클래스 (계속)

✍ 추상 클래스와 실체 클래스




```
abstract class Employee
{
    public abstract void insertEmployee();
    public queryEmployee() { }
}
```




```
class Employee
{
    public void insertEmployee() { }
}
```

클래스 (계속)

- 은닉성(Encapsulation)
  - 클래스는 "인터페이스(Interface)와 구현(Implementation)으로 구성된 것"으로 볼 수 있음
    - 인터페이스
      - 다른 객체들이 볼 수 있고, 사용할 수 있는 요소
    - 구현
      - 다른 객체들에 대해 은폐되어 있고 인터페이스에 대한 서비스 구현
  - 객체의 구체적인 구현부분을 은폐(hiding)
    - Encapsulation [Information hiding]
  - Encapsulation이 제공하는 두 가지 종류의 Protection
    - 다른 객체에 의한 객체 내부 상태 (Internal state)의 Corruption 방지
    - 객체 구현 변경에 따른 클라이언트 코드의 변경 최소화

클래스 (계속)

- **Visibility 정의**
  - Operation visibility 는 Encapsulation 개념을 강조
  - 종류
    - Public : 모든 클라이언트에서 접근 가능
    - Protected : 서브 클래스의 인스턴스에 의해서만 접근 가능
    - Private : 자신의 인스턴스에서만 접근 가능

클래스 (계속)

- 클래스 스테레오타입
  - 정의
    - UML의 확장 메커니즘
    - 클래스, 컴포넌트, 노드 등의 모델요소 들을 의미적인 관점에서 분류한 것
    - UML에서 약 40여 가지의 스테레오타입을 미리 정의하고 있으며, 분석가의 주관적인 관점에서 추가적인 정의도 가능
    - 모든 모델 요소들은 적어도 하나 이상의 Stereotype을 가질 수 있음
  - 종류
    - 바운더리 클래스(Boundary class)
    - 콘트롤 클래스(Control class)
    - 엔터티 클래스(Entity class)
    - 예외처리 클래스(Exception class)
    - 메타 클래스(Meta class)
    - 유틸리티 클래스(Utility class)

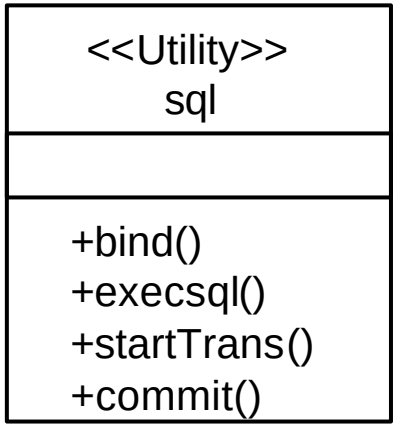
클래스 (계속)

Stereotype (계속)

표기법

클래스 (계속)

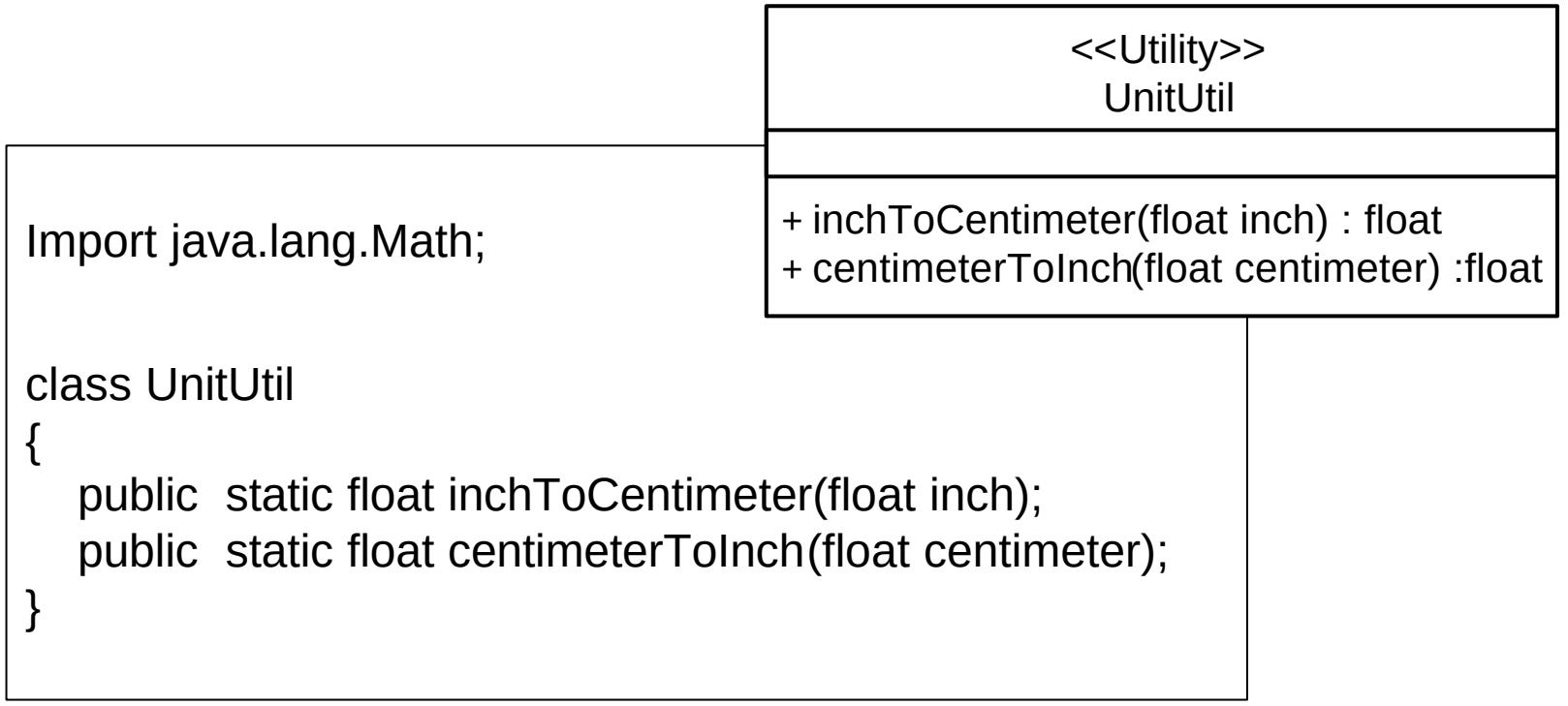
- 유틸리티 클래스
  - 특징
    - Global Attribute와 Global Operation를 제공
    - <<Utility>> 라는 클래스 스테레오 타입으로 표현
    - 유틸리티는 모델링측면의 기본 구조보다는 프로그래밍 측면에서 고려
    - 유틸리티 클래스는 Instance가 없다.
  - 목적
    - 일반적인 알고리즘 서비스 제공
    - 객체지향 라이브러리나 어플리케이션이 아닌 것들을 Wrapping



클래스 (계속)

Class Utility 예

거리측정 단위를 변환하는 함수를 단일 인터페이스를 갖도록 package화해서 생성한 Class Utility



오퍼레이션

- 정의
  - 클래스의 행위를 정의하는 책임(Responsibility)들의 집합
  - 이름, 파라메터, 리턴타입 으로 구성

[<<stereotype>>] [visibility] name [(parameter_list, …)] [: return_type]

- 클래스의 책임은 오퍼레이션을 통해 수행됨
  - 책임과 오퍼레이션은 반드시 1:1로 매핑될 필요는 없음
  - 예
    - "제품" 클래스의 책임: 가격정보를 제공
    - 책임 수행을 위한 오퍼레이션 : getInformation(), getPrice()

오퍼레이션 (계속)

- 오퍼레이션은 업무 영역(Business Domain)에 종속적임
- 예
  - "고객" 클래스의 오퍼레이션
    - 해당 클래스가 속해 있는 업무영역에 따라 각기 달라짐

"은행고객"의 관점

고객
계좌이체하다() 대출하다() 입금하다()

"병원고객"의 관점 ●●●

고객
약을사다() 진료받다()

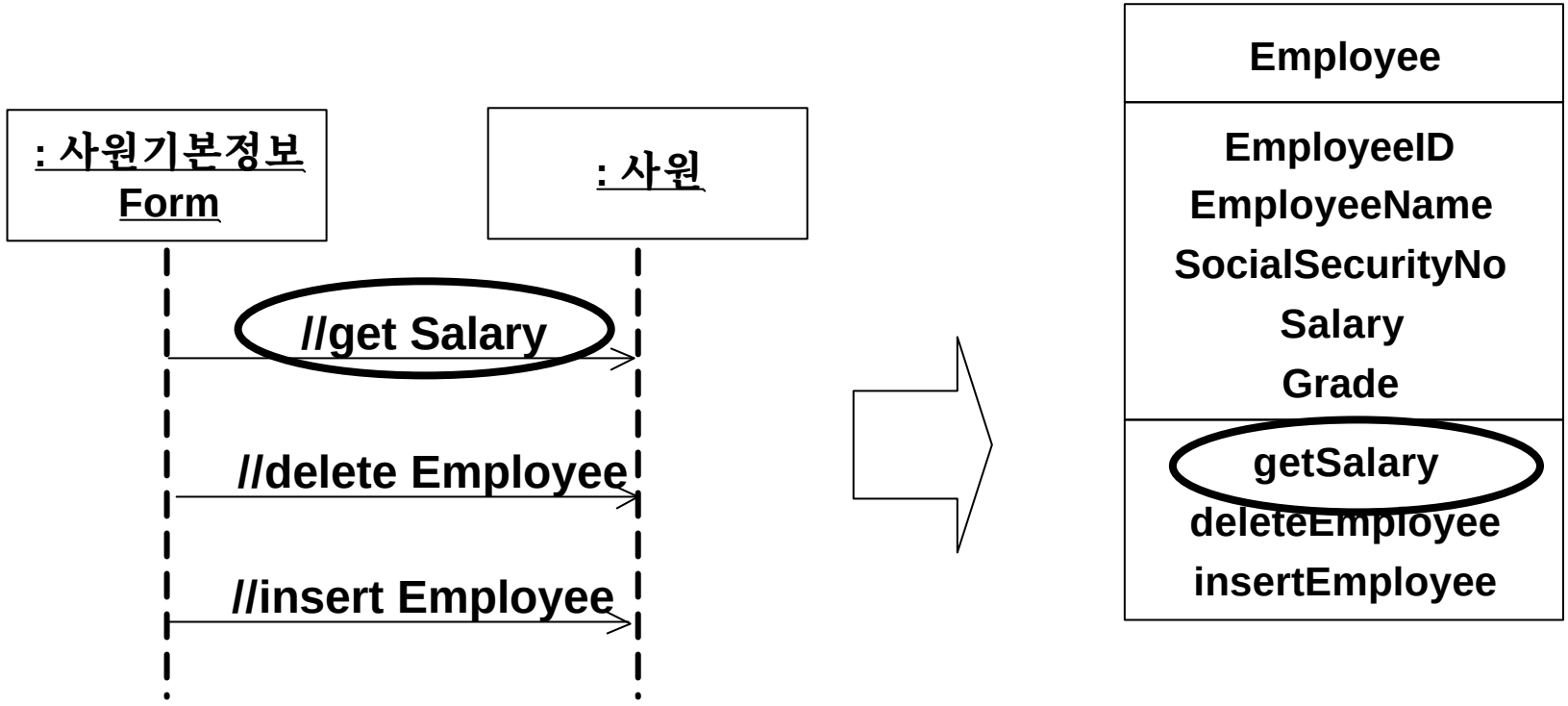
●●●

오퍼레이션 (계속)

- 명명 규칙(Naming Rules)
  - 수행 결과가 표시되도록 할 것
    - Poorly named
      - calculateBalance()
      - Balance가 반드시 계산되어야 함을 강조
    - Well named
      - getBalance()
      - Balance(결과)를 얻기 위한 Operation임을 의미
  - 공급자(Supplier)의 관점
    - Good names
      - dispense(), giveOil()
    - Bad names
      - receiveOil()

오퍼레이션 (계속)

- **Interaction Diagram으로 부터 오퍼레이션 추출**
  - Interaction Diagram에 정의된 메시지는 수신하는 클래스의 오퍼레이션이 됨



애트리뷰트

- 정의
  - 클래스의 인스턴스에 적용되는 데이터 정의(Data Definition)
  - Attribute 명은 의미가 명확한 명사형으로 정의
    - 클래스내에서 유일
  - 연관된 관계도 애트리뷰트가 됨
    - "학생" 클래스의 애트리뷰트
      - 성명 (Attribute)
      - 전공 (Attribute)
      - 소속부서 (Relationship)

애트리뷰트 (계속)

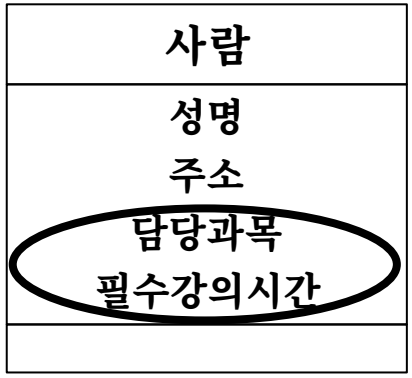
- **Attribute values**
  - 특정 객체의 상태 값
  - 모든 객체는 해당 클래스에 정의된 모든 Attribute에 대해 값을 가짐
  - 예
    - "사원" 클래스의 Attribute
      - 사번
      - 사원이름
      - 급여

애트리뷰트 (계속)

- **Attribute는 업무영역(Business Domain)에 종속적**
  - **예**
    - "사람" 클래스의 Attribute

"은행고객"의 관점

"교수"의 관점

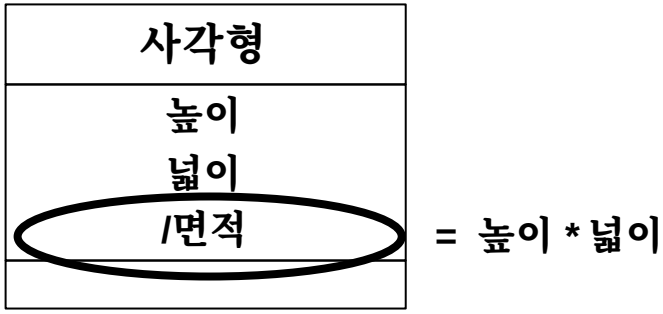


●●●

애트리뷰트 (계속)

Derived Attribute

- 다른 Attribute들의 값으로 부터 계산된 값을 가지는 Attribute
  (Attribute 이름 앞에 "/"를 붙여 표시)
- 값이 필요할 때마다 다시 계산할 수 있는 시간적 여유가 없을 경우 주로 사용
- 필요한 메모리 용량과 시스템 성능을 고려하여 결정해야 함
- 예
  - "사각형" 클래스의 Attribute



애트리뷰트 (계속)

✍ 데이터 타입 & 초기값

- ✍ 모든 Attribute는 Data type과 Initial value를 가짐

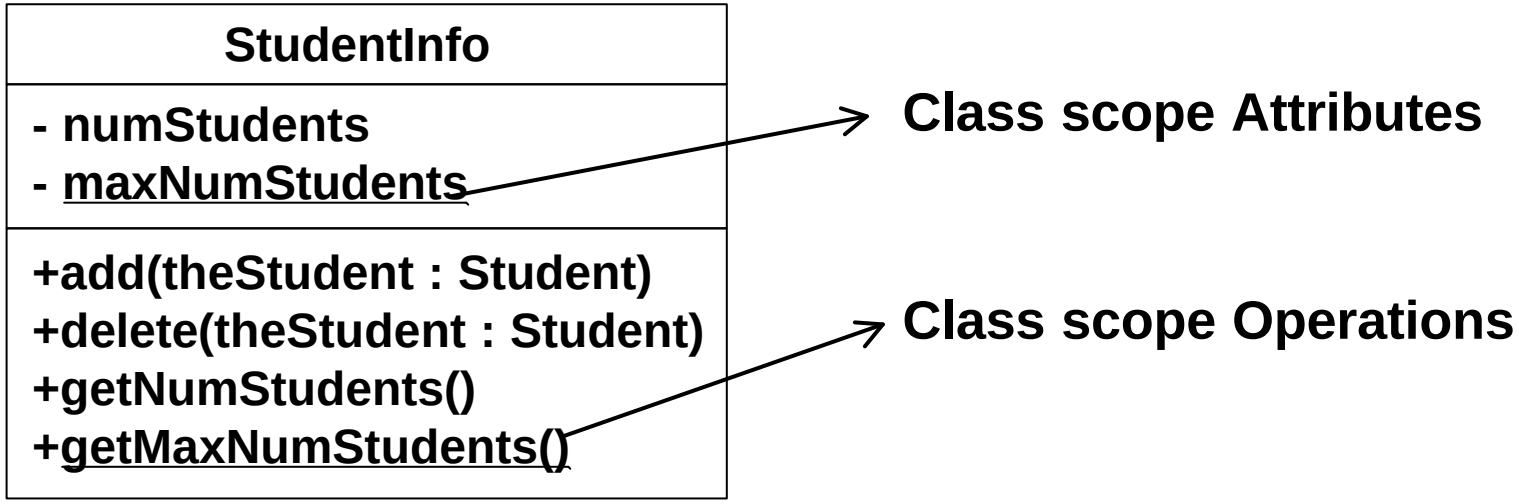
[visibility] AttributeName [multiplicity] [: Type] [= Default]

✍ Attribute 추출

- ✍ 대부분의 Attribute는 유즈 케이스에 나타난 “Flow of Event”에서 도출
  - ✍ 명사 중심으로
- ✍ 클래스 설계단계에서 추가적으로 도출 가능
  - ✍ 오퍼레이션 파라메터 및 리턴 값에 의한 클래스 도출 가능
- ✍ 업무영역에 대한 전문가[실무자]가 좋은 Attribute를 찾아 줄 수 있음
  - ✍ Attribute는 문제영역에 종속적이기 때문

범위

- 클래스 오퍼레이션 (Class Scope Operation)
  - 클래스의 인스턴스가 생성되지 않더라고 호출 가능
  - 클래스범위의 애트리뷰트 만 접근 가능



- getMaxNumStudents() 오퍼레이션은 maxNumStudents 애트리뷰트만 접근 가능

범위 (계속)

클래스 오퍼레이션 (계속)

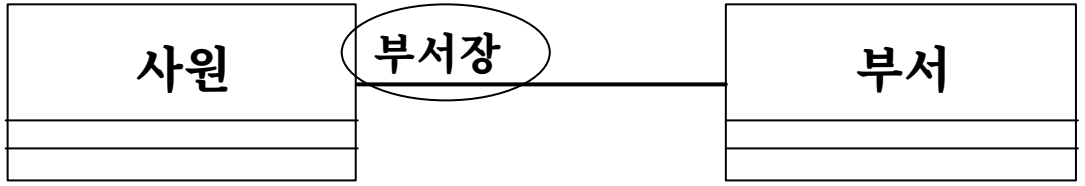
```
// StudentInfo class define
public class StudentInfo {
   private static int maxNumStudents = 100;
   private int numStudents;
   public StudentInfo() {}
   public static int getMaxNumStudents() {
            return maxStudent;
         }
}
// “getMaxNumStudents” Class Method call
public static void main(String[] args) {
        System.out.println("Class's value is : "+StudentInfo.getMaxNumStudents());
}
```

관계성

- 정의
  - 객체들은 서로 연관이 되어서 시스템의 행위를 지원하며, 객체들간의 Collaboration은 Relationship을 통해 형성
- 종류
  - 연관관계
    - Independent link
  - 포함관계
    - Whole-part Relationships
  - 상속관계
    - Is-A Relationships
  - 종속관계

관계성 (계속)

- 역할(Role)
  - Association 관계에 있는 클래스들에 대해 해당 클래스가 갖는 목적/역할을 가리킴
  - 역할 이름은 명사(구) 형태로 표현
  - 역할 이름은 Association 관계에 있는 양쪽 클래스 모두가 가질 수도 있고, 한쪽 클래스만 가질 수도 있음



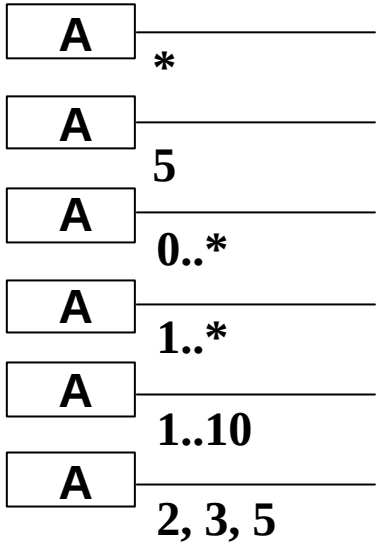
관계성 (계속)

- 역할 (계속)
  - 역할이름은 애트리뷰트 명으로 사용된다.

```
class Department{
    private String deptID ;
    private String deptName ;
    private Employee theManager ;
    ...
}
```

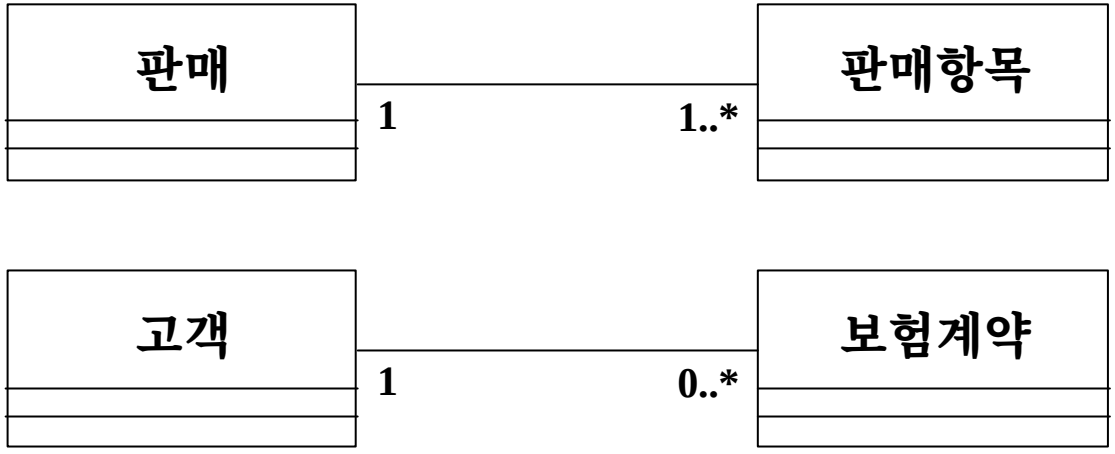
관계성 (계속)

- **Multiplicity**
  - 정의
    - 클래스와 클래스가 인스턴스화되어 연결관계를 맺을때, 관계에 참여하는 객체의 개수를 정의
    - 특정 클래스의 인스턴스에 대해 Association 관계에 있는 다른 클래스의 인스턴스가 몇 개 관련되어 있는가를 의미
  - 모든 Association에 대해 서로 간의 Multiplicity를 고려
  - 종류
    - Many (*)
    - Exactly five (5)
    - Zero or more (0..*)
    - One or more (1..*)
    - One to ten (1..10)
    - Exactly two, three or five (2,3,5)



관계성 (계속)

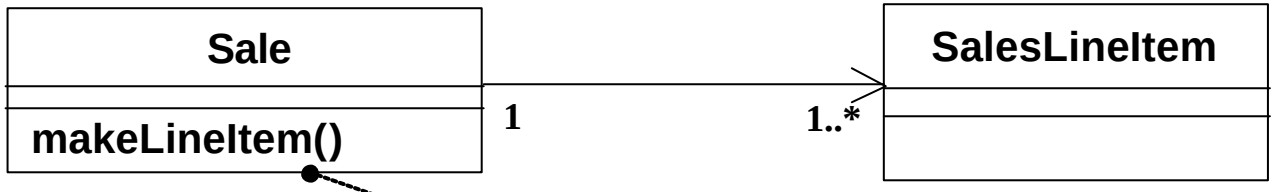
- **Multiplicity (계속)**
  - Multiplicity가 의미하는 것
    - Association의 Mandatory / Optional 여부
    - Minimum / Maximum Instance의 수
  - 예



관계성 (계속)

Multiplicity (계속)

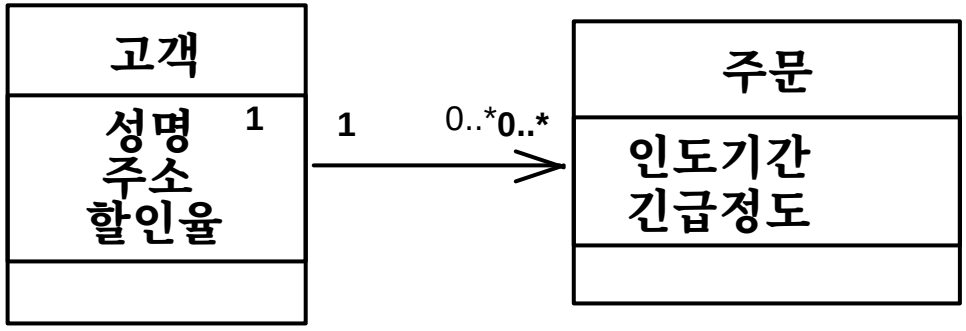
one-to-Many Relationship : Container Class 구현




```
Import java.util.*;
class Sale{
   private Vector lineItems ;
   ....
   public makeLineItem(ProductSpecification spec, int quantity)
   {
         lineItems = addElement(new SalesLineItem(spec, quantity));
    }
    ...
}
```

관계성 (계속)

- **Navigation**
  - 분석단계에서, Association은 양방향성(bi-directional)
  - 설계단계에서, Association은 단방향성(uni-directional)



관계성 (계속)

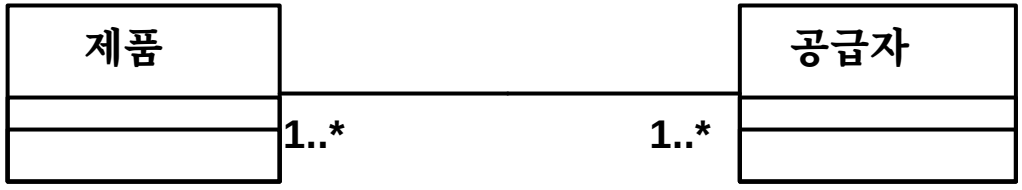
Navigation (계속)

Navigation Decision

- 설계단계에서, 양방향으로 Navigable할 필요가 있는지를 검토
- Use Case와 Scenario에 표현되는 navigation
  - 특정 ClassA Instance에 대해, ClassB의 관련된 모든 Instance들을 파악할 필요가 있는가?
  - 특정 ClassB Instance에 대해, ClassA의 관련된 모든 Instance들을 파악할 필요가 있는가?

Navigation Question

- 내가 사고자 하는 물건을 어떤 공급자에게 구매해야 하는가?
- 이 공급자로 부터 구매 가능한 물품은 무엇이 있는가?

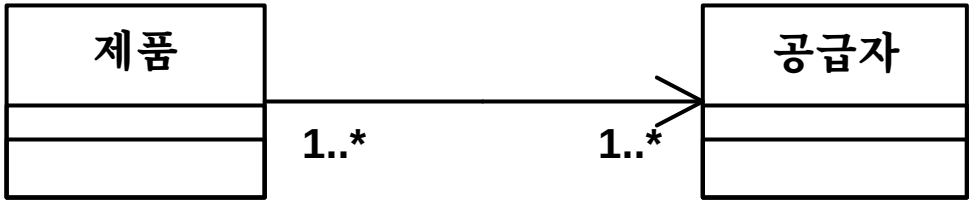


관계성 (계속)

- **Navigation (계속)**
  - 고려사항
    - Two-way Navigation은 One-way Navigation보다 구현하기가 복잡하고 많은 노력이 필요
    - 만약 Two-way Navigation이 필요하다고 하더라도, 특정 상황을 고려하여 One-way Navigation으로 구현하는 것이 좋음
  - 판단 기준
    - 둘 중 어느 한쪽으로의 navigation이 매우 빈번히 발생
    - 두개의 클래스 중 어느 하나의 클래스 Instance 발생 수가 적음

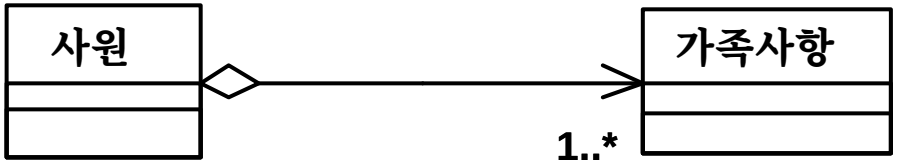
관계성 (계속)

- **Navigation (계속)**
  - 경우 1: Navigation Frequency 고려
    - 어떤 제품을 공급하는 공급자에 자주 주문을 요청하지만, 그 공 급자가 공급하는 모든 제품정보를 열람하는 일은 송장처리를 위해 분기마다 발생
    - Product-to-Supplier Direction 구현
  - 경우 2: Instance 고려
    - 공급자가 두 곳만 존재
    - Product-to-Supplier Direction 구현



관계성 (계속)

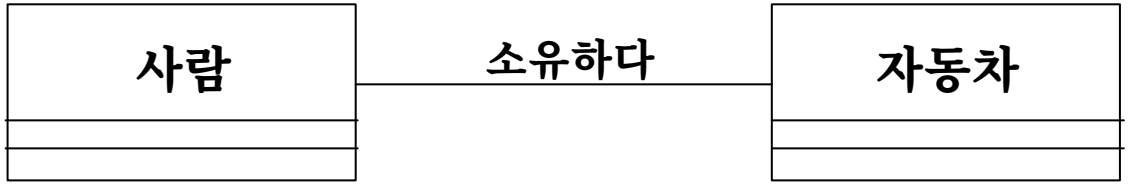
- **Navigation (계속)**
  - Aggregation의 경우
    - Aggregation도 단방향성(uni-directional)이 되어야 함
    - Aggregation Line에 화살표가 추가



연관관계

- 정의
  - 클래스간에 표현되는 개념적 관계
  - 클래스간의 의미적 연결(bi-directional)
    - Association 관계에 있는 클래스들의 객체간에도 link가 존재함을 의미

- **Notation**

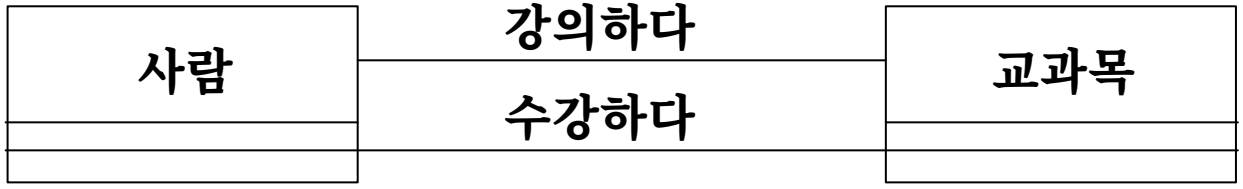


  - Class Diagram에서 표현됨
  - Data flow는 link를 따라 단방향/양방향 모두 가능
  - 의미를 명확히 하기 위해 이름을 부여
    - 동사(구) 형태

연관관계 (계속)

Multiple Associations

- 클래스간에 두 가지 이상의 Association이 존재하는 경우
- 각 Association들 마다 반드시 Association Name을 부여할 것

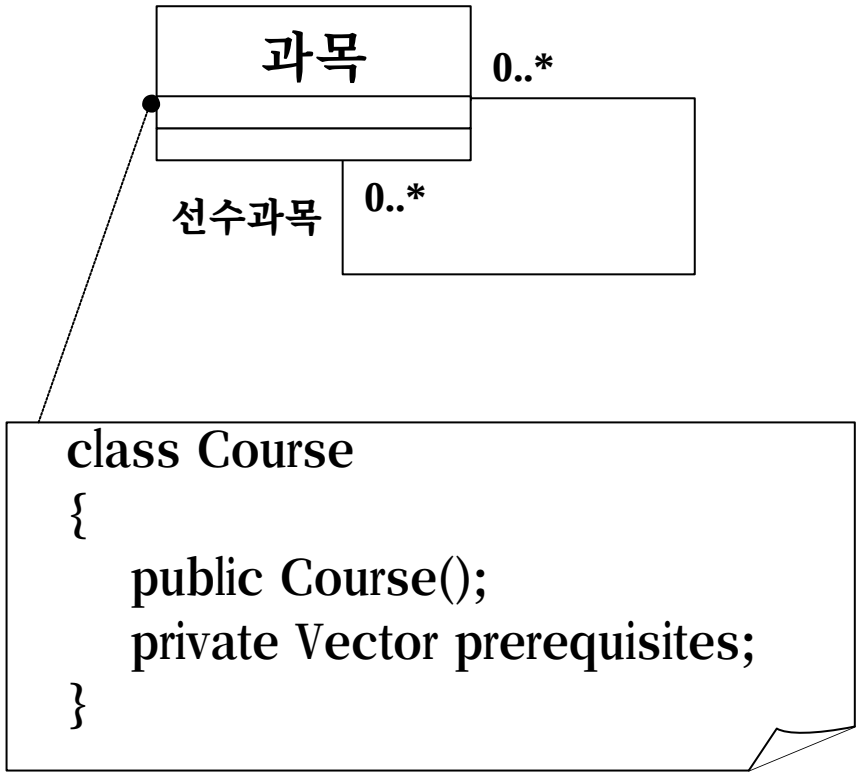


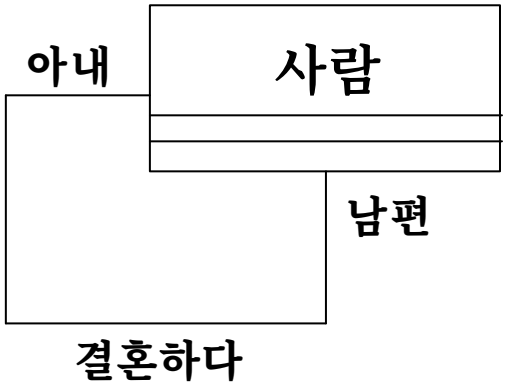
- Multiple Association은 바람직 하지 않으며, 클래스를 분할하는 방식으로 해결하는 것이 좋음

연관관계 (계속)

재귀 연관관계(Reflexive Association)

- 같은 클래스의 객체끼리 관계가 존재한다.



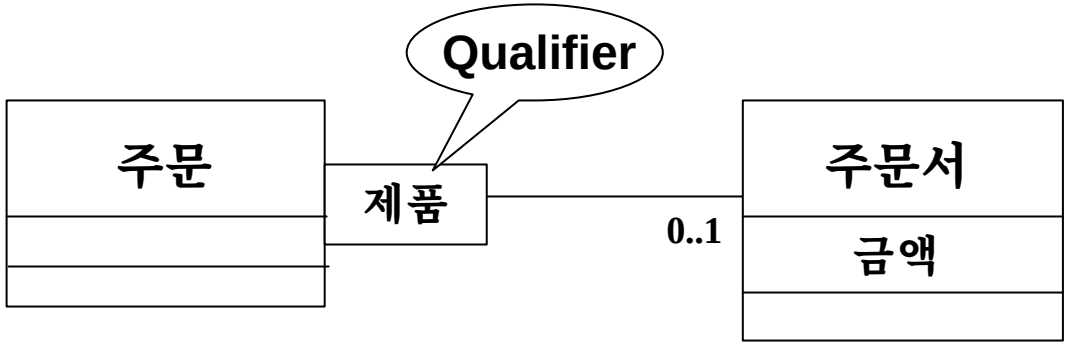


연관관계 (계속)

Qualified 연관관계

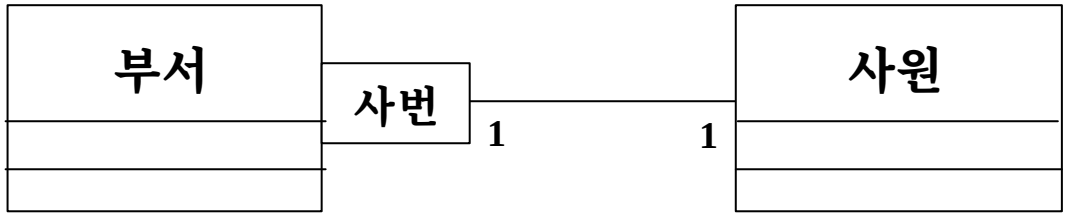
- One-to-many, many-to-many Association에서 사용
- Qualifier
  - Attribute또는 Attribute의 집합이며, 이들 값으로 Association으로 관련된 다른 객체 집합을 분리
  - Association에서 객체의 집합을 구별시키는 역할
  - Association에서 객체를 분리하기 위한 Key의 일종으로 간주되기도 함

표기법

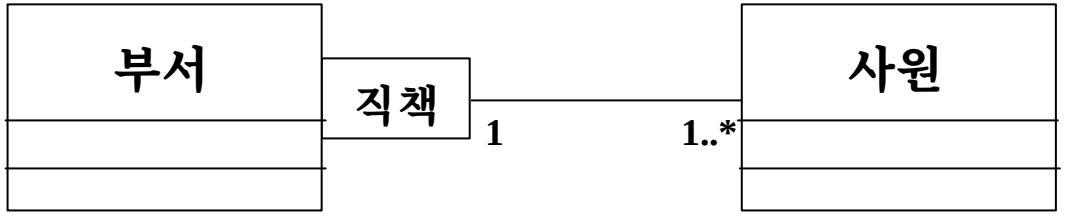


연관관계 (계속)

Qualified 연관관계의 예



- 한 부서에 특정 사번과 관련된 하나의 사원 객체가 존재한다.

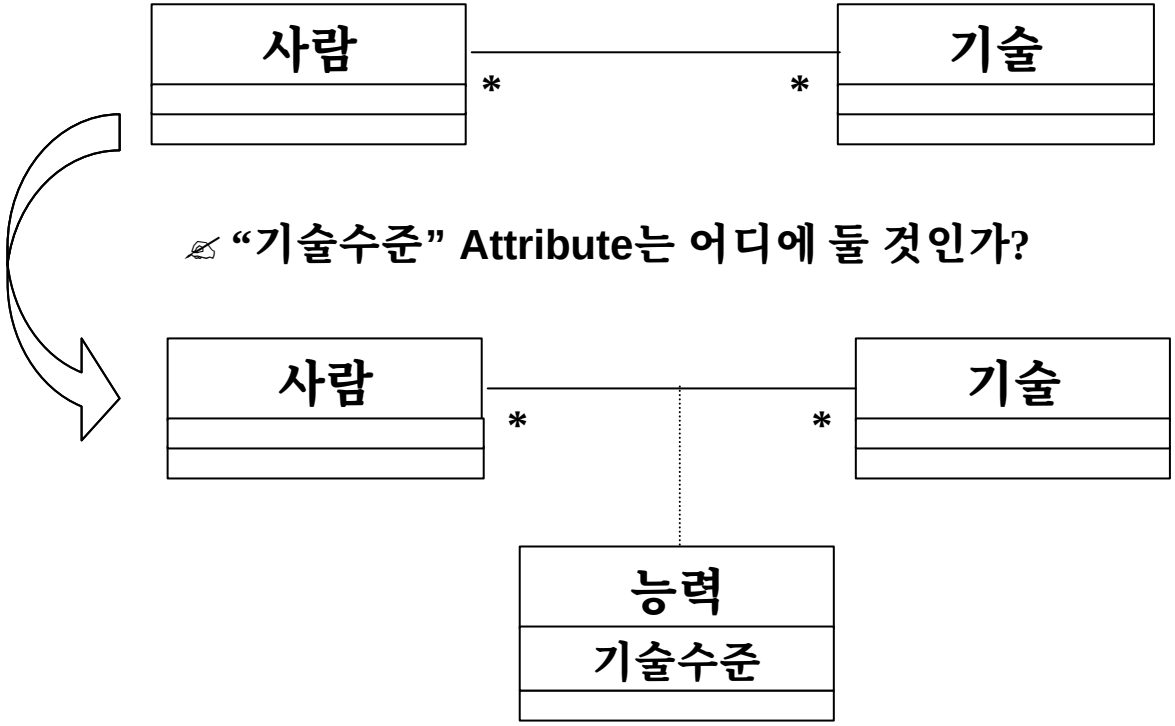


- 한 부서에서 특정 직책과 관련된 여러 개의 사원 객체가 존재한다.

연관 관계 (계속)

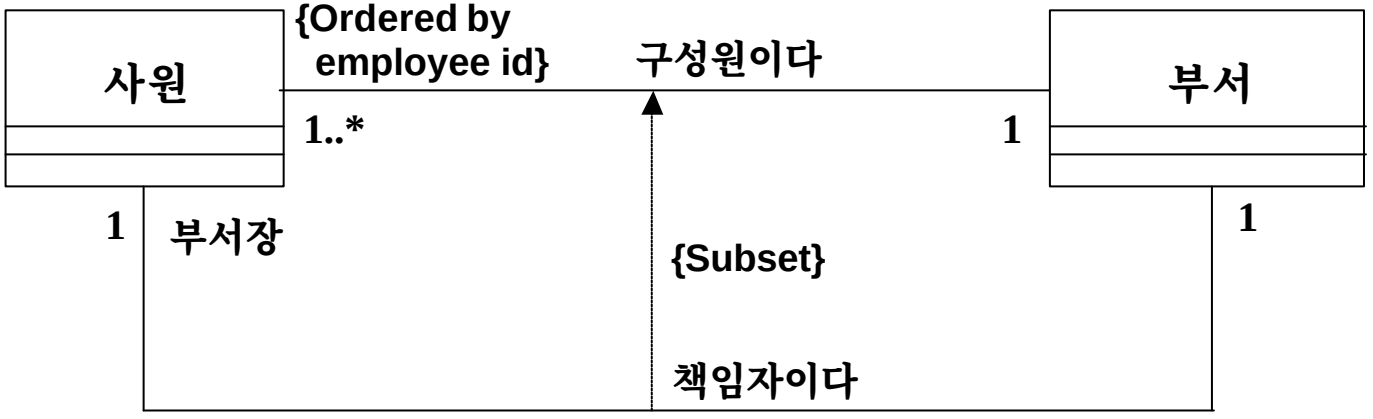
연관 클래스(Association Class)

- Many-to-many relationship에서 도출
- 연관관계에 대해 Attribute, Operation 추가할 경우
- 연관관계 당 하나의 연관 클래스만 도출 가능



연관관계 (계속)

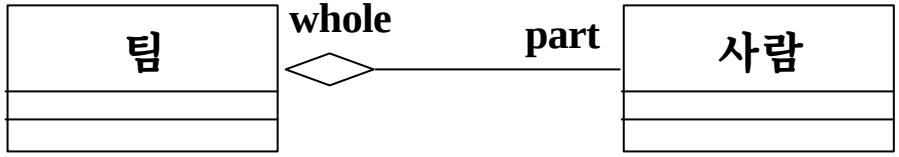
- 제약(Constraints)
  - 정의
    - 반드시 유지되어야 하는 하는 조건(condition)에 대한 표현
  - 표기법
    - 대괄호({ })안에 적절한 단어 사용



포함관계

- 연관관계의 특수한 경우
  - "Part-of" (또는 "Containment") Relationship
  - 종류
    - By Reference
    - By Value

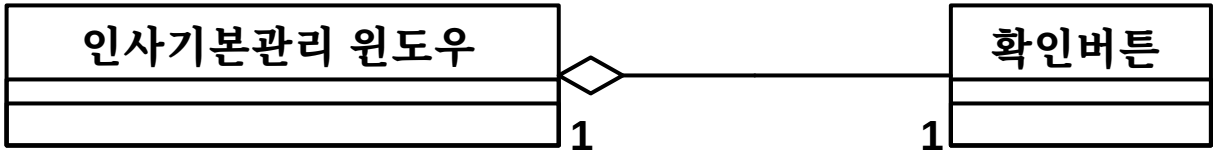
- 표기법



- **Multiplicity**
  - **Association과 동일하게 적용됨**

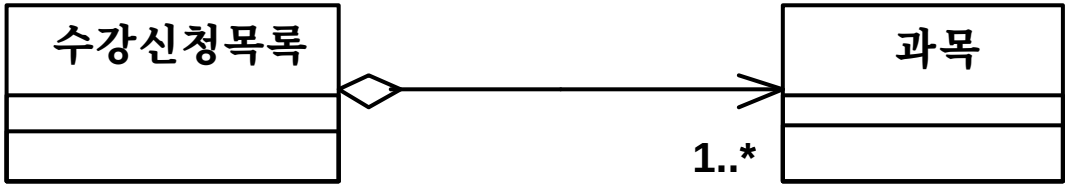
포함관계 (계속)

- **By Value Relationship**
  - Dependent lifetime
    - Whole - 객체를 생성하면, Part - 객체도 생성됨
    - Whole - 객체를 소멸되면, Part - 객체도 소멸됨
  - 예
    - "인사기본관리 윈도우"가 생성되면, "확인버튼"도 생성되며,
    - "인사기본관리 윈도우"가 소멸되면, "확인버튼"도 소멸됨



포함관계 (계속)

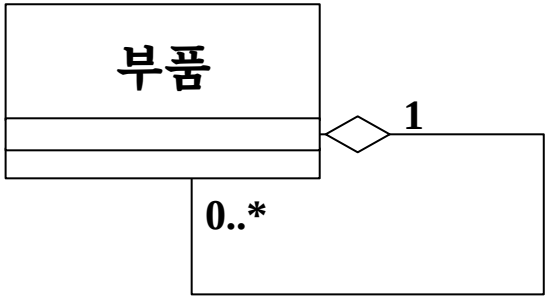
- **By-Reference Relationship**
  - Independent lifetime
  - Unfilled diamond 로 표현



포함관계 (계속)

재귀 포함관계(Reflexive Aggregation)

- Recursive Relationship



상속관계

- 정의
  - 클래스간의 관계
  - “Is a” 또는 “Kind of” 관계

    <SubClass> “Is a[Kind of]” <SuperClass>
  - 한 클래스의 Attribute/Operation을 다른 클래스들이 공유하는 관계
  - 하위 클래스는 하나 이상의 상위 클래스로 부터 상속 받는 추상화 계층 (Hierarchy of abstractions)을 정의
- 종류
  - 단일 상속
    - 하위 클래스는 오직 하나의 하위 클래스로 부터 상속 받음
  - 다중 상속
    - 하위 클래스는 하나 이상의 상위 클래스들로 부터 상속 받음

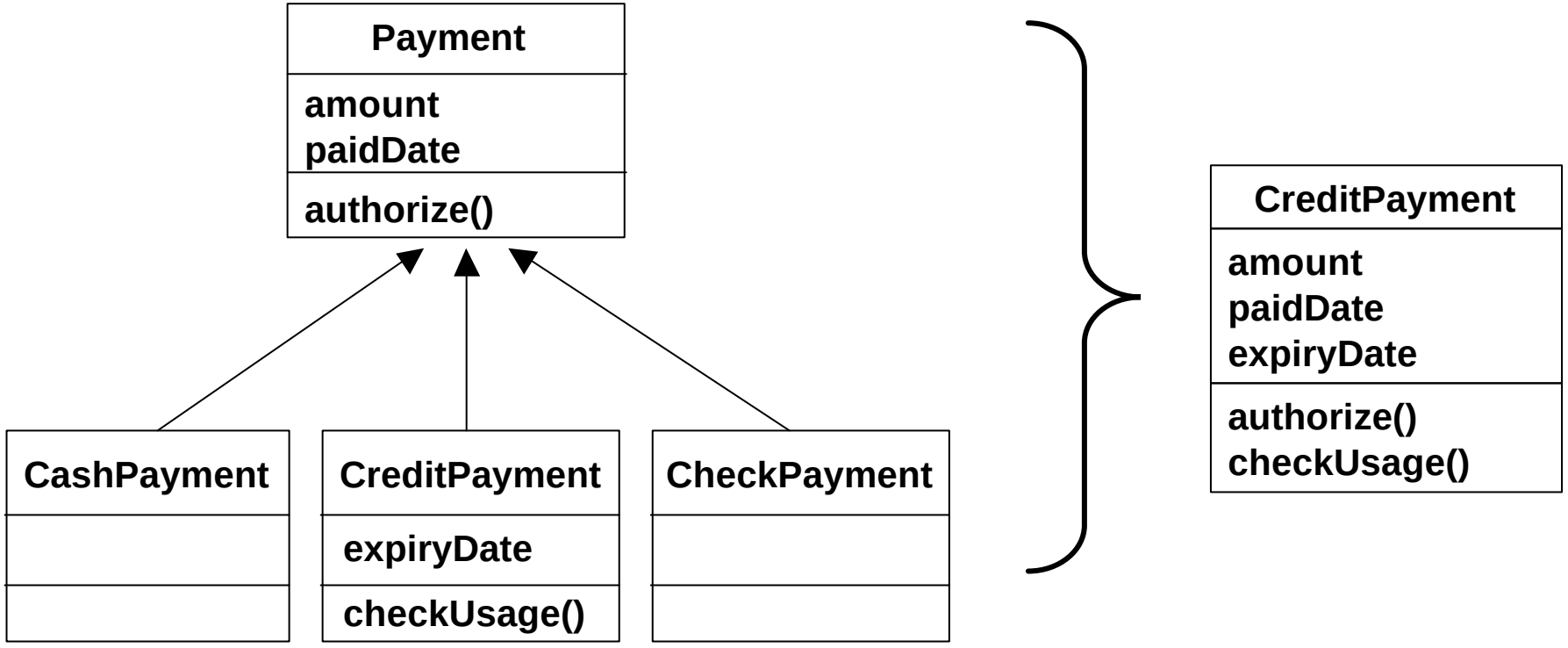
상속관계 (계속)

- 상속 관계(Inheritance relationship)는 개별적인 객체들과는 상관 없음
  - Relationship Name이 없음
  - Multiplicity는 의미가 없음

- 하위 클래스는 상위 클래스로 부터
  - Attributes, Operations, Relationship을 상속 받음
  - Operation을 상속 받았을 경우, 하위 클래스에서 재정의 가능 (Overriding)
  - 상속 받은 것 들 이외의 부가적인 Attributes, Operations, Relationship들도 추가할 수 있음

상속관계 (계속)

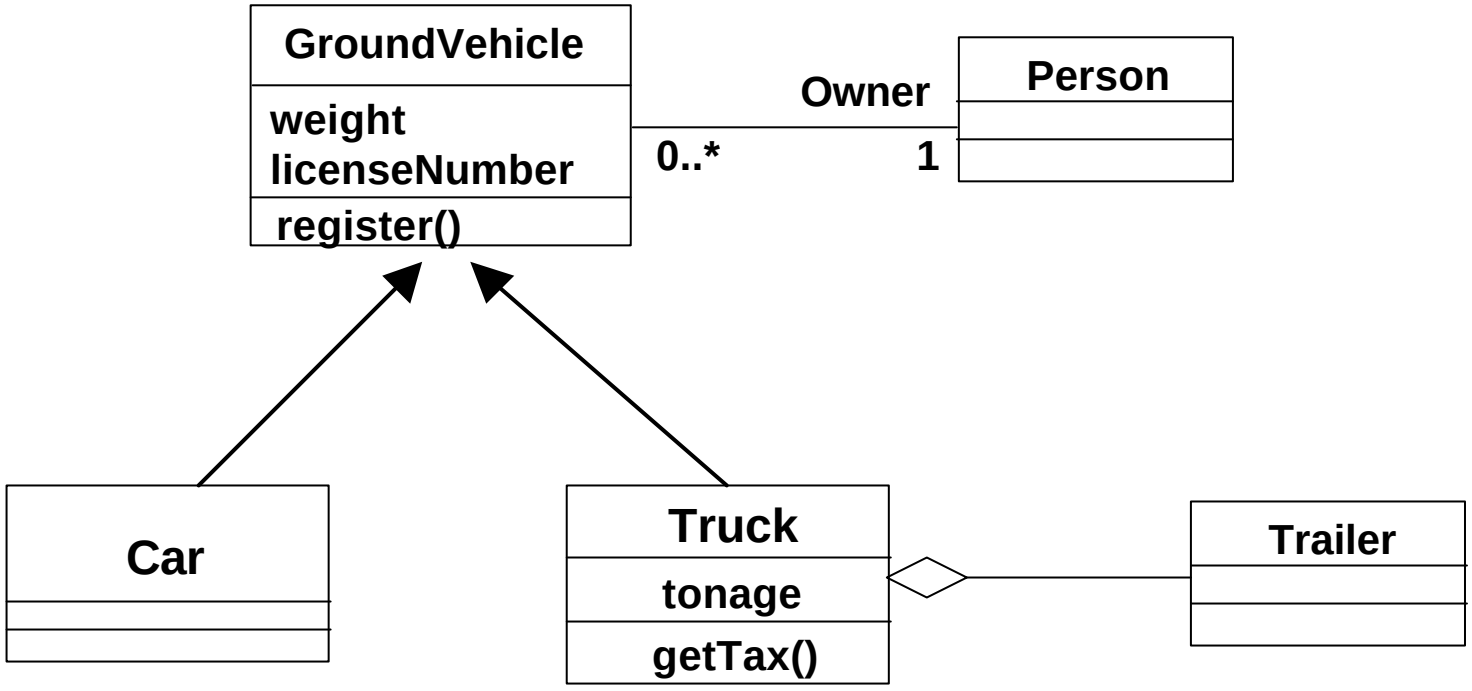
✍ 애트리뷰트와 오퍼레이션 상속



상속관계 (계속)

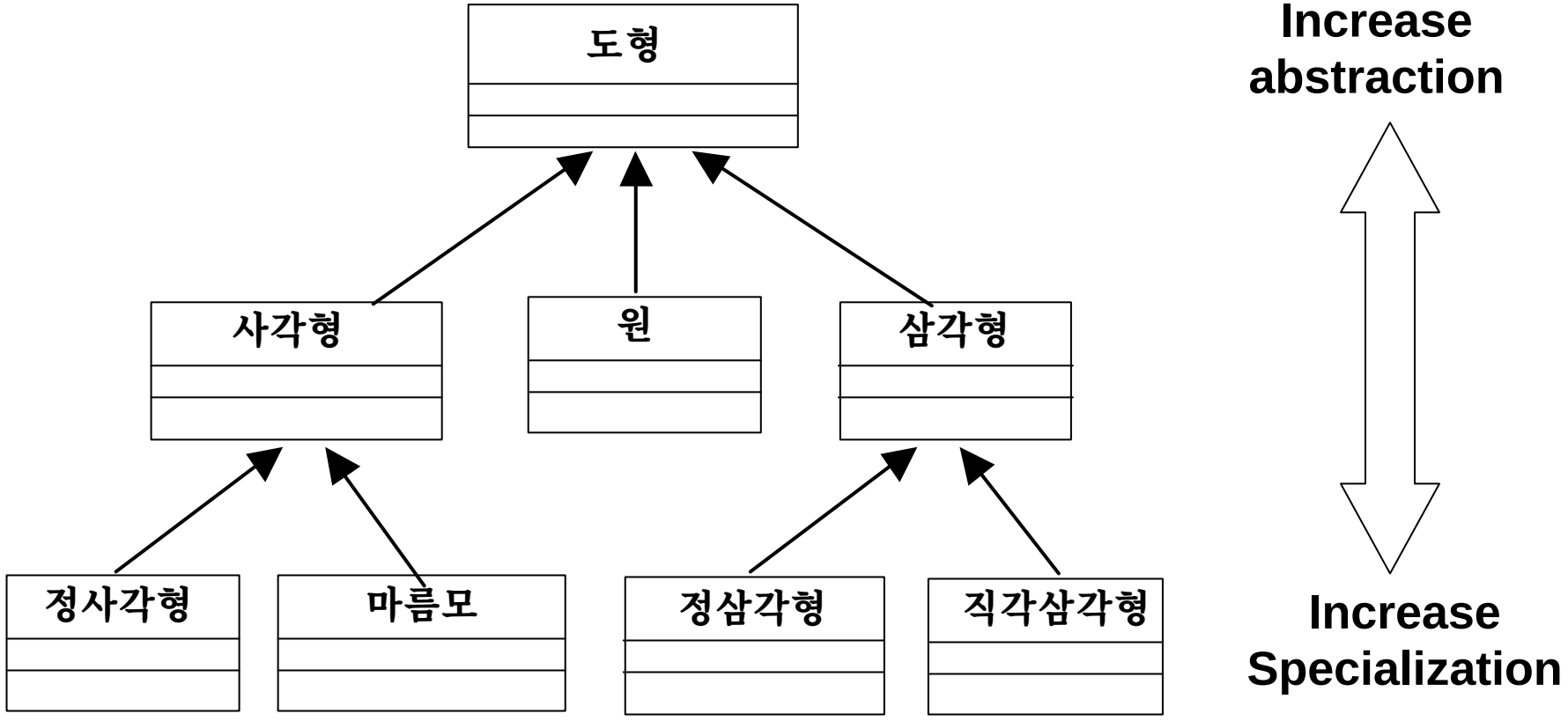
관계성 상속

- "Car" vs. "Owner" : 연관관계
- "Truck" vs. "Owner" : 연관관계
- "Truck" vs. "Trailer" : 포함관계



상속관계 (계속)

✍ 클래스의 일반화/구체화 (Generalization/Specialization)

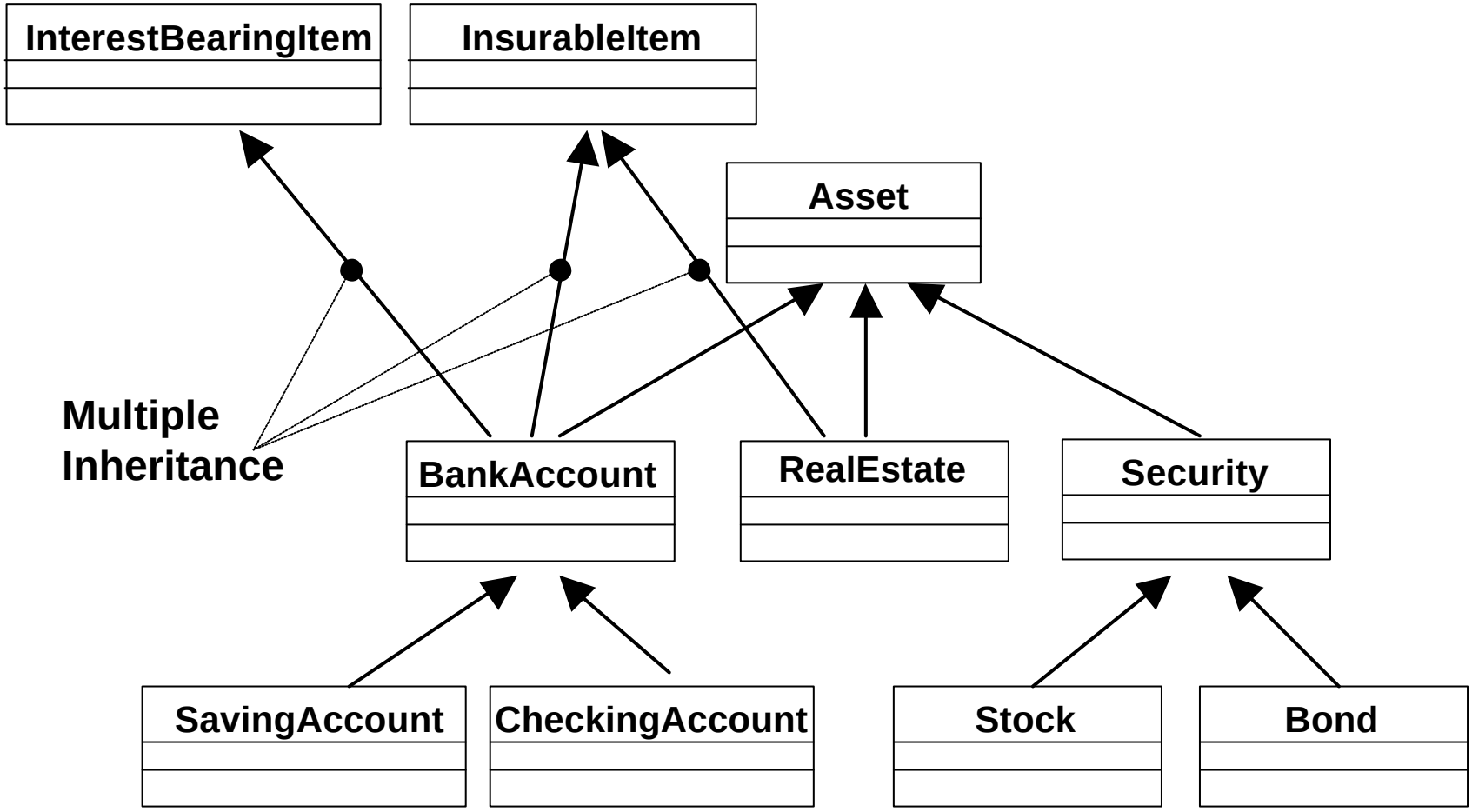


상속관계 (계속)

- 다중 상속 (Multiple Inheritance)
  - 반드시 필요할 경우에만 사용
    - 부작용(side effect) 고려
  - 프로그래밍 언어의 지원 여부 반드시 고려
  - 다중 상속 (Multiple Inheritance)의 문제점
    - **Naming Conflict**
    - **Repeated Inheritance**

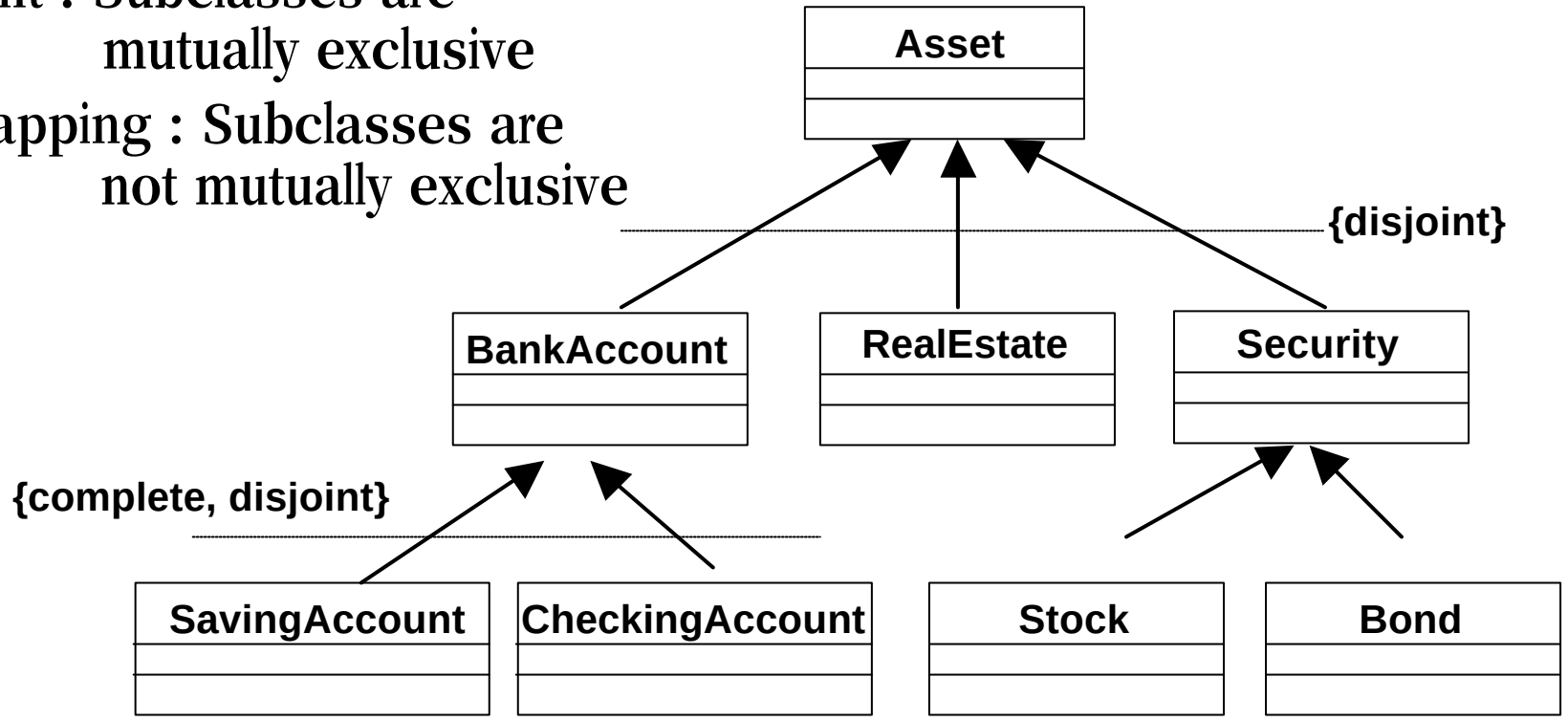
상속관계 (계속)

다중상속


InterestBearingItem
InsurableItem
Asset
Multiple
Inheritance
BankAccount
RealEstate
Security
SavingAccount
CheckingAccount
Stock
Bond

상속관계 (계속)

- **Inheritance Constraint**
  - Complete : no additional children are permitted
  - Incomplete : additional children are permitted
  - Disjoint : Subclasses are mutually exclusive
  - Overlapping : Subclasses are not mutually exclusive



상속관계 vs. 포함관계

- 상속관계
  - "Is a"["Kind of"] Relationship
- 포함관계
  - "Has a" Relationship

Inheritance	Aggregation
"is a"	"has a"

종속관계

- 종속관계
  - 종속관계는 어떤 서비스에 대해 하나의 클래스가 다른 클래스에 종속됨을 의미
  - Dashed arrow로 표현

  - 의미
    - 클라이언트 클래스의 오퍼레이션이 공급자 클라이언트의 객체를 생성
    - 클라이언트 클래스의 오퍼레이션이 공급자 클래스의 instance인 전달인자를 Return하게 될 것이라는 것임을 의미

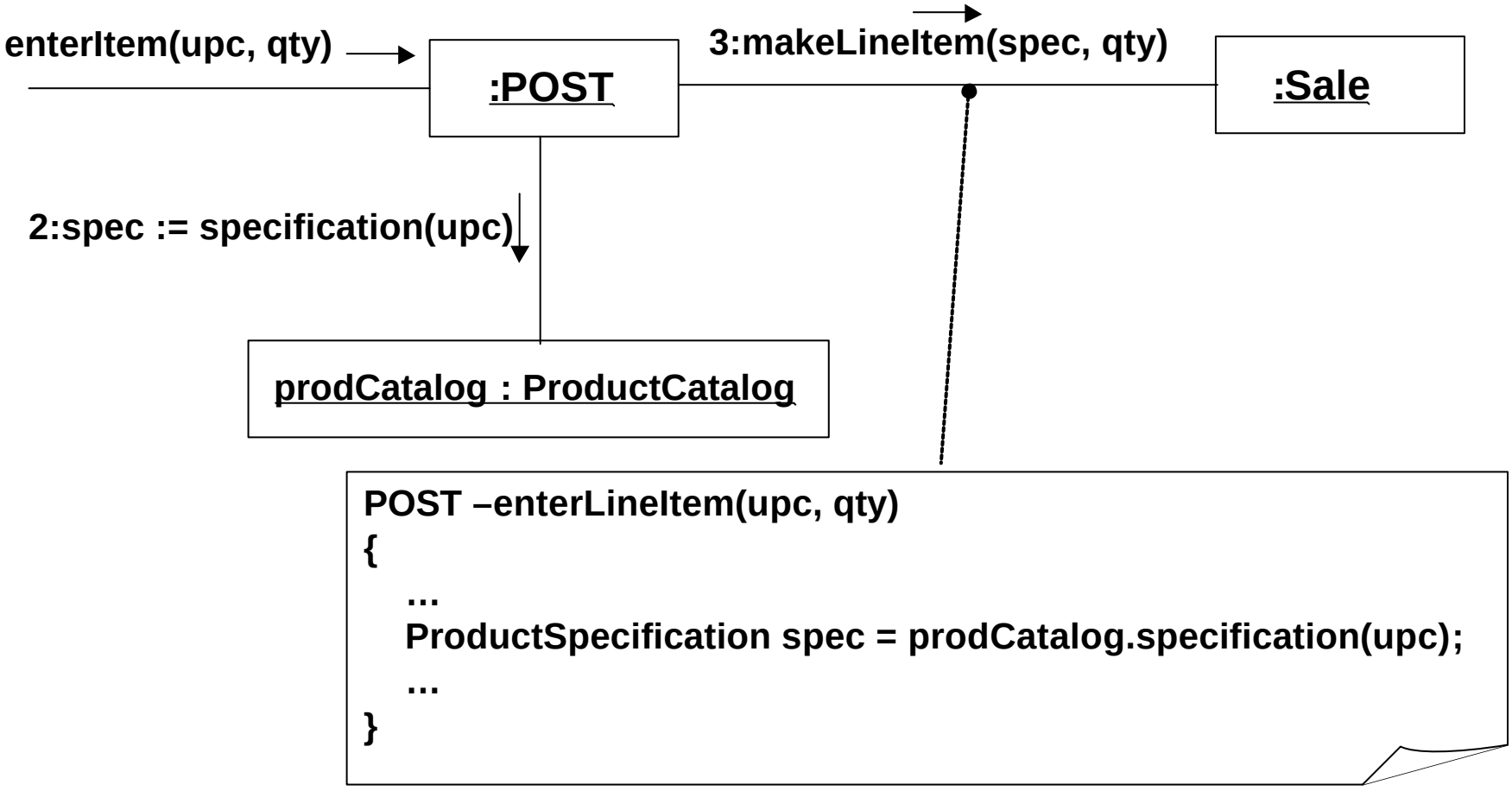
관계성 재정의

- **Object Visibility Options**
  - Global visibility : 종속관계
    - supplier 객체는 전역 객체이다.
  - Parameter visibility : 종속관계
    - supplier 객체는 client 객체 오퍼레이션의 파라메터이다.
  - Local visibility : 종속관계
    - supplier 객체가 client 객체 안에서 선언되어 사용된다.
  - Attribute visibility : 연관관계
    - supplier 객체가 client 객체의 Field 로 선언된다.

- **Relationship Design**시 클래스 추가 도출 가능

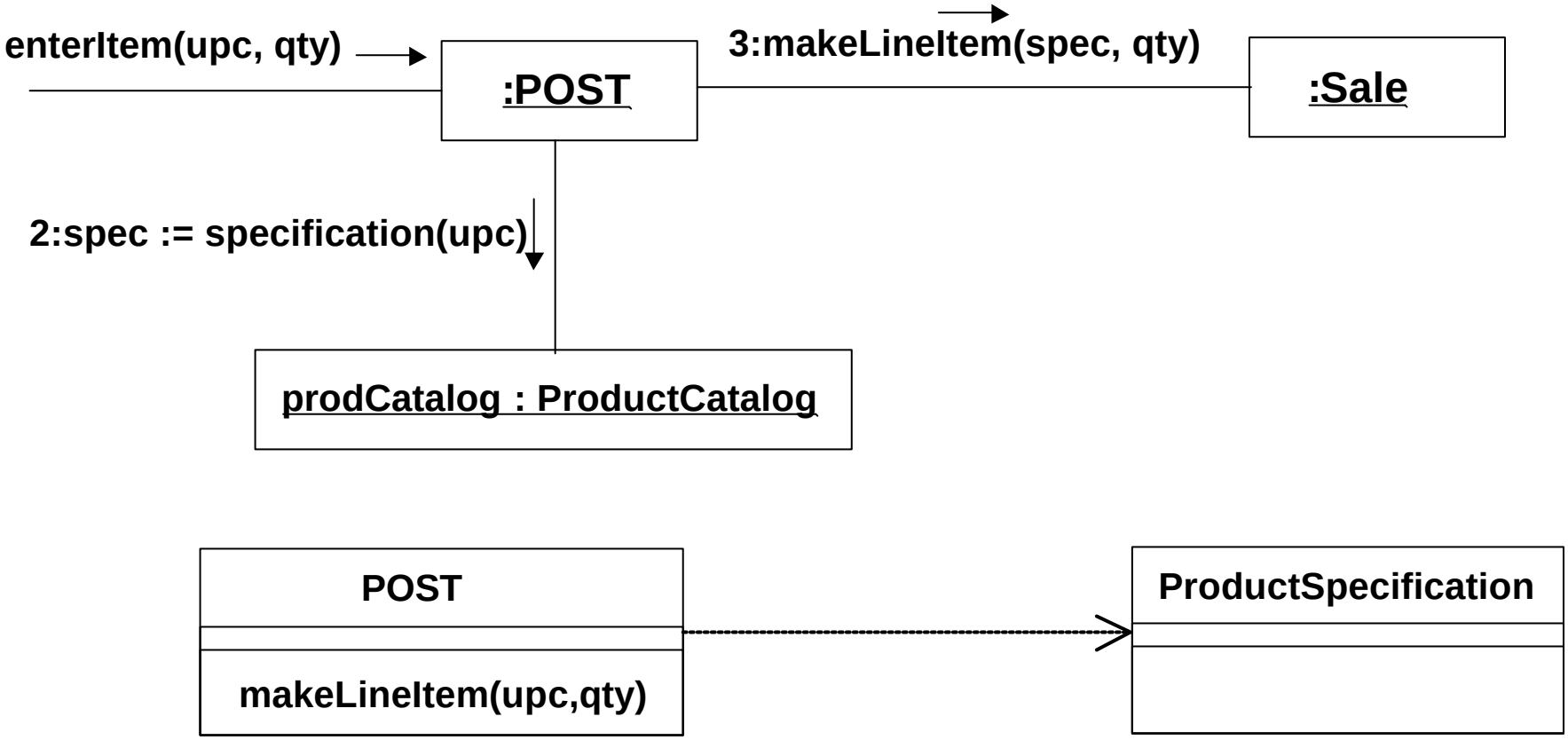
관계성 재정의 (계속)

- Local **visibility**



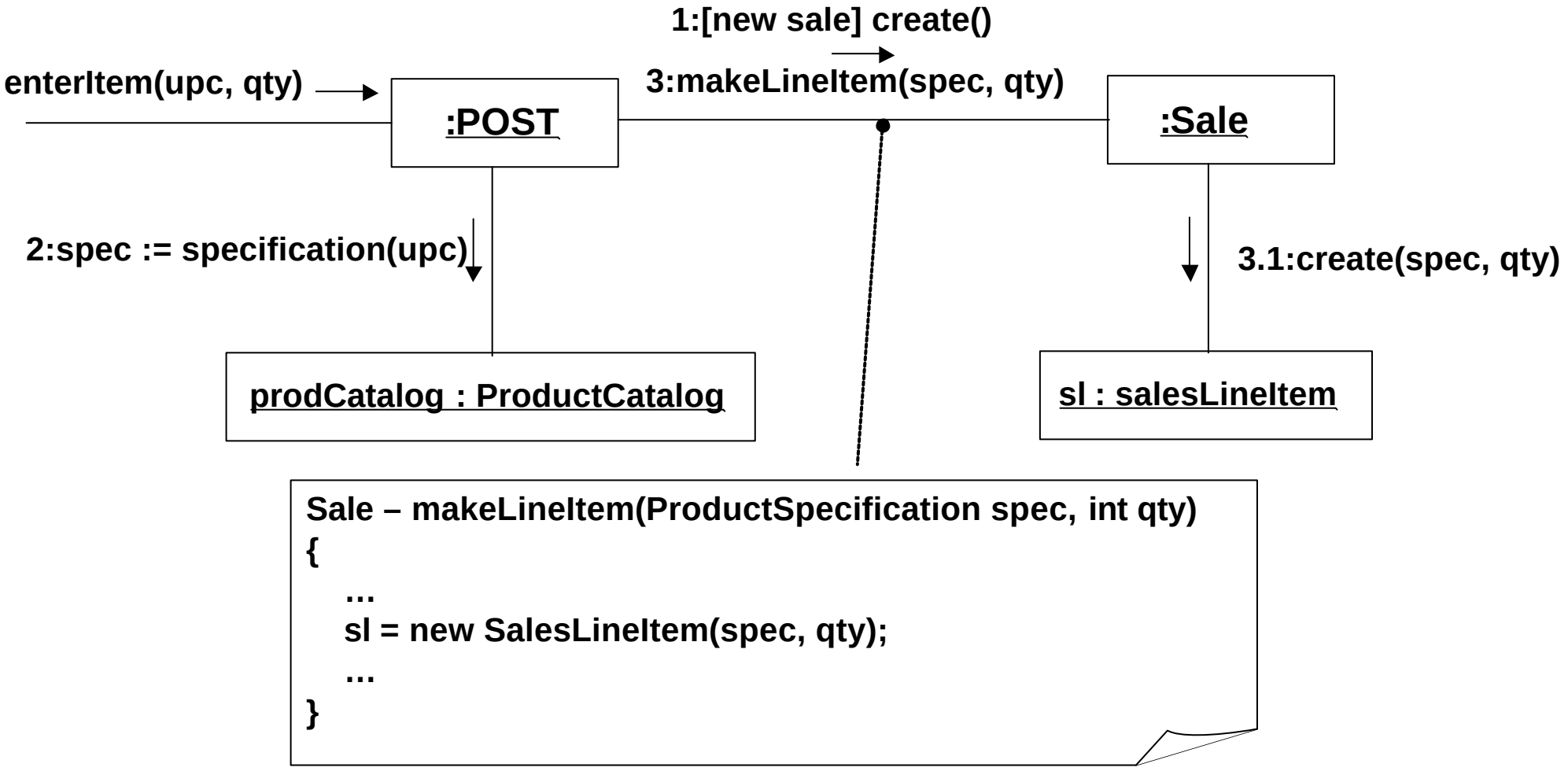
관계성 재정의 (계속)

- Local **visibility**



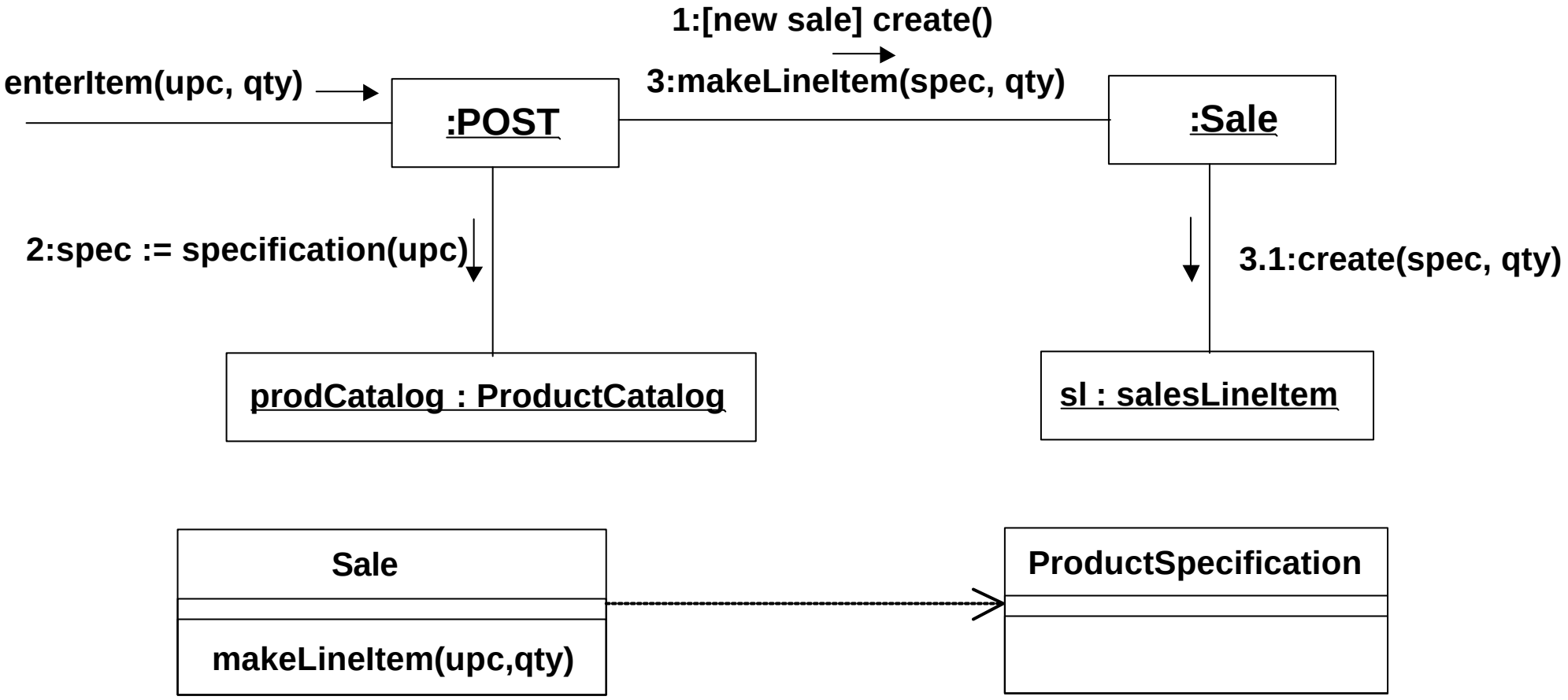
관계성 재정의 (계속)

- Parameter **visibility**



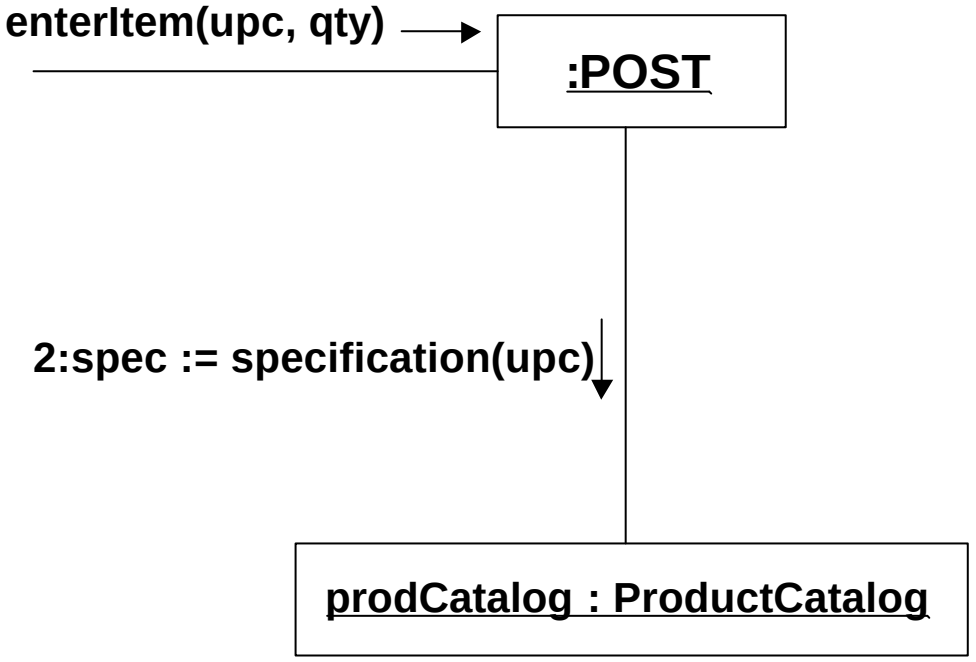
관계성 재정의 (계속)

- Parameter **visibility**



관계성 재정의 (계속)

- Attribute **visibility**

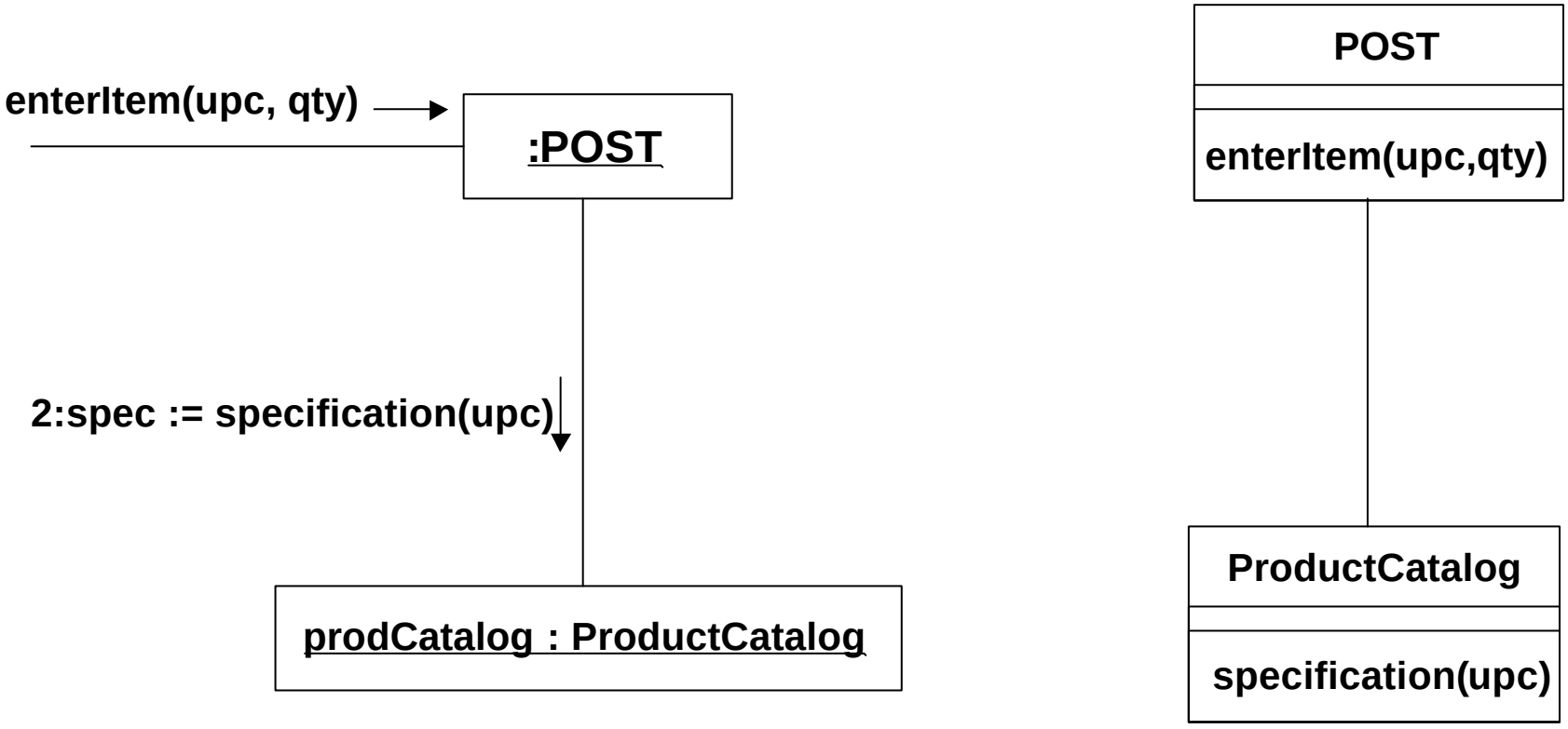



```
class POST
{
    …
    private ProductCatalog prodCatalog;
    …
}
```

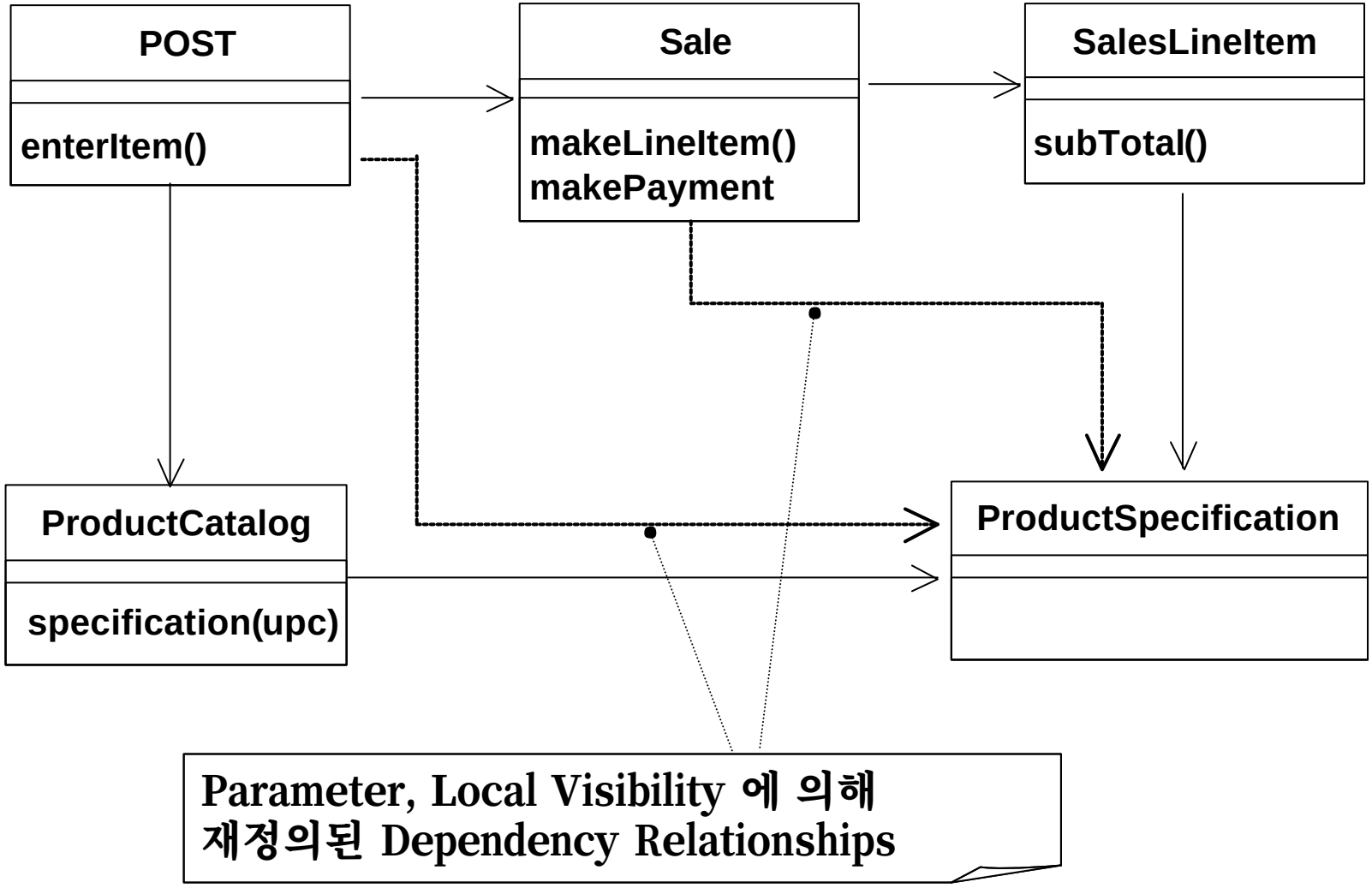
```
POST – enterItem(upc, qty)
{
    …
    spec = prodCatalog.specification(upc)
    …
}
```

관계성 재정의 (계속)

- Attribute **visibility**

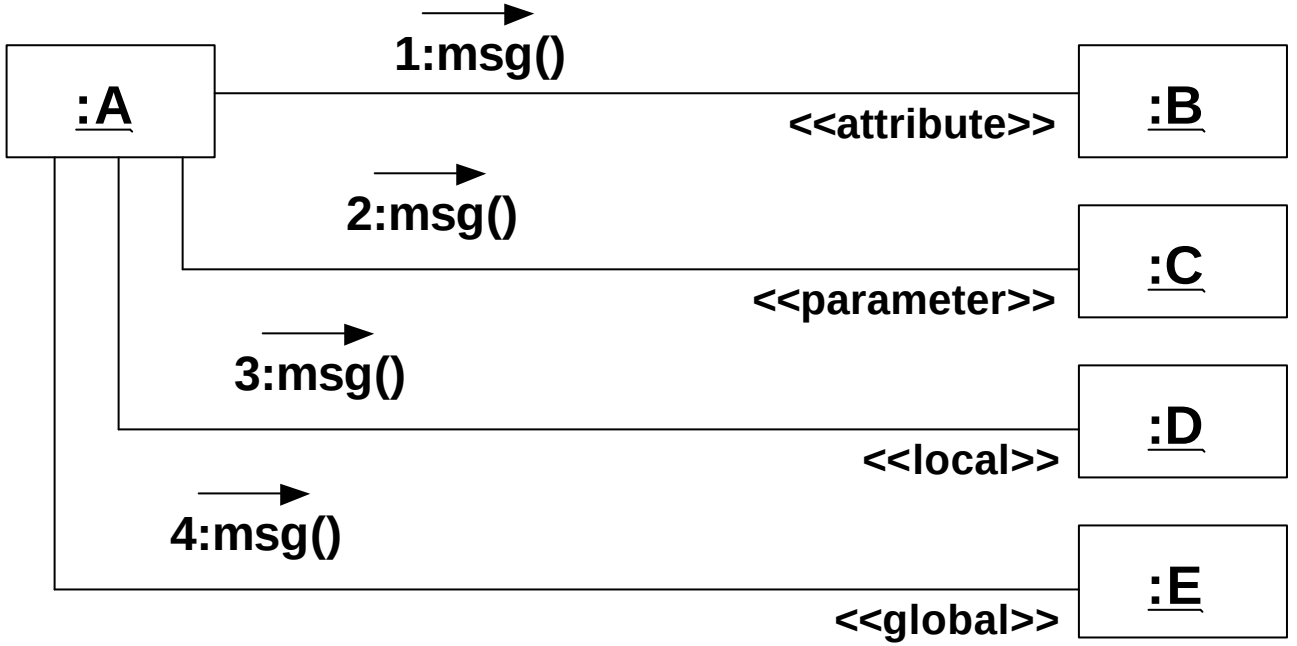


관계성 재정의 (계속)


POST
enterItem()
Sale
makeLineItem()
makePayment
SalesLineItem
subTotal()
ProductCatalog
specification(upc)
ProductSpecification
Parameter, Local Visibility 에 의해
재정의된 Dependency Relationships

관계성 재정의 (계속)

- Stereotypes을 이용하여 Visibility 표현

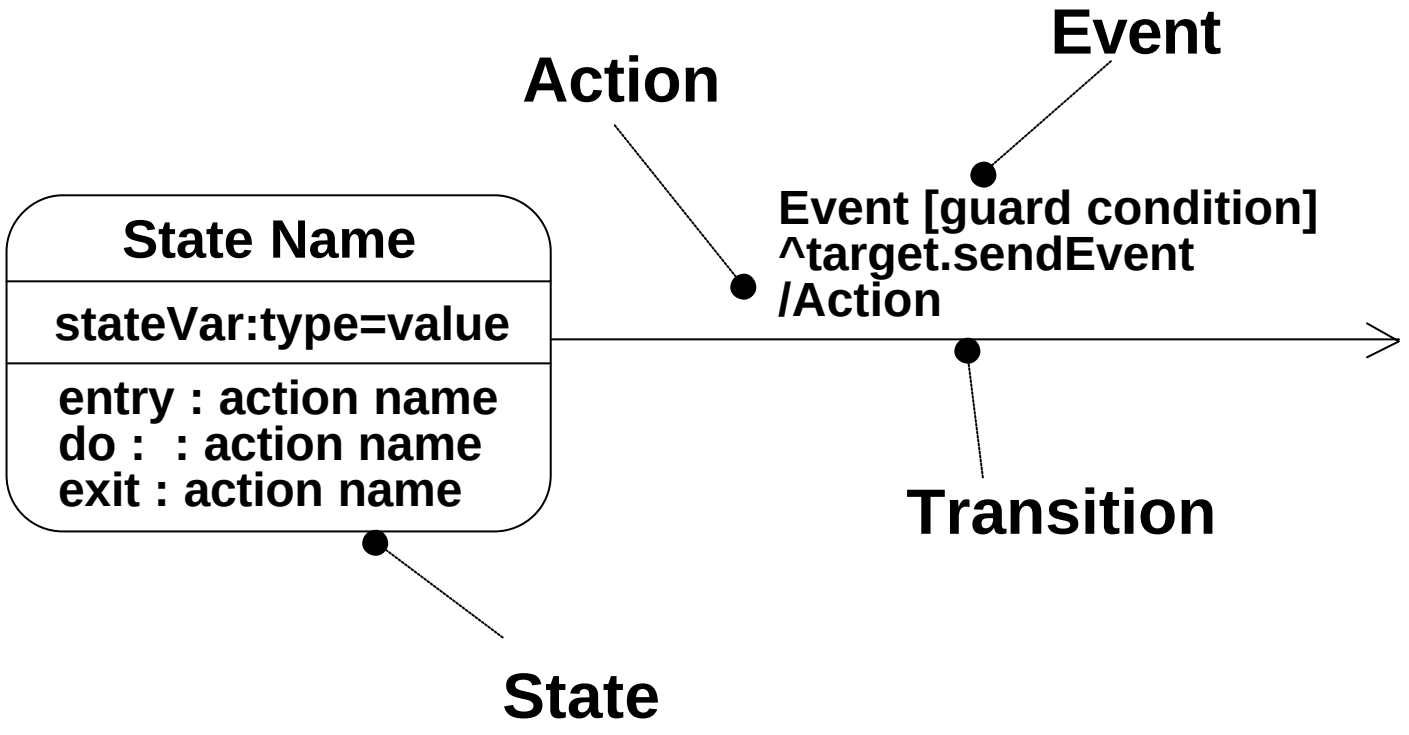


Statechart Diagram

- **Behavior Modeling**
  - Use Case
    - 시스템의 행위(System's Behavior)
  - Interaction Diagram
    - 유즈 케이스내의 객체의 행위 표현
  - Statechart Diagram
    - 객체의 행위(Object Behavior) 표현
    - 특정 클래스의 전체 유즈 케이스에 걸친 상태 표현
    - 클래스로 부터 생성된 객체의 Life history를 표현
      - 상태(State)
      - 이벤트(Events)
        : 상태 전이(Transition)의 원인
      - 액션(Action)
        : 상태 전이에 대한 결과

Statechart Diagram

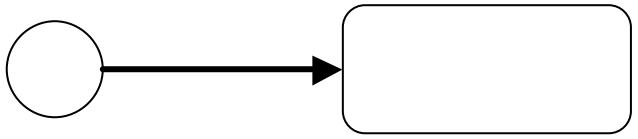
Statechart Diagram 표기법



Statechart Diagram (계속)

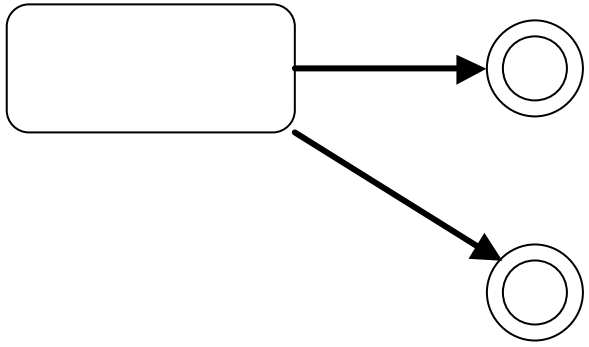
Special States

- Initial State
  - 필수(Mandatory)
  - Only one



Initial state

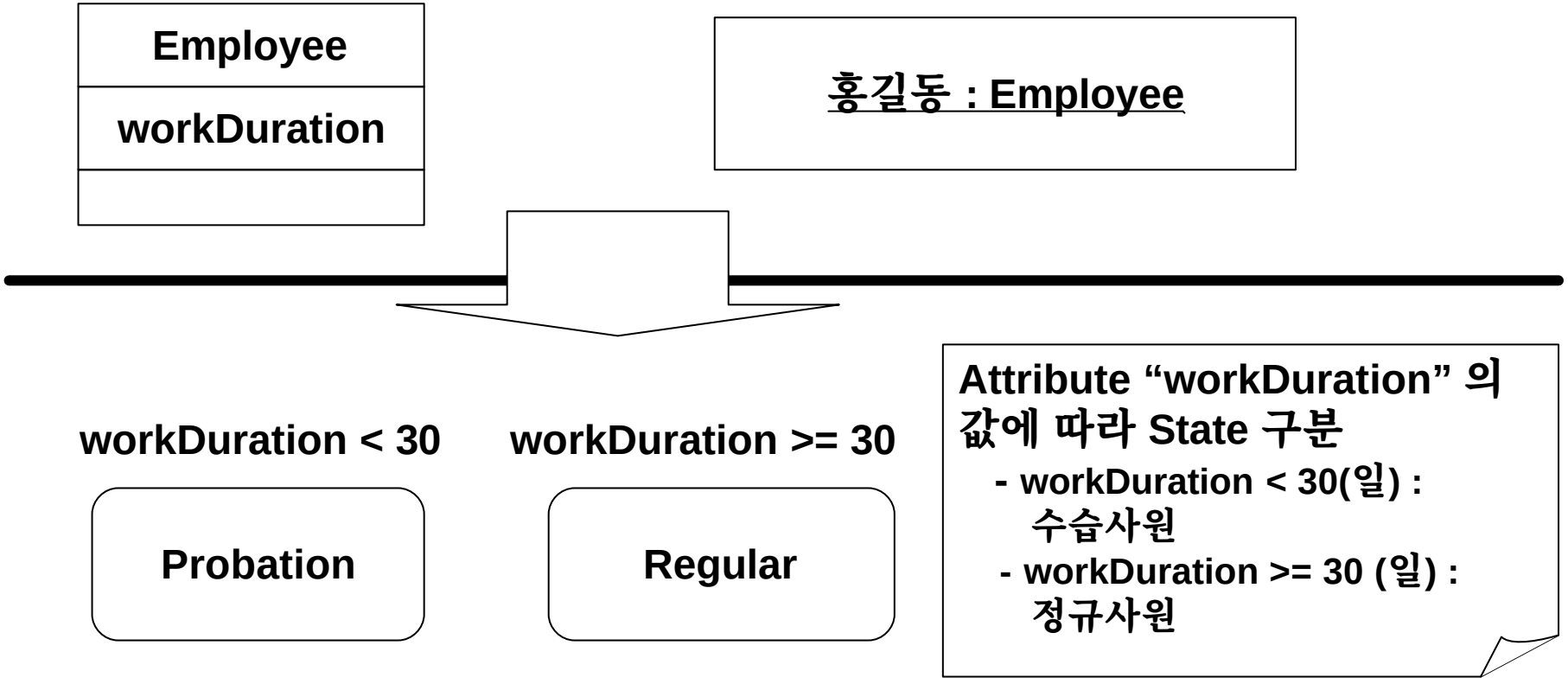
- Final State
  - 선택적(Optional)
  - More than one



Final state

Statechart Diagram (계속)

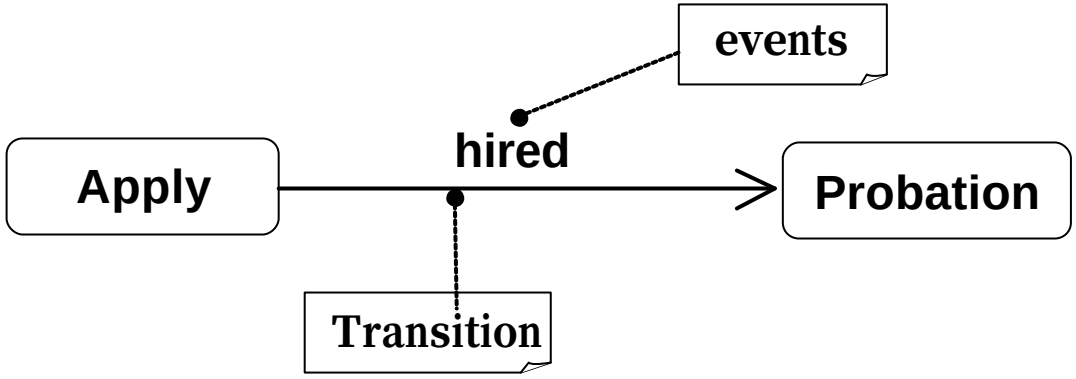
- 상태와 Attribute
  - 상태는 특정 Attribute의 값에 따라 구분 가능



Statechart Diagram (계속)

- 이벤트(Events)
  - 특정한 시점에서 발생
  - 상태 전이(Transition)의 원인
  - 예 : 채용 (hired) , 휴직 신청 (request leave)

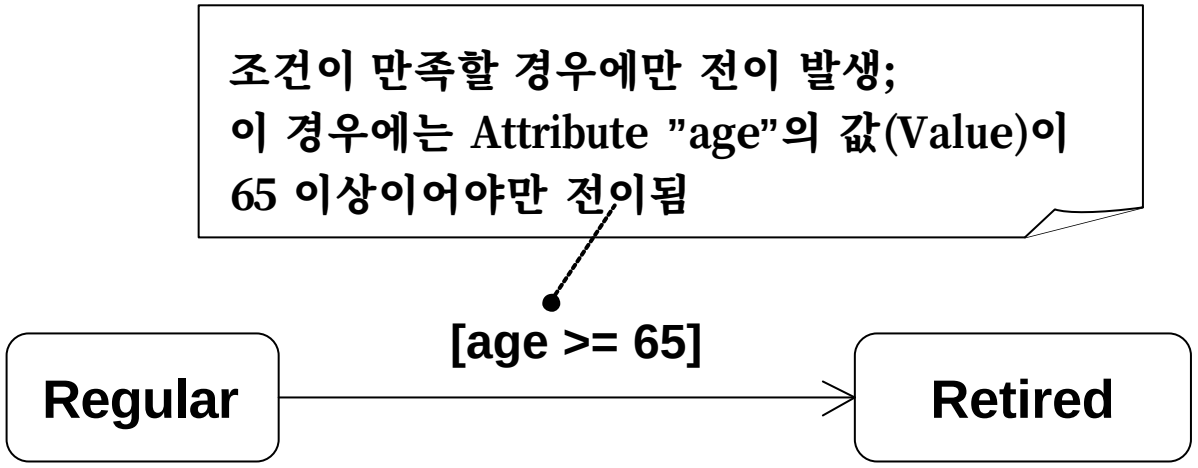
- 전이(Transitions)
  - 객체가 이벤트로 인해 원래 상태(originating State)에서 다음 상태 (successor State)로 변화되는 것



Statechart Diagram (계속)

Guard Conditions

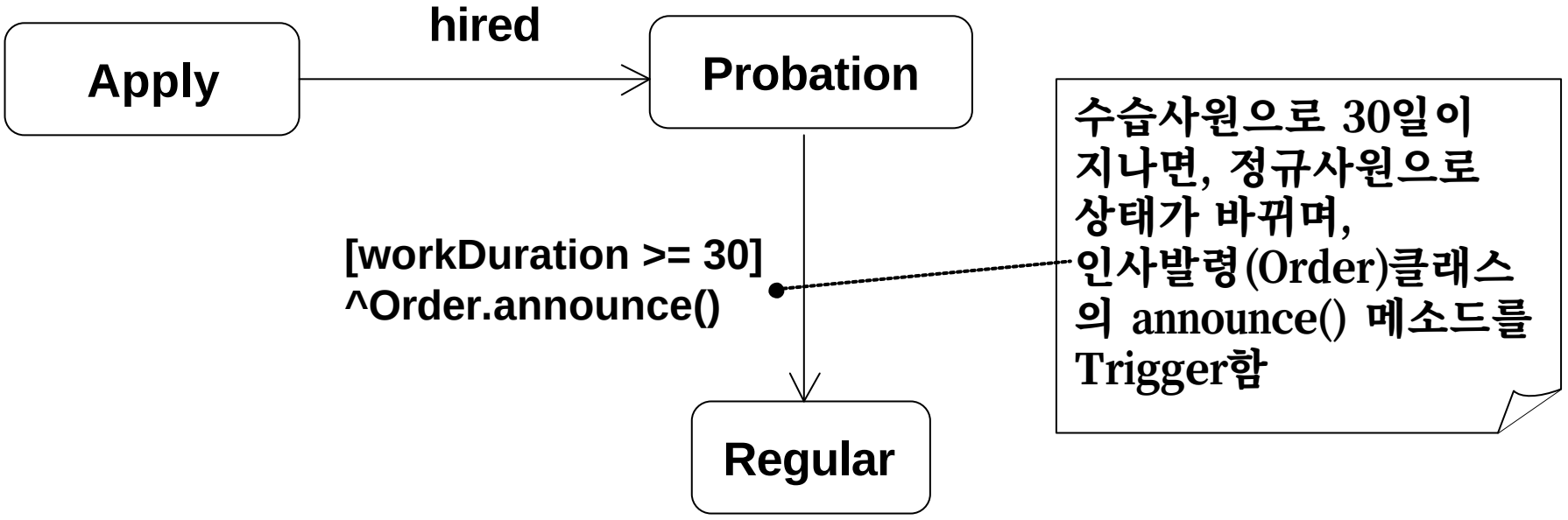
- 상태 전이 시에 특정조건을 만족할 경우에만 전이가 이루어지도록 하기 위해 사용되는 Attribute 값의 Boolean expression



Statechart Diagram (계속)

Sending Events

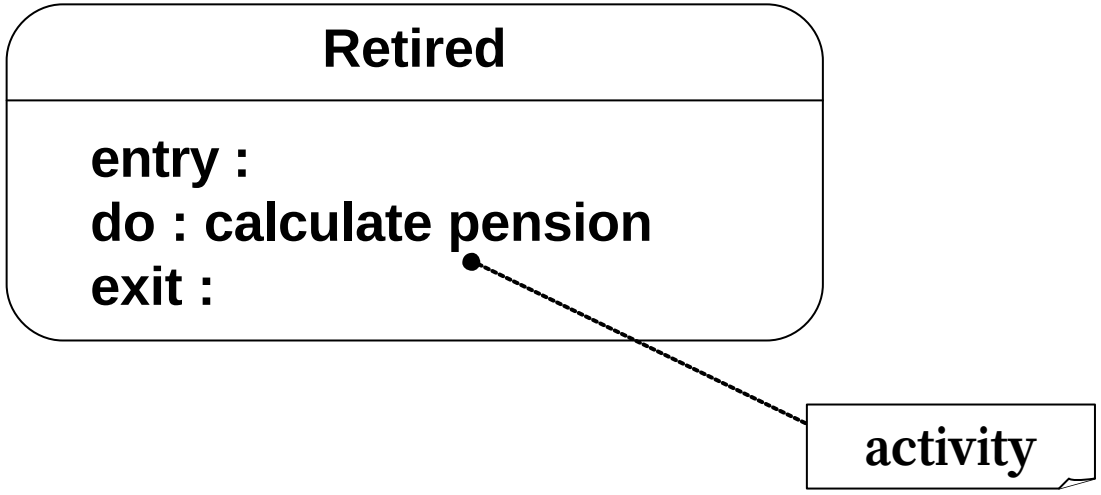
- Trigger Event
- Format
  - **^Class Name. Event Name**



Statechart Diagram (계속)

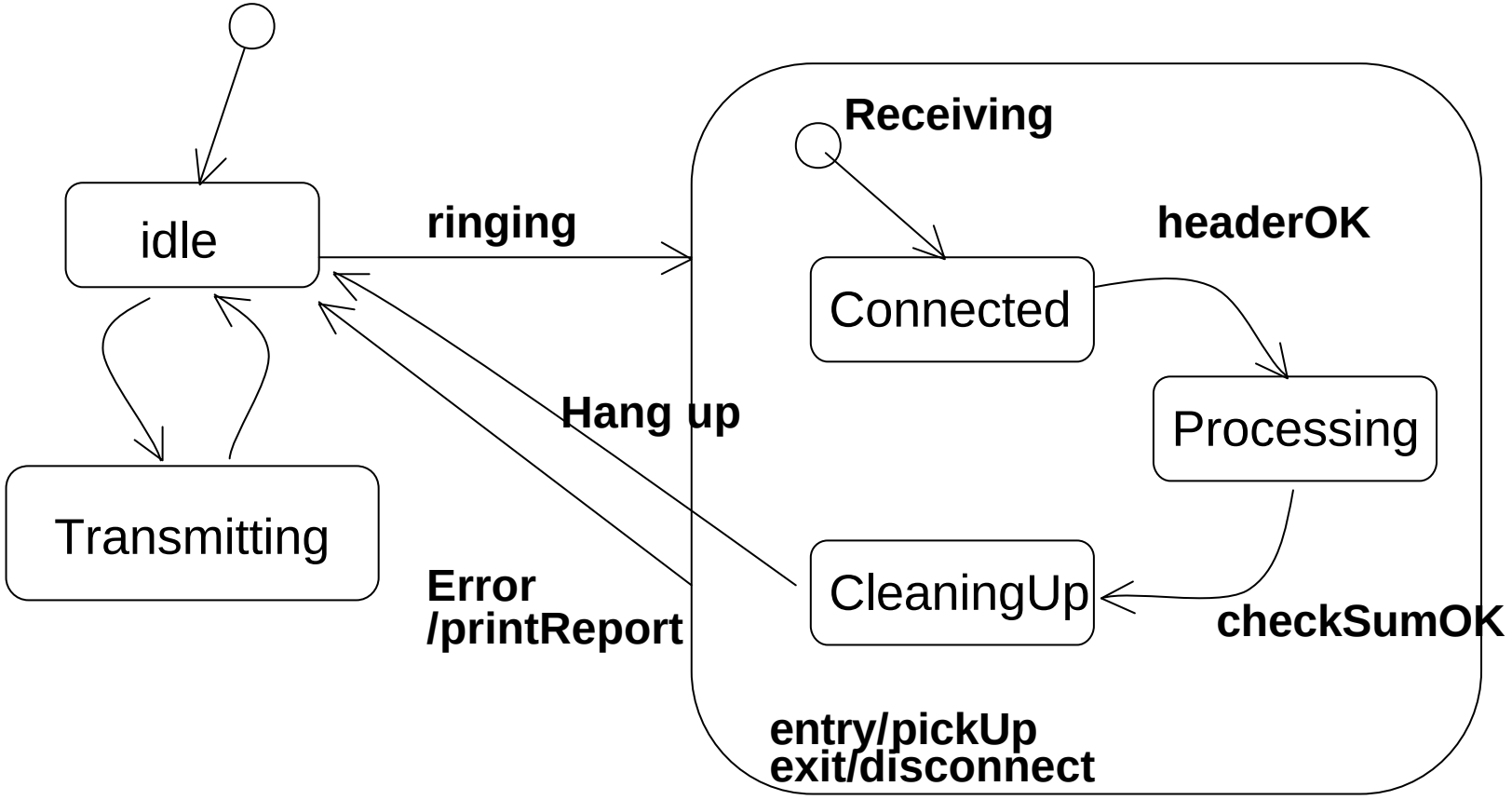
Activities

- 객체가 특정 상태에서 수행하는 Operation
  - 상태에 진입(Entered)하면서부터 시작하여 종료(Complete)할 때까지 계속 수행
  - 수행을 완료하거나 수행 도중 중단(outgoing transition) 될 수 있음



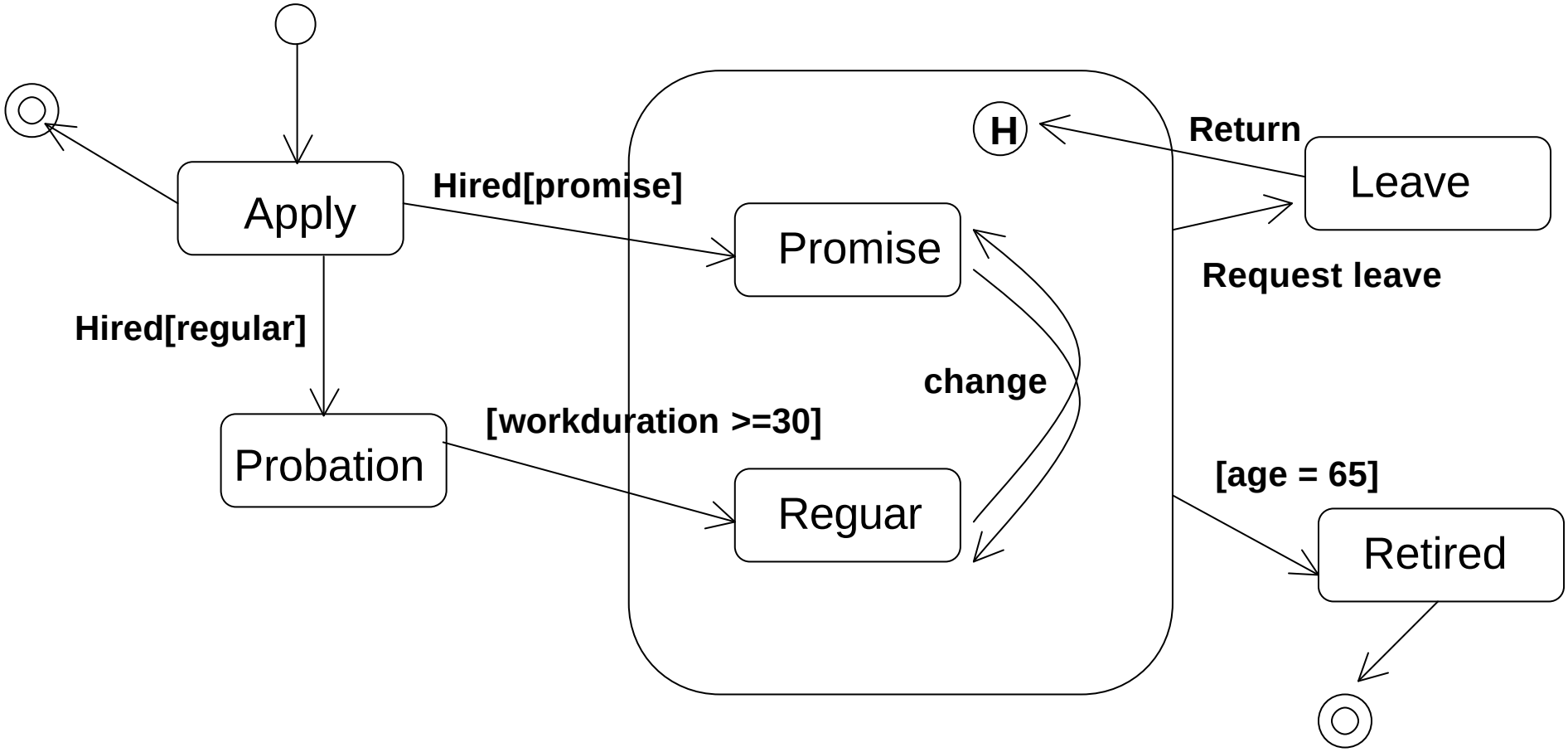
Statechart Diagram (계속)

✍ 예: StateChart with Nested States



Statechart Diagram (계속)

예: Statechart with History



II-5. Component 설계 및 배치

- Logical vs Physical Architecture
- Component and Interface
- Component Diagram
- Porcess Modeling
- Deployment Diagram

Logical vs Physical Architecture

Logical Architecture

- 시스템이 제공해야 할 기능은 무엇인가?
- 어떤 클래스들이 존재하며, 이 클래스들은 서로 어떻게 관계를 맺고 있는가?
- 클래스와 객체들이 기능을 수행하기 위해 어떻게 연동되어 있는가?

Diagram

- Use Case Diagram
- Class Diagram
- State Diagram
- Activity Diagram
- Collaboration Diagram
- Sequence Diagram

Logical vs Physical Architecture

Physical Architecture

- 클래스 또는 객체들이 어떤 프로그램이나 프로세스에 물리적으로 위치하는가?
- 어떤 컴퓨터에서 프로그램이나 프로세스가 실행되는가?
- 시스템을 어떤 컴퓨터와 하드웨어 장치들이 구성하고 있으며, 그들은 어떻게 연결되어 있는가?
- 여러 컴포넌트들이 서로 종속적인가? 만일, 특정 컴포넌트가 변경되면, 영향을 받는 어플리케이션은 어떤 것이 있는가?

Diagram

- Component Diagram
- Deployment Diagram

Component and Interface

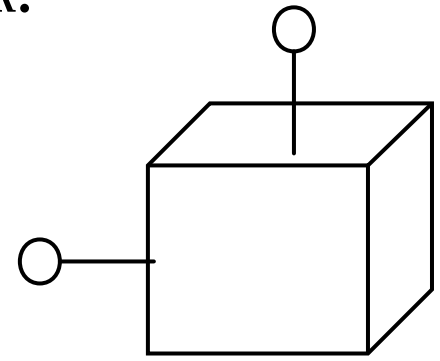
컴포넌트(Component)

정의

- 잘 정의된 인터페이스를 가진 독립적(independently)이고 캡슐화 된 (encapsulated) 소프트웨어 단위(software units)
- 인터페이스를 통해 서비스를 제공하는 소프트웨어 패키지(software package)

Object that are also part of a framework.

- Part of business frameworks (Ledger,Cost acct,Hum.Res..)
- Part of technology frameworks (COM/ActiveX, JavaBeans)

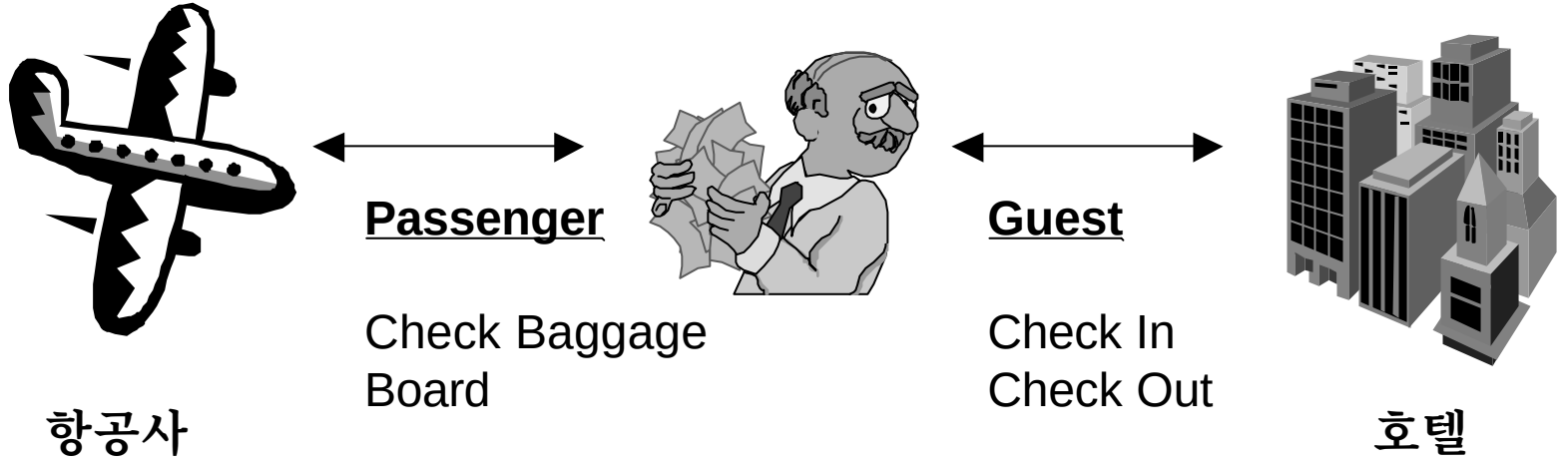


Component and Interface (계속)

인터페이스 (Interface)

- 서비스(service) 또는 오퍼레이션(operation)의 집합
- 인터페이스 모델링이 재사용을 위한 컴포넌트 설계의 핵심

- 객체(또는 컴포넌트)는 많은 상호작용을 하며, 객체의 역할은 다를 수 있다.



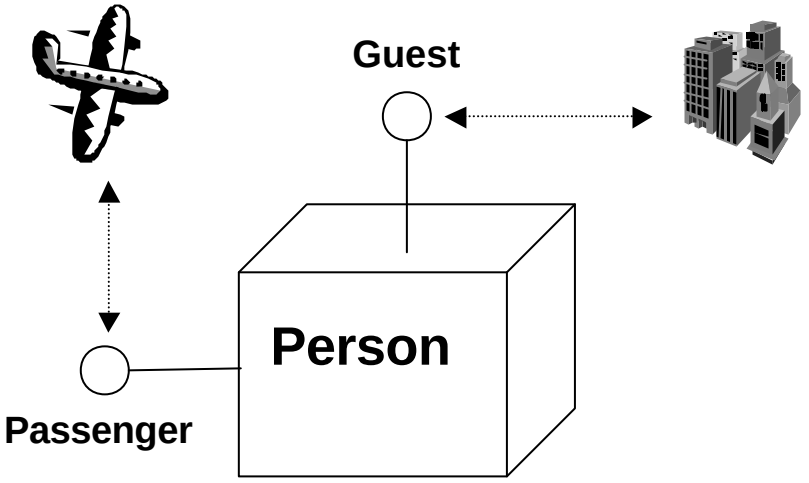
Component and Interface (계속)

인터페이스 (Interface)

- 컴포넌트는 여러 개의 인터페이스를 구현

```
Interface Guest {
    checkIn();
    checkOut();
}

Interface Passenger {
    checkBaggage();
    board();
}
```




```
Class Person
        implements Guest, Passenger {

        //implementation…
}
```

Component and Interface (계속)

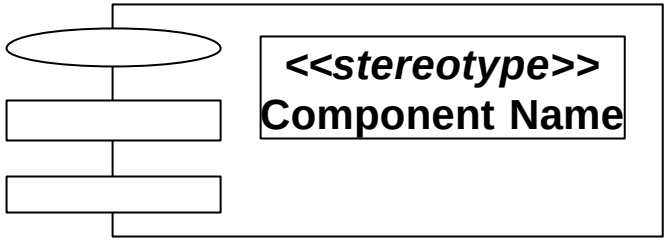
- 인터페이스 설계의 장점
  - 유연성 (Flexibility)
    - 다양한 요구에 따른 각기 다른 인터페이스의 집합을 쉽게 패키지화
  - 확장성 (Extensibility)
    - 기존 컴포넌트에 인터페이스를 추가함으로써 간단하게 새로운 기능 추가
  - 이식성 (Pluggability)
    - 같은 인터페이스만 유지한다면 컴포넌트를 새로 교체 가능
    - 재사용은 클래스뿐만 아니라 Framework, pattern도 가능
  - 기술 독립적(Technology Independence)
    - 추상적이고 상세한 컴포넌트 정의를 통해 여러 컴포넌트 기술(예, ActiveX, CORBA, Java Beans)로 구현 가능
  - 기존 어플리케이션 Wrapping
    - 인터페이스 랩퍼를 사용함으로써 기존의 어플리케이션을 재 모델
  - 다형성 (Polymorphism)
    - 동일한 인터페이스와 동일한 오퍼레이션 집합을 제공한다면, 클라이언트는 객체 타입을 고려하지 않고 서비스 요청 가능

Component Diagram(계속)

- **Software Component**
  - logical architecture(Classes, Objects, Relationships, Collaborations)안에서 정의된 기능들을 구현한 것
  - Component 내의 Class들은 상호 운영적인 아주 밀접한 관계가 있음
  - 개발환경에서의 구현 파일

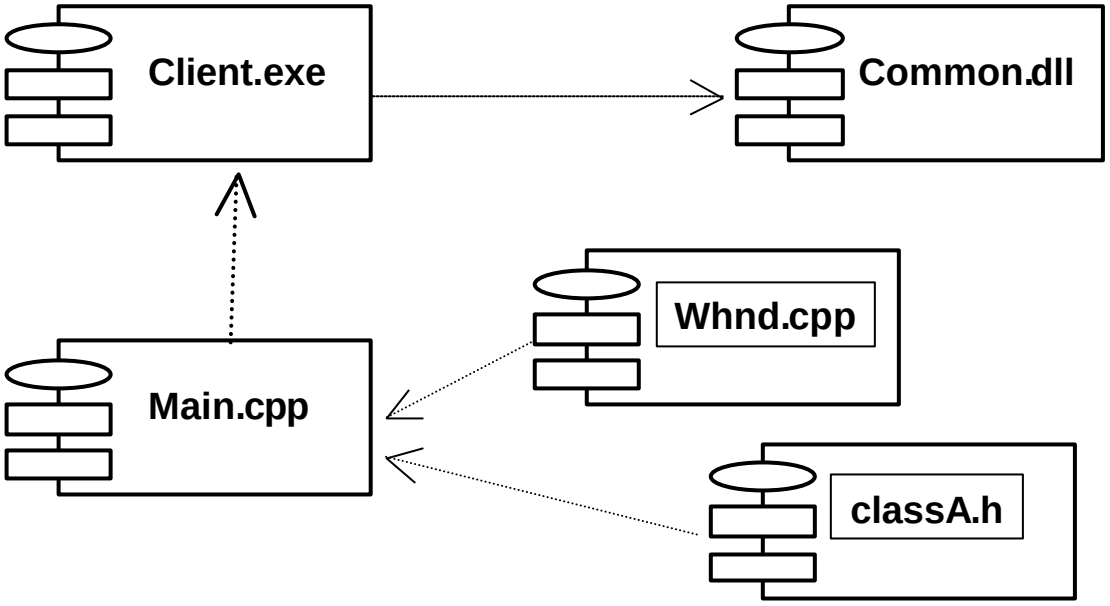
- **Stereotypes**
  - exe, dll, main program, headers, modules, forms

- **Component Notation:**



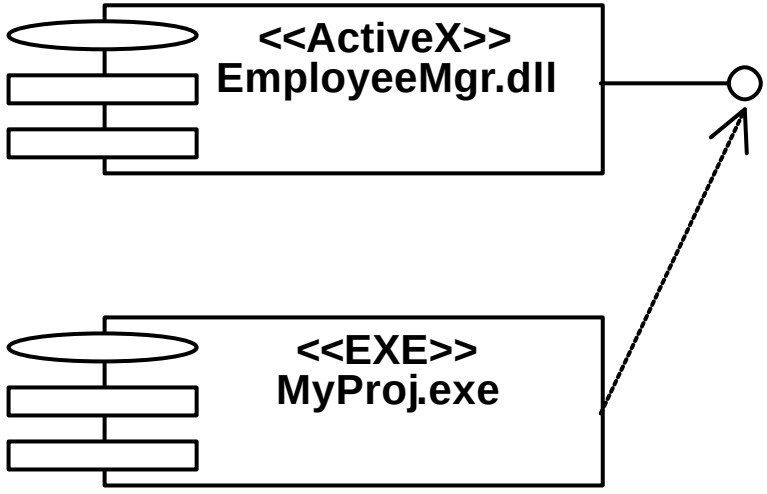
Component Diagram(계속)

- **Component Name**
  - 구현하고 있는 Class와 Object들을 표현
  - Component 이름은 실제 구현하는 Physical File이름과 관련됨



Component Diagram(계속)

- Component간의 Relationships
  - Dependency만 존재(dashed line)
  - 구현 파일들의 Compile-time의 종속관계 표현



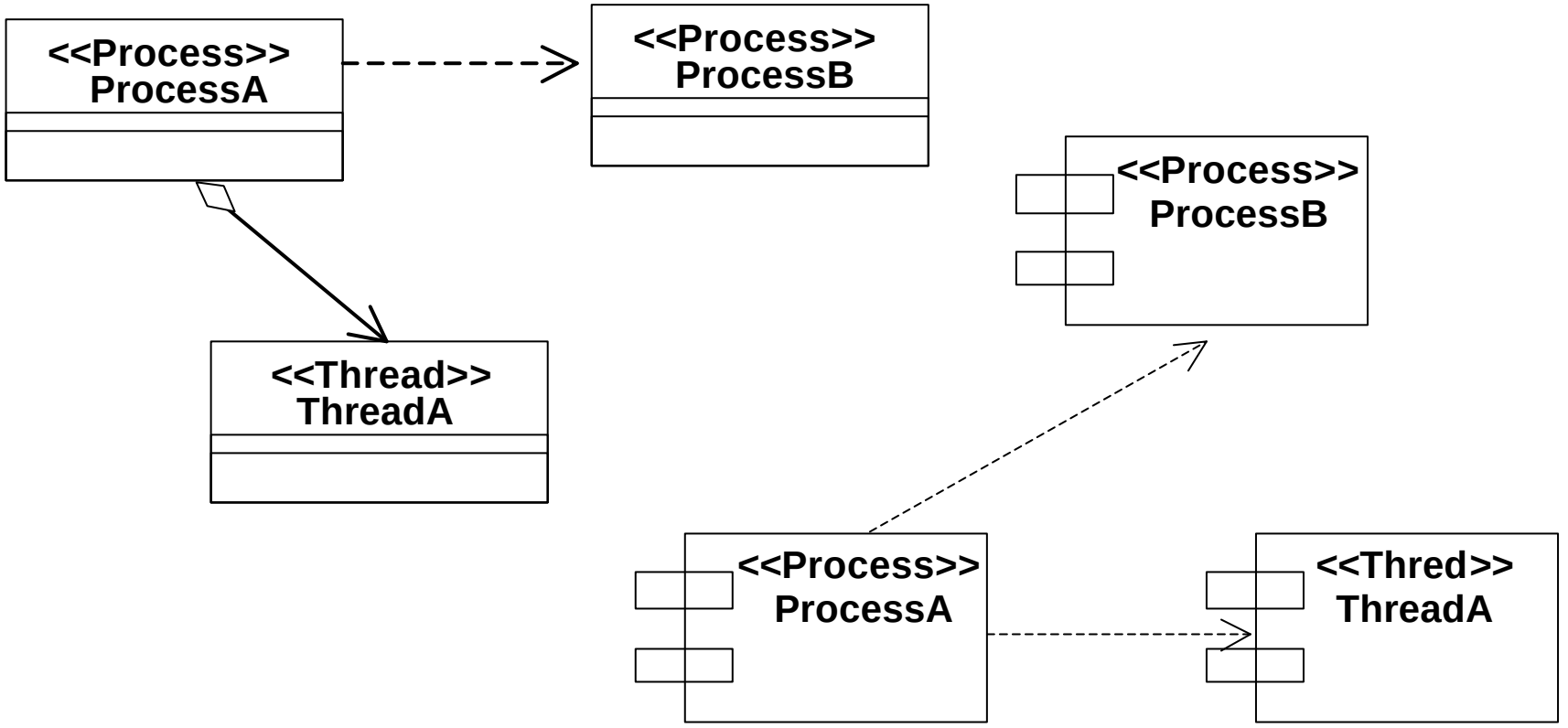
Process Modeling

- 정의
  - 프로세스 (process) : Heavyweight flow that can execute concurrently which other processes
  - 쓰레드 (thread) : Lightweight flow that can execute concurrently with other threads within the same process

- **Process 표기법**
  - <<process>>, <<thread>> 스테레오 타입 정의
    - **Active Class와 Object 로 표현**
    - **Component 로 표현**

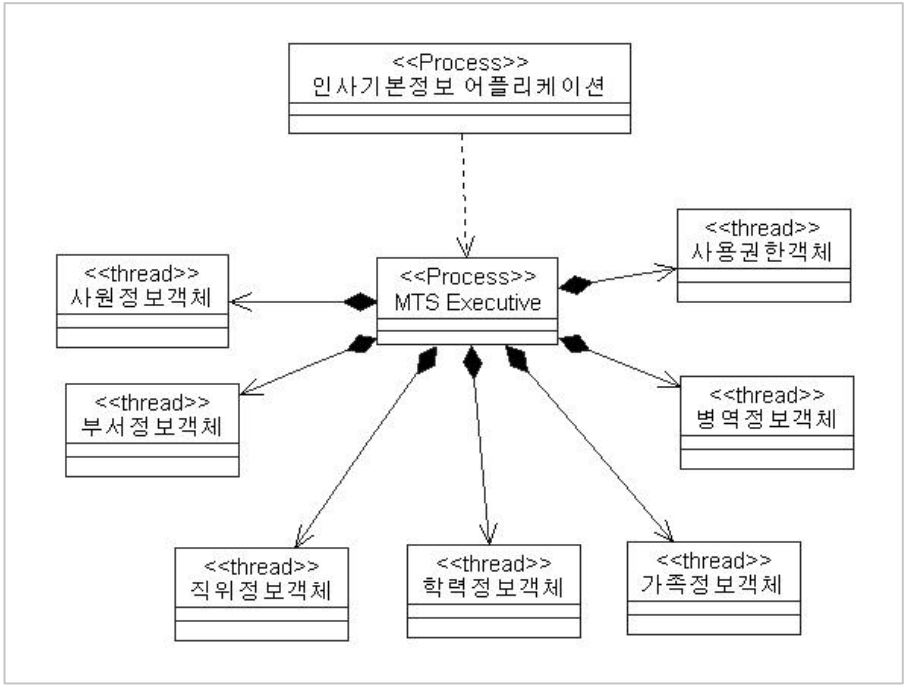
Process Modeling (계속)

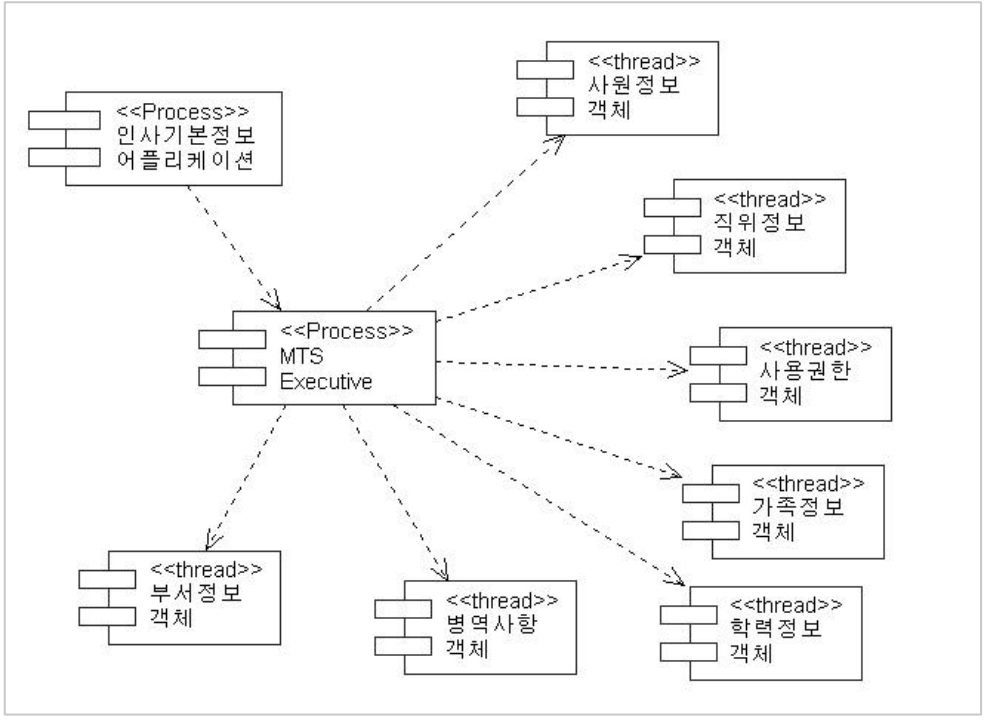
Process간의 Relationship



Process Modeling (계속)

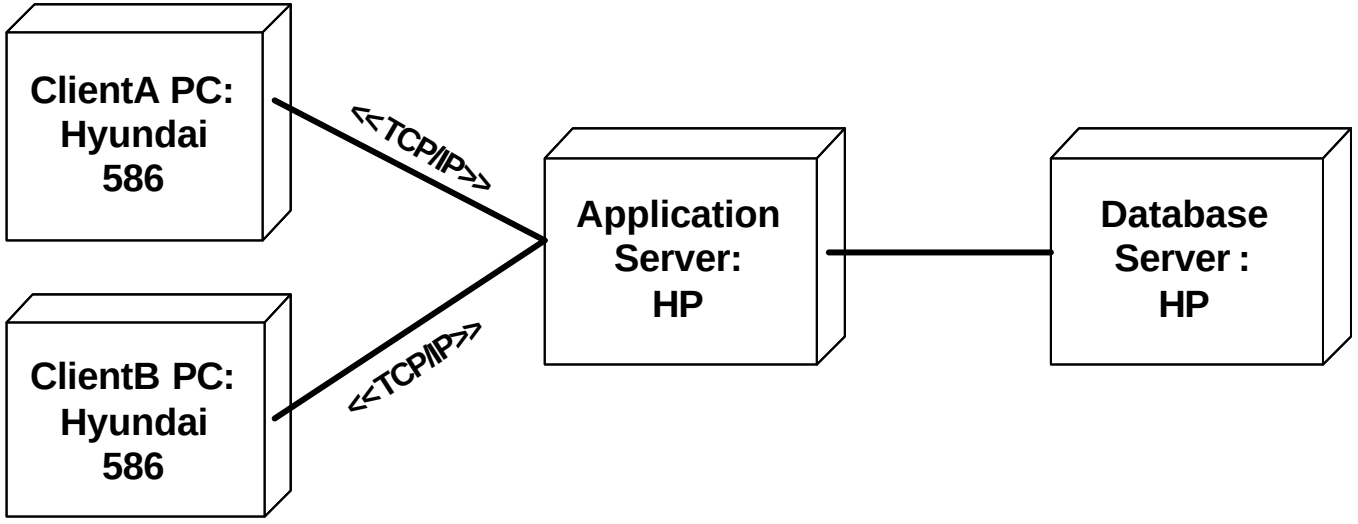
Process Model 예





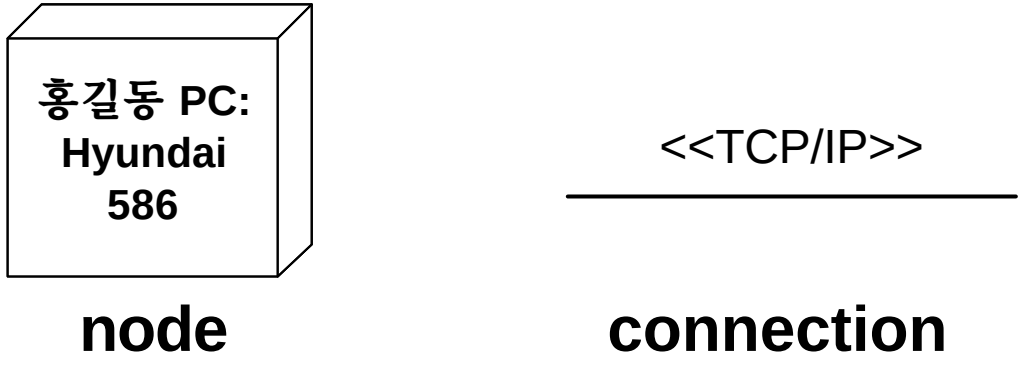
Deployment Diagram

- 특징
  - Processors, Devices, 그리고 Software Component들의 Run-time Architecture를 설명
  - 시스템 구현관점에서, Component들을 Node에 할당
  - 시스템과 다른 Node(processor, device)들간의 관계 설명
  - 다이어그램상에서 연결되어 있는 Node들은 Communication path를 반영



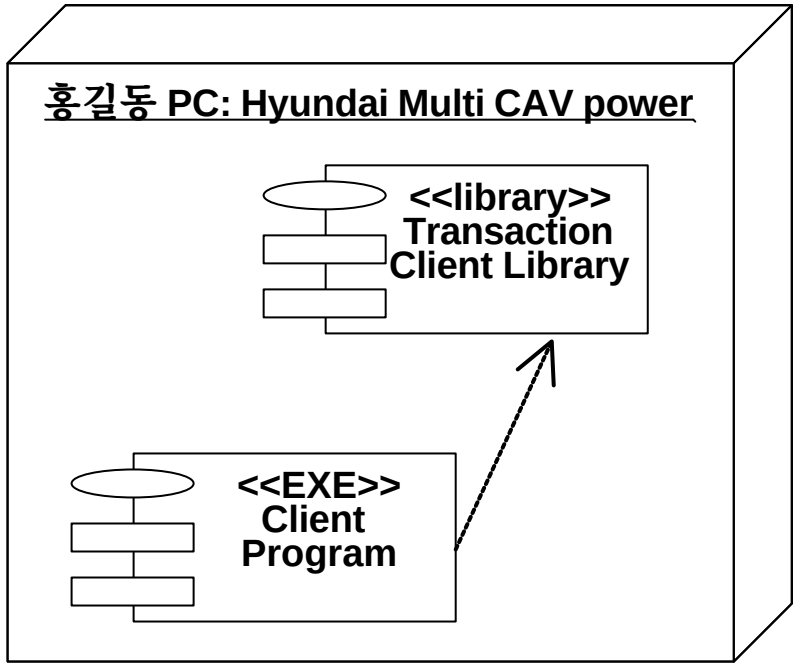
Deployment Diagram(계속)

- 주요 구성요소
  - Node :
    - Run-time physical object
    - Type과 Instance로 표현(Node is class)
    - Computer, Printer, Card Readers, ...
  - Connection : Node들 간의 Communication Association
    - Communication Type : Stereotype으로 표현
      (communication protocol 또는 network)



Deployment Diagram(계속)

- **Components and Nodes**
  - Executable Component Instance들이 Node Instance에 존재하며 실행 가능
  - Run-time Component만 표현



Deployment Diagram(계속)

- Node에 Component 할당
  - 하나의 Component는 적어도 하나 이상의 Node Instance에서 실행됨
  - Allocation시 고려사항
    - Hardware Resource Usage
    - 지리적 위치
      - Where specific functionality is required(on which nodes)
      - What functionality must be available locally
    - Access to devices
      - Can the printer be connected to the server, or does every client need a local printer?
    - 보안(Security)
      - Access 권한 및 정보 보호
    - Performance
    - Extensibility and Portability