Software Modeling & Analysis

- UML Report

T6 200811425 김평석
200811435 신성호
200811449 이찬희
200811454 전인서
200811462 최현빈

Contents

- History of UML & Rational Unified Process
- Construction of UML & Diagram
- Use Case, Sequence Diagram
- Class Diagram With Java
- UML Tools
- Reference

History of UML

& Rational Unified Process

- **History of UML**

- **Rational Unified Process**

History of UML (Unified Modeling Language)

- **UML 개발배경**

  객체지향적 분석과 디자인에 대해 다양한 방면으로 실험적인 접근을 하던 방법론자들에 의해서 다양한 객체지향(Object- oriented) 모델링 방법이 등장하게 된다. 그 중에서도 Booch의 OOD(Object-Oriented Design)과 Ivar Jachobson의 OOSE(Object-Oriented Software Engineering), Jim Rumbaugh의 OMT(Object Modeling Technique) 방법론이 유명하였는데, 이렇게 많은 방법론이 존재하다 보니 서로 다른 방법론을 사용하는 조직 간에 모델 정보를 공유하기 위해서는 각 조직에서 사용하는 방법론의 표현기호를 익히고 이해해야 했다. 따라서 모델을 표현하기 위한 동일한 기호, 모델링 언어를 사용하는 것에 대한 필요성을 느끼게 된다.

History of UML (Cont.)

- **UML 개발**

  UML 개발은 Booch와 Rumbaugh의 방법론을 통합시킨 Unified Method 0.8로 시작되었고, 이듬해 OOSE와 다른 기능들을 통합하면서 UML 0.9가 탄생하였다. 1997년 9월, UML1.1이 발표되었고, OMG(Object Management Group)가 표준으로 채택하였다. 그 후 UML은 2004년 UML 2.0 발표, 2010년 UML 2.3 발표, 2011년 UML 2.4 발표 등 꾸준히 수정 · 보완되고 있다.

Rational Unified Process

- **RUP란?**

  UML언어를 기초로 정의된 Unified Process를 Rational사에서 개발도구와 통합하여 개발한 객체지향 방법론이다.

- **RUP의 특징**

  Use-case기반으로, 분석,설계,테스트로의 추적성 및 일관성을 유지한다. 또한, 아키텍쳐를 중심으로 복잡한 프로젝트를 운영하고, 시스템의 무결성을 유지하도록 프로젝트 전체에 대한 통제를 할 수 있게 한다. RUP는 점진적인 개발과정을 채택하고 있는데, 이러한 개발단계 조정과 Workflow간에 적절한 조정이 매우 중요하다. 이 때 적절한 조합을 통해 개발을 진행하게 되며, 반복을 계속해서 소프트웨어 개발을 관리하며 최종적으로 완성된 시스템을 만들어내게 된다. RUP는 4개의 단계와 9개의 Workflow로 구분된다.

Rational Unified Process (Cont.)

- **RUP의 4단계**
  - Inception phase(개념화) : 프로젝트 범위 설정
  - Elaboration phase(상세화) : 문제인식 후 설계 및 위험제거
  - Construction phase(구축) : 반복, 점증적으로 개발
  - Transition phase(이행) : 사용자에게 시스템 교육 및 전달

Rational Unified Process

- **9단계 WorkFlow**
  - Business Modelling : 시스템이 사용될 조직의 업무 모델링
  - Requirements : 요구사항 분석
  - Analsys and Design : 분석 및 시스템 설계
  - Implementation : 시스템 구축
  - Test : 전체 시스템 검증
  - Deployment : 제품출시
  - Configuration and Change Management : 형상 및 제품관리
  - Project management : 프로젝트 과정 관리
  - Environment : 제품 및 개발 환경 관리

Construction of UML & Diagram

- **Construction of UML**
- **UML Diagram**
  1. **Class Diagram**
  2. **Object Diagram**
  3. **Use Case Diagram**
  4. **Collaboration Diagram**
  5. **Sequence Diagram**
  6. **State Chart Diagram**
  7. **Activity Diagram**
  8. **Component Diagram**
  9. **Deployment Diagram**

Construction of UML

- 측면
  - 정적(Static) 측면 : 시간을 흐름을 무시하고, 모델링의 대상 범위에서의 오브젝트의 구조와 관계를 나타냄
  - 기능적(functional) 측면 : 어떠한 것을 주고받으며, 화면 표시를 언제, 어떻게 바꾸는가를 나타냄
  - 동적(Dynamic) 측면 : 시간의 경과와 이벤트의 발생에 따라 오브젝트의 상태가 어떻게 변화하는지
  - 구조적(Implemental) 측면 : 시스템을 실행하기 위해 필요한 컴퓨터와 기억매체의 공간적 배치

Construction of UML (Cont.)

- 관점(perspective), 레벨
  - 개념 레벨 : 문제영역(도메인)의 해석, 구현과 상관없이 해당 영역에서 다루는 개념과 관계를 정리함으로 써 문제영역의 다양한 사항을 이해할 수 있게 함
  - 사양 레벨 : 설계 작업에 대응하고 문제에 대한 해결책을 완성함. 즉, 필요기능과 실현방법을 고안
  - 구현 레벨 : 개발 작업에 대응하며, 사양 레벨에서 고안된 실현방법을 기초로 하여 소프트웨어를 작성하는데 필요한 정보를 덧붙임

- 하지만 측면과 관점에 대한 구분기준이 명확하지 않아서, 개발자가 수행하고 있는 모델링 작업의 목적이 무엇인지를 확실히 아는 것이 중요하다

UML Diagram

- 다이어그램의 목적은 시스템을 여러가지 시각에서 볼 수 있는 뷰를 제공하는 것이다. 구분 요소에 따라 다이어그램을 나누면, 다음과 같다.

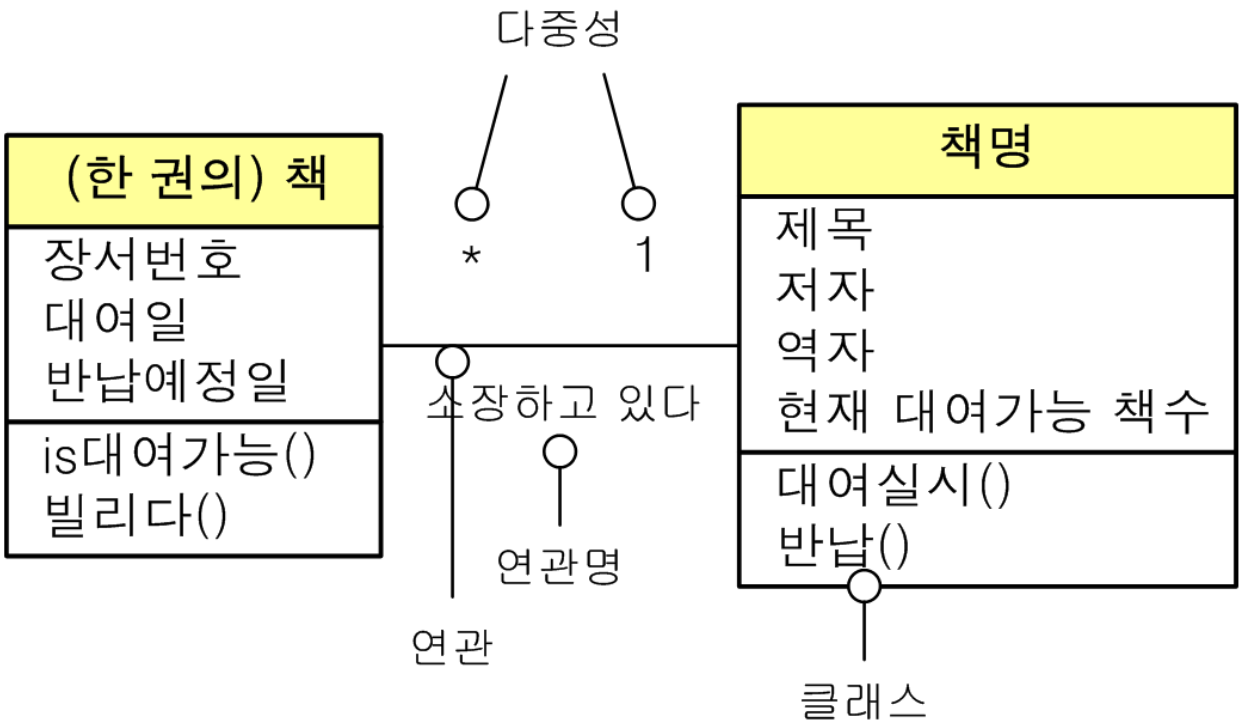
<table>
<tr><th>관점 / 측면</th><th>정적 측면</th><th>구조적 측면</th><th>동적 측면</th><th>물리적 측면</th></tr>
<tr><td>개념 레벨</td><td rowspan="3">Class Diagram

Object Diagram</td><td>Use Case Diagram</td><td rowspan="3">Activity Diagram

State Chart Diagram</td><td rowspan="2"></td></tr>
<tr><td>사양 레벨</td><td rowspan="2">Collaboration Diagram
Sequence Diagram</td></tr>
<tr><td>구현 레벨</td><td rowspan="2">Component Diagram
Deployment Diagram</td></tr>
</table>

Diagram 1. Class Diagram

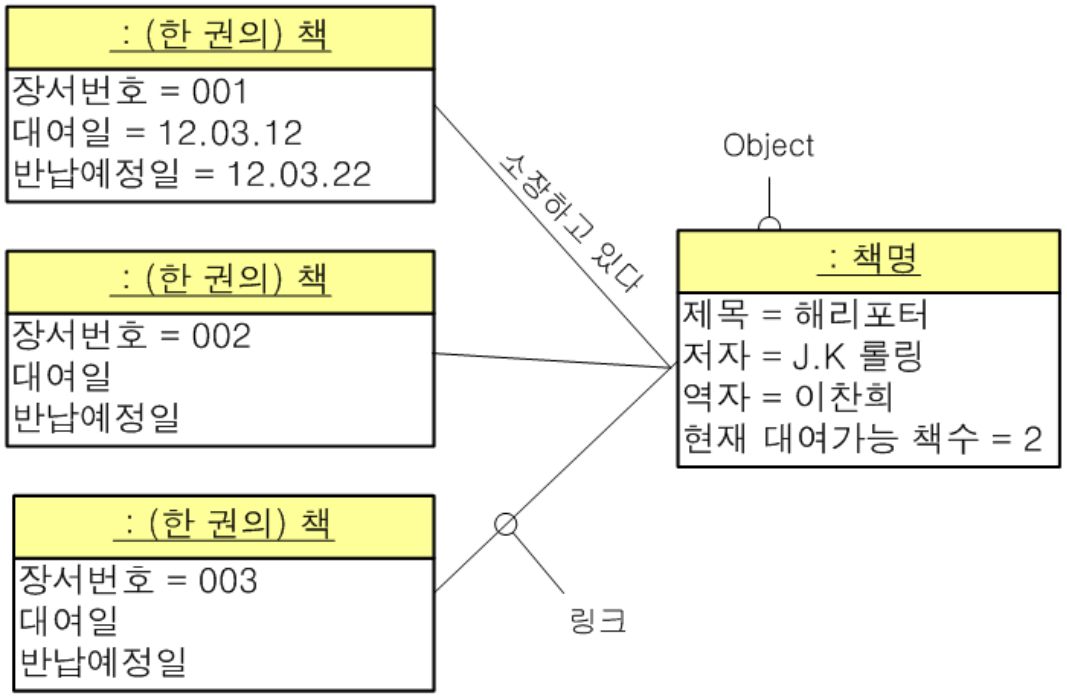
- Class Diagram은 문제영역을 구성하는 모델 요소간의 관계를 나타내는 다이어그램



< 도서관 대여관리 시스템의 예 >

Diagram 2. Object Diagram

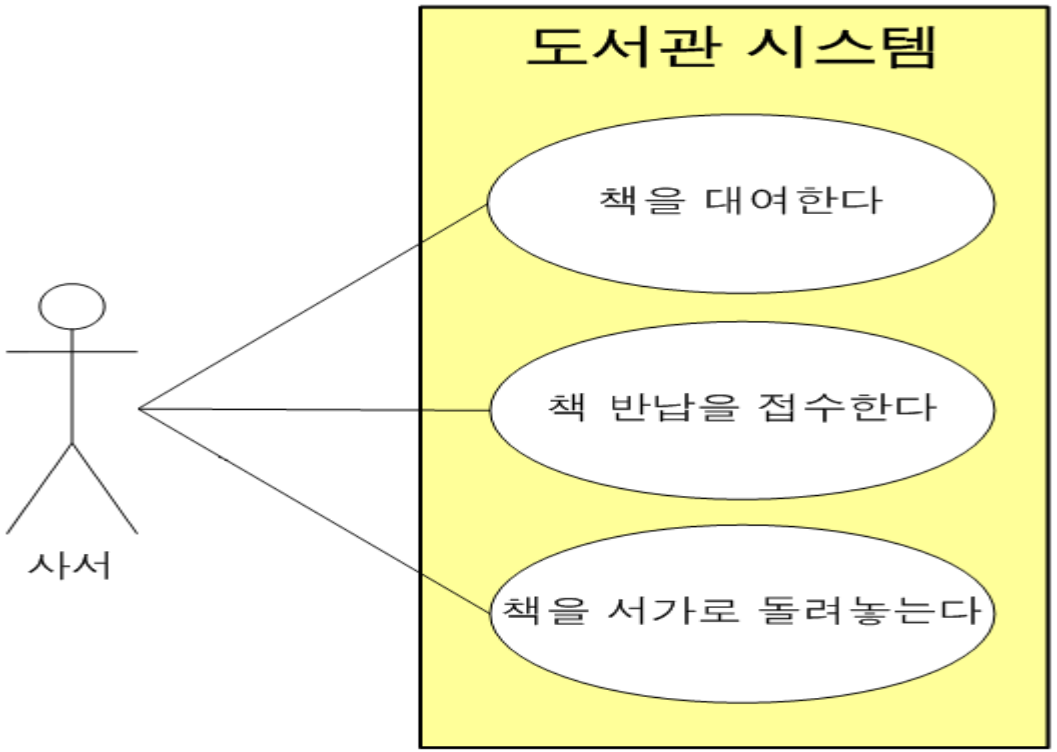
- 구체적인 예(인스턴스)에 의한 구조를 기술
- Class Diagram를 이해하기 쉽게 한 Diagram



〈 도서관 대여관리 시스템의 예 〉

Diagram 3. Use Case Diagram

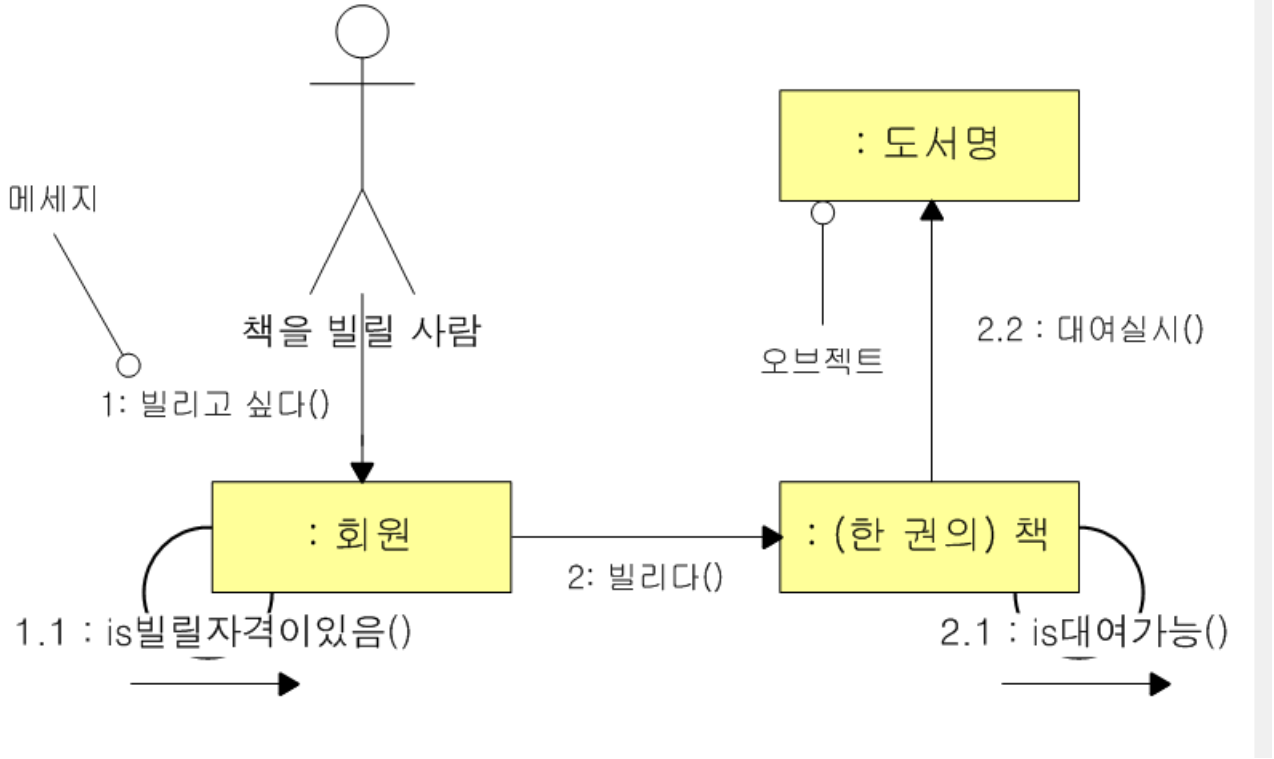
- 액터(시스템과 커뮤니케이션 하는 주체(사람, 시스템 등))의 시점에서 본 시스템의 기본적인 행동을 기술



< Use Case Diagram >

Diagram 4. Collaboration Diagram

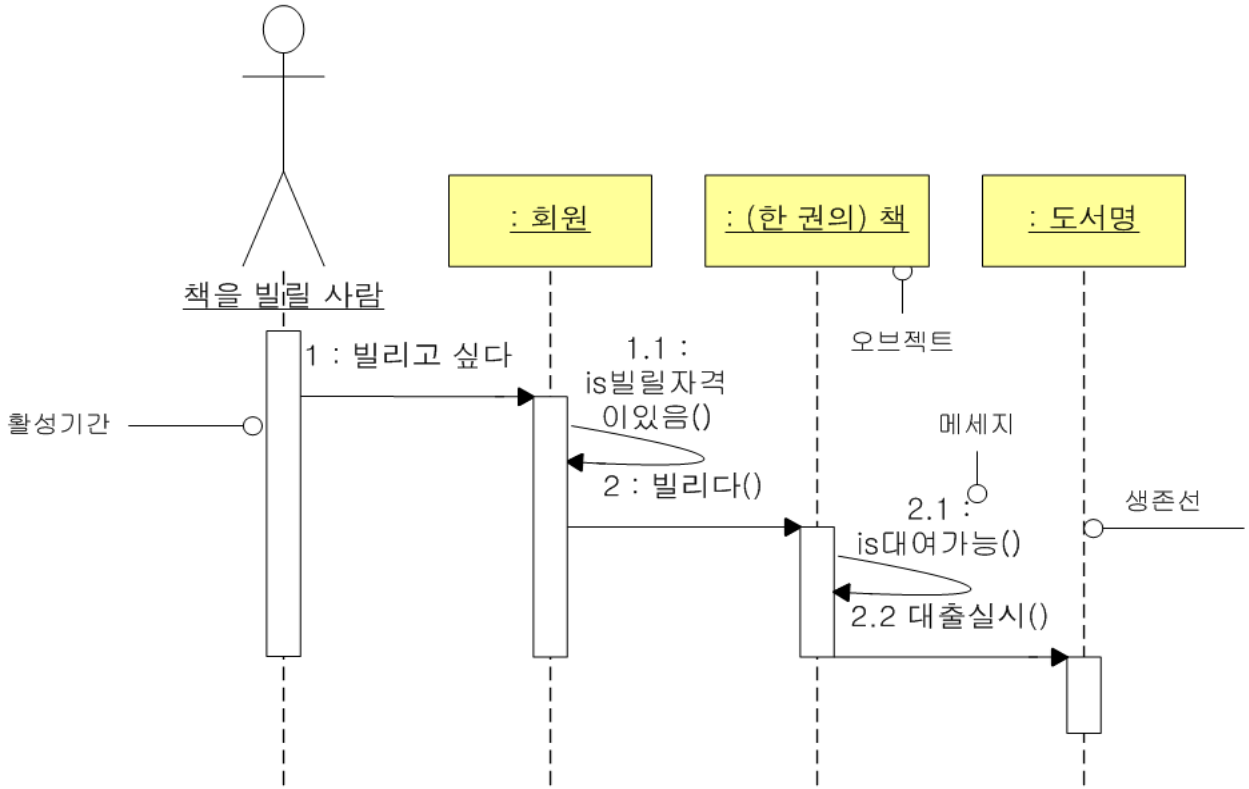
- Object Diagram에 메세지의 화살표로 송신관계를 부가한 것
- 메시지는 조작의 출발점, 번호를 적어 메시지가 전달되는 순서를 나타냄



< Collaboration Diagram >

Diagram 5. Sequence Diagram

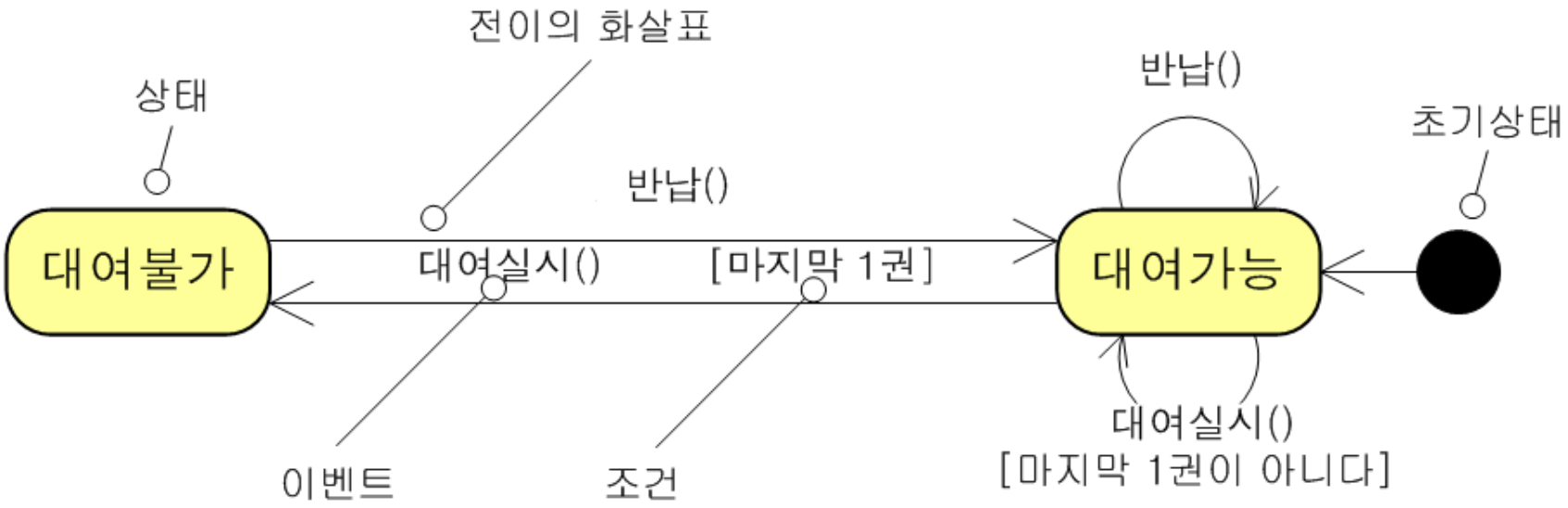
- 그림의 아래로 향하여 시간이 흐르고, 옆으로 오브젝트 간을 메시지가 왕래함
- Collaboration Diagram과는 다르게 시간의 흐름을 표현



< Sequence Diagram >

Diagram 6. State Chart Diagram

- 오브젝트가 가질 수 있는 상태의 변화를 나타낸 것
- 오브젝트의 상태를 모서리가 둥근 사각형으로 나타내고, 전이를 나타내는 화살표(열린 화살표)의 방향으로 그 사이의 전이 관계를 나타낸 것

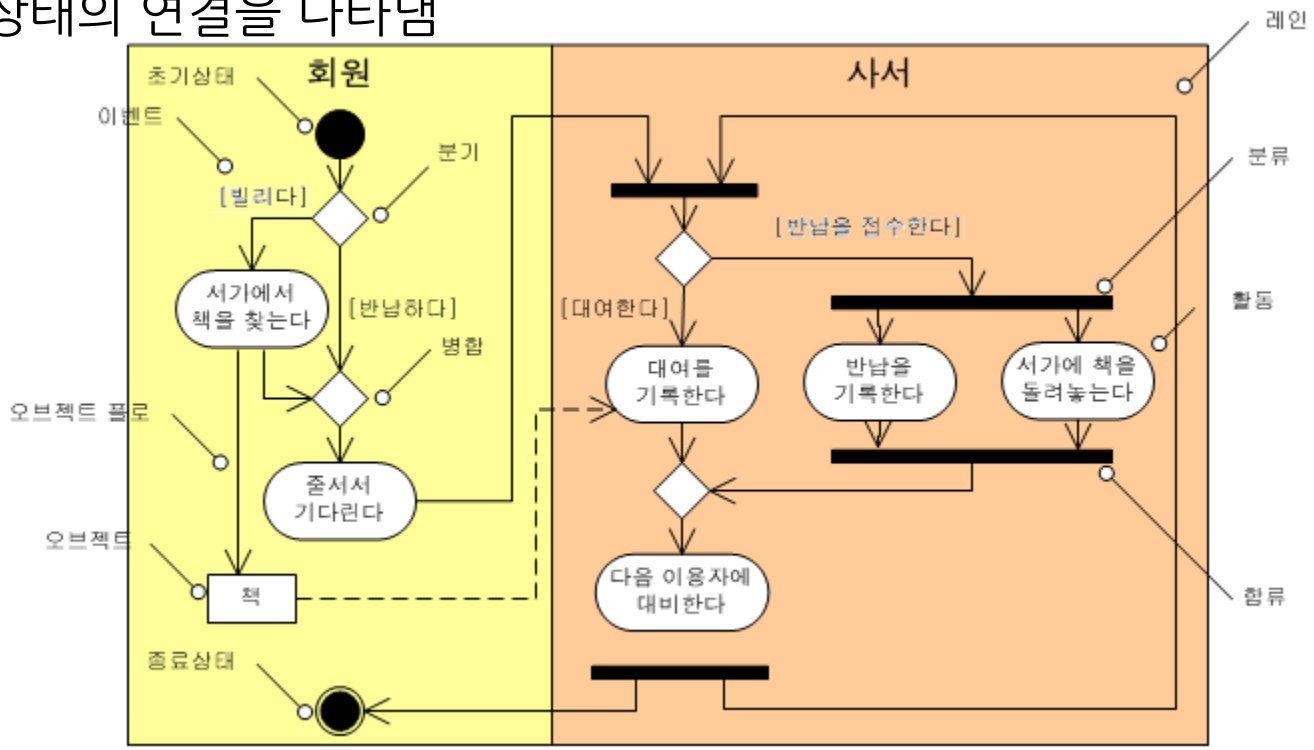


< State Chart Diagram >

'도서명'이라는 오브젝트가 취할 수 있는 상태를 나타냈다.
상태변화가 시스템의 행위에 큰 영향을 미치는 오브젝트에 대해서 이 다이어그램을 작성한다

Diagram 7. Activity Diagram

- State Diagram의 특수한 형태
- 어떤 활동의 동작 상태가 완료되면, 다음 활동이 동작상태로 바뀌는 일련의 동작상태의 연결을 나타냄

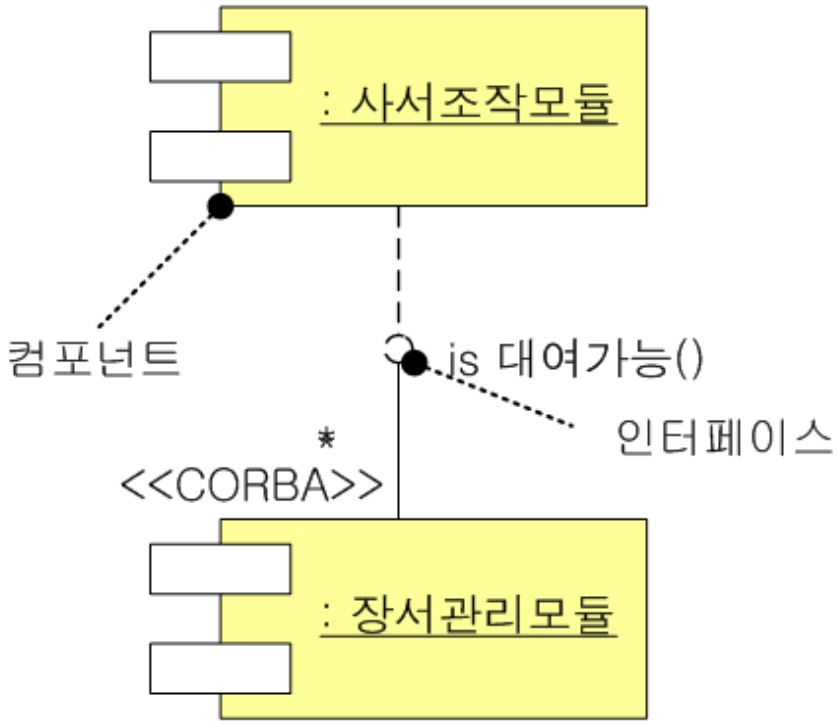


<Activity Diagram>

책의 대여 및 반납의 Workflow, 회원과 사서의 활동을 구분할 수 있어, 보기가 편하다.
다만 무리한 일체화는 오히려 문제가 될 수 있다

Diagram 8. Component Diagram

- 컴포넌트들을 나타내고, 의존을 나타내는 화살표 등으로 그 의존 관계를 기술

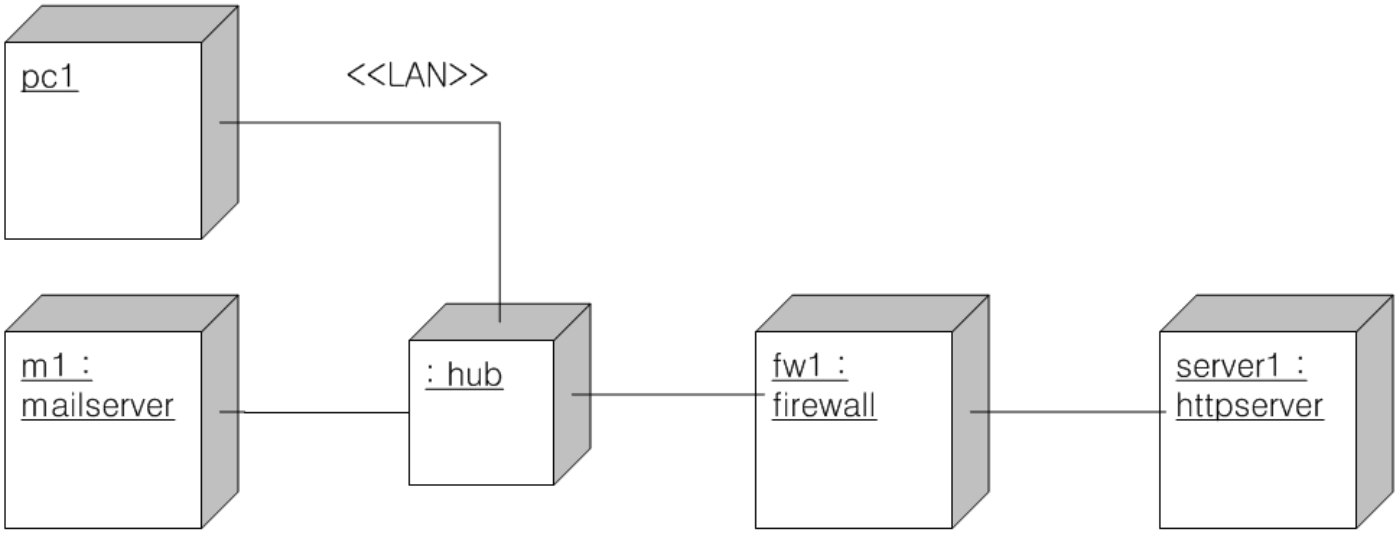


< Component Diagram >

사서조작 모듈 컴포넌트가 장서관리 모듈 컴포넌트의 'is대여가능()' 이라는 인터페이스를 사용하고 있다는 것을 나타낸다

Diagram 9. Deployment Diagram

- 시스템의 실행 시에 필요로 하는 물리적인 처리자원, 실행 모듈, 소프트웨어 컴포넌트의 인스턴스를 노드라고 함
- 노드를 입체로 표현하고, 그 사이를 의존 화살표와 접속 관계를 나타내는 실선으로 연결, 이들의 통신 관계를 나타냄



< Deployment Diagram >

컴퓨터나 LAN 등이 어떤 물리적 관계로 접속되어지는가를 보여준다

Use Case Diagram

& Sequence Diagram

- **Use Case Diagram**

- **Sequence Diagram**

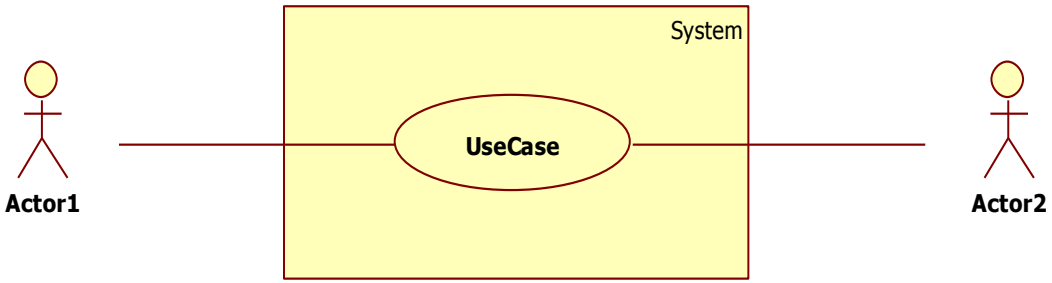
Use case Diagram

- Use case와 Actor, 그들 간의 관계를 함께 나타내는 것으로서, 시스템의 동적인 측면 모형을 작성한다. 시스템이나 하위 시스템, 클래스의 행위 모형을 작성하는데 중심이 된다.

- 참고사항
  - 사용자 요구분석이 선행되어야 한다.
  - Forward Engineering에서 실행 시스템을 시험할 때 중요
  - Reverse Engineering에서 실행 시스템을 파악할 때 중요

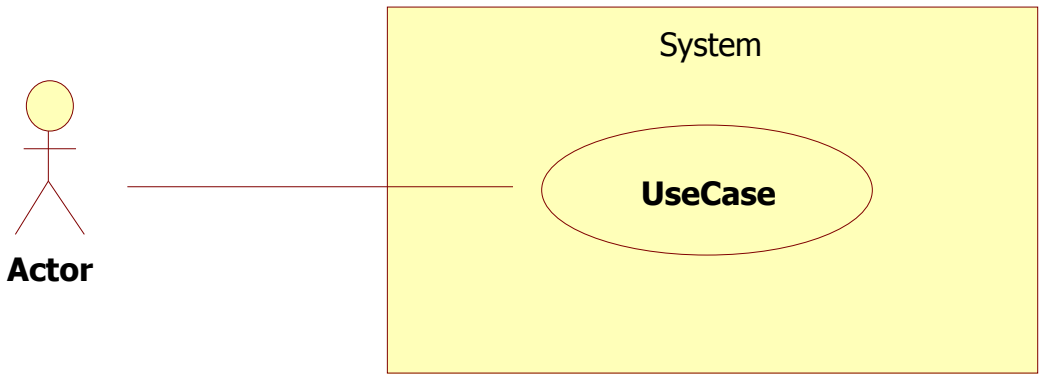
Use case Diagram (Cont.)

- 포함되는 내용
  - Subject(System)
  - Use case
  - Actor
  - Association, Generalization, Include, Extend관계 명시



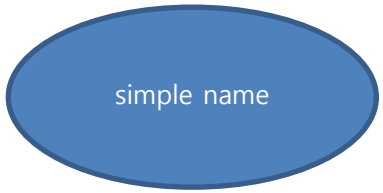
Use case Diagram (Cont.)

- Subject
  - 전체시스템(Subject)을 큰 사각형으로 나타낸다.
  - 사각형 밖에 위치할 Actor를 배치하고 Subject와의 교류를 표시한다.
  - Use case Diagram을 이용하여 Actor의 역할을 명시한다.
  - 시스템 생명에 필요한 Actor만 포함시킨다.

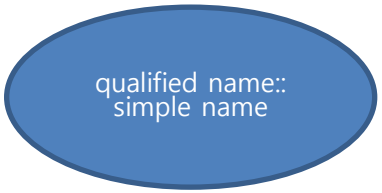


Use case Diagram (Cont.)

- Use case
  - 순차적으로 발생하는 동작(시스템이 제공해야 하는 서비스)을 기술하는 것이며 변이를 가질 수 있고, 시스템은 이러한 활동을 수행하여 Actor에게 결과를 준다.
    - ex) (네이버 이용시) “회원가입” “이메일” “검색”
  - Use case는 이름으로 구별하며 이때 simple name은 use case자체의 이름이고 qualified name은 use case가 속한 group의 이름이다.



< simple name의 use case 도식의 예 >



< qualified name의 use case 도식의 예 >

Use case Diagram (Cont.)

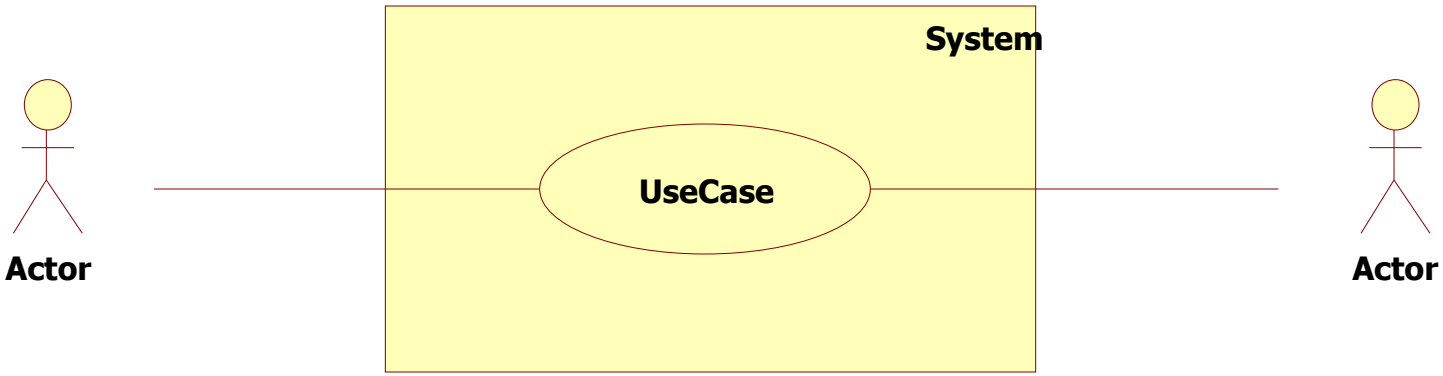
- Collaboration
  - Use case가 하는 일을 명시적으로 나타낸다.
  - 대부분의 경우 하나의 use case는 하나의 collaboration으로 구현 가능하기 때문에 이 경우 명시적으로 작성할 필요 없다.
    - 모든 use case에 명시된 흐름을 만족시키면서 최소의 개수로 좋은 구조로 된 collaboration을 찾는 것이 시스템 아키텍처의 논의대상

< use case와 collaboration의 실현화 관계 도식의 예 >

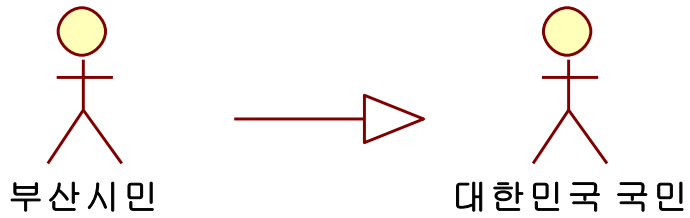
Use case Diagram (Cont.)

- Actor
  - Use case와 교류할 때 사용자가 맡고 있는 역할을 나타낸다.
  - 사람이나 하드웨어 장비, 또는 다른 시스템이 시스템과 맺고 있는 역할을 나타낸다.
  - 시스템을 제외한 모든 외부요소를 의미한다고 볼 수 있다.
  - 시스템에 속하는 것이 아니라. 시스템 바깥에 위치한다.
  - 한 개체가 여러 Actor의 역할을 수행할 수 있다.
  - Actor의 관계에 따라 Generalization, Specialization 관계를 나타낼 수 있다.
  - 허수아비 모양으로 나타낸다.
  - 빈 세모의 화살표는 Generalization관계를 표현한다.

Use case Diagram (Cont.)



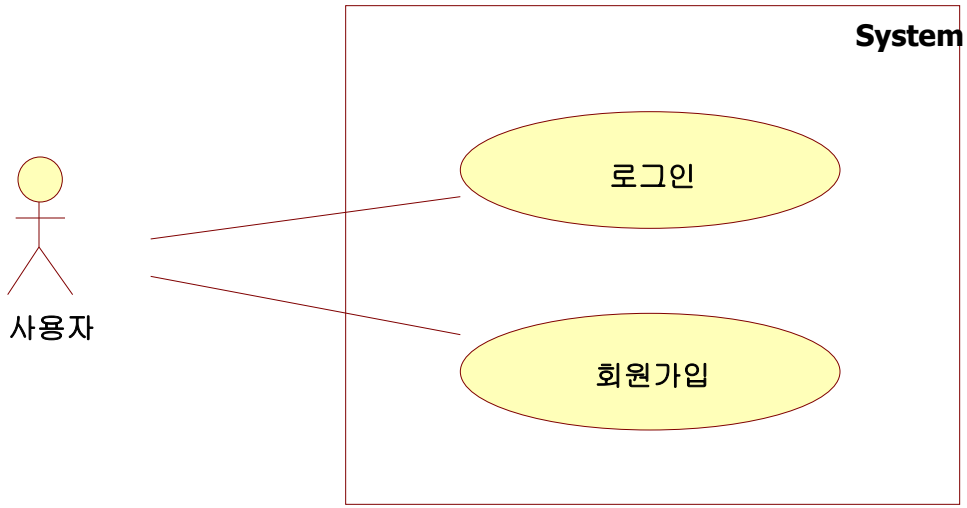
< 전체적인 Diagram에서 Actor도식의 예 >



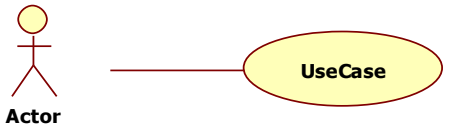
< Actor간의 Generalization도식의 예 >

Use case Diagram (Cont.)

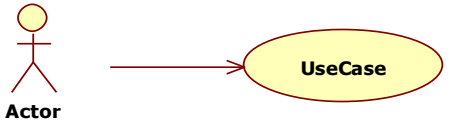
- Association
  - Actor는 Use case에 연결되게 되는데 이는 연결된 use case와 대화한다는 표시이고, 메시지를 주고 받을 수 있다. 실선으로 표현되며 메시지의 방향을 표현할 수도 있다.



< Actor와 Use case간의 관계를 표현한 도식의 예 >



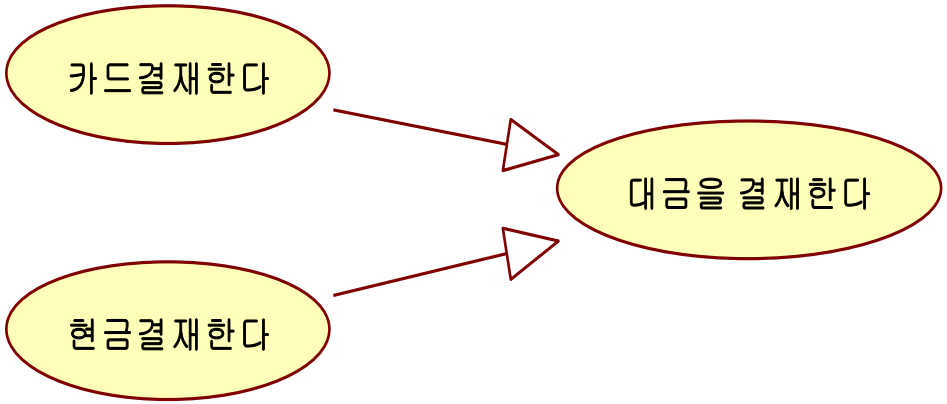
< 양방향 >



< Actor만 메시지를 보내는 단방향 >

Use case Diagram (Cont.)

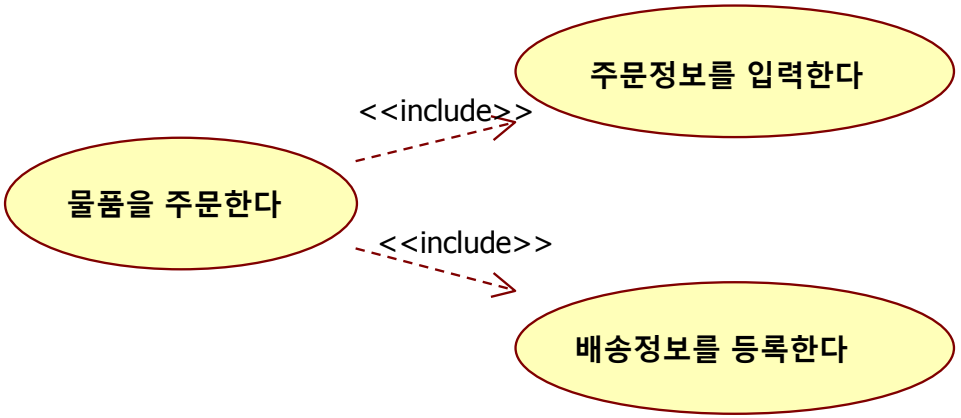
- Use case간의 관계
  - Generalization
    - 클래스의 Generalization과 동일하다.
    - 하위 Use case가 상위 Use case에게 기능을 상속받음을 표현 한다.



< Use case의 Generalization도식의 예 >

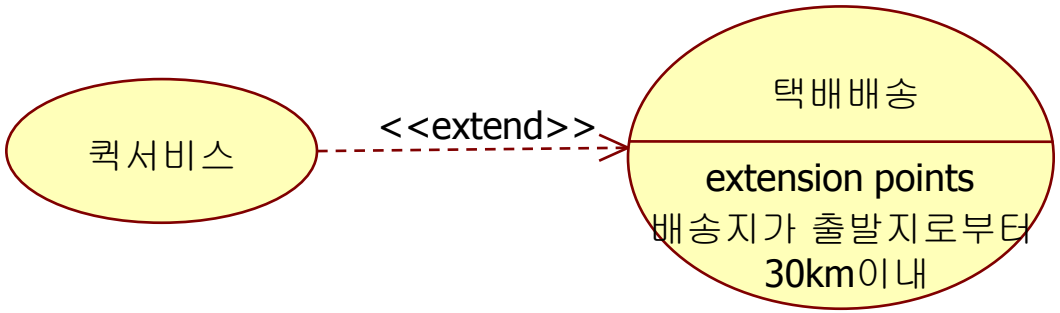
Use case Diagram (Cont.)

- Use case간의 관계
  - Include
    - Use case가 다른 Use case를 가져다 쓰게 될 때의 관계를 Include를 통해 표현한다. << include >>
      - ex) 물품 주문 시 주문정보를 입력 받고 배송정보를 입력 받는 것
    - 포함되는(주문정보 입력, 배송정보입력)은 독자적으로 인스턴스를 생성하지 못한다.



Use case Diagram (Cont.)

- Use case간의 관계
  - Extend
    - 기본 Use case수행 시 특별한 조건(Extension points)을 만족할 때 수행되는 Use case를 표시



Sequence Diagram

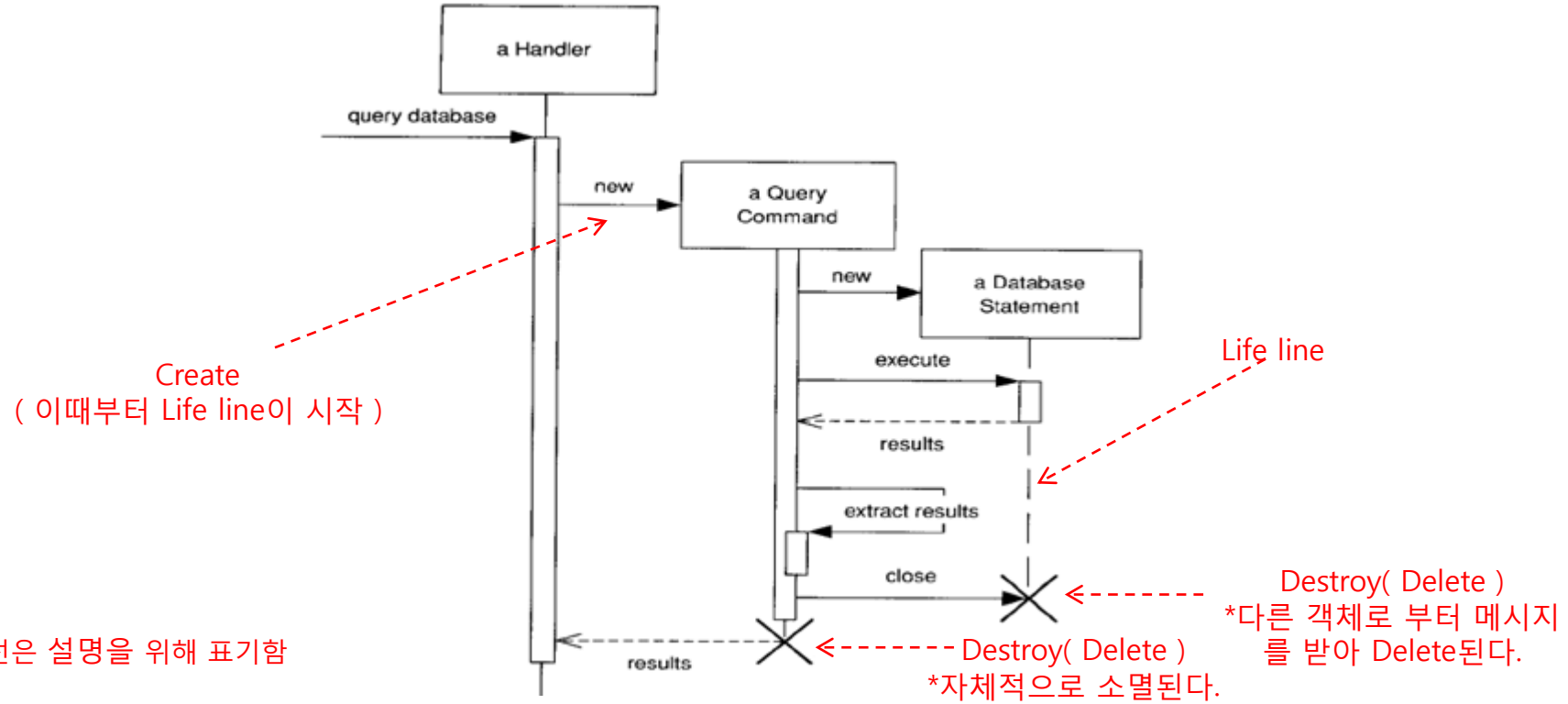
- 시간 진행에 따른 메시지 순서를 강조하며 객체간의 동작을 설명할 필요 없이 잘 표시되어 있다.
- 시스템의 동적 구조, 즉 객체와 객체그룹 사이, 객체와 객체 사이, 객체그룹과 객체그룹 사이의 동적인 행위를 기술한다.
  - 하나의 Use case에서 객체간의 Interaction을 확인해야 하는 경우에 사용한다.
  - Action의 정의를 표현하는 데는 적합하지 않다.

Sequence Diagram (Cont.)

- 작성요령
  - Interaction을 하는 개체를 가로축에 배치한다.
  - Interaction을 주도하는 개체를 왼쪽에 배치하고 부속되는 개체를 순차적으로 오른쪽에 배치한다.
  - 개체가 주고받는 메시지는 세로축에 배치시키되 시간의 흐름과 개체의 관계가 일치 하도록 한다.

Sequence Diagram (Cont.)

- Life line
  - 수직 점선으로 나타내며 특정 기간 동안 객체가 살아있음을 나타낸다.
  - Life line은 create메시지를 받은 시점에서부터 시작된다.
  - 이때 Life line시작점(맨 위쪽)에 사각형을 그려 객체를 나타낸다.
  - destroy메시지를 받으면 끝난다.(대문자 X를 표시하여 나타낸다.)

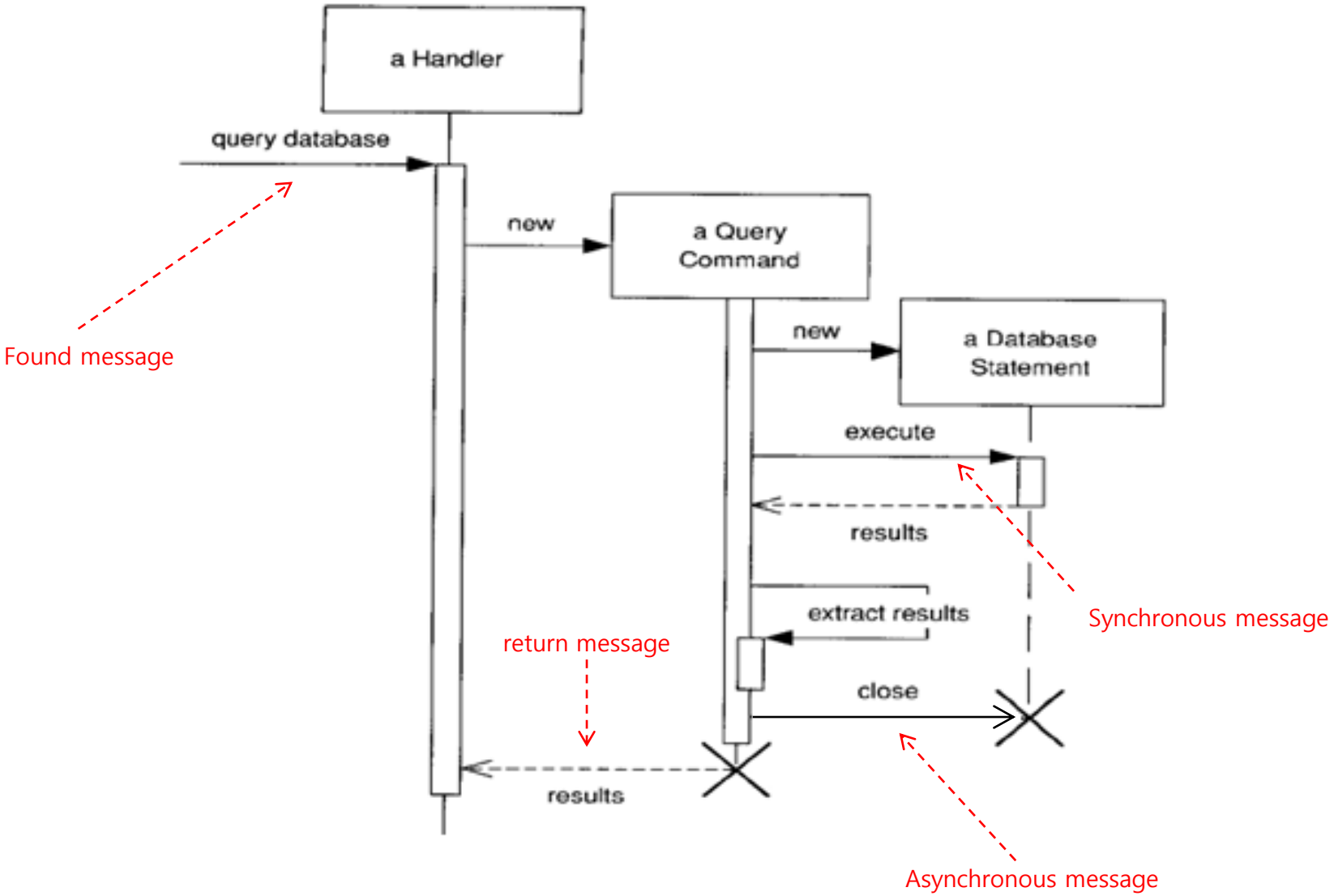


Sequence Diagram (Cont.)

- Message
  - 한 life line에서 다른 life line으로 화살표로 나타낸다.
  - 화살표 방향은 메시지를 받는 쪽이다.
  - 매개변수와 클래스 이름을 명시할 수 있다.
    - name : Class
  - Synchronous & Asynchronous Call
    - 동기 : 호출하는 객체가 return을 대기해야 하는 경우(실선의 속이 찬 세모화살표) ⟶
    - 비동기 : 호출하는 객체가 return을 대기할 필요가 없는 경우(실선의 화살표) ⟶
  - 첫 번째 메시지는 확인되지 않은 source로부터 발생되기 때문에 메시지를 생성한 참가자가 없다. 이런 메시지를 found message라고 한다.
  - return
    - 점선의 화살표 ---⟩
    - 생략할 수 있지만 표시하는 것이 유리한 경우도 있다.

- Focus of control
  - Life line위에 길이가 긴 직사각형으로 나타내고 개체가 활동함을 의미한다.
    - 직접 수행하든가 하위 프로시저를 호출할 때

Sequence Diagram (Cont.)



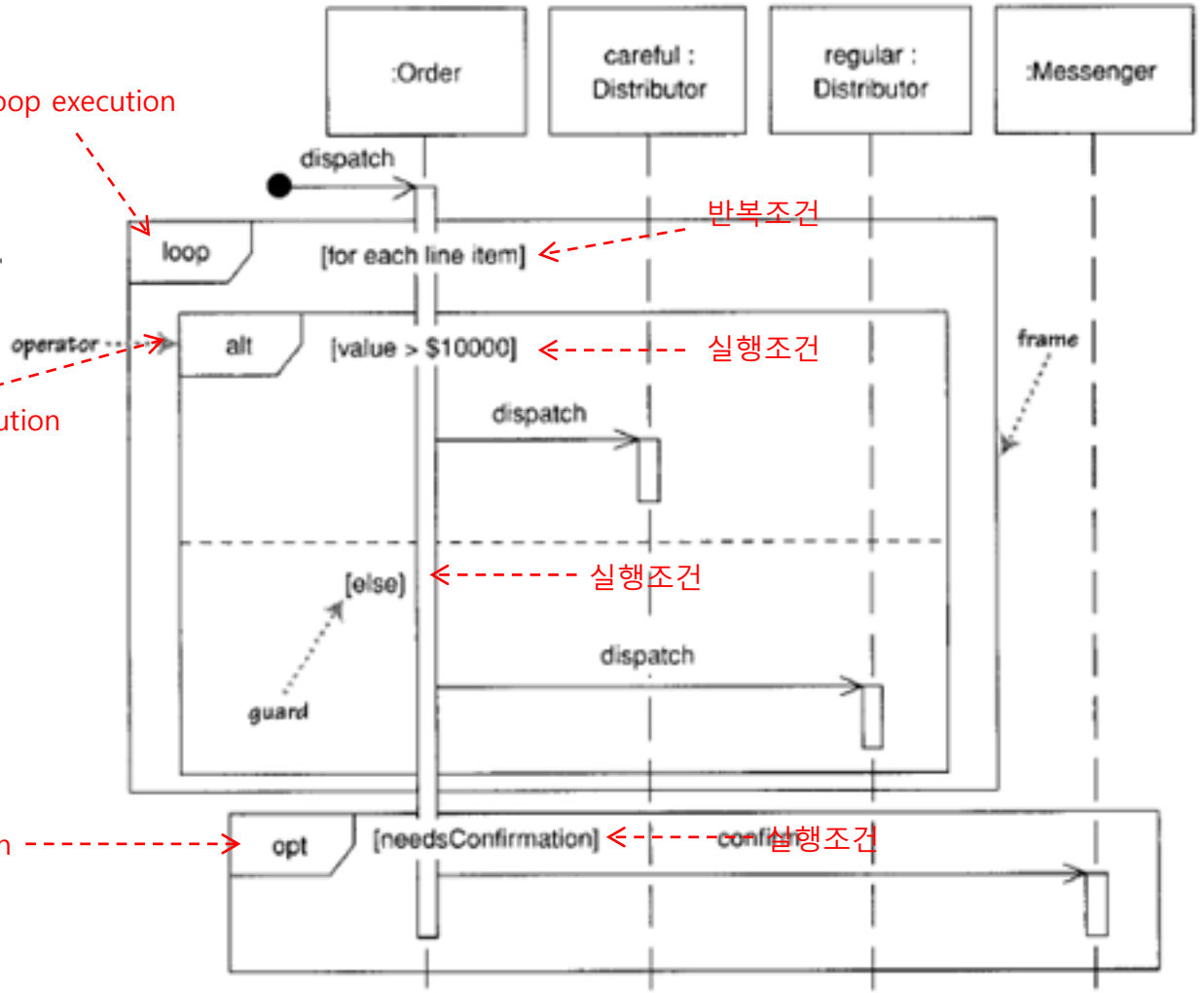
Sequence Diagram (Cont.)

- Loops, Conditionals, and the Like
  - 각 부분을 사각형 영역으로 나타낸다.
  - 왼쪽 위의 라벨형태로 어떤 종류인지 명시한다.
    - Optional execution : opt 로 명시. 참이 되는 조건식을 명시하며 그 조건이 참일 경우 몸체(사각형영역)이 실행된다는 의미이다.
    - Conditional execution : alt로 명시. 사각형 영역을 복수개로 나누고 점선으로 구분하여 표시한다. 이때 조건이 맞는부분의 영역이 실행되며 위쪽부터 순차적으로 접근한다. 모든 조건이 맞지 않으면 그영역을 빠져나간다고 볼 수 있다.
    - Loop(iterative) execution : loop로 명시. 반복조건이 참인경우 계속해서 사각형 영역을 반복하게 된다. 거짓이면 그 영역을 빠져나간다.
    - 이 밖에도 많은 연사자들이 있다.
  - Sequence Diagram은 loop나 conditional을 표현하기에는 적합하지 않다.

Sequence Diagram (Cont.)

```
procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confir
end procedure
```

< 소스코드 예제 >



Sequence Diagram (Cont.)

- 여러 Use case에 걸친 하나의 객체의 behavior를 보고 싶으면 state diagram을 사용해야 한다.

- use case또는 thread에서 동작하는 객체의 behavior를 보고 싶으면 activity diagram을 사용해야 한다.

Class Diagram With Java

- **Java와 UML**
- **Class Diagram**
  - **Class**
    - **Concrete Class**
    - **Abstract Class**
  - **Interface**
  - **Exception**
  - **Relation**
    - **Association**
    - **Generalization**
    - **Realization**
    - **Dependency**
  - **Package**

Java와 UML

- java programming 에서 가장 중요한 것은 “적절한 객체 지향 설계를 하고 있는가?” 이다. 자바가 객체지향 언어이기 때문에 객체 지향 분석 설계에 맞게 구성되어야 할 것이다.

- 결국 Class, Use Case Component, Package라는 기본요소와 Dependency, Generalization이라는 기본 관계를 사용하면 여러 문제를 정적으로 모델화 할 수가 있다.

- 또한 State chart Diagram이나 Sequence Diagram Component Diagram,의 기본적인 기능을 사용하면 대부분의 문제를 동적인 측면에서 모델화 할 수 있다.

Class

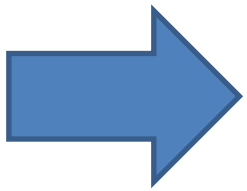
- 클래스는 객체 지향의 중심이 되는 개념으로 당연히 자바에서도 가장 중요한 기능이다.
- UML에서는 여러 가지 종류의 클래스를 적절한 방법으로 표현할 수 있게 기능을 제공하고 있다.

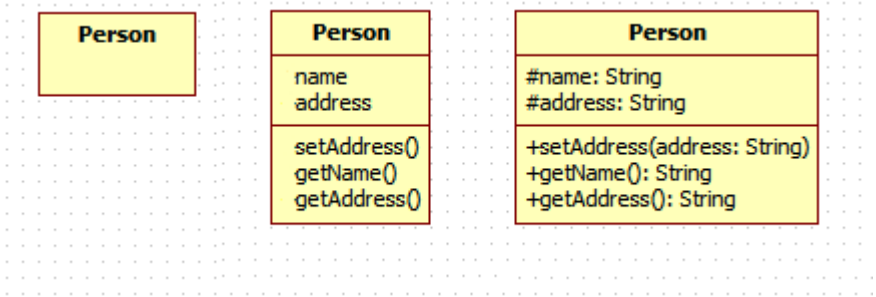
Concrete Class

- concrete 라는 것은 구체화 되어 있다는 의미로, 객체지향에 있어서 Instance로 실체화 되어 있는 클래스로 자바로 말하면 New 키워드로 생성할 수 있는 Class를 말한다.
  - Concrete Class
  - Abstract Class

Concrete Class (Cont.)

```java
public class Person{

    protected String name;
    protected String address;

    public Person(String name)
    {
        this.name=name;
    }

    public void setAddress(String address)
    {
        this.address =address;
    }

    public String getName(){
        return (name);
    }

    public String getAddress()
    {
        return (address);
    }
}
```





왼쪽에 있는 부분이 간략화된 형태의 concrete class이다. 가운데 있는 것이 속성과 메소드를 반영한 것이다. 보다 자세하게 작성한 경우가 오른쪽에 있는 경우다. 오른쪽에 있는 아이콘에서 메소드의 왼쪽에 있는 기호는 다음과 같은 의미를 나타낸다.

[+] : public

[#] : protected

[-] : private

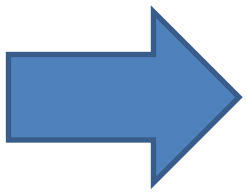
[~] : package

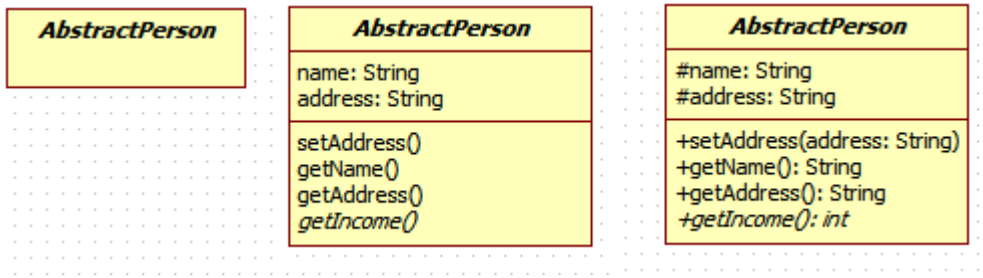
Abstract Class

- java의 abstract class는 아래와 같이 나타낼 수 있다.

  Concrete class 이름이 기울임 꼴 글꼴로 표시되어 있다. 추상화 메소드도 기움임 꼴 글꼴로 표시되어 있다.

```
public abstract class AbstractPerson{
    protected String name;
    protected String address;

    public AbstractPerson(String name)
    {
        this.name = name;
    }
    public void setAddress(String address)
    {
        this.address = address;
    }
    public String getName()
    {
        return (name);
    }

    public String GetAddress(){
        return (address);
    }

    public abstract int getSalary()
    {}

}
```



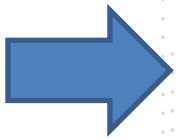


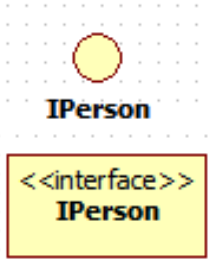
Interface

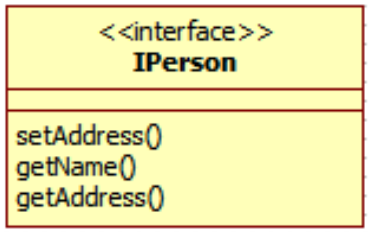
- java에는 interface라는 기능이 있다. interface는 메소드만을 정의한 클래스로 객체의 동작과 실제 구현을 분리하여 정의하는데 사용된다.

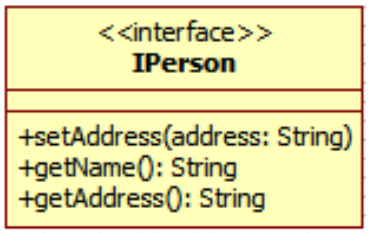
Interface (Cont.)

```
public interface IPerson
{
    void setAddress(String address);
    String getName();
    String getAddress();
    int getSalary();
}
```





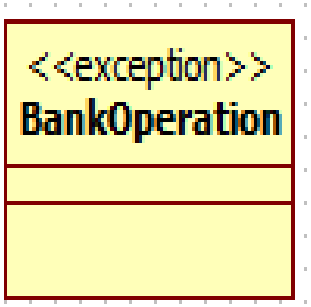




java interface는 UML의 interface로 모델화 된다. UML의 interface는 기본적으로 interface의 스테레오 형식의 클래스이다. 스테레오 형식이란 기존의 모델 엔트리로부터 새로운 모델 엔트리를 정의 하기 위한 구조이다. 스테레오 형식은 그 형식이름을 “<<” 와 “>>” 로 감싸 표시한다.

인터페이스는 concrete 클래스나 abstract 클래스와 동일한 아이콘을 사용하여 표현할 수 있지만, 간략화한 표현도 준비되어 있다. 간략화 한 표현은 위의 그림과 같이 원으로 표현한다.

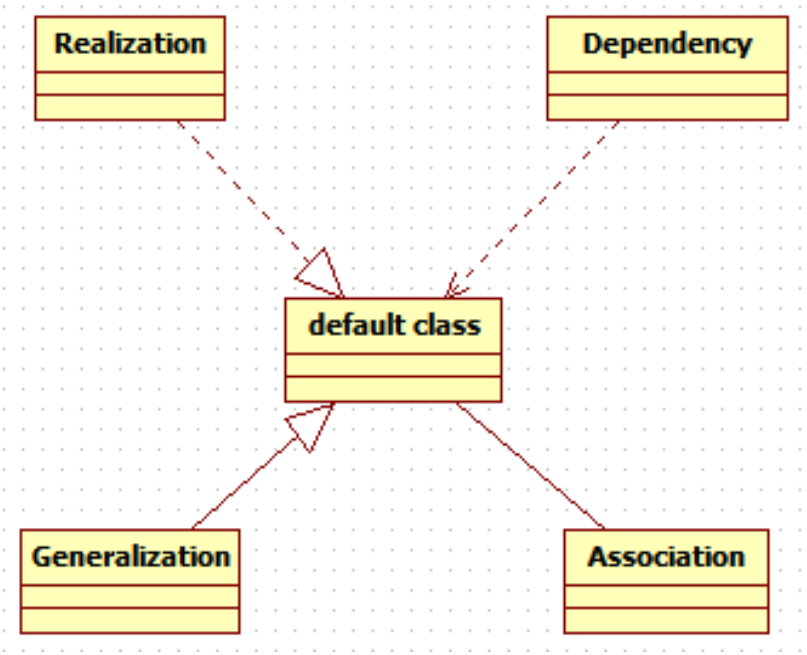
Exception



- java에서 에러 처리의 기술에는 예외가 사용된다. java 예외는 UML에서는 exception이라는 스테레오 타입에 의해 표현된다.

Relation

- UML 에서는 객체와 객체 간에 관계는 relation으로 표현한다.
- 클래스 다이어그램에서 사용하는 relation은 크게 4 종류가 있다.
  - Association
  - Generalization
  - Realization
  - Dependency



객체 간의 관계에 따른 UML 표기

Relation	UML	
Is - a	Generalization Realization	
has - a	Association	Aggregation Composition
etc	Association Dependency Realization	

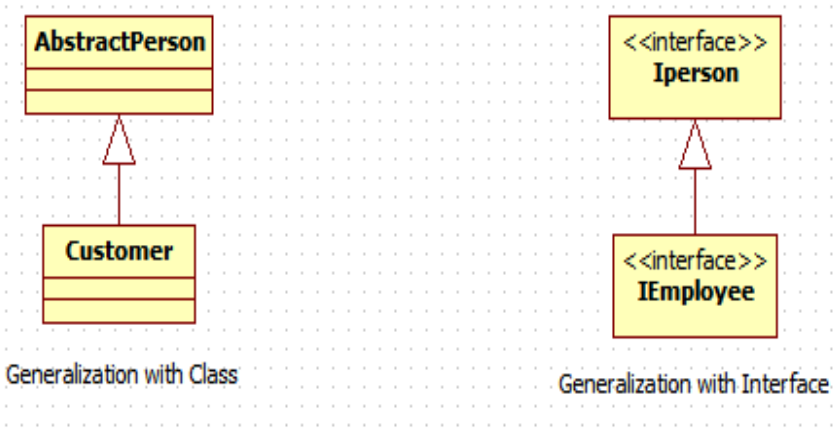
Inheritance

- inheritance를 구현하는 자바의 기능은 3개로 나누어 진다.
  - extends에 의한 클래스 상속
  - extends에 의한 인터페이스 상속
  - implements에 의한 인스턴스 구현

Generalization

- generalization의 객체 지향의 일반적인 용어는 inheritance이다. Java 에서는generalization으로 extends 키워드를 사용한다.

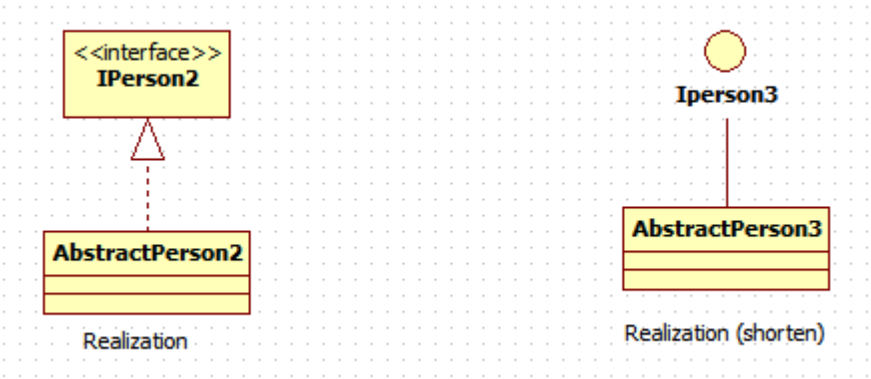
```
public class Customer extends AbstractPerson
    public Customer(String name){
        super (name);

    }

    public int getSalary(){
        return 0;
    }
}
```



Generalization

- 인터페이스를 클래스로 구현하는 것을 UML에서는 realization으로 모델화 한다. 객체지향의 일반적인 용어에서는 inheritance, java implements라는 키워드를 사용한다.

```
public abstract class AbstractPerson implements IPerson
{
    protected String name;
    protected String address;

    public AbstractPerson(String name){
        this.name=name;
    }

    public void setAddress(String address)
    {
        this.address =address;
    }
    public String getName()
    {
        return (name);
    }
    public String getAddress()
    {
        return (address);
    }
}
```

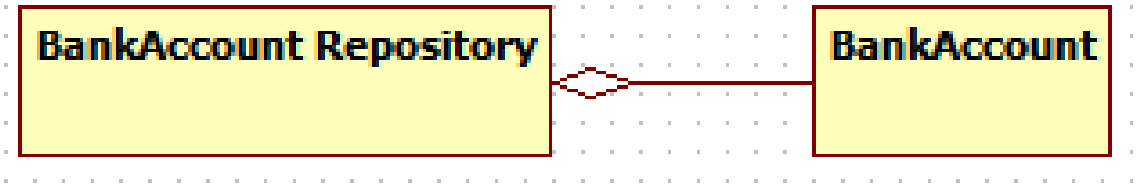


Association

- Relation 중에서 가장 빈번하게 보이게 되는 것이 Association이다.
- Association은 객체와 객체 간에 구조적, 정적인 관계가 있는 것을 나타낸다.
- Association은 다음의 3개의 종류로 나눌 수 있다.
  - Aggregation
  - Composition
  - 기타
- 보다 정확한 표현을 하려면 Association 중에 객체의 구성을 나타내는 연관 관계가 Aggregation이고 Aggregation중에서 구성되는 개체와 구성하는 객체의 관계가 아주 강한 것이 Composition이다.

Aggregation

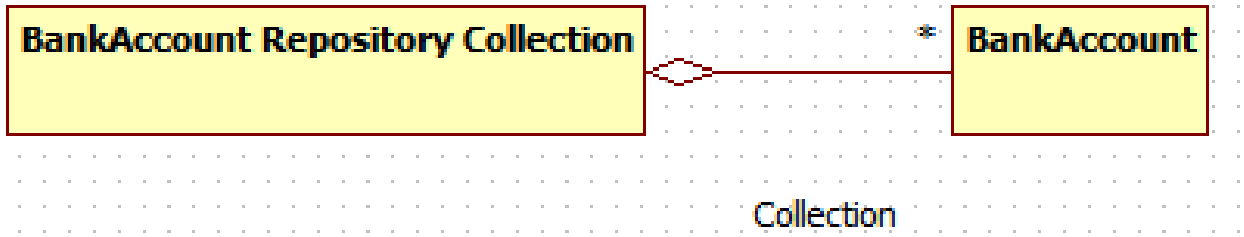
- Aggregation 은 객체 구성을 표현하기 위한 Association이다.
- Aggregation 구현의 기본은 객체 변수이다.
- BankAccountRepository가 여러 개의 BankAccount를 관리할 수 있게 배열을 사용하고 있다.



Aggregation (Collection)

- 객체 지향의 강력함의 하나는 객체의 집합의 처리를 할 수 있다는 점이다. 이런 집합 처리를 위해 java에 준비되어 있는 기능은 Java2에서 제공되는 JCF (Java Collection Framework)라는 컨테이너 클래스이다.

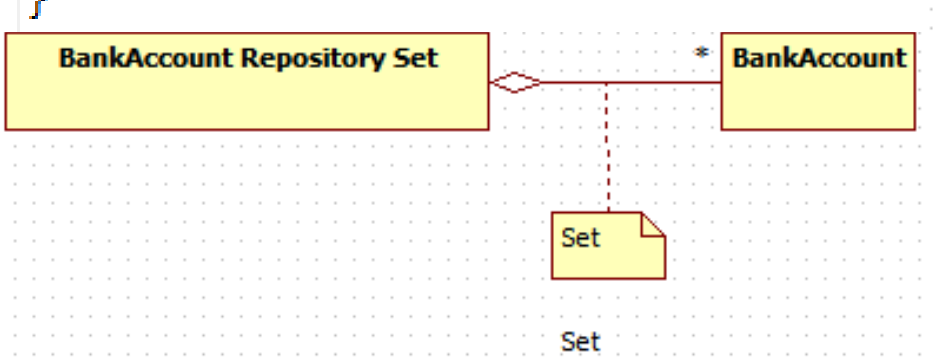
```
import java.util.*;

public class BankAccountRepositoryCollection{

    protected Collection collection = new ArrayList();

}
```



Aggregation (Set)

- Set은 집합을 나타내는 컬렉션이다. 수학적인 집합은 동일한 요소를 중복하여 갖지 않기 때문에, 이것과 같은 것을 이 Set라는 컬렉션으로 구현할 수 있다. Set은 형을 나타내는 인터페이스이기 때문에, 실체를 갖는 클래스로 java.util.HashSet 를 사용한다.

```
import java.util.*;

public class BankAccountRepositorySet{

    protected Set set = new HashSet();

}
```

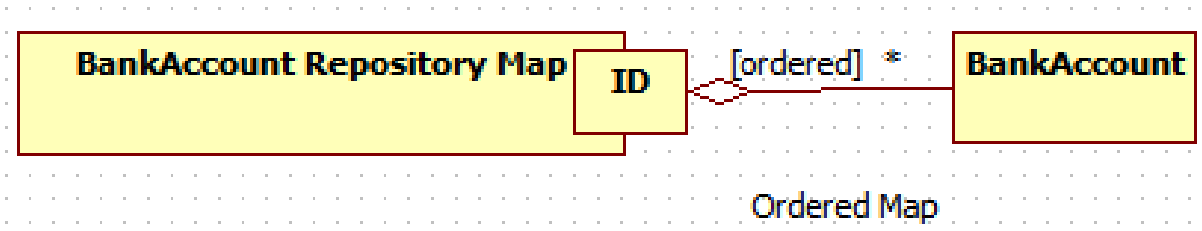


Aggregation (Map-ordered)

- 맵은 사전 검색 기능을 갖는 컬렉션이다. 컬렉션에 보관되어 있는 객체가 분류되어 있는지를 보증하는 기능 동시에 제공하려는 경우가 있다. 이러한 컨테이너를 표현하도록 자바에서는 SortedMap이 준비되어 있다.

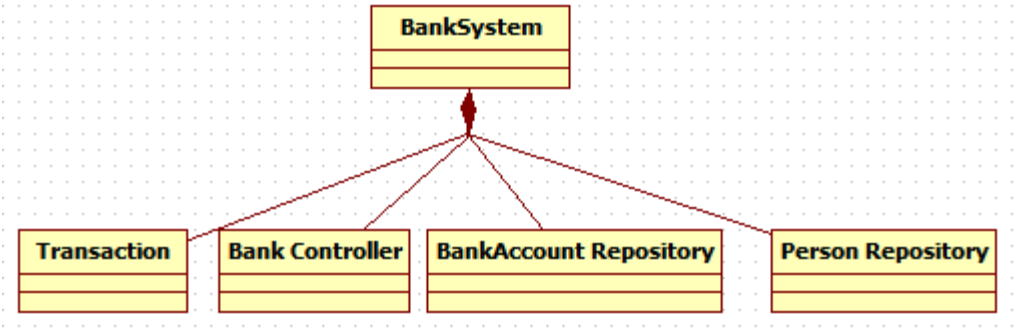
```
import java.util.*;

public class BankAccountRepositoryStoredMap{

    protected SortedMap map = new TreeMap();
}
```

- 이와 같이 java.util.Collection 사용하는 경우에는 단순한 Aggregation 으로 모델화된다.



Composition

- Aggregation의 하나의 종류로써 기존의 Aggregation보다 더 강력한 결합을 나타낸다. 학교와 학생이나 회사와 종업원의 관계를 구현하는 각 객체 간의 관계는 시간이 흐르면 당연히 변경될 수 있다. 학교에 입학하거나 회사 입사로 인원은 증가되고, 인원은 증가되고, 학교 졸업이나 퇴학, 퇴사로 인원은 줄어든다.
- 이에 반해 2개의 결합을 고정하여 객체의 설정이 보존되게 하는 관계가 있다. 자동차와 엔진 이나 사람과 심장의 관계에서 자동차에서 엔진을 제거하면 자동차가 아닌 고철에 불과하고 사람으로부터 심장을 제거하면 그 사람은 죽게 된다.
- 이런 강한 관계를 Composition이라고 한다.



Dependency

- Dependency는 클래스 다이어그램에서 사용하는 경우에 Use의 관계를 나타낸다.

- Use의 관계는 Association과 같은 정적인 구조 관계가 아니라 동적인 일시적인 관계이다. 메소드의 인수로 전달되는 객체와 메소드를 갖고 있는 개체의 관계가 이 Use의 관계이고, Dependency로 표현된다.

Dependency

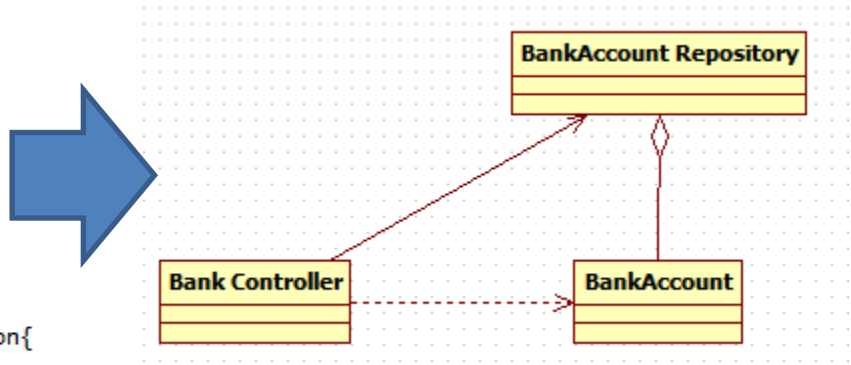
```
public class BankController{

    protected BankAccountRepository repository;
    public void deposit(String id, int amount) throws Bank OperationException{

        BankAccount account = repository.getAccount(id);

        if(account ==null)
        {
            throw (new BankOperation Exception());
        }

        account.amount += amount ;
        repository.putAccount(account);

    }

    public void withdraw(String id, int amount) throw BankOperationException{
        BankAccount account = repository.getAccount(id);
        if(account == null)
        {
            throw (new BankOperationException());
        }
        if(account.amount < amount){
            throw (new BankOperationException());
            account.amount -= amount ;
            repository.putAccount (account);
        }
    }
}
```

이 Controller 클래스를 중심으로 하는 클래스 다이어그램은 Dependency 형으로 표현된다. BankController로 부터 BankAccountRepository로의 관계는 객체 변수로 구현된 정적구조적인 관계인 Association으로 표현되지만, 문제는 BankController로 부터 ankAccount에의 관계이다. 이 관계는 deposit 메소드나 withdraw 메소드 중에서 동적으로 나타나 사라지는 관계이다.

Dependency

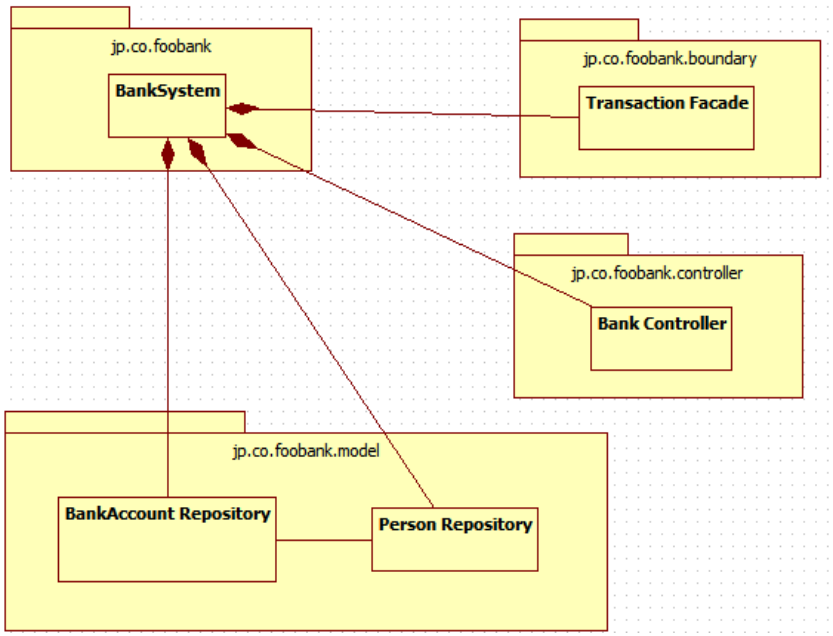
- 결국 BankController가 BankAccount를 일시적으로 사용하는 관계지만, 이 관계를 표현하려면 Use의 의미를 갖는 Dependency를 이용한다.

```
public class BankController{

    protected BankAccountRepository repository;
    public void deposit(String id, int amount) throws Bank OperationException{

        BankAccount account = repository.getAccount(id);

        if(account ==null)
        {
            throw (new BankOperation Exception());
        }

        account.amount += amount ;
        repository.putAccount(account);

    }

    public void withdraw(String id, int amount) throw BankOperationException{
        BankAccount account = repository.getAccount(id);
        if(account == null)
        {
            throw (new BankOperationException());
        }
        if(account.amount < amount){
            throw (new BankOperationException());
            account.amount -= amount ;
            repository.putAccount (account);
        }
    }
}
```



Package

- java에서 패키지라는 기능을 사용하여 여러개의 클래스를 그룹화 하여 관리할 수 있다. UML에서도 동일한 기능으로서 패키지가 준비되어 있다.
- UML에서의 패키지는 자바의 패키지 보다 넓은 의미를 갖지만, 클래스 다이어그램에서 사용하는 경우에는 자바의의 패키지는 UML에서도 패키지로 표현된다.
- 그림은 Composition으로 BankSystem을 패키지로 분해하여 본 것이다.



UML Tools

- **UML Tools**
- **StarUML**

UML Tools

Name	Creator	Platform / OS	First public release	Latest stable release	Open source	Software license	Programming language used
AgileJ StructureViews	AgileJ	Cross-platform (Java)			No	Commercial	Java
Altova UModel	Altova	Microsoft Windows	2005-05		No	Commercial	Java, C#, Visual Basic
ArgoUML	Tigris.org	Cross-platform (Java)	1998-04	2011-12-15[1]	Yes	EPL	Java
astah*	Change Vision, Inc.	Multi-platform		2011-09-19	No	Commercial, Free trial, Free edition (Community version)	Java, C++, C#
ATL	Obeo, INRIA Free software community	Cross-platform (Java)		2010-06-23	Yes	EPL	Java
Borland Together	Borland	Cross-platform (Java)		2008	No	Commercial	
BOUML	Bruno Pagès	Cross-platform		2011-10	No	Commercial starting from v5.0[2], GPL before v5.0	C++/Qt
Dia	Alexander Larsson/GNOME Office	Cross-platform (GTK+)	2004?	2011-12-18	Yes	GPL	C
Eclipse UML2 Tools[3]	Eclipse Foundation	Cross-platform (Java)	Planning	Planned	Yes	EPL?	Java
Enterprise Architect	Sparx Systems	Windows (Supports Linux & Mac installation)	2000	2011-12-01	No	Commercial	C++
MagicDraw UML	No Magic	Cross-platform (Java)	1998	2010-11-29	No	Commercial	Java
Modelio	Modeliosoft	Windows, Linux	2009	2012-01-25	Yes	GPL V3, Apache 2.0	Java, C++
Objecteering	Objecteering Software	Windows, Linux	1992		No	Commercial	
objectiF	microTOOL	Microsoft Windows	1992	2010-09-21	No	Commercial	Java, C#, C++
Open ModelSphere	Grandite	Cross-platform (Java)	2002-02	2009-11-04	Yes	GPL	Java
Papyrus	Commissariat à l'Énergie Atomique, Atos Origin	Windows, Linux		2010-12-15	Yes	EPL	Java
Poseidon for UML	Gentleware	Cross-platform (Java)		2009	No	Commercial	Java
PowerDesigner	Sybase	Windows	1989	2010	No	Commercial	
RISE	RISE to Bloome Software	Windows (.NET)	2008	2010-09-03	No	Freeware	C#
Software Ideas Modeler	Dusan Rodina	Windows (.NET), Linux (Mono)	2009-08-27	2012-02-06	No	Commercial, Freeware for non-commercial use	C#
StarUML	Plastic Software	Windows	2005-11-01	2006-08-07	Yes	GPL, modified	Delphi
Umbrello UML Modeller	Umbrello Team	Unix-like, Windows	2006-09-09	2009-08-04	Yes	GPL	C++, KDE
Visual Paradigm for UML	Visual Paradigm Int'l Ltd.	Cross-platform (Java)	2002-06-20	2011-09-19	No	Commercial, Free Community Edition	Java

UML Tools - StarUML

- Name : StarUML
- Creator : Plastic Software
- Platform / OS : Windows
- First public release : 05. 11. 01.
- Latest stable release : 06. 08. 07
- Open Source, UML 2, MDA, Templates : Yes
- Language generated : Java, C#, C++
- Reverse engineered languages : Java Profile,C++ Profile,
  C# Profile Code Generator and Reverse Engineer

UML Tools - StarUML (Cont.)

- StarUML은 빠르고, 유연하고, 확장가능하며, 풍부한 기능에 Win32 플랫폼에서 UML 1/MDA 플랫폼(툴)을 개발하기 위한 오픈 소스 프로젝트입니다.
- StarUML 프로젝트의 목적은 IBM Rational Rose, Together와 같은 상업적 도구를 비싼 돈을 들여 사용하지 않더라도 그에 준하는 기능을 갖춘 오픈 소스 소프트웨어 모델링 도구 및 플랫폼을 개발하는 것입니다.
- StarUML은 국내 소프트웨어 업체인 플라스틱 소프트웨어에서 개발된 Plastic에서 유래되었습니다. 즉 StarUML은 다른 tool들과 달리 그 매뉴얼을 비롯하여 구성이 모두 한글로 이루어져 있어 그 활용도에 있어서 이번 프로젝트에 매우 적합하다고 생각됩니다.

Reference

- Wikipedia (http://en.wikipedia.org/wiki/List_of_UML_tools)
- The UML User Guide (Grady Booch, James Rumbaugh, Ivar Jacobson) [Addision Wesley]
- UML Distilled A Brief Guide To The Standard Object Modeling Language(Chapter 4. Sequence Diagram)
- UML 모델링의 본질 (아옥공신 / 성안당 / 2005)
- Java 객체지향 언어로 배우는 디자인 패턴 (신재호 / 정보문화사)
- 자바 개발자를 위한 UML contact J (송호중 / 대림 / 2001)