Introduction to OOAD using UML tools

Software Engineering

Team Report #1

Team 8

200611458 김영승

200611478 성두훈

200611494 원스타

200611518 조민경

개요

OOAD란 무엇일까? 그리고 또 UML이란 것은 무엇일까? 소프트웨어 공학을 하는 사람이라면 한 번 쯤은 접해 볼 수 밖에 없는 단어들이고, 또 어떤 사람은 앞으로 사용하게 될 것들이다. 간단하게 말하자면, UML은 OOAD를 좀 더 효과적으로 수행할 수 있도록 그래픽으로 시각화해 주는 CASE의 하나라고 할 수 있다. 이 글에서는 먼저 간단하게 OO와 OOAD, 그리고 UML에 대하여 알아보고, UML의 구성요소, 그리고 UML의 가장 핵심적인 아니 UML의 전부라고 할 수 있는 여러 다이어그램의 실제 사용에 대하여 알아본다.

OO(Object-Oriented)

객체 지향의 목적은 실제 세계의 일부 혹은 전체를 본뜬 것이라는 점이다. 속성과 행동들이 좀더 많이 반영된 클래스 일수록 실제 세계에 더 가까운 모델이 만들어진다. 간단한 예로, 우리가 사용하는 컴퓨터의 경우, 내부 CPU, M/B, HDD등의 속성을 추가하면 좀 더 정확한 모델이 될 수 있을 것이다.

OO을 이루는 몇 가지 개념들

객체 지향 개념은 단지 속성과 행동만 가지고는 설명할 수 없는 객체의 성질을 가진다. 가장 흔히 사용되는 원리로 추상화(Abstraction), 상속(Inheritance), 다형성(Polymorphism), 캡슐화(Encapsulation)가 있으며 이에 못지 않게 다루는 것으로 메시지 전송(Message Sending), 연관(Association), 집합 연관(Aggregation Association)이 있다.

- 추상화 - 객체를 모델링 할 때 필요로 하는 만큼의 속성과 오퍼레이션을 추출해 내는 것이다.
- 상속 – 하나의 객체는 클래스의 인스턴스로서 그 클래스의 모든 특성을 이어받는다. 이것을 상속이라고 한다. 상속은 슈퍼클래스-서브클래스의 관계를 가지게 된다. 즉, 슈퍼클래스로부터 상속을 받는 서브클래스가 생성되는 것이다. 일반적으로『서브클래스는 슈퍼클래스이다』라는 식의 관계가 성립한다.
- 다형성 – 다른 클래스인데 같은 이름의 오퍼레이션이 존재하는 것으로 각각의 클래스는 자신의 오퍼레이션의 동작 형태를 알고 있다.
- 캡슐화 – 객체는 자신의 오퍼레이션을 수행하고 결과를 내놓지만, 그 오퍼레이션의 동작 원리는 가려져 있다. 즉, 각각의 오퍼레이션의 수행의 결과는 알 수 있지만, 어떠한 과정에 의해서 이루어지는 알 수 없게 되어 있는 것이다. 이를 정보 은닉(Information hiding)이라고도 부른다.
- 메시지 전송 – 한 객체가 다른 객체에게 메시지를 보내 어떤 오퍼레이션을 수행하도록 하면, 메시지를 받은 객체는 지시 받은 대로 해당 오퍼레이션을 수행하는 것이다.
- 연관(Association) – 한 객체가 다른 객체와 관계를 맺는다. 이 관계는 한 개 이상일 수도 있으며, 다중성과도 관계가 있다.
- 집합연관 – 하나의 객체가 다른 여러 객체들의 조합에 의해 만들어 진 것으로, 예를 들어, 컴퓨터는 CPU, M/B, HDD, 키보드, 마우스, 모니터 등 다양한 객체의 조합에 의해서 만들어진 것을 생각할 수 있다.

OOAD(Object-Oriented Analysis Design)

OOAD란?

객체 지향 분석 설계로 OOAD는 객체 그룹간의 상호작용으로서 시스템 모델을 접근하는 소프트웨어 공학이다. 각각의 객체는 클래스의 특성, 상태, 그리고 동작들에 대해 나타내고, 다양한 모델은 정적 구조와 동적 행위 그리고 이러한 객체의 집합에 대한 런타임 실행을 볼 수 있게 만들 수 있다. UML같은 모델을 표현하는 많은 표기법이 존재하고 있다.
객체 지향 분석(OOA)은 시스템의 기능적인 요구사항 분석을 위한 객체 모델링 기법을 제공하고, 객체 지향 설계(OOD)는 명세서 구현에 의한 분석 모델을 잘 만들 수 있도록 한다. OOA는 시스템이 무엇을 할지에 중점을 두고, OOD는 시스템이 어떻게 할지에 중점을 둔다.

Object-Oriented systems

객체 지향 시스템은 객체의 구성이고, 객체들의 조합으로 인해 생기는 시스템의 작용 결과이다. 객체들간의 동작은 서로 메시지를 송신하는 것에 의해 결정된다. 메시지를 송신하는 것은 함수를 호출하는 것과는 다르다. 함수를 호출하는 것은 해당 함수를 실행한다는 단순한 의미이지만, 메시지를 송신하는 것은 객체가 다른 객체로부터 수신한 메시지에 의해서 실행해야 하는 것을 결정하는 것을 의미한다. 동일한 메시지가 여러 개의 다른 함수들에서 실행될 수 있고, 이 경우 어느 함수가 실행의 주체가 되느냐에 대해서는, 해당 객체의 상태에 따라 결정된다. "메시지 송신"의 실행은 모델화의 대상이 되는 시스템의 아키텍쳐에 의해 다양해 지고, 상호작용하는 객체 집합이 동일 컴퓨터 내에 배치되어 있는지, 아니면 여러 개의 컴퓨터에 분산되어 배치되어 있는지에 의해서 다르다.

Object-Oriented analysis

객체 지향 분석은 시스템화의 대상이 되는 problem domain을 대상으로 하여 분석하고, 이 domain에 존재하는 다양한 정보의 개념 모델을 만드는 것을 목표로 하는 과정이다. 분석 모델에서는 구현 단계에서 생길 가능성이 있는 다양한 종류의 제약들에 대해서는 전혀 고려하지 않는다. 즉, 분석 모델에서는, 시스템이 어떻게 구축되는가 하는 것은 전혀 고려하지 않는다는 것이다. 구현 단계에서의 제약 사항들에 관한 것은 다음 과정인 객체 지향 설계 단계에서 다루기 때문이다.
객체 지향 분석 작업의 근원이 되는 것은, 기술된 형식의 요구 사양, 앞으로의 발전에 관한 기업 전략을 적은 서류, 이해 관계자나 그 외의 관계자와의 인터뷰 등이 있다. 시스템은 다수의 domain에 의해 분할된다. 이 분할은 시스템이 여러 비즈니스와 기술, 그리고 다양한 관심 분야에 관여하고 있을 때 일어난다. 이렇게 분할된 시스템은, 각각 개별적으로 분석된다.
객체 지향 분석의 결과물은 개발하는 시스템이 기능적으로 무엇을 하는 것이 필요한가라는 것을 개념적인 모델의 형태로 기술한 다이어그램이나 문서이다. 객체 지향 분석의 결과물은 Use Case 및 UML 다이어그램, 그리고 여러 개의 interaction 다이어그램으로 나타낸다. 이 결과물은, 시스템의 유저 인터페이스를 모의 작성하여 기술한 자료를 포함하기도 한다. 객체 지향 분석의 목적은 고객의 요구사항들을 만족하는 동작을 수행할 수 있는 컴퓨터 소프트웨어를 묘사한 모델을 개발하는 것이다.

Object-Oriented design

객체 지향 설계는 객체 지향 분석으로 얻은 분석 모델을 통하여, 다양한 종류의 제약 조건을 고려한 실제 구현 가능한 모델로 변환하는 과정이다. 여기서 말한 다양한 종류의 제약에는 선택한 아키텍쳐에 인한 제약, 비기능적 제약(기술적, 환경적 제약 등)을 포함한다. 구체적으로 트랜잭션 throughput, 응답 시간, 실행시의 플랫폼, 개발 환경, 프로그래밍 언어 등이 있다.
객체 지향 설계에서는 분석 모델로 명세화된 많은 개념을 클래스와 인터페이스 대응 시킨다. 객체 지향 설계의 결과물은, 문제 domain에 도달해 시스템이 어떻게 구축될까를 상세하게 적은 모델 다이어그램과 문서이다.

UML

UML(Unified Modeling Language) 이란?

Requirement, design, implementation 등의 시스템 개발 과정에서, 개발자간의 의사소통을 원활하게 이루어지게 하기 위하여 표준화한 모델링 언어이다.
UML은 글로써 이루어진 문서가 아닌 그래픽 표기법(graphical notation)의 집합으로, 단일 메타모델(meta-model)을 기초로 하고 있으며 소프트웨어 시스템, 특히 객체지향(object-oriented) 방식을 사용하여 구축되는 소프트웨어 시스템을 표현하고 설계하는 것을 도와준다. 이 말은 UML은 그래픽적인 정보를 통하여 사용자들에게 정보를 전달한다는 것과 화면에 정보를 표시하기 위해 메타모델을 사용한다는 것이다. 또한 객체 지향 방식을 사용하는 소프트웨어 시스템에서 UML을 사용하여 설계하면 좋다는 것을 알 수 있다.

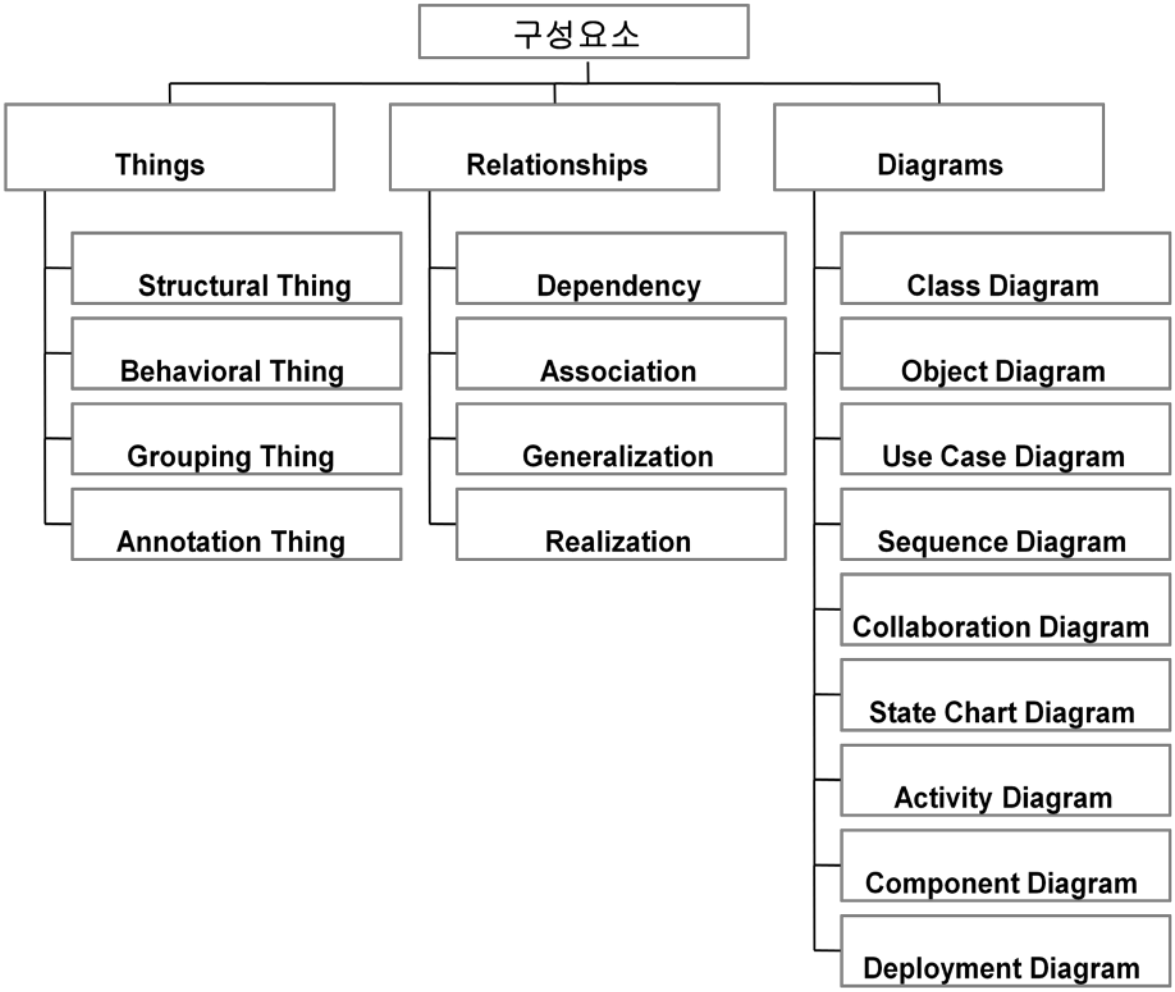
UML의 탄생 배경

UML은 Grady Booch, James Rumbaugh, Ivar Jacobson의 머리에서 태어났다. Tree Amigo라고도 불리는 이 세 명은 원래 80년대 전반과 90년대 초반까지 객체 지향 분석 설계분야에서 각자의 영역의 방법론을 연구해 왔었다. 그들이 발표한 방법론은 동일한 분야의 다른 경쟁자들보다 항상 탁월한 위치에 있었으며, 세 사람은 90년대 중반에 이르러 각자의 아이디어를 교환하기 시작하였고 결국 각자의 방법을 하나로 모아 합치기에 이른다. 이렇게 해서 탄생한 것이 UML이다.
UML의 드래프트 버전은 소프트웨어 업계를 뒤흔들기 시작하였고, 그 결과로 돌아온 피드백은 바로 변경점에 반영되었다. UML을 지지하는 회사가 늘어남에 따라 그 결과 UML 컴소시엄이 발족하게 되었다. UML 컴소시엄의 멤버로는 DEC, HP, Microsoft, Oracle, Rational 등이 있었다. 1997년 UML 컨소시엄은 UML 버전 1.0을 만들어 내고, 오브젝트 매니지먼트 그룹(OMG)이 표준 모델링 언어의 제안서를 제출하라는 요구에 따라 이것을 제출하였다.
그 후, UML 컴소시엄은 계속 발전하였으며, OMG에 다시 상정된 UML 1.1은 1997년 말에 표준 모델링 언어로 채택되었다. OMG는 UML의 관리기법을 받아들여 1998년에 새로운 수정안을 발표하였다. UML 1.1 이후 많은 버전의 UML 버전이 소개 되었지만, 대부분의 것들은 UML 1.1과 크게 달라진 것이 아닌 기존의 것의 버그나 문제점을 수정한 것이었다. 하지만 UML 2.0은 기존의 것에 비해 크게 발전했다. 그리고 이 UML 2.0 현재의 OMG 표준이다.
그 후에도 꾸준히 UML은 발전하고 있으며, 표준안으로 발표된 것은 아니지만, 2007년에 UML 2.1과 UML 2,1,2를 발표하였으며, 2009년 2월에는 UML 2.2를 2010년 6월에는 UML 2.3 버전이

출시 되었다.

UML의 구성

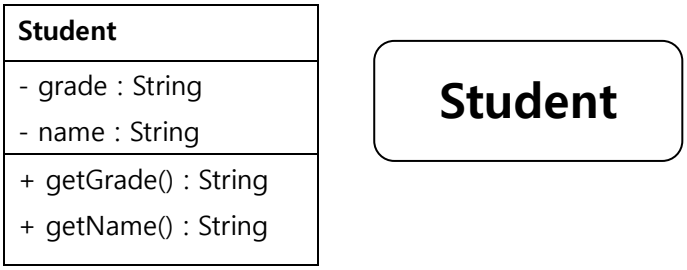
UML은 크게 Things, Relationships, Diagrams으로 구성되어 있다.



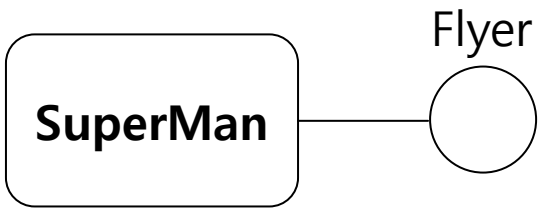
1. Things는 추상적인 개념으로 UML을 이용한 모델링의 기본요소이다.

1.1 Structure Thing

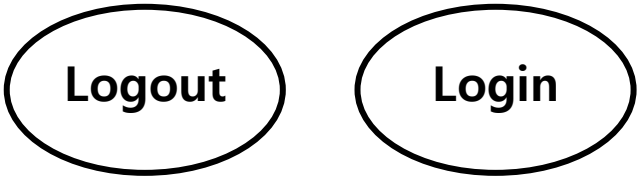
- UML 모델의 명사형으로 모델의 정적인 부분이며 개념적이거나 물리적인 요소를 표현한다.
- Class : 속성(Attribute)과 동작(Operation)으로 구성된 객체를 표현한 사각형의 박스이다.



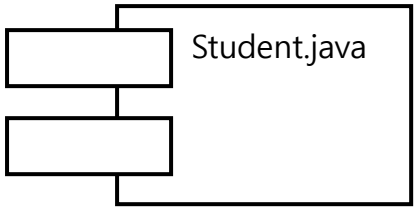
- Interface : 클래스 또는 컴포넌트의 동작을 명세화한 operation들의 집합이다.



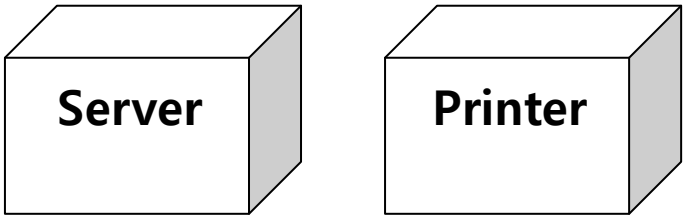
- Use Case : 시스템이 수행해야 하는 기능을 기술한 것으로 타원형태로 표현한다.



- Component : 객체지향에서 모듈화된 자원이다.



- Node : 실행 시에 존재하는 물리적인 요소로서, 주로 컴퓨터 자원을 표현한다.



1.2 Behavioral Thing

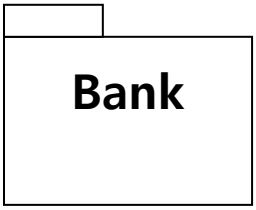
- UML 모델에서의 동사형이다.
- 모델의 동적인 부분이며 시간과 공간에 따른 행위 요소 표현한다.
- Interaction : 행위를 의미하며 특정문맥에 속한 객체들간의 Message들로 구성된다.

- State Machine : 객체의 시간에 따른 변화하는 상태를 표현한다.

1.3 Grouping Thing

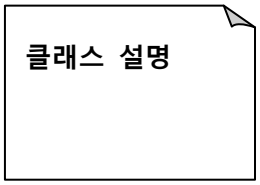
모델을 그룹화하여 하나의 요소로 표현할 수 있도록 해준다.

- Package : 요소들을 그룹으로 묶는 방법이다.
- Framework, 서브시스템 표현 시 사용한다.



1.4 Annotation Thing

- UML 모델요소를 설명하고, 명확히 하는 표현 방법이다.
- Note : comment로서 모델 요소를 명확하게 표현 설명하기 위한 방법이다.
- 주로 제약조건이나 내용을 설명한다.



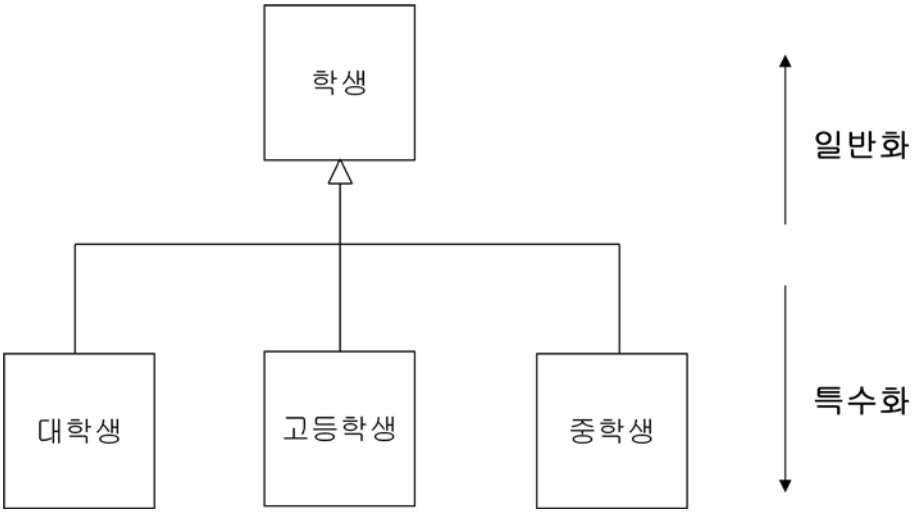
2. Ralationships은 구성요소들간의 의미 있는 연관성을 표현하고, 주로 클래스간의 관계를 표현한다.

2.1 Dependency Relationship

- 한 클래스가 다른 클래스를 사용하는 사용 관계로, 하나의 클래스의 변화가 다른 클래스에 영향을 주는 관계를 나타낸다.
- UML 표기법은 점선으로 된 화살표로 표현한다.

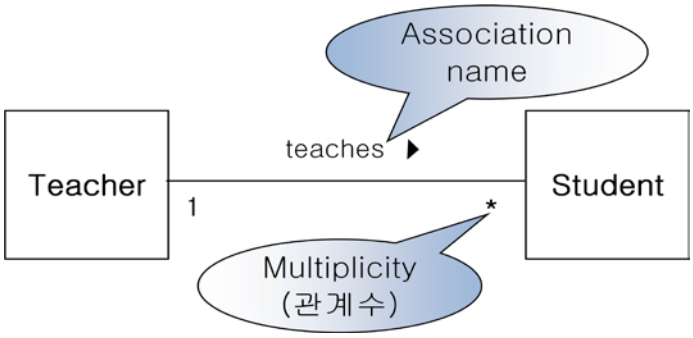
2.2 Generalization Relationship

- 일반화(Generalization), 특수화(Specialization) 관계가 있으며, 이것은 객체지향의 상속의 개념이라고 생각할 수 있다.



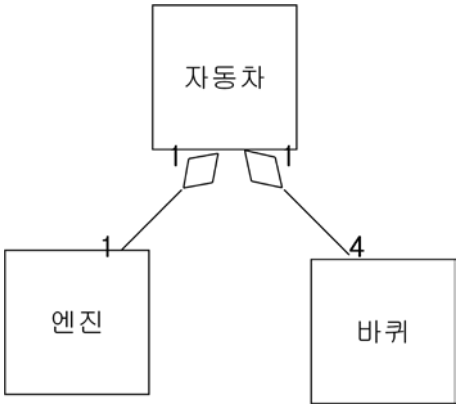
2.3 Association Relationship

- 클래스로부터 생성된 객체간의 일반적 연관 관계다.



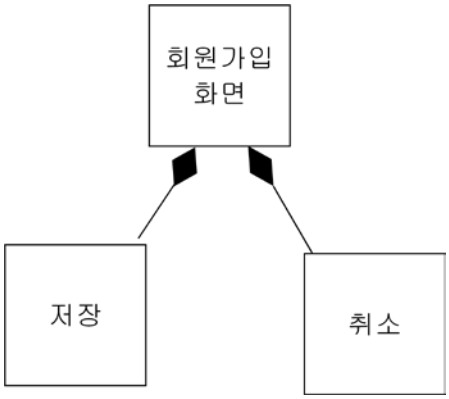
2.3.1 Aggregation Relationship

- 두 클래스간의 전체-부분 관계(Whole-part)로 이루어 진다.
- 각 클래스가 독립적인 생명주기를 갖는다.
- 하나의 클래스가 여러 개의 컴포넌트 클래스로 구성된다.



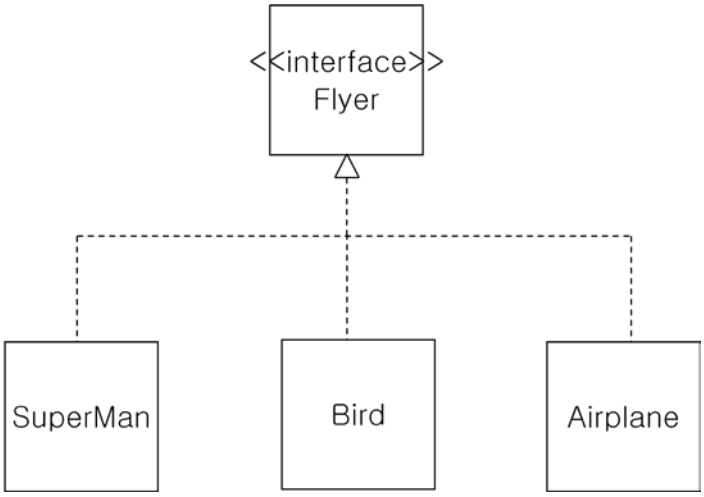
2.3.2 Composition Relationship

- 두 클래스간의 부분-전체 관계를 이룬다.
- 부분의 생명주기가 전체의 영향을 받는다
- 하나의 클래스가 여러 개의 컴포넌트 클래스로 구성된다.



2.4 Realization Relationship

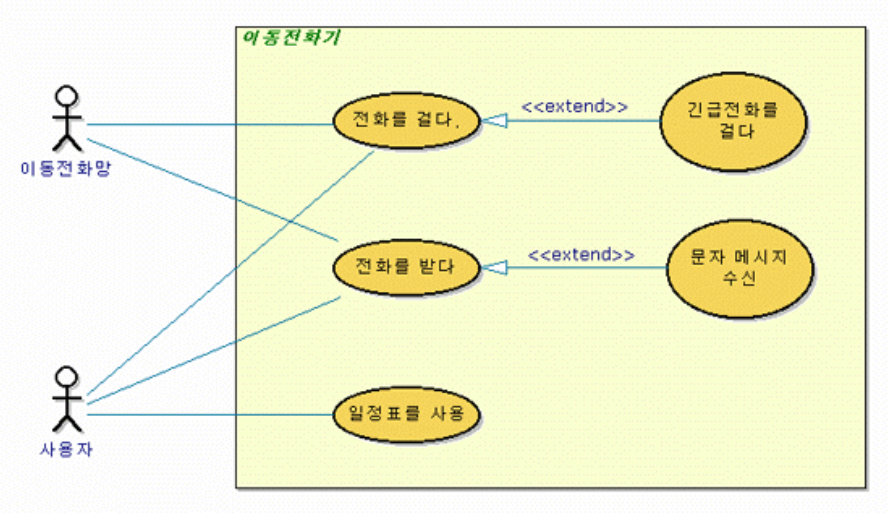
- 인터페이스와 실제 구현된 클래스간의 관계다.



3 Diagram : Things + Relationship

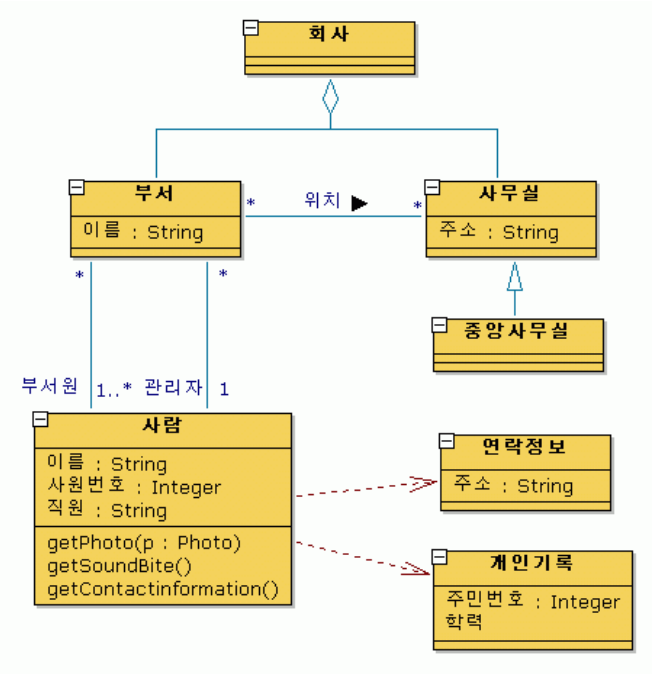
3.1 Use Case Diagram

- 사용자관점에서 논리적인 시스템의 기능을 정의한다.
- 시스템 구축 초기에 이 시스템이 어떤 일을 하는지에 대해 사용자 입장에서 이해할 수 있을 정도로 기술한다.
- 시스템 전체의 개발범위를 결정한다.



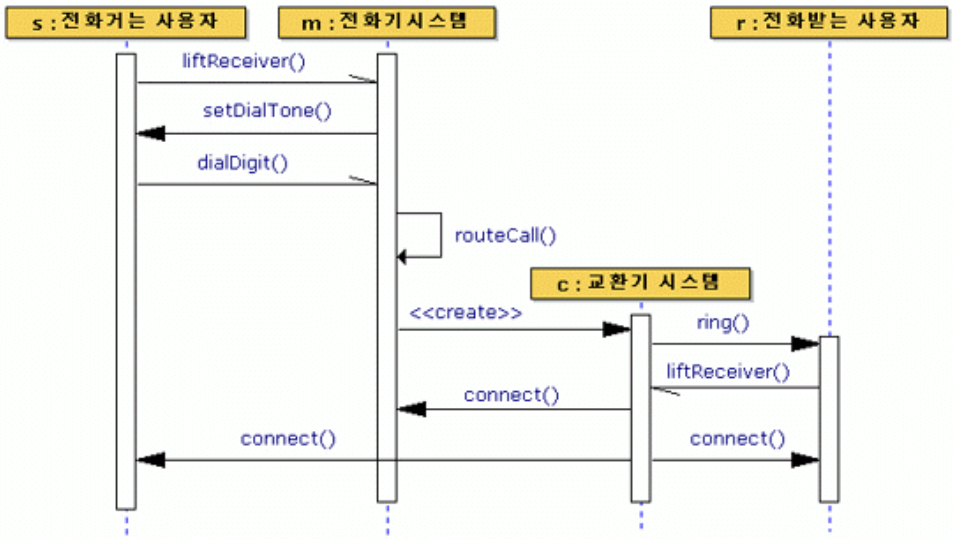
3.2 Class Diagram

- 시스템 내부의 클래스의 속성과 행위를 기입하고 각 클래스간의 관계를 정의한다.



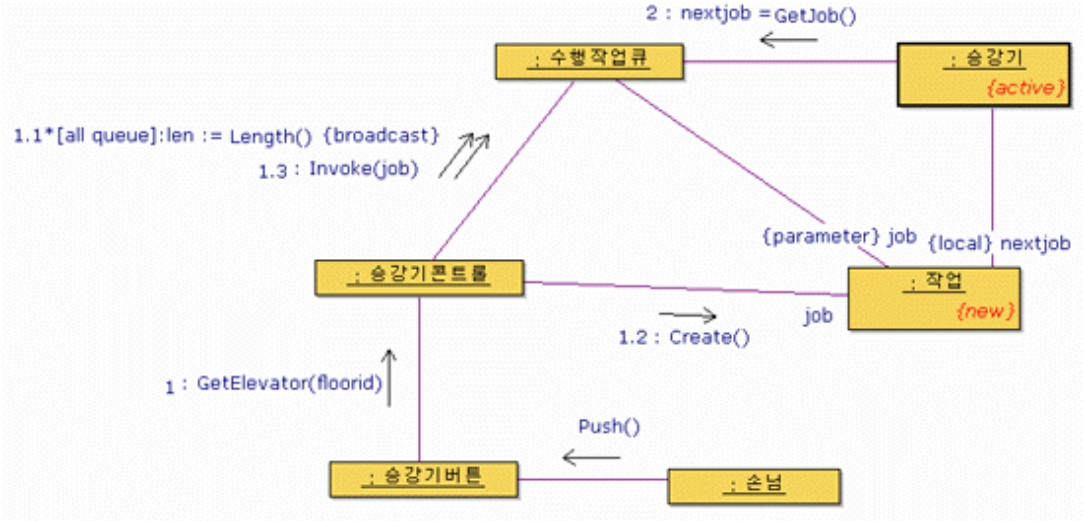
3.3 Sequence Diagram

- 시스템의 동적인 면을 나타내는 Diagram이다.
- 시스템 실행시 생성되고 소멸되는 객체와 객체들 사이에 주고 받는 메시지를 나타낸다.
- 횡축은 시간의 흐름을 나타내며 메시지의 순서에 중점을 둔다.



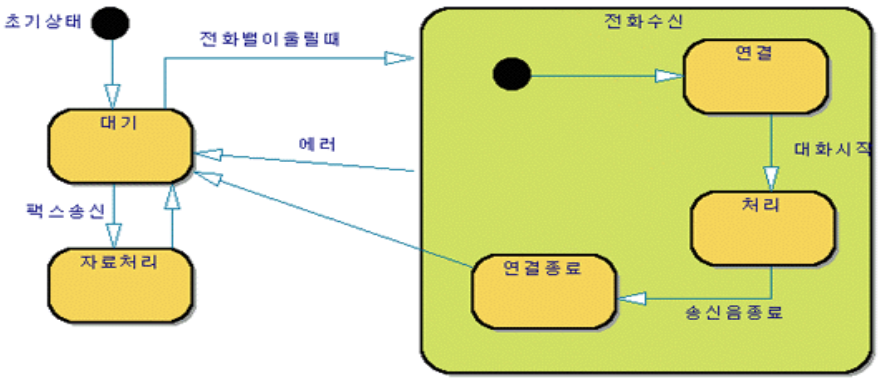
3.4 Collaboration Diagram

- 메시지의 흐름과 객체와 객체들 사이의 관계를 나타낸다



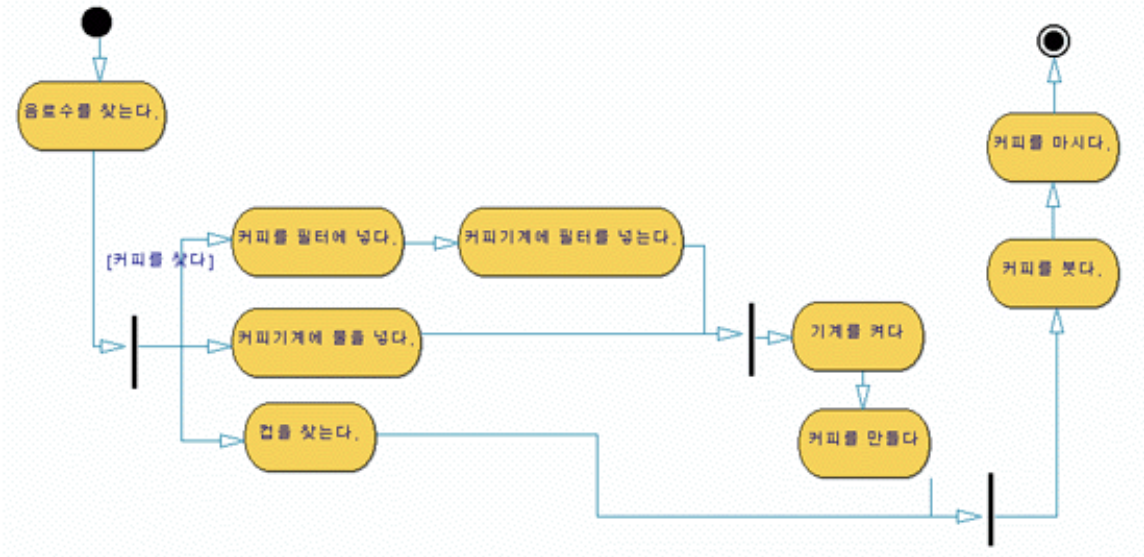
3.5 Statechart Diagram

- 한 객체의 상태 변화를 다이어그램으로 나타냄



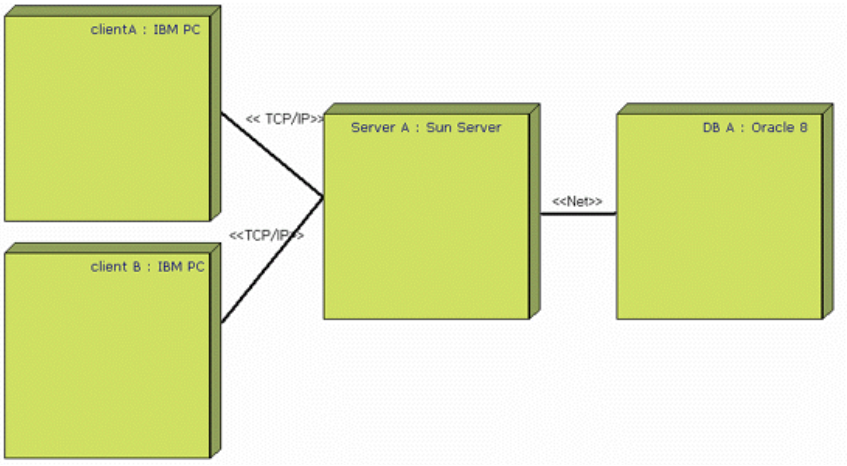
3.6 Activity Diagram

- Activity의 수행 순서와 처리흐름을 나타낸다.



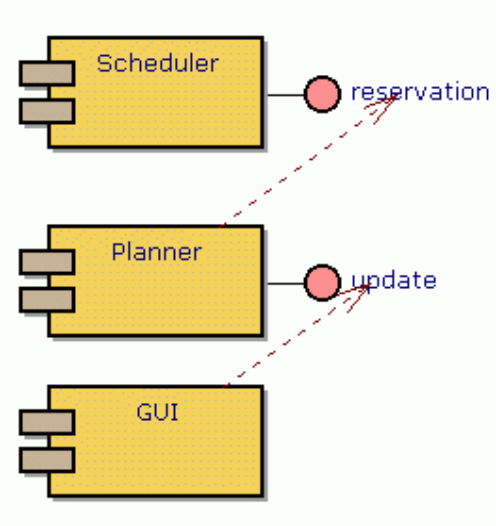
3.7 Deployment Diagram

- 실제 하드웨어적인 배치와 연결상태를 나타낸다.



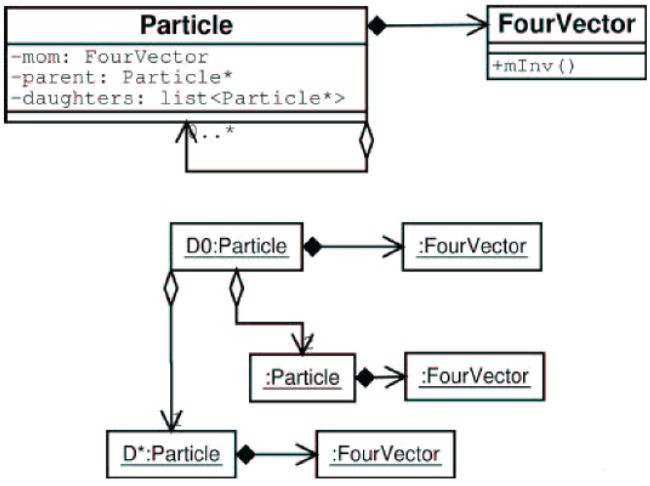
3.8 Component Diagram

- 소프트웨어의 물리적 단위의 구성과 연결상태를 나타냄



3.9 Object Diagram

- 클래스 Instance 간의 관계를 보여준다.



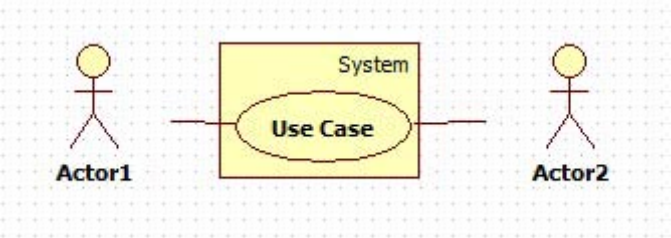
UML 다이어그램

UML의 여러 가지 그래픽 요소는 하나의 큰 그림, 즉 다이어그램을 그리는데 사용된다. UML은 언어이기 때문에, 이들 그래픽 요소들을 서로 맞추는 데에는 규칙이 필요하다. 이 규칙은 각각의 다이어그램을 통해 자세히 알아보고, 그 전에 다이어그램에 대해서 알아보자.

다이어그램의 목적은 시스템을 여러 가지 시각에서 볼 수 있는 뷰를 제공하는 것이며, 이러한 뷰의 집합을 모델이라고 한다.

1. Use Case 다이어그램

Use Case는 기능적인 요구사항을 수집하는데 있어서 매우 강력한 도구이며, Use Case 다이어그램은 더욱 강력하다. Use Case 다이어그램은 Use Case를 시각화 하기 때문에 분석가와 사용자, 분석가와 고객 간의 의사소통을 원활히 해준다. Use Case 다이어그램에서 Use Case나타내는 기호는 타원이며 Actor를 나태는 기호는 막대 인간이다. Actor와 Use Case 사이는 실선으로 연결



하며, Use Case는 대개 시스템 경계를 나타내는 사각형에 둘러싸여 있다.

Use Case 분석을 하면 얻을 수 잇는 이익 중 하나는 시스템과 외부 세계와의 경계를 효과적으로 보여줄 수 있다는 점이다. Actor는 대개 시스템의 외부에 잇는 반면, Use Case는 시스템의 내부에 있다. 이때 시스템과 외부 세계의 경계는 사각형을 그려서 구분하며, 이사각형은 시스템의 쓰임새를 둘러싸는 상태로 만든다. 이 형태를 Use Case 모델이라고 한다.

Use Case는 시나리오로 진행되며, 각 시나리오는 진행 단계로 이루어진다. 하지만, 진행 단계는 다이어그램에 나타나 있지 않다. 그렇다면 시나리오의 진행 단계는 어떻게 작성해야 할까?

Use Case 다이어그램은 대개 의뢰인과 개발팀이 참조하는 설계 문서의 한 부분으로 사용된다. 각 다이어 그램은 한 페이지당 하나씩 그려지는 것이 보통이며, 각 Use Case를 구성하는 시나리오는 각각의 페이지에다가 텍트로 적어두면 된다. 시나리오는 다음과 같이 작성한다.

· Use Case를 시작하는 Actor

· Use Case가 시작하는데 필요한 선행 조건

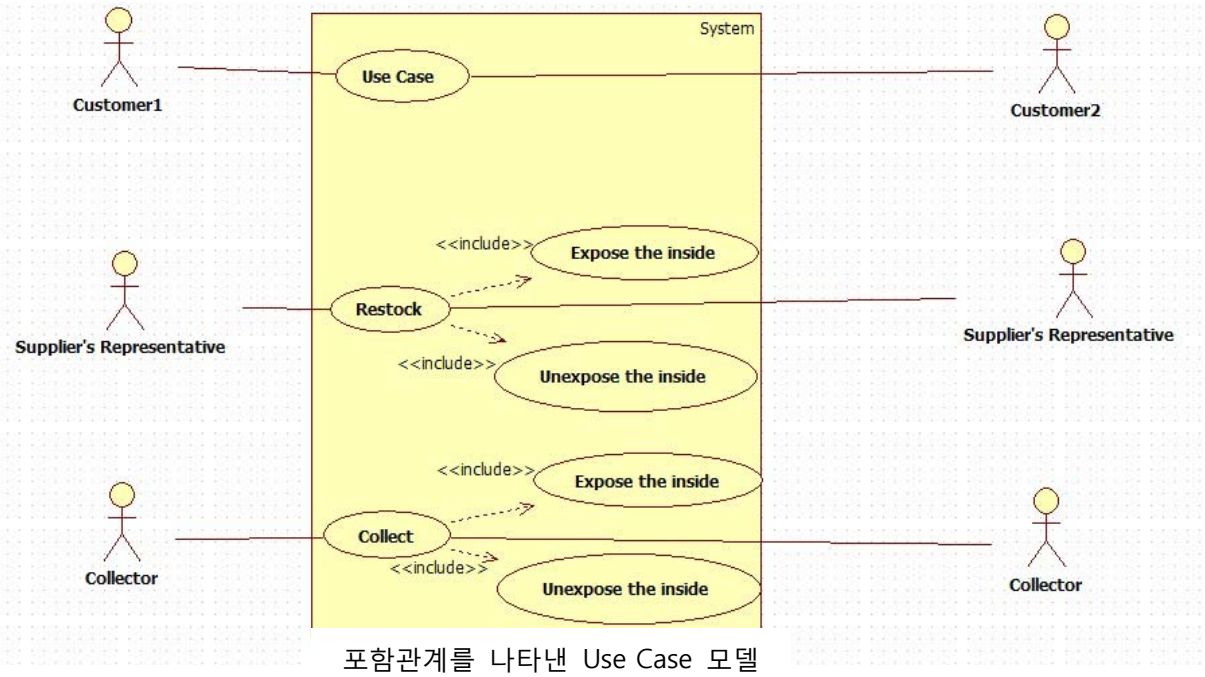
· 시나리오의 진행 단계

· Use Case가 끝나는데 필요한 종료 조건

· Use Case의 결과를 받는 행위자

필요시에는 가정도 세울 수 있으며, 한 문장 정도의 설명문도 적어둘 수 있다. 이 외에 Activity 다이어그램으로도 표현이 가능하다.

Use Case에서 중요한 또 하나는 Use Case 사이에도 관계를 가질 수 있다는 것이다. 첫째는 포함관계로서, 다른 Use Case에서 기존의 쓰임새를 재사용하고 있는 관계이고, 둘째는 확장 관계



포함관계를 나타낸 Use Case 모델

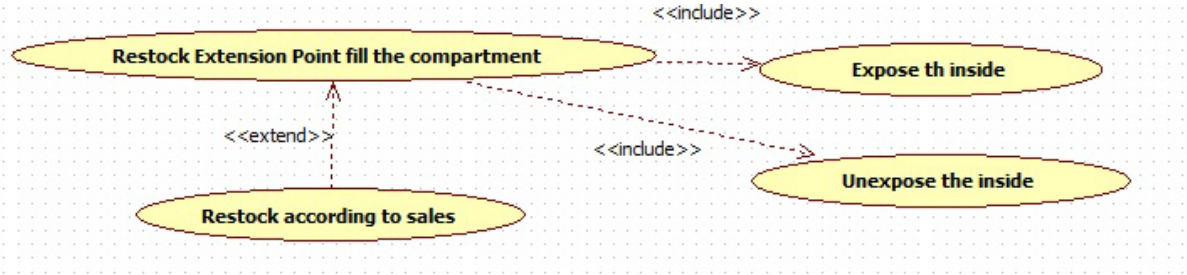
로서, 기존의 Use Case에 진행 단계를 추가하여 새로운 Use Case로 만들어 내는 관계이다. 이외에 일반화와 그룹화가 있다. 일반화는 한 Use Case가 다른 Use Case를 상속한 관계이며, 그룹화는 여러 개의 Use Case를 조직화하는 단순한 방법이다.

- **포함**

  아래 그림을 통해 볼 수 있다. Restok과 Collect Use Case는 둘 다 자동 판매기의 보안 장치를 해제하고 앞문을 열며, 앞문을 닫고 보안장치를 거는 단계를 가지고 있다. "Expose the inside" Use Case는 전자의 단계에, "Unexpose the inside" Use Case는 후자의 단계에 해당한다. 이렇게 포함된 Use Case는 절대로 단독으로 존재할 수 없으며, 전체 Use Case의 부분으로만 존재한다.

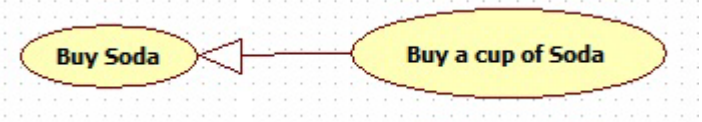
- **확장**

  포함관계와 비슷하게 의존 관계선을 사용하여 연결함으로써 나타내며, <<extend>>스테레오타입을 붙여줌으로써 끝맺는다. 기본 Use Case내의 확장 포인트는 Use Case의 이름 아래에 위치 시킨다.



- **일반화**

  클래스가 다른 클래스로부터 상속받을 수 있듯 Use Case도 상속이 가능하다. Use Case 상속의 경우, 자식 Use Case는 부모 Use Case가 가진 모든 행동과 의미를 물려 받으며, 여기에

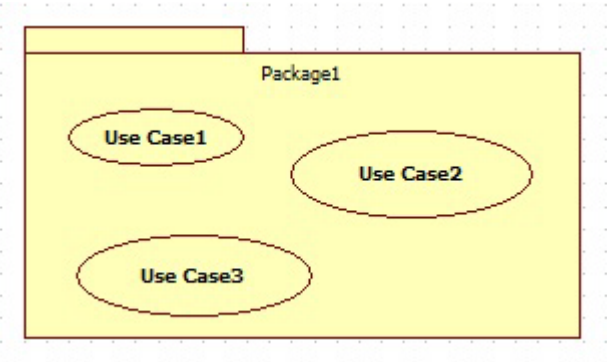


  자신만의 행동을 추가할 수도 있다. 또한, 부모 Use Case 등장한 곳에는 항상 자식 Use Case를 대신 놓을 수 있다.

  이것은 Use Case뿐만 아니고 Actor사이에도 있을 수 있다.

- **그룹화**

  Use Case 다이어그램은 여러 개의 Use Case를 가지고 있는 형태로 나타날 수 있다. 이 때 Use Case를 조직화 하는 것이 좋다.시스템이 여러 개의 서브시스템으로 구성되어 있거나 시스템에 대한 요구사항을 수집하기 위하여 사용자의 의견을 조사할 때가 조직화에 알맞은 경우이다.



조직화는 하나의 패키지로 그룹화하는 것이 가장 간단하 방법으로, 패키지를 나타내고 이 안에 관련된 Use Case를 넣으면 된다.

2. Statechart 다이어그램

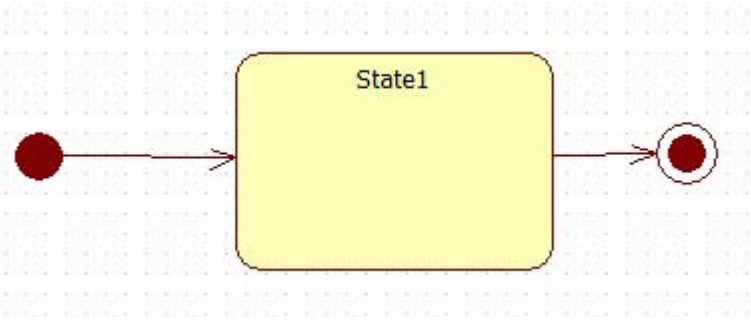
2.1 상태 다이어그램이란?

시스템에 일어나는 변화를 표현하는 방법 중 한 가지는 "사건이나 시간에 따라 시스템 내의 객체들이 자신의 상태(state)를 바꾼다"고 말하는 것이다. 몇 가지의 예를 들어보면 다음과 같다

- **스위치를 누를 때마다, 탁상 전등의 상태는 켜지고 꺼진다.**
- **리모컨의 버튼을 누르면, 텔레비전의 상태는 한 채널을 보여주다가 다른 상태를 보여주게 된다.**
- **얼마간의 시간이 흐르면, 세탁기의 상태는 세탁에서 행궁으로 바뀐다.**

상태 다이어그램은 다른 다이어그램과는 근본적인 차이점을 보이고 있다. 다른 다이어그램들은 시스템이나 최소한 한 개 그룹의 클래스 혹은 객체의 행동을 모델링하는데 사용되는 것인데 반해, 상태 다이어그램은 "단일 객체"의 상태를 나타낸다.

2.2 상태 다이어그램을 구성하는 기호



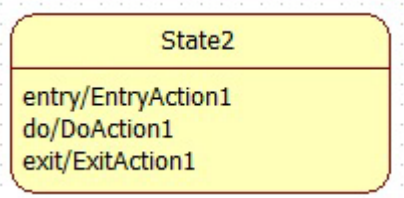
상태를 나타내는 아이콘은 모서리가 둥근 사각형이며, 상태 전이를 나타내는 기호는 화살표 머리가 달린 실선이다. 속을 칠한 원은 상태 흐름의 시작점을 나타내며, 종료점은 원을 둘러싼 원으로 나타낸다.

2.3 상태 아이콘에 넣는 정보들

상태 아이콘에도 클래스 아이콘처럼 세 영역(이름, 속성, 오퍼레이션)으로 나누어 상세한 정보를 써 넣을 수 있다. 가장 위 부분에는 상태의 이름이, 중간부분에는 상태 변수가, 가장 아랫

부분에는 활동이 들어간다.

상태 변수는 타이머나 카운터처럼 상태 진행에 도움을 주는 데이터이다. 활동은 사건과 동작으로 이루어져 있으며, 자주 쓰이는 세 가지는 잘 알아 두는 것이 좋다.



- **진입(entry) – 시스템이 상태로 들어갈 때 일어남**
- **탈출(exit) – 시스템이 상태에서 빠져 나올 때 일어남**
- **활동(do) – 시스템이 상태 안에 있는 동안 일어남**

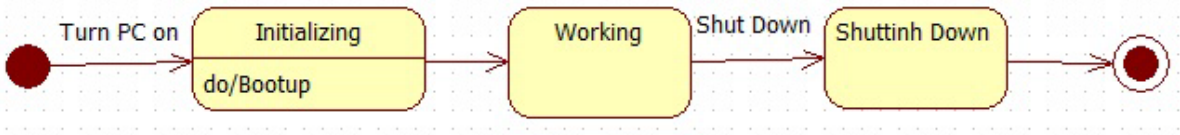
2.4 상태 전이선에 추가되는 정보 : 사건과 동작

GUI를 예로 들어보자. 일단 GUI는 다음의 세가지 상태 중 하나에 있을 수 있다.

- **초기화(initializing)**
- **작동중(working)**
- **끝 마무리(Shutting Down)**

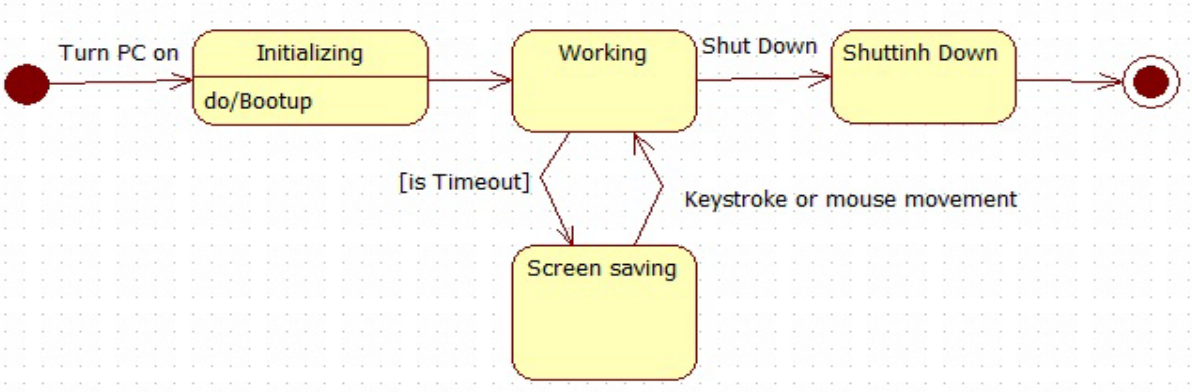
PC를 켜면, 부팅이 수행된다. 따라서 PC를 켜는 일은 GUI가 Initializing 상태로 전이되도록 하는 촉발 사건이며, 부팅은 이 상태 전이가 수행되는 도중에 일어나는 동작이다.

Initializing 상태에 설정된 활동의 결과로, GUI는 Working 상태로 전이된다. 여기서 PC를 끄게 되면 Shutting Down 상태로 전이되도록 하는 촉발 사건이 만들어지고, 결국 PC는 꺼진다.



2.5 상태 전이선에 추가되는 정보 : 전이 조건

만약 Working 상태에서 사용자의 입력이 15분간 없을 때 Screen saver가 작동한다고 하자. 여기서 '15분'의 시간 간격을 UML에서는 전이조건이라고 한다. 즉, 이 조건이 만족 되면 연관된 전이가 일어나는 것이다. 아래 그림은 Screen saving 상태와 이 상태와 관련한 전이 조건이 추가된 GUI의 상태 다이어그램이다. 전이 조건은 Bollean 수식으로 표현한다.



2.6 하위 상태

GUI가 Working 상태에 있을 때 일어나는 동작들은 무척 다채롭고 복잡하다. GUI는 사용자가 키 입력, 마우스 이동, 마우스 버튼 등등 어떤 일을 하는 지를 기다리고 있다. 사용자의 입력이 들어오면 GUI는 이것을 등록하고 적절한 동작이 수행한 결과를 화면에 나타낸다.

GUI는 이렇게 Working 상태 안에 있는 동안에도 변화한다. 이러한 변화는 상태의 변화이며, 주어진 상태 내부에서 일어나는 것이기 때문에 하위 상태라는 이름으로 불린다. 하위 상태는 두 가지로 나뉜다. 하는 순차적 하위 상태이며, 또 하나는 동시적 하위 상태이다.

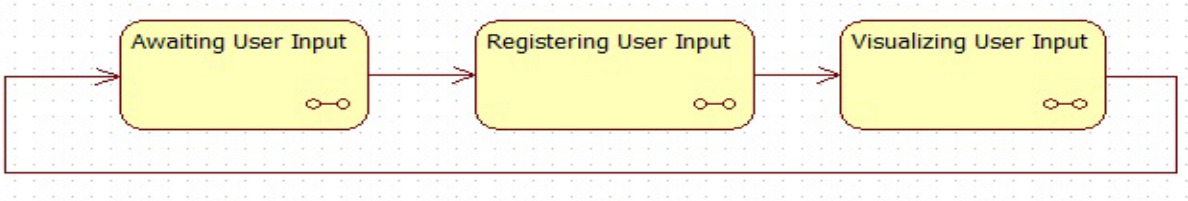
- 순차적 하위 상태

순차적 하위 상태는 차례 차례로 이어진다. GUI의 Working 상태 내에서 변화는 다음과 같은 순차적인 흐름으로 정리 할 수 있다.

· 사용자 입력을 대기

· 사용자 입력을 등록

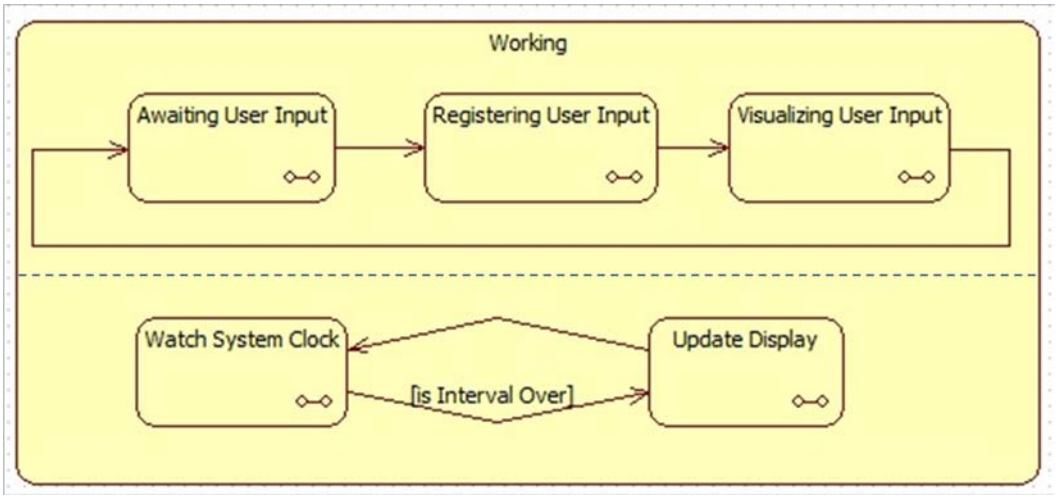
· 사용자 입력을 화면에 나타냄



사용자 입력은 사용자 입력 대기 상태에서부터 전이를 촉발시킨다. 사용자 입력 등록 상태 내에서 활동은 해당 상태를 사용자 입력을 화면에 나타내는 상태로 전이시킨다.

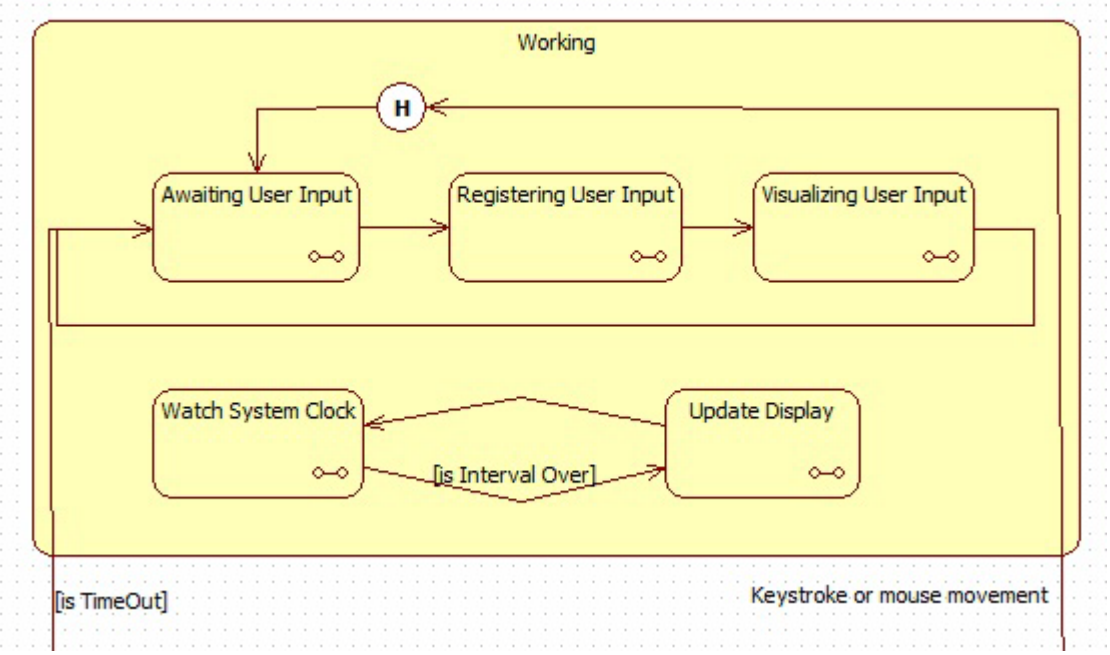
- 동시적 하위 상태

GUI는 시스템 클럭도 체크하고 특정한 시간간격이 지나면 애플리케이션 디스플레이도 갱신해야 한다. 시스템 클럭을 체크하고 디스플레이를 갱신하는 일은 순차적 하위 상태이지만, 입력을 대기하고 갱신하는 순차적 하위 상태와 동시에 진행되고 있다. 이러한 동신 진행성은 순차적 하위 상태 사이를 점선으로 구분 지음으로써 설정할 수 있다.



2.7 이력 상태

UML은 어떤 기호를 써서 복합 상태로 하여금 주어진 객체가 복합 상태를 벗어날 때 활성중인 하위 상태를 기억해 두도록 한다. 이 기호는 원으로 둘러싸인 "H" 문자로서, 이 기호로 나타나는 상태를 이력 상태라고 한다.



2.8 메시지와 신호

Screen saving 상태에서 Working 상태로 전이하게 하는 촉발 사건은 키 입력, 마우스 이동, 마우스 클릭이었다. 사실, 이 사건들은 사용자가 GUI에게 보내는 메시지이다. 객체들은 서로 메시지를 주고받으며 자신의 행동을 수행하기 때문에 이 사실은 매우 중요한 개념이다. 즉, 이 경우 촉발 사건은 한 객체가 다른 객체로 전송하는 메시지라고 할 수 있다.

2.9 상태 다이어그램의 중요성

상태 다이어그램은 분석가, 설계자, 개발자들이 시스템내의 객체 행동을 이해하는데 큰 도움을 준다. 클래스 다이어그램과 객체 다이어그램은 시스템의 정적인 모습만을 보여줄 뿐이다. 즉, 계층, 연관, 행동이 '어떠한가'에만 초점을 두고 있지, 각 행동의 동적인 상황은 보여주지 않는다.

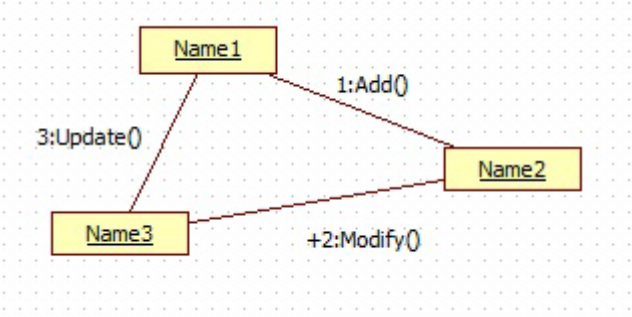
특히, 개발자는 객체 자체를 소프트웨어로 구현해야 하기 때문에 객체들이 어떻게 행동하는지를 정확히 파악하여야 한다. 개발자는 객체를 구현하는 것으로 충분치 않으며, 그 객체에게 무엇인가를 시켜야 한다. 상태 다이어그램은 "객체가 어떤 행동을 하도록 되어 있는지"에 대해 개발자들이 고민할 필요가 없도록 만든다. 객체의 행동이 명확하게 그려져 있는 상태 다이어그램이 있기에, 개발팀이 요구사항에 맞는 시스템을 구축할 가능성이 더 높아지는 것이다.

3. Collaboration 다이어그램

Collaboration 다이어그램은 객체 다이어그램을 확장한 것이다. 객체 사이의 연관 관계뿐만 아니라, 각 객체들이 주고받는 메시지들을 나타낸다.

메시지는 두 객체 사이의 연관선과 가깝게 화살표를 그려 주면 된다. 화살표는 메시지를 받는 객체를 향한다. 화살표에는 레이블을 달아 이 메시지가 어떤 메시지인지를 명시해주어야 한다. 이렇게 설정된 메시지는 대개 "메시지를 받는 객체로 하여금 어떤 오퍼레이션을 실행하라"는 뜻이다. 메시지의 끝에는 괄호(())가 놓인다. 필요하면 이 괄호 안에 매개변수를 써넣을 수도 있다.

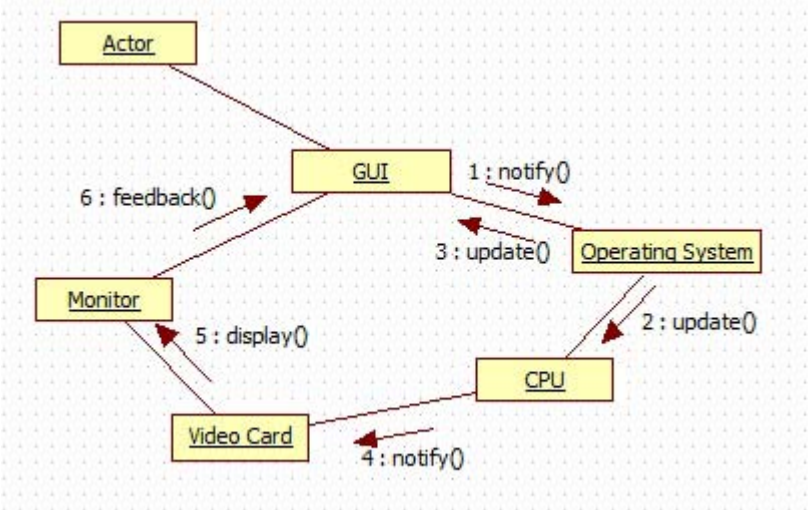
Collaboration 다이어그램은 순차적인 흐름 정보도 나타낼 수 있다. 이를 위하여 메시지에 번호를 매겨 각 메시지의 처리 순서를 나타낸다. 번호와 메시지 사이는 콜론(:)으로 구분한다.



3.1 GUI의 Collaboration 다이어그램

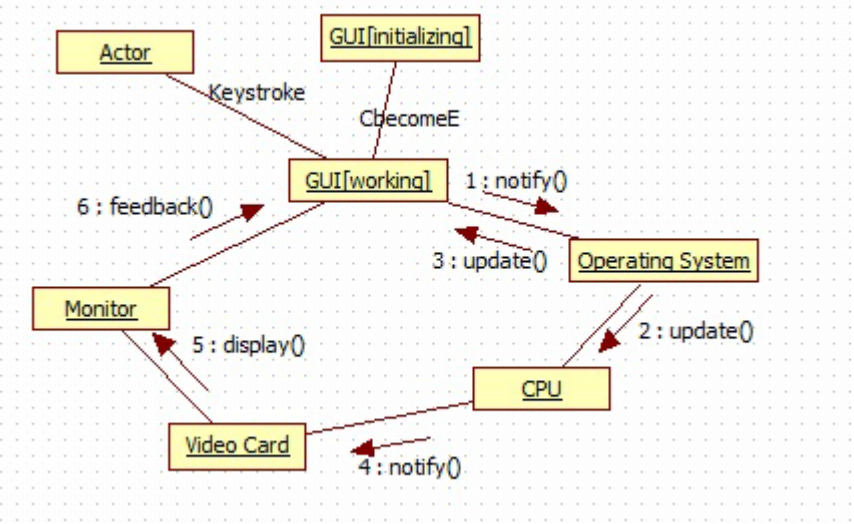
이 예제는 단순한 경우이다. 행위자는 키 입력으로 GUI의 동작을 시작하게 하며, 메시지는 순차적으로 발생한다. 각 진행 단계는 다음과 같을 것이다.

1. **GUI는 키 입력을 운영체제에게 알린다.**
2. **운영체제는 CPU에게 그 사실을 알린다**
3. **운영체제는 GUI를 갱신한다.**
4. **CPU는 비디오카드에게 GUI 갱신에 필요한 명령을 내린다.**
5. **비디오 카드는 모니터로 메시지를 전송한다.**
6. **모니터는 화면에 알파뉴메릭 문자를 표시하여 사용자에게 피드백을 제공한다.**



3.2 상태 변화의 추가

Collaboration 다이어그램에 객체의 상태 변화를 나타낼 수 있다. 객체 사각형 안에 객체의 상태를 나타낸다. 다이어그램에 변경된 상태의 객체를 나타내는 사각형을 하나 더 그려 넣는다.

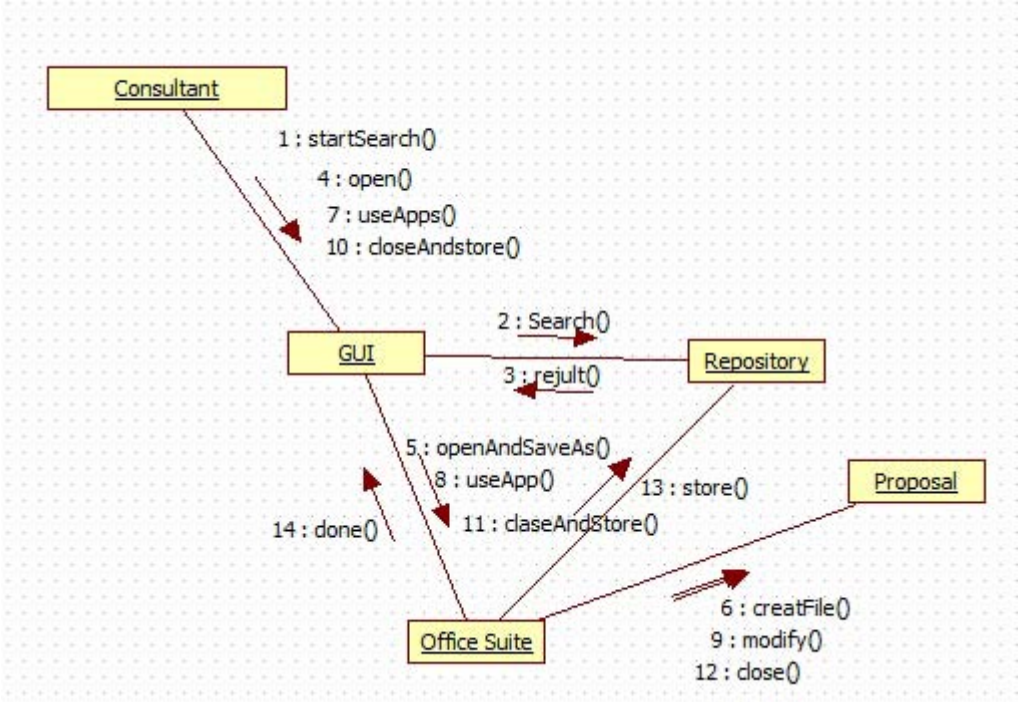


3.3 객체의 생성

객체의 생성을 다루기 위해 컨설팅 기업의 LAN 구축의 "Create a Proposal" Use Case에 대해 알아보자. 모델링할 진행 단계는 다음과 같다

1. **컨설턴트는 기존의 제안서를 사용했으면 하기 때문에, 네트워크에 물려 있는 중앙 저장소에서 적당한 제안서를 찾는다.**
2. **만일 컨설턴트가 적당한 제안서를 찾아내면, 이 파일을 열고, 이 과정에서 오피스 도구모음 소프트웨어가 실행된다. 컨설턴트는 이 파일을 새로운 이름으로 저장하고 새 제안서를 위한 새 파일을 만든다.**
3. **만일 컨설턴트가 제안서를 찾아내지 못했다면, 오피스 도구모음 소프트웨어를 실행시키고 제안서를 위한 새 파일을 만든다.**
4. **제안서를 작성하는 동안, 컨설턴트는 오피스 도구모음 소프트웨어가 가진 애플리케이션을 사용한다.**
5. **컨설턴트가 제안서 작업을 마쳤을 때, 컨설턴트는 중앙 저장소에 이 제안서를 저장한다.**

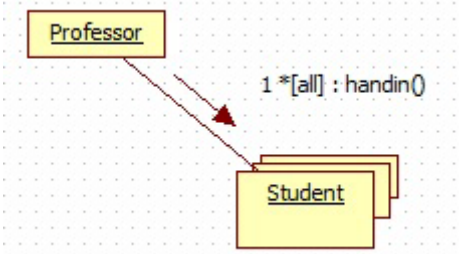
객체의 생성을 나타내기 위해서는 객체를 생성하는 메시지에 <<create>> 스테레오타입을 붙여주면 된다.



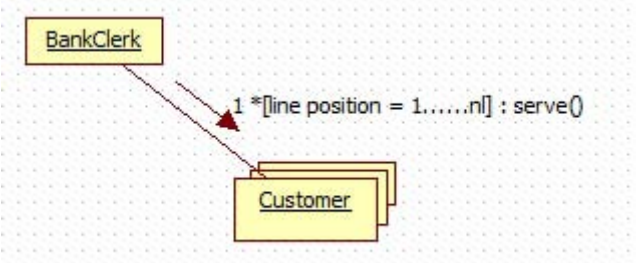
3.4 그 외의 개념들

- 다중 객체로의 메시지 전송

  동일한 클래스에서 만든 여러 개의 객체에게 메시지를 보내는 경우가 있다. 예를 들어, 교수는 여러 학생에게 숙제를 내라고 말할 수 있다. 이것을 Collaboration 다이어그램으로 나타내려면 객체 사각형을 사선 방향으로 쌓는다. 그리고 객체로 전송되는 메시지에는 애스터리스크가 붙은 대괄호 조건문을 붙여준다.

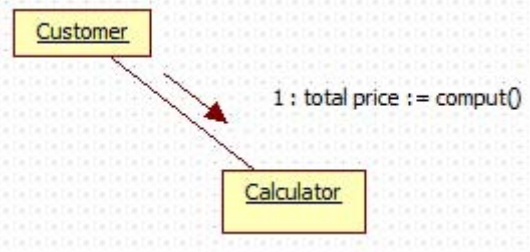


  메시지를 보내는 순서가 중요한 때가 있다. 은행원은 창구에 늘어선 고개들을 순서대로 맞아 서비스를 해준다. 이 상황은 순서("line position = 1...n")를 고려한 "while" 조건으로 나타낼 수 있다.



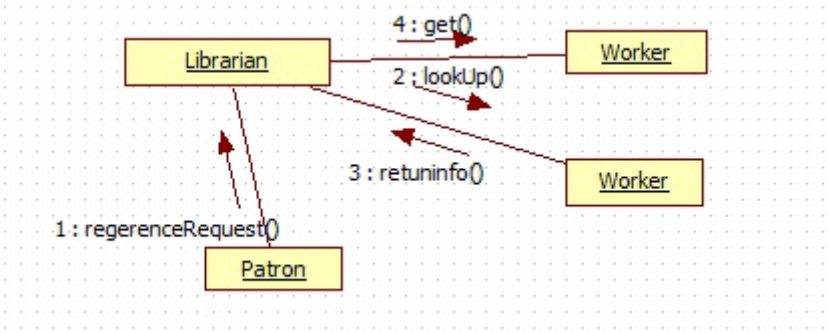
- 반환된 결과 나타내기

  오퍼레이션의 결과값 반환은 "반환되는 값의 이름 := 오퍼레이션" 형태의 수식을 써줌으로써 나타낼 수 있다. 이러한 수식을 메시지-시그니쳐 라고 한다.



- 활성객체

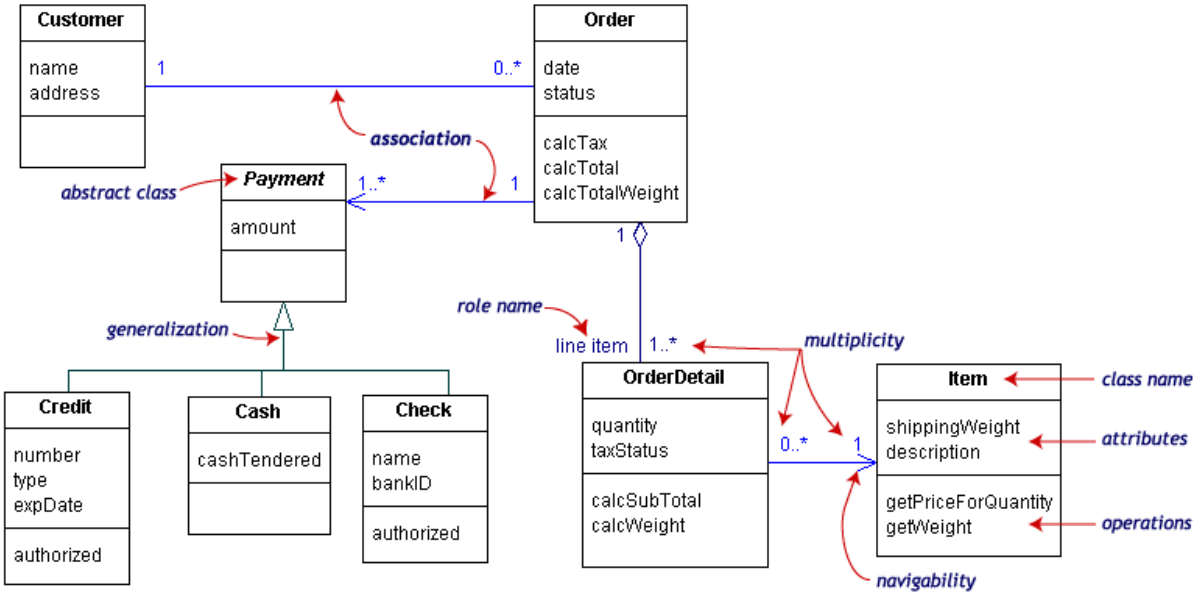
  객체간의 교류에 있어서 특정한 객체가 흐름을 제어하는 경우가 있다. 흐름을 제어하는 이 객체를 활성 객체라고 하며, 활성 객체는 수동 객체에게 메시지를 보내며 다른 활성 객체들과 교류한다. 도서관을 예로 들면, 사서는 고객의 요청을 받아 데이터베이스로부터 참고 정보를 찾아 알려주며, 일꾼에게 책을 다시 꽂아 두라는 등의 책임을 할당한다. 또한, 사서는 자신과 동일한 일을 수행하는 다른 사서들과 메시지를 주고 받는다. 시스템 내에서 두 개 이상의 객체가 동시에 작동될 수 잇는데, 이것을 동시성이라고 한다.



4. Class 다이어그램

클래스 다이어그램(Class Diagram)은 클래스와 클래스간의 관계를 통해 시스템의 전체적인 모습을 보여주는 Static Structure Diagram이다.

아래 클래스 다이어그램은 사용자의 쇼핑 주문에 대한 모델링 예제이다. 여기서 중심이 되는 클래스는 Order인데, 이것은 Customer가 Payment를 지불하도록 되어 있다. Payment는 Cash, Check 또는 Credit의 세가지중 하나다.



UML클래스 표기는 사각형안에 클래스명(class name), 속성(attributes), 오퍼레이션(operation)의 세가지 부분으로 나누어 진다. Payment와 같이 추상클래스(abstract class)는 이탤릭체로 나타내고 클래스간의 관계는 선으로 연결된다.
위의 다이어그램은 세가지 관계를 가지고 있다.

- **Association** -- 두 클래스의 인스턴스간의 관계. 한쪽 클래스의 인스턴스는 반드시 반대편 클래스에 대해 알고 있어야만 재대로 작동할 수 있다.
- **Aggregation** -- 한 클래스가 컬렉션으로 포함되는것을 나타낸다. Aggregation은 다이아몬드 모양의 끝이 포함하는 클래스를 향하도록 나타낸다. 위의 다이어그램에서는 Order클래스가 OrderDetails의 컬렉션을 포함하고 있다.
- **Generalization** -- generalization관계는 상속(inheritance) 특성을 가진다. 상속연결은 한 클래스가 다른 클래스들의 상위클래스(super class)임을 나타낸다. 일반화는 삼각형 모양이 상위클래스(super class)를 향하도록 표현한다. 위의 다이어그램에서 Payment는 Cash, Check, Credit의 상위클래스(super class)다.

하나의 관계는 두개의 끝점(end point)을 가진다. 끝점은 연관의 형태를 명확히 하기위해 역할명(role name)을 가진다. 예를들어, OrderDetail은 각 Order의 항목(line item)이다.
Navigability은 화살표가 가르키는 방향에 있는 클래스를 탐색하거나 질의할 수 있음을 나타낸다. OrderDetail은 그것의 Item에 대해 질의(query)할 수 있지만 그 반대로는 할 수 없다. 화살표는 누가 관계에서 구현을 하는 "주체"인지 알 수 있게 해준다. 이경우, OrderDetail은 Item을 가지고 있다. 연관에서 화살표를 생략할 경우 묵시적으로 양방향 관계를 나타낸다.
Multiplicity은 하나의 인스턴스에 연관된 다른쪽 클래스의 가능한 인스턴스의 수를 의미한다. Multiplicity은 하나의 숫자 또는 수의 범위로 나타낼 수 있다. 위의 예제에서는 각 Order에 하나의 Customer만 연관이 되지만, Customer는 다수의 Order를 가질 수 있다.

아래는 일반적인 다중성의 표현방법이다.

표기법	의미
0..1	0 또는 하나의 인스턴스. n..m 표기법은 n에서 m까지의 범위
0..* 또는 *	0을 포함한 무한수. 제한없음
1	명백한 하나
1..*	적어도 하나이상

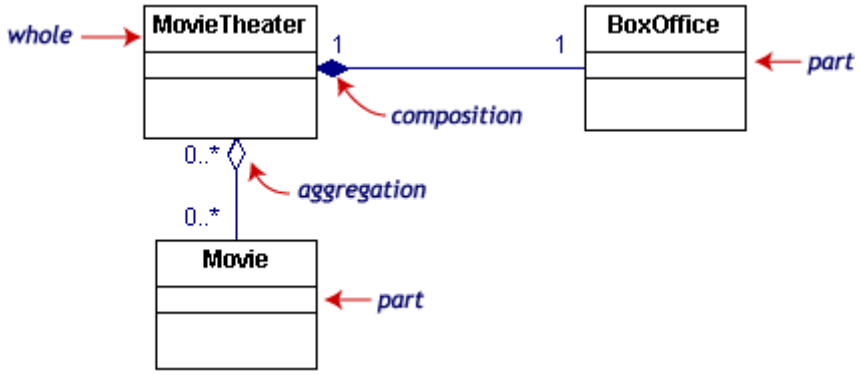
모든 클래스 다이어그램은 Class, Association, 그리고 Multiplicies을 가진다. Navigability과 역할은 다이어그램내에서의 위치를 명확히 하기위한 옵션이다.

모든 클래스 다이어그램은 Class, Association, Multiplicies을 가지지만 더 많은정보를 보여줄 수도 있다. 이미 Generalization, Aggregation, 그리고 Navigability에 대해 알아보았다. 지금부터 다음의 항목들에 대해 더 알아 보겠자.

Composition
클래스 멤버에 대한 Visibility와 Scope
Dependencies와 Constraints
인터페이스(interfaces)

4.1 Composition and aggregation

객체가 특정클래스의 일부로 속하는 관계를 Aggregation이라 했다. Composition은 이보다 더 강한 관계의 의미로서 객체가 부모(whole)의 일부일 뿐만 아니라 부모없이는 존재할 수 없는 것을 의미한다. Composition은 집합의 표기에서 다이아몬드를 채운형태로 표현된다.
아래 다이어그램은 BoxOffice가 반드시 MovieTheater에 속함을 보여준다. MovieTheater를 제거하면 BoxOffice도 함께 제거된다. 하지만, Movie 컬렉션은 이처럼 MovieTheater와 밀접하게 연관되어있지 않다.

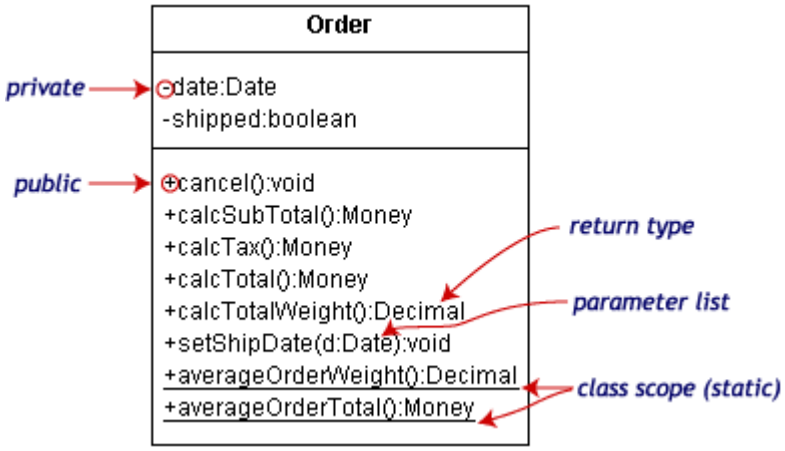


집합은 대부분의 경우 연관과 특별히 다른점이 없다. UML은 명확한 집합의 정의를 제공하지 않는다. 그렇기 때문에 집합에 관해 많은 혼란이 생겼으며 실제 UML 2.0에서 집합은 배제되었다. 집합에서 명확한 규칙은 두개 또는 그 이상의 객체가 자기자신의 부분이 될수 없고 서

로 상대 객체의 부분이 될수 없다. 즉, 객체간의 관계의 고리를 만들수 없다는 것이다.
합성도 집합과 같은 규칙이 적용된다. 그리고, 합성은 주인이 피보호자의 수명 전체에 책임을 진다. 만약, 주인이 복사되면 피보호자도 함께 복사된다. 피보호자의 한 인스턴스를 두 주인이 동시에 소유할 수 없다. 합성이 사용되는 경우는 deep copy이다. 즉, 복사된 값이 별도로 변경되는 경우와 같이 특수한 경우에 사용될 수 있다.

4.2 visibility and scope

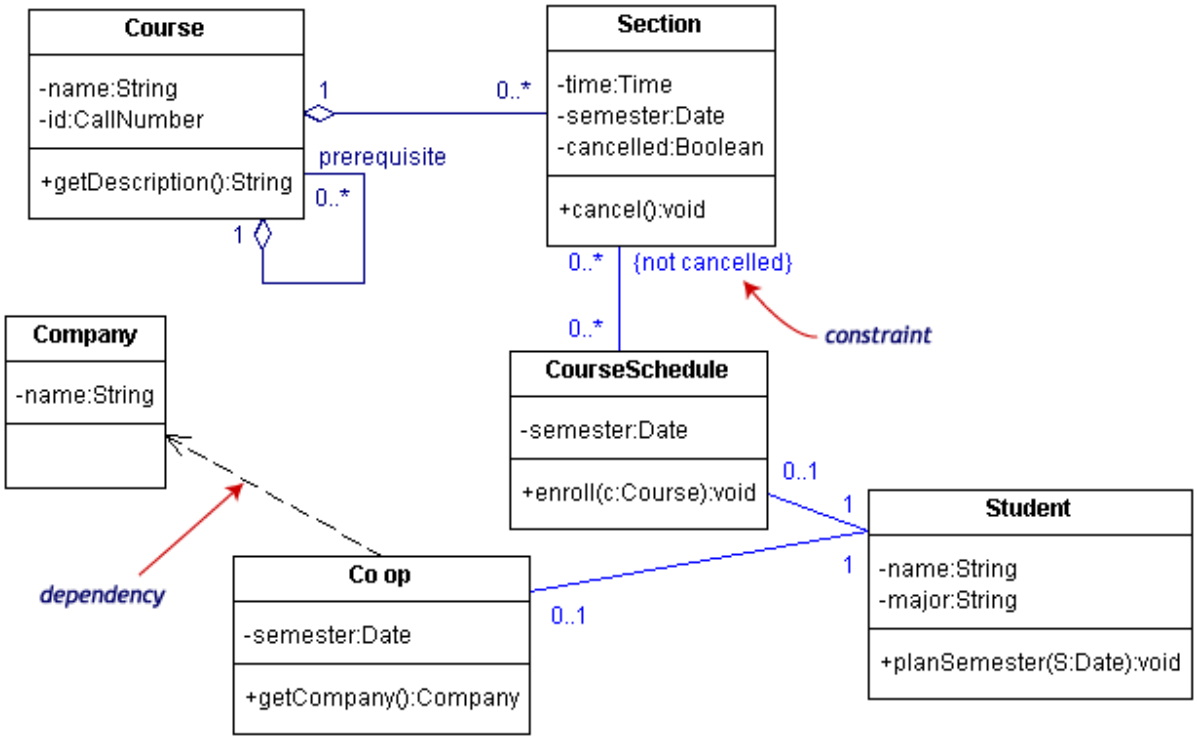
클래스표기는 클래스명(class name), 속성(attributes), 오퍼레이션(operation)의 세영역으로 나눈다. attributes과 operation은 접근성(access)과 범위(scope)에 따라 레이블을을 붙일 수 있다.



- **static 멤버는 밑줄로 표현한다. 반대로, 인스턴스는 밑줄이 없다.**
- **오퍼레이션은 다음의 형식을 따른다.**
  <접근 구분자> <이름> (<파라미터 목록>) : <리턴 타입>
- **파라미터 목록에서 각 타입은 콜론(colon)뒤에 따라온다.**
- **접근 구분자는 각 멤버의 맨앞에 위치한다.**
  + : public
  - : private
  # : protected

4.3 Dependencies과 Constraints

Dependency은 두 클래스간의 관계에서 한쪽이 변경되면 다른 한쪽도 영향을 받을 경우를 말한다. 종속성은 점선으로 표현된다. 아래 클래스 다이어그램에서 Co_op는 Company에 종속적이다. 만일 Company를 변경하기로 마음먹었다면 Co_op또한 변경해야 된다.

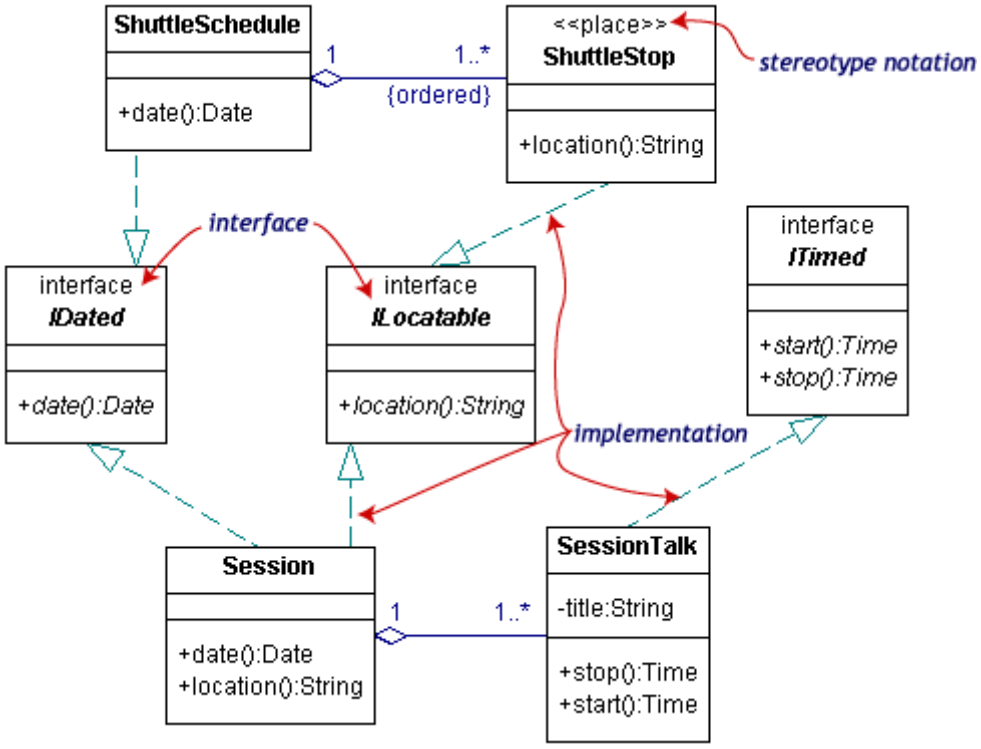


Constraints은 디자인이 연관이 성립되기 위해 만족시켜야할 구현조건을 명시한 것이다. Constraints은 중괄호({ })사이에 쓰여진다. 위의 다이어그램에서 constraint 는 Section이 CourseSchedule의 부분이 될수 있음을 나타낸다.

4.4 Interfaces와 Stereotypes

Interface는 오퍼레이션 시그니처의 집합이다. C++에서 인터페이스는 순수한 가상멤버(virtual members)로만 구성된 추상클래스(abstract class)처럼 구현된다. 자바에서는 직접 구현된다.

아래의 클래스 다이어그램은 컨퍼런스과정을 모델링한 예제이다. 여기서 핵심적인 클래스는 하나의 프레젠테이션을 나타내는 SessionTalk와 하루에 진행되는 SessionTalk의 컬렉션인 Session이다. ShuttleSchedule과 그 리스트인 ShuttleStops은 호텔에서 기다리고 있는 참석자들에게 매우 중요하다. 이 다이어그램은 ShuttleStops가 순서화(ordered)되어 있다는 하나의 Constraints을 가진다.

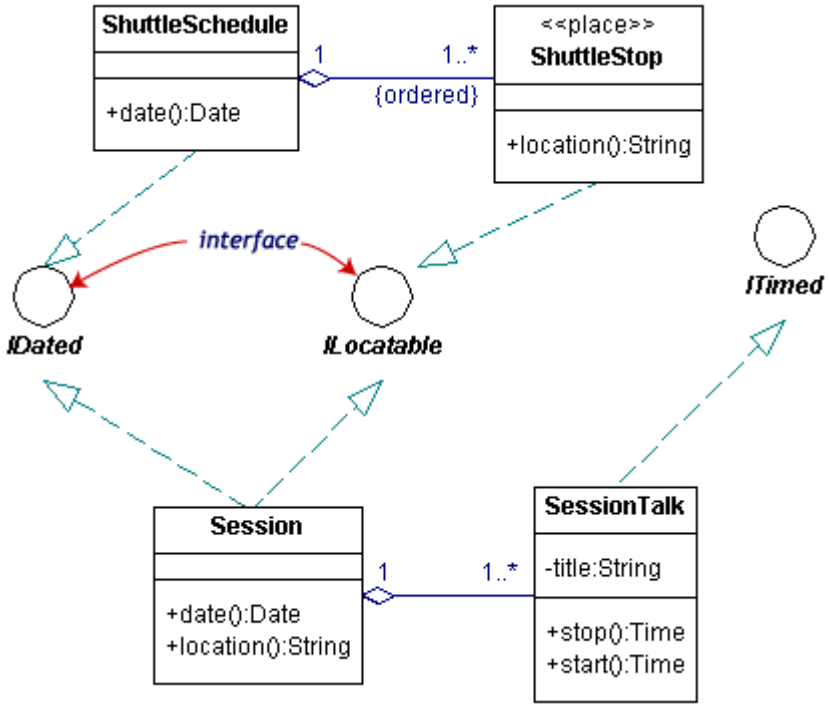


위의 다이어그램은 IDated, ILocatable, ITimed 세개의 인터페이스(interfaces)를 가지고 있다. 인터페이스의 이름은 일반적으로 "I"로 시작한다. 인터페이스명을 포함한 오퍼레이션 시그니처들은 이탤릭(Italic)체로 표현되어 있다.

ShuttleStop은 ILocatable을 구현(또는 구체화)하기 위해 동일한 오퍼레이션(location():String)을 구현하고 있다.

ShuttleStop을 <<place>>라는 스테레오타입을 가지고 있다. 스테레오타입(stereotypes)은 존재하지 않는 새로운 종류의 모델을 정의하여 UML을 확장시키는 방법을 제공한다. 그런의미에서 인터페이스(Interface) 또한 스테레오타입의 한종류라 할 수 있다. 스테레오타입은 guillemets라는 특수문자 사이에 삽입되지만 통상 "<"와 ">"을 두개씩 겹쳐서 "<<" , ">>"으로 표현하기도 한다.

인터페이스(Interfaces)를 UML로 표현하는데는 두가지 방법이 있다. 아래 다이어그램은 위의 다이어그램과 동일하며 Lollipop, 또는 Circle 표기법이라 부른다.



이 표기법에서, 인터페이스는 원형으로 표현되며 구현클래스와 연결되어 있다. 이 표기법은 다이어그램의 전체적인 가독성을 높이고 단순화한 장점은 있으나 상세한 정보는 표현되지 않는다.

5. Sequence 다이어그램

시퀀스 다이어그램은 객체들 간 인터랙션을 발생 순서대로 보여줄 때 쓰인다. 클래스 다이어그램과 마찬가지로 개발자들은 이 시퀀스 다이어그램이 자신들을 위한 것이라고 생각한다. 하지만 영업 부서의 직원들 역시 비즈니스가 어떻게 돌아가고 있는지를 설명할 때, 시퀀스 다이어그램을 사용할 수 있다. 기업의 현업을 문서화 하는 것 외에도 비즈니스 레벨의 시퀀스 다이어그램은, 앞으로의 시스템 구현에 필요한 요구 사항들을 기록하는 문서로서 사용된다. 프로젝트의 요구 사항을 분석하는 동안 분석가들은 사용 케이스를 다음 레벨에 적용할 수 있다. 사용 케이스들은 보다 잘 정리되어 시퀀스 다이어그램 안으로 들어간다.

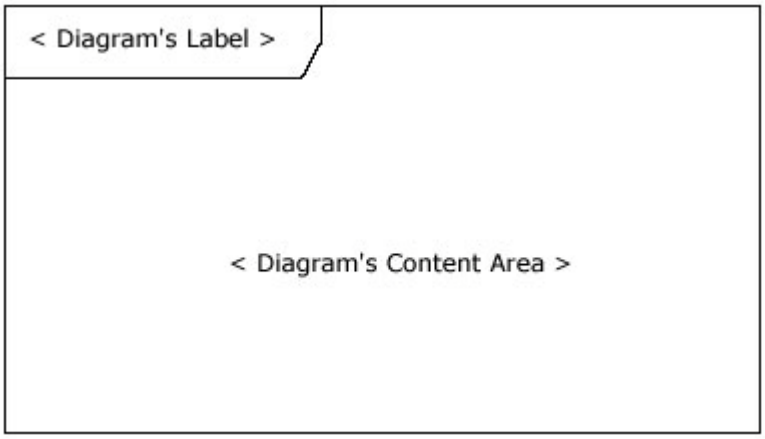
기업의 기술부 사원들은 앞으로의 시스템이 어떻게 작동해야 하는지를 문서화할 때 시퀀스 다이어그램을 활용할 수 있다. 디자인 단계에서, 아키텍트와 개발자들은 이 다이어그램을 사용하여 시스템 객체의 인터랙션을 실행해 보고, 전체 시스템 디자인을 완성한다.

시퀀스 다이어그램은 주로 형식적인 정련 단계에서 사용된다. 사용 케이스들은 한 개 이상의 시퀀스 다이어그램으로 세분화된다. 새로운 시스템을 설계할 때 사용되기도 하지만, 시퀀스 다이어그램은 기존 "레거시(legacy)" 시스템의 객체들이 현재는 어떻게 인터랙팅 하는지를 문서화하는데 사용될 수 있다. 시스템을 이동할 때 문서화는 매우 유용하다.

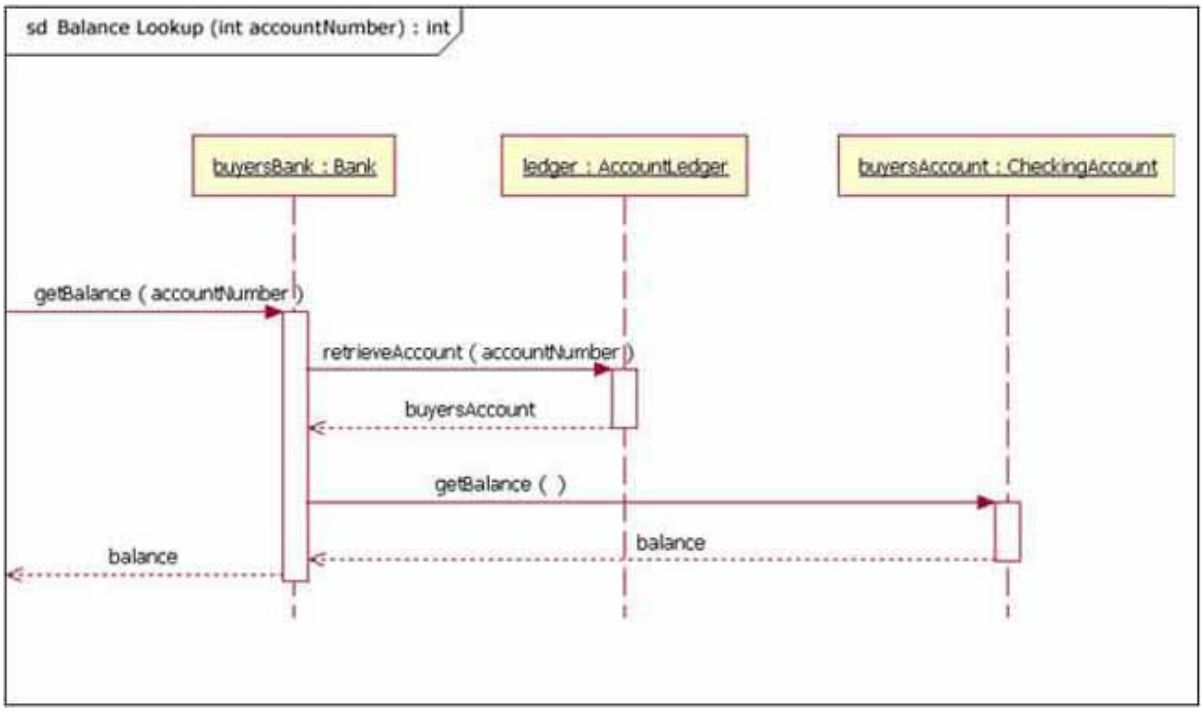
5.1 표기법

이 UML 2 다이어그램의 표기법에 추가된 프레임 엘리먼트는 UML 2의 다른 많은 다이어그램 엘리먼트의 기초로 쓰이지만, 처음에 대부분의 사람들은 이 프레임 엘리먼트를 다이어그램의

그래픽 영역이라고 생각한다. 프레임 엘리먼트는 다이어그램의 레이블을 위한 지정된 장소를 제공하고, 다이어그램의 그래픽 영역을 제공한다. 프레임 엘리먼트는 UML 다이어그램에서는 선택 사항이다. 그림 1과 2에서 보듯, 다이어그램의 레이블은 프레임의 "네임박스(namebox)" 라고 부르게 될 왼쪽 코너의 상단에 놓인다. 실제 UML 다이어그램은 더 큰 직사각형 안에서 정의된다.



시각적으로 경계선을 표시하는 것 외에도 이 프레임 엘리먼트는 인터랙션을 설명하는 다이어그램(시퀀스 다이어그램)에서도 중요한 기능도 한다. 시퀀스 다이어그램에서 시퀀스에 대한 인커밍 메시지와 아웃고잉 메시지(인터랙션)는, 이 메시지들을 프레임 엘리먼트의 경계선에 연결하여 모델링 된다.



다이어그램을 위한 프레임 엘리먼트를 사용할 때 다이어그램의 레이블은 다음 포맷을 따라야 한다.

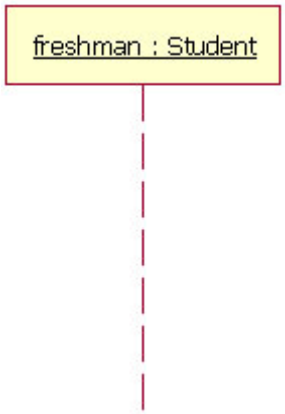
다이어그램타입 (sd) 다이어그램명 (Diagram Type Diagram Name)

5.2 기초

시퀀스 다이어그램의 주요 목적은 어떤 결과를 만들어내는 이벤트 시퀀스를 정의하는 것이다. 메시지 보다는 메시지가 발생하는 순서에 초점이 더 맞춰진다. 대부분 시퀀스 다이어그램은 system 객체들 간 어떤 메시지들이 보내지는지, 그리고 어떤 순서로 발생하는지를 나타낸다. 다이어그램은 이 정보를 수직적 측면과 수평적 측면으로 전달한다. 수직 측면에서는 탑다운(top down) 방식으로 메시지/호출이 발생한 시간 순서를 나타내고, 수평 측면에서는 왼쪽에서 오른쪽으로 메시지가 보내진 객체 인스턴스를 보여준다.

5.3 Lifelines

시퀀스 다이어그램을 그릴 때 Lifeline 표기법 엘리먼트는 다이어그램 상단에 놓인다. Lifeline은 모델링되는 시퀀스에 개입된 역할 또는 개게 인스턴스들을 나타낸다. Lifeline은 박스의 아래쪽 중심에서 대시(dash) 라인을 그리며 내려간다. 이 Lifeline의 이름은 박스 내부에 있다



Lifeline의 UML의 네이밍 표준은 다음 포맷은 따른다.

인스턴스명(객체명) : 클래스명

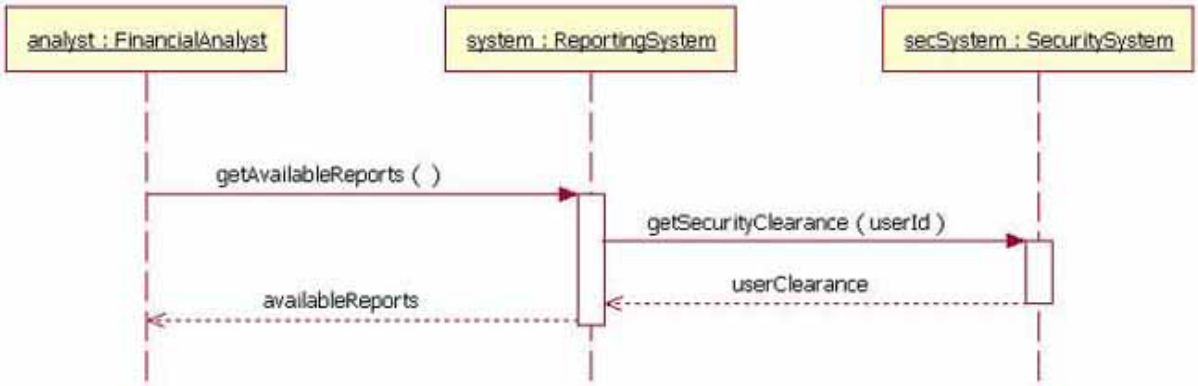
위의 예제에서, Lifeline은 Student 클래스의 인스턴스를 나타낸다. 이것의 인스턴스 이름은 freshman이다. Lifeline 이름 밑에 그어진 밑줄에 주목하라. 밑줄이 사용될 때는 Lifeline이 한 시퀀스 다이어그램에서 클래스의 특정 인스턴스를 나타낸다는 것을 의미한다. 특정 종류의 인스턴스(예를 들어, '역할')가 아니다. 구조 모델링에 대해서도 살펴볼 것이다. 지금까지 누가(Bill과 Fred) 그 역할을 수행하는지를 지정하지 않은 시퀀스 다이어그램에는 buyer와 seller 등의 역할이 포함되어 있다는 것을 알 수 있다. 이런 경우 다이어그램은 다른 정황에서도 재사용된다. 시퀀스 다이어그램에 역할 이름이 아닌 인스턴스 이름에 밑줄을 긋는다.

위의 Lifeline 예제는 네임드 객체이다. 하지만 모든 Lifeline이 네임드 객체를 나타내는 것은 아니다. 대신 익명 또는 이름없는 인스턴스를 나타내는데도 Lifeline이 사용될 수 있다. 시퀀스 다이어그램에 이름없는 인스턴스를 모델링 할 때, Lifeline의 이름은 네임드 인스턴스와 같은 패턴을 따른다. 그러나 인스턴스 이름을 주는 대신에, Lifeline의 이름의 부분이 공백으로 된다. 그림 3을 다시 보자. 만약 이 Lifeline이 Student 클래스의 익명 인스턴스를 나타낸다면, Lifeline은 " Student." 이다. 또한 시퀀스 다이어그램은 프로젝트의 디자인 단계에서 사용되기 때문에 유형이 지정되지 않은 객체를 갖고 있는 것이 맞다. 예를 들어 "freshman."이 바로 그것이다.

5.4 메시지

시퀀스 다이어그램의 첫 번째 메시지는 언제나 상단에서 시작하고 다이어그램의 왼쪽에 위치한다. 뒤따르는 메시지들은 이전 메시지보다 약간 낮게 다이어그램에 추가된다.

메시지를 또 다른 객체에 보내는 객체(lifeline)를 나타내기 위해서 수신 객체에 실선 화살표(동기식 호출일 경우)를 긋는다. 또는 (비동기식일 경우) 막대 화살표를 긋는다. 메시지/메소드 이름은 화살표 위에 놓인다. 수신 객체로 보내지는 메시지는 수신 객체의 클래스가 구현하는 작동/메소드를 나타낸다. 그림 4의 예제에서, analyst 객체는 ReportingSystem 클래스의 인스턴스인 system 객체를 호출한다. analyst 객체는 system 객체의 getAvailableReports 메소드를 호출한다. system 객체는 secSystem 객체에 userId의 인자와 함께 getSecurityClearance 메소드를 호출한다. 이것이 바로 SecuritySystem 클래스 유형이다.

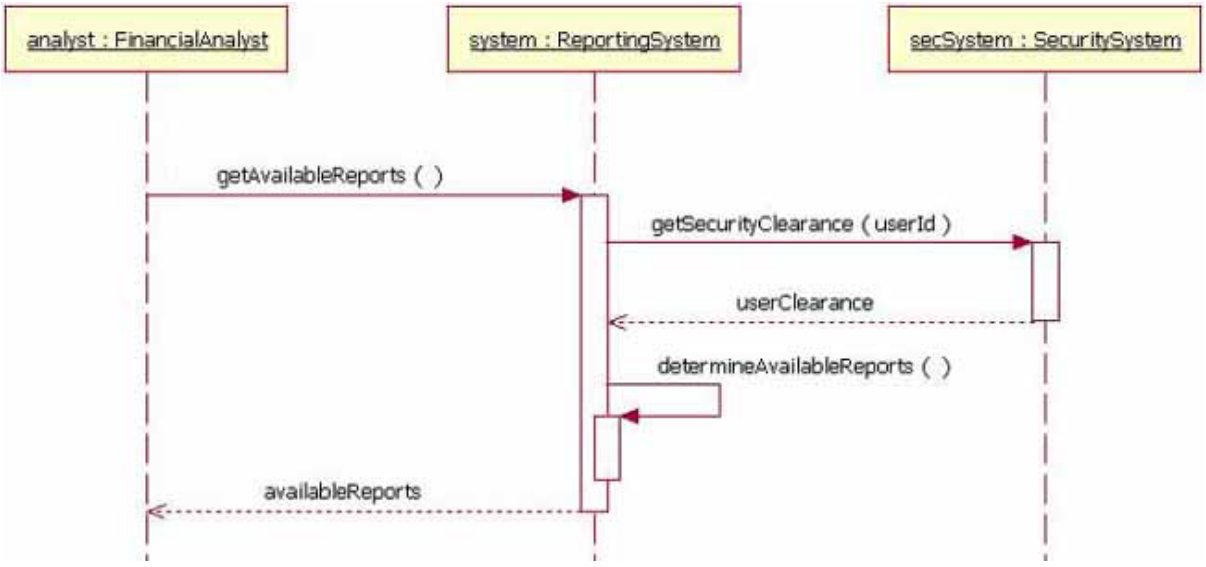


시퀀스 다이어그램에 대한 메시지 호출을 보여주는 것 외에도 위 예제의 다이어그램에는 리턴 메시지가 포함되어 있다. 이 리턴 메시지들은 필수요소는 아니다. 리턴 메시지는 원래 lifeline을 향하도록 점선 화살표로 그려지고 그 위에 리턴 값을 배치한다. 위의 예제에서, getSecurityClearance 메소드가 호출될 때 secSystem 객체는 system 객체에 userClearance를 리턴한다. 이 system 객체는 getAvailableReports 메소드가 호출되면 availableReports를 리턴한다.

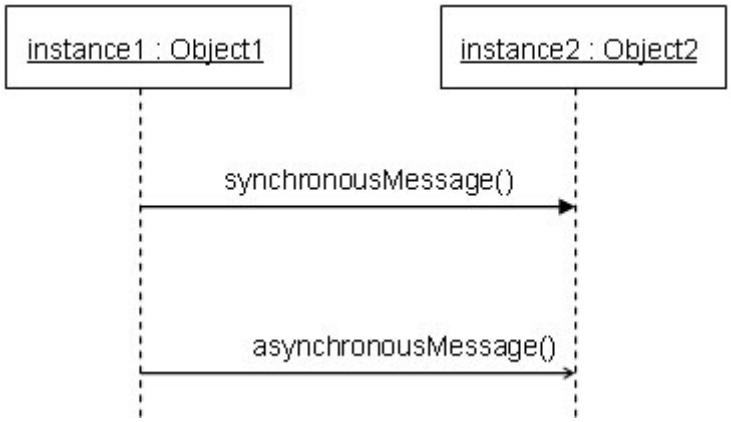
다시 말하지만, 리턴 메시지는 시퀀스 다이어그램의 선택 사항이다. 리턴 메시지의 사용 여부는 모델링되는 것의 상세함 정도에 달려있다. 리턴 메시지는 보다 상세한 것을 원할 때 유용하다. 하지만 호출 메시지로도 충분하다. 개인적으로는 값이 리턴될 때마다 리턴 메시지를 삽입한다.

시퀀스 다이어그램을 모델링 할 때, 객체가 자신에게 메시지를 보내야 할 때가 있다. 언제 객체가 자기자신을 호출할까? 순수주의자들은 객체는 메시지를 객체 자신에게 보내서는 안된다고 주장한다. 하지만 자신에게 메시지를 보내는 객체를 모델링 하는 것도 어떤 경우에는 유용하다. 아래 예제는 위 예제를 개선한 것이다. 아래 예제는 determineAvailableReports 메소드를 호출하는 system 객체를 보여준다. 그 system 객체에 "determineAvailableReports," 메시지를 보여줌으로써 모델은 이 프로세스가 system 객체에서 발생한다는 사실에 주목할 수 있다.

자기자신을 호출하는 객체를 그리기 위해서는 정상적인 방법으로 메시지를 그리되 또 다른 객체로 연결하는 대신, 메시지를 다시 객체 자신으로 연결한다.

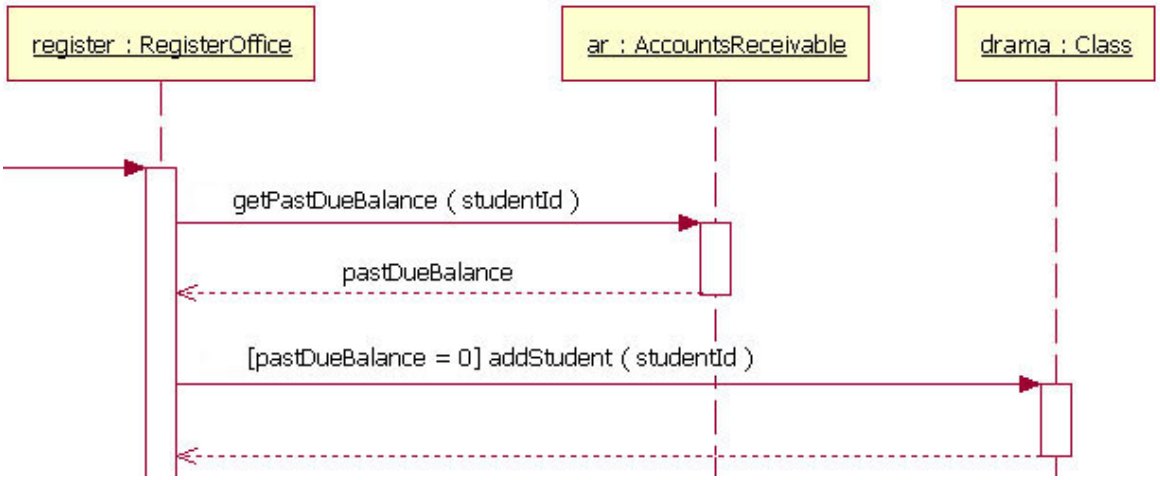


위의 예제 메시지는 동기식 메시지이다. 하지만 시퀀스 다이어그램에서는 비동기식 메시지도 모델링 할 수 있다. 비동기식 메시지는 동기식 메시지와 비슷하게 그려지지만 메시지 라인은 막대 화살표로 표시된다.



5.5 가드(guard)

객체 인터랙션을 모델링 할 때 객체로 보내지는 메시지 조건이 부합해야 할 때도 있다. 가드(guard)는 흐름을 제어하는 UML 다이어그램에서 쓰인다. UML 1.x 와 UML 2.0 모두 가드를 언급했다. UML 1.x에서 보호는 하나의 메시지에만 할당될 수 있었다. UML 1.x의 시퀀스 다이어그램에 가드를 그리려면 보호되고 있는 메시지 라인 위, 메시지 이름 앞에 guard 엘리먼트를 둔다. 아래 예제는 메시지 addStudent 메소드에 대한 가드가 있는 시퀀스 다이어그램이다.

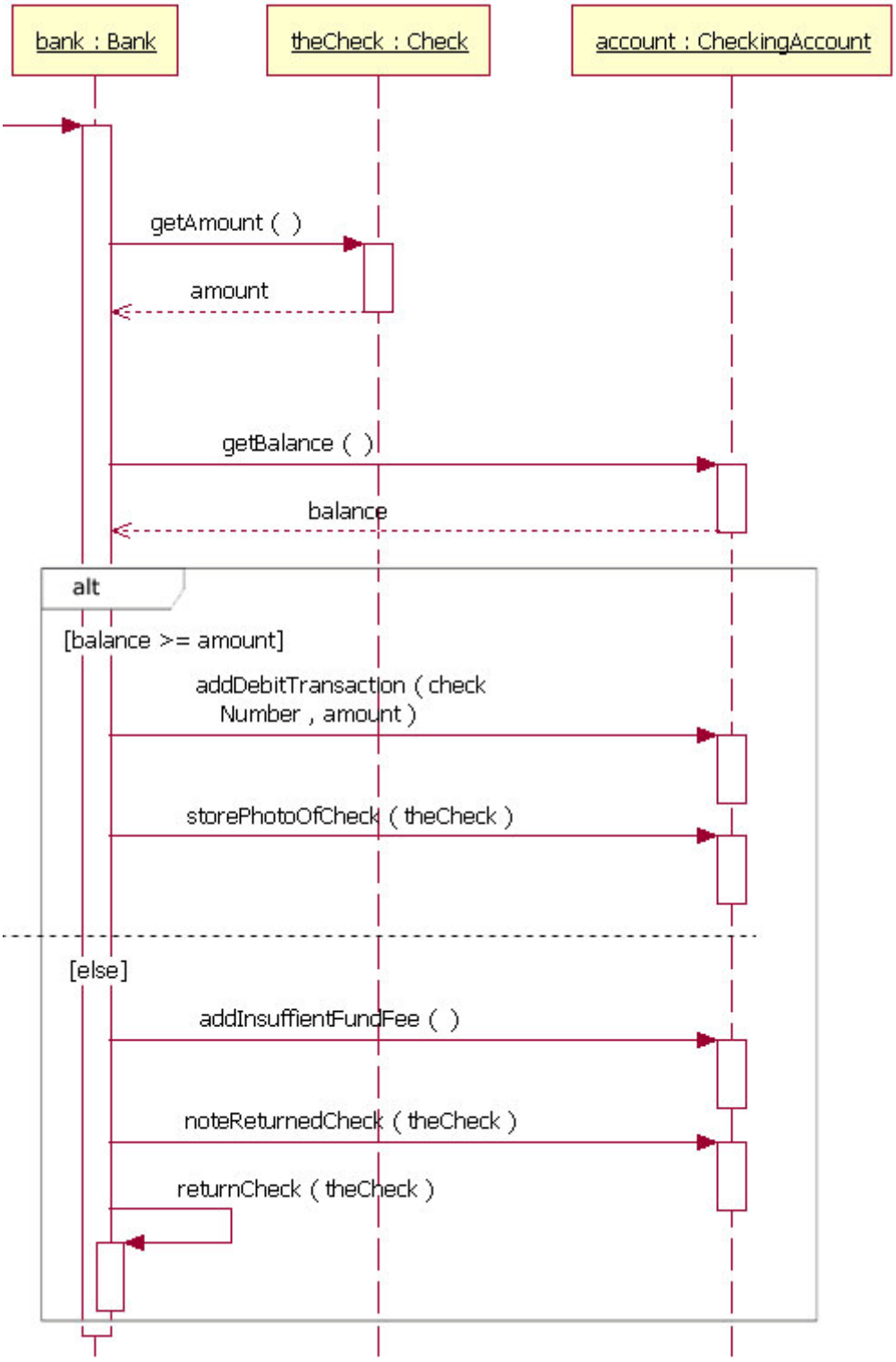


예제에서 가드는 텍스트 "[pastDueBalance = 0]" 이다. 이 메시지에 가드가 있기 때문에 addStudent 메시지는 시스템 계정이 [pastDueBalance = 0]을 리턴할 경우에만 보내진다.

5.6 대안

대안은 두 개 이상의 메시지 시퀀스들간 상호 배타적인 선택을 나타낼 때 사용된다. 대안은 전통적인 "if then else" 직 (만일 내가 세 개의 아이템을 구매하면 구매금액의 20%를 할인 받는다; 그 외에는 10%의 할인을 받는다.)의 모델링이 가능하다.

아래 그림에서 보듯, 대안 엘리먼트는 프레임을 사용하여 그려진다. "alt" 라는 단어는 이 프레임의 네임박스 안에 놓인다. 더 큰 직사각형은 피연산함수로 나누어진다. 피연산 함수는 대시(dash) 라인으로 분리된다. 각 피연산 함수에는 가드가 주어지고 이 가드는 lifeline 상단에 피연산 함수의 왼쪽 상단 부분을 향해 배치된다. 피연산함수의 가드가 "true,"로 되면 그 피연산함수를 따라야 한다.



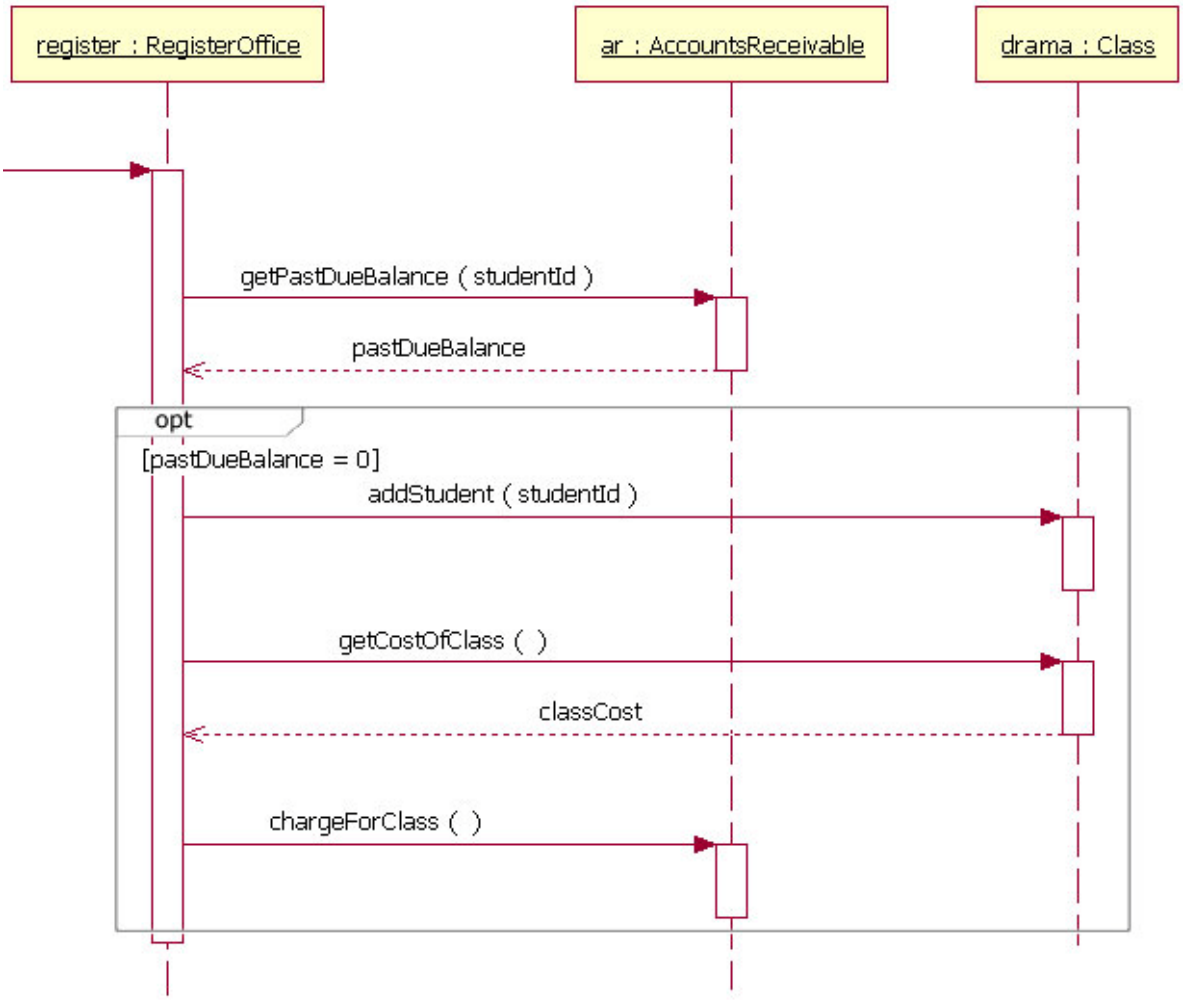
대안이 어떻게 읽혀지는지를 보여주는 예제로서 위의 그림은 상단에서 시작하는 시퀀스를 보여준다. check amount와 account의 balance 정보가 있는 bank 객체가 있다. 이 부분에서 대안이 사용된다. 가드 "[balance >= amount]" 때문에 account의 balance이 보다 크거나 같을 때 시퀀스는 addDebitTransaction과 storePhotoOfCheck 메시지를 account 객체로 보내는 bank 객체를 사용하여 시퀀스를 지속시킨다. 하지만 balance가 amount 보다 작거나 같을 때 시퀀스는 addInsuffientFundFee와 noteReturnedCheck 메시지를 account 객체로 보내고,

returnCheck 메시지를 자기 자신에게 보내는 bank 객체로 처리한다. "[else]" 가드 때문에 balance가 amount 보다 작거나 같을 때 두 번째 시퀀스가 호출된다. 대안을 사용하면 "[else]" 가드가 필요 없다. 하지만 피연산함수가 이것에 대한 명확한 가드를 갖고 있지 않다면 "[else]" 가드가 필요하다.

대안은 "if then else"에만 국한되지 않는다. 필요한 만큼 대안 경로를 취할 수 있다. 더 많은 대안이 필요하면 시퀀스의 가드와 메시지를 포함한 직사각형에 피연산함수를 추가하면 된다.

5.7 옵션

옵션 Combined Fragment는 특정 상황에서 발생하는 시퀀스를 모델링 할 때 사용된다. 다른 경우, 이 시퀀스는 발생하지 않는다. 이 옵션은 간단한 "if then"문장을 모델링 하는데 쓰인다. 옵션 표기법은 대안과 비슷하다. 단 한 개의 피연산 함수를 가져야 하고, "else" 가드가 전혀 없다는 것을 제외하고는 말이다. 옵션을 그리려면 프레임을 그려야 한다. "opt" 텍스트가 이 프레임의 네임박스 안에 배치되고, 이 프레임의 콘텐트 영역에 옵션의 가드가 lifeline의 상단에, 왼쪽 상단 코너를 향해 배치된다. 그런 다음 옵션의 메시지 시퀀스가 나머지 영역에 배치된다.

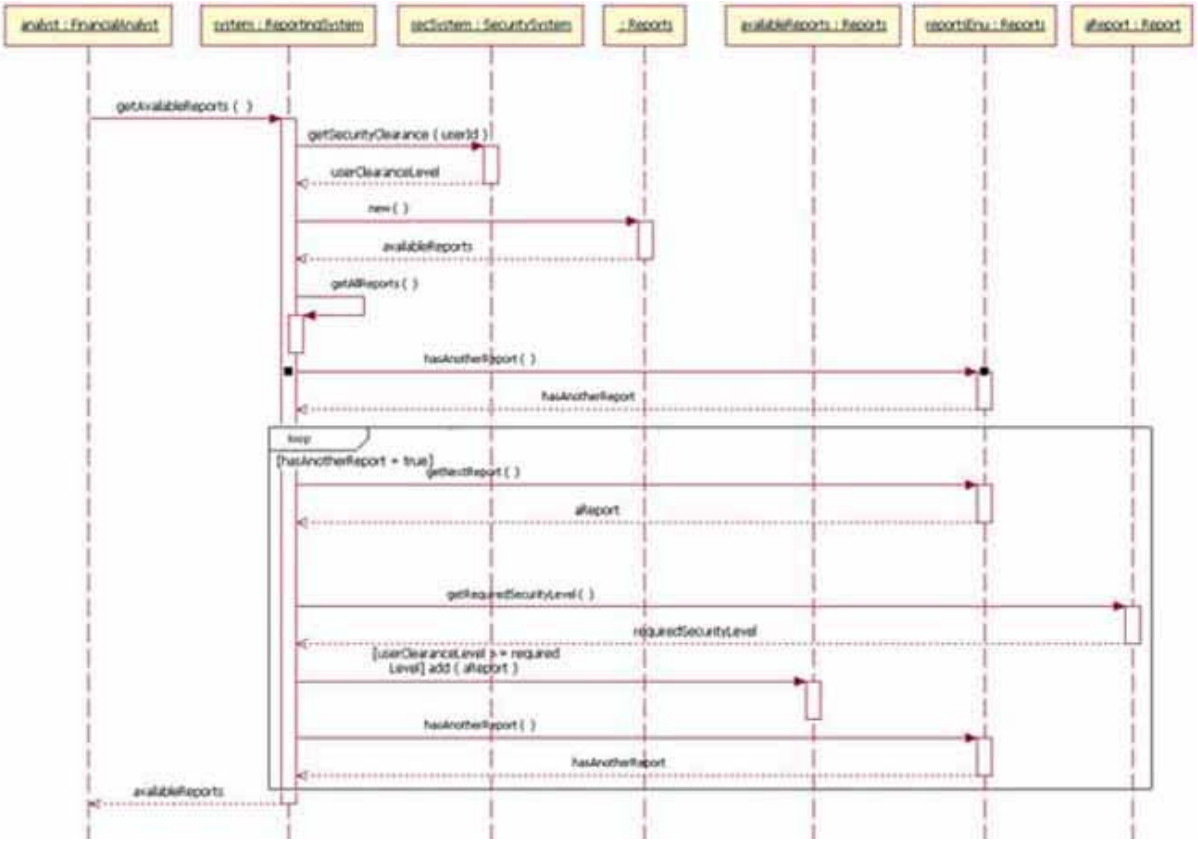


옵션 Combined Fragment는 읽기 쉽다. 위의 그림은 이전 예제의 시퀀스 다이어그램을 재구성 한 것이다. 하지만 여기에서는 student의 과거 해당 balance가 0일 경우 보내져야 하는

메시지가 더 많기 때문에 옵션을 사용한다. 위 예제의 시퀀스 다이어그램을 보면, student의 과거 balance가 0 이면 addStudent, getCostOfClass, chargeForClass 메시지들이 보내진다. student의 과거 balance가 0이 아니라면 시퀀스는 어떤 메시지도 보내지 않는다.
위 예제의 시퀀스 다이어그램에는 이 옵션용 가드가 포함되어 있다. 하지만 이 가드는 필수 엘리먼트는 아니다. 추상 시퀀스 다이어그램에서는 이 옵션의 조건을 지정한다. 이것이 옵션 fragment 라는 것을 가리키면 된다.

5.8 루프(loop)

가끔 반복적인 시퀀스를 모델링 해야 할 때도 있다. UML 2에서 반복되는 시퀀스의 모델에 루프 Combined Fragment를 사용한다.
루프는 외형상 옵션과 매우 흡사하다. 프레임을 그리고 그 프레임의 네임박스에 "loop"라고 쓴다. 프레임의 콘텐트 영역 안에서 루프의 가드는 lifeline의 상단에, 왼쪽 상단 코너 쪽을 향하여 놓인다. 그런 다음 루프의 메시지 시퀀스는 프레임의 나머지 콘텐트 영역에 배치된다. 루프에서 가드는 두 가지 특별한 조건을 가질 수 있다. 이 특별 가드 조건들은 "minint = [the number]" ("minint = 1")라고 하는 최소 반복과 and maximum iterations written as "maxint = [the number]" ("maxint = 5")라고 하는 최대 반복이다. 최소 반복 가드를 사용하여, 루프는 지정된 최소한의 수만큼 실행해야 하고 최대 또한 마찬가지이다.



이 예제에서, 루프는 reportsEnu 객체의 hasAnotherReport 메시지가 false를 리턴할 때까지 실행된다. 이 시퀀스 다이어그램의 루프는 루프 시퀀스가 실행되는지를 확인할 때 부울 테스트를 사용한다. 이 다이어그램은 위에서부터 읽어 내려간다. 루프에 다다르면 hasAnotherReport 값이 true 인지를 확인하기 위해 테스트가 실행된다. HasAnotherReport 값

이 true 면 시퀀스는 루프로 간다.

6. Activity Diagram

액티비티 다이어그램은 업무영역이나 시스템 영역에서 다양하게 존재하는 각종 처리로직이나 조건에 따른 처리흐름을 순서에 입각하여 정의한 모델이다. 액티비티 다이어그램은 하나의 액티비티에서 다음 액티비티로 순서가 바뀌면서 처리되는 과정을 표현하기 때문에 순서와 분기와 처리절차의 표현을 필요로 하는 대상에 대해 제한 없이 적용이 가능하다.
액티비티 다이어그램을 작성하는 목적과 용도는 다음과 같다.

- **대상에 상관없이 처리 순서를 표현하기 위해 작성한다.**
  액티비티 다이어그램은 액티비티와 액티비티의 순서를 표현할 목적으로 작성된다. 그 대상이 비즈니스 영역이든 시스템 영역이든 로직과 처리순서의 표현이 필요할 경우, 액티비티 다이어그램을 사용한다. 그래서 그 용도는 무척 다양하다. 시스템 관점에서 프로그램 사양을 작성하는 곳, 비즈니스 관점에서 영업사원의 영업업무 프로세스를 표현하는 곳에도 사용할 수 있다.
- **비즈니스 프로세스를 정의한다.**
  액티비티 다이어그램의 적용 영역에서 가장 훌륭하게 사용되는 대상 중의 하나는 비즈니스 프로세스의 분석이다. 시스템화 대상영역에 속한 현재 업무분야의 비즈니스 처리흐름을 표현(As-Is 프로세스 분석) 하거나 향후 변화된 비즈니스 처리 흐름(To-Be 프로세스 분석)을 작성할 수 있다.
- **프로그램 로직을 정의한다.**
  액티비티 다이어그램은 프로그램의 사양을 정의하는데 보조적으로 사용된다. 프로그램은 다양한 처리 흐름을 가지고 있다. 복잡한 처리 흐름을 자연언어로 기술하는 것은 부적절하다. 작성하는 과정도 어렵거니와 작성된 사양을 정확히 이해하기도 무척 힘이 든다. 액티비티 다이어그램은 처리 흐름을 도식화하여 간단하고 명료하게 처리로직을 표현함으로써 작성과 이해가 용이하다.
- **유즈케이스를 실현(Realization)한다.**
  프로젝트 초기에 정의된 유즈케이스는 프로그램으로 의해 구현되기 전에 설계되어야 한다. 유즈케이스를 액티비티 다이어그램을 이용해 실현하는 경우, 객체를 정의하거나 객체간 상호작용을 분석하는 형태가 아니라, 유즈케이스의 처리흐름을 순서도처럼 상세히 기술하는 형태로 작성된다. 그러나 이 경우 비슷한 용도로 작성되는 유즈케이스 정의서가 존재하기 때문에 액티비티 다이어그램으로 유즈케이스를 실현하는 것은 흔한 사례는 아니다.

6.1 작성시기

액티비티 다이어그램을 작성하는 시기는 그 적용 영역이 다양한 것처럼 한정되어 있지 않고, 다음의 시기에 작성될 수 있다.

- **업무 프로세스 정의 시점**
  비즈니스 프로세스를 정의하는 용도로 액티비티 다이어그램을 작성할 수 있다.

- **유즈케이스 정의서(Use case Description) 작성 시점**
  유즈케이스 정의서에서 유즈케이스의 처리절차를 기술하는 부분에 액티비티 다이어그램을 작성할 수 있다.
- **오퍼레이션 사양 정의 시점**
  클래스 오퍼레이션의 사양을 액티비티 다이어그램을 적용하여 작성할 수 있다.
- **기타**
  기타 처리흐름이나 처리절차가 필요한 시점이면 언제나 액티비티 다이어그램이 작성될 수 있다.

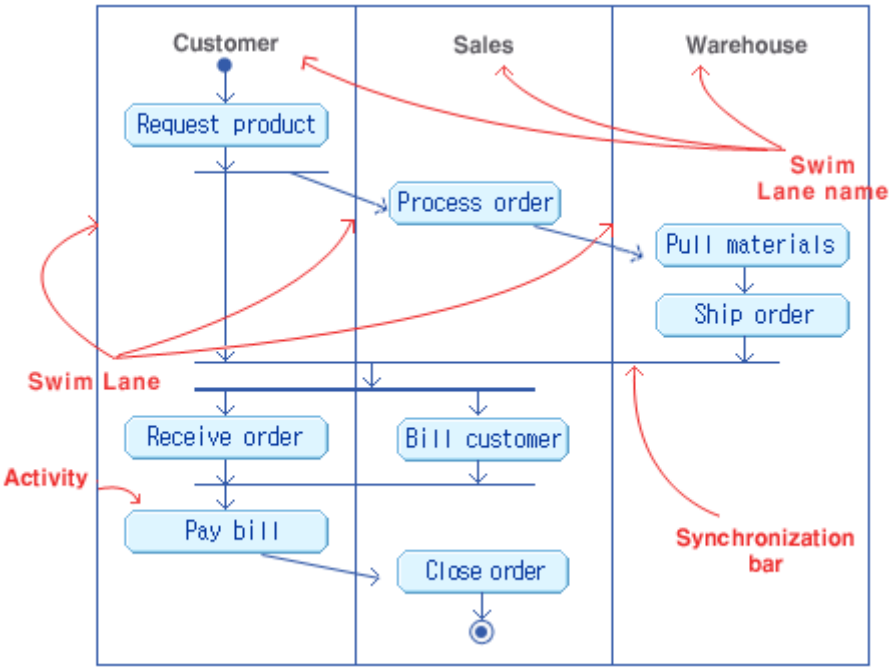
액티비티 다이어그램을 작성하기 위해서 준비물 및 선행과정은 특별히 없다.

6.2 액티비티 다이어그램의 구성요소

Things 혹은 심볼 : 액티비티(Activity), 시작점(Initial State), 종료점(Final State),
판단(Decision,Branch), Synchronization Bar

Relationships : 전이(Transition)

기타 요소 : Swim Lane



6.3 액티비티의 표기

액티비티는 모서리가 둥근 사각형으로 표기하며, 액티비티 명은 심볼 내에 표기한다.

액티비티의 정의 및 의미

- **액티비티는 행위나 작업을 의미한다.**
- **액티비티의 크기는 작성 대상에 따라 유동적이며, 한 액티비티 다이어그램에서는 액티비티의 크기가 균일한 것이 바람직하다.**

- **액티비티는 최소 단위가 아니며 내부적으로 구조를 가질 수 있는 단위이다.**
- **액티비티는 해당 작업의 종료 시점을 명확히 정의하기가 힘들다.**

- 시작점, 종료점의 표기

시작점과 종료점은 원 모양으로 표기하는데, 시작점은 속이 꽉 채워진 원으로, 종료점은 속이 채워진 원에 바깥의 또 다른 원이 둘러싸고 있는 모양으로 표기한다.

표기 예) ● : 시작점

표기 예) ⊙ : 종료점

- 시작점, 종료점의 정의

시작점은 처리흐름이 시작하는 곳을 의미한다. 모든 처리흐름은 시작점으로부터 개시되어 전개된다.

종료점은 처리흐름이 종료하는 곳을 의미한다. 모든 처리흐름은 종료점에서 처리흐름을 완료한다.

- 판단(Decision)의 표기

판단은 속이 빈 마름모꼴로 표기하며, 명칭이나 기타 장식이 붙지 않는다.

표기 예) ◇

- 판단(Decision)의 정의

판단은 분기가 일어나는 곳이다. 논리식의 결과 값에 따라 두 곳 이상의 흐름으로 분기가 일어난다. 처리 흐름은 논리식의 결과에 따라 처리 흐름이 나누어져 전개되는 것은 매우 흔하게 일어나는 일이다.

- Synchronization Bar의 표기

두꺼운 실선으로 표기하며, Synchronization Bar는 대부분 수평선으로 표기되나, 수직선으로 표기할 수도 있다.

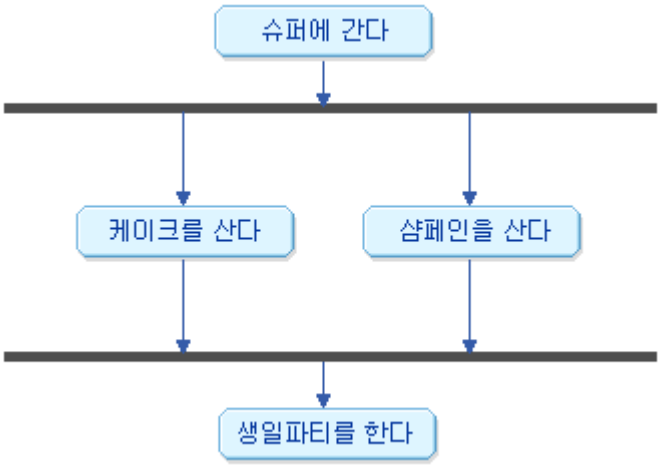
- Synchronization Bar의 의미

Synchronization Bar는 병렬 처리절차가 시작되거나 모이는 곳이다.

종종 둘 이상의 처리 절차가 그 수행순서에 상관없이 병렬로 진행될 경우가 있다.

Synchronization Bar로부터 분기해서 다음 Synchronization Bar로 모일 때까지의 처리 절차

는 병렬로 수행된다. Synchronization Bar에 이어진 액티비티가 수행되기 위해서는 병렬로 수행되는 Synchronization Bar상의 모든 처리절차가 끝나야 한다.



- **전이(Transition)의 표기**

화살표가 달린 실선으로 표기하며, 액티비티의 배치에 따라 수평선이나 수직선으로 표기한다.

- **전이(Transition)의 의미**

전이(Transition)는 하나의 액티비티가 행위를 완료하고 다른 액티비티로 처리순서가 옮겨지는 제어 흐름을 표현한다. 하나의 액티비티에서 여러 개의 전이(Transition)가 나가기도 하고, 들어오기도 한다.

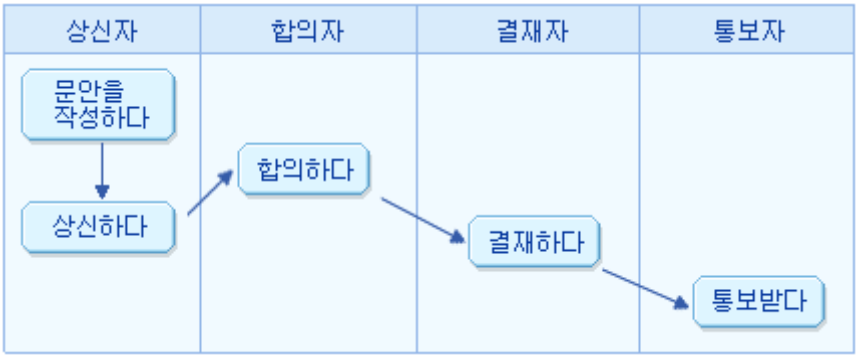
- **Swim Lane의 표기**

Swim Lain은 영역으로 표현을 하며, 액티비티 다이어그램의 제일 위쪽에서 아래쪽까지 수직방향으로 공간을 구분하는 방식으로 표현한다. 이것은 그 모양이 마치 실내 수영장의 트랙(swim lane) 같다고 해서 swim lane이라는 이름이 붙여졌다.

- **Swim lane의 의미**

Swim lane은 여러 가지 용도로 쓰일 수 있다. 업무 조직의 구분일 수 있고, 개인의 역할에 따른 구분이기도 하다. Swim lane의 영역 내에 정의된 액티비티는 그 Swim lane이 관장하고 ownership을 가진다.

Swim lane을 표현함으로써 누가(swim lane) 무엇을 한다(액티비티)라는 식의 표현이 가능해진다.



6.4 작성순서

- **작성 대상 선정**

  액티비티 다이어그램의 작성 대상을 선정한다. 대부분의 경우 액티비티 다이어그램은 업무 프로세스를 모델링하거나, 오퍼레이션 사양을 정의하는 용도로 사용된다.

- **Swim lane 정의**

  대상영역에 명확한 역할을 정의할 수 있을 경우,역할을 식별하여 swim lane으로 표현한다. swim lane은 필수적으로 정의해야 하는 것은 아니다.

- **처리 절차 모델링**

  처리 절차를 모델링 할 경우 시작점과 끝점이 표현되어야 하고 처리흐름이 도중에 끊겨 미아상대가 되지 말아야 한다.

6.5 주의사항

- **해당 부분을 이해하는데 필수적인 요소들만 표현한다.**

  모델에서 장황한 부가 요소를 포함하는 것은 바람직하지 않다. 분석 대상의 본질을 이해하는 데 꼭 필요한 요소들만 모델에 정의하는 것이 좋다.

- **추상화 수준에 맞는 상세성을 일관되게 제공한다.**

  모든 모델이 마찬가지이지만, 한 장의 모델에는 동일한 상세화 레벨이 유지되어야 한다. 서로 다른 추상화 레벨의 액티비티들이 섞여 있으면 의미를 파악하기 힘들게 된다.

- **중요한 의미를 이해하기 적절한 단위로 표현한다.**

  액티비티의 크기는 일정해야 한다. 서로 완전히 다른 단위의 액티비티가 섞여 있을 경우 모델의 완전성을 도모할 수 없다.

- **목적을 전달할 수 있는 명칭의 부여한다.**

  액티비티 명칭을 비롯해 쓰이는 모든 명칭들은 명확한 표현을 사용해야 한다. 모호한 명칭으로 정의되면 혼란만 야기 시키는 결과를 초래한다.

- **주 흐름으로부터 시작하여 전이, 분기, 동시성을 표현한다.**
- **교차선이 최소화 하도록 요소를 배치해야 한다.**
- **중요한 부분은 Note, Color 등을 이용하여 시각적 효과를 사용하면 좋다.**

7. Deployment Diagram

디플로이먼트 다이어그램은 시스템을 구성하는HW 자원 간의 연결 관계를 표현하고, HW 자원

에 대한SW 컴포넌트의 배치 상태를 표현한 다이어그램이다. 그리고 디플로이먼트 다이어그램은 시스템의 설계 단계의 마지막에 작성한다. 즉, 모든 설계가 거의 마무리되어 SW 컴포넌트가 정의되고, 시스템의 HW 사양도 확정된 후 디플로이먼트 다이어그램이 작성될 수 있다.
디플로이언트 다이어그램 작성하는 목적은 다음과 같다.

- SW시스템이 배치, 실행될 HW자원들을 정의한다.

디플로이먼트 다이어그램은 다른 UML 다이어그램들과는 달리 HW자원들을 명시적으로 정의하는 용도로 작성된다. 그러나 이렇게 HW를 정의하는 목적이 HW 자체의 사양을 정의하고 설명하기 위한 것은 아니다. 오히려 SW 시스템이 탑재되어 동작하는 매개체로서, HW자원을 정의한다라는 관점에서 정의한다.

- SW 컴포넌트가 어떤 HW 자원에 탑재되어 실행될지 정의한다.

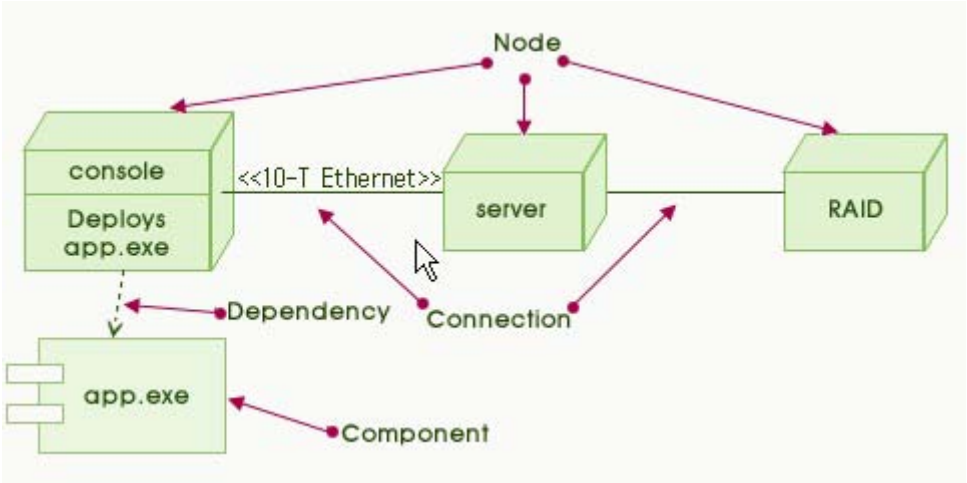
디플로이먼트 다이어그램은 실행모듈(컴포넌트)을 분산된 HW자원에 적절히 배치하여 원하는 성능과 효율을 낼지를 정의하는 목적으로 작성된다. 따라서 디플로이먼트 다이어그램에는 SW자원과 HW자원이 동시에 표현된다.

- HW 자원의 물리적인 구성을 정의한다.

SW컴포넌트가 탑재된 HW자원들은 적절한 성능을 내기 위해 물리적인 연결을 가지고 있어야 한다. 디플로이먼트 다이어그램은 어떤 HW자원간에 연결이 있는지, 그 연결은 어떠한 성능을 가진 연결인지를 정의한다.

7.1 구성요소

디플로이먼트 다이어그램의 구성요소는 다음과 같다.
Things 혹은 심볼 : 노드(Node), 컴포넌트(Component)
Relationships : Connection, Dependency



7.1.1 Node

Node는 직육면체로 표기하며, Node Name은 심볼 내에 표기한다.

Node는 SW 컴포넌트가 탑재되어 처리되는데 관련된 HW 자원을 의미한다. 주로 연산능력(computing power)이 있는 HW 즉, SW를 탑재하여 운용할 수 있는 능력을 가진 하드웨어가 표현된다. 그러나 표현할 수 있는 HW 자원의 종류가 제한된 것은 아니고, 아래와 같은 다양한 장비들이 노드로 정의될 수 있다.

[HW 장비들의 예]

Sensor, Printers ,Card readers, Communication devices, Mechanical processing resources

[Node의 예]

Web Server, DB Server

7.1.2 Component

Component는 탭이 달린 직사각형으로 표기하며, Component Name은 심볼 내에 표기한다. Component는 독립적으로 배포되고 교체되며 재사용될 수 있는 SW조각를 의미한다. 보통의 경우 실행모듈을 말하지만, 실제 통용되는 Component라는 용어는 항상 실행모듈만을 가리키지는 않는다. 컴포넌트가 가끔은 아주 광의로 사용되어서 소스코드나 UI(User Interface), 분석, 설계 산출물들을 포함한 것을 의미하기도 한다. 컴포넌트라는 용어의 의미는 문맥에서 말하는 사람의 의도를 생각해서 받아 들여야 한다.

[Component의 예]

결재 시스템에서 결재, 사원 등, 전자 상거래 시스템에서 우편번호 검색, 신용카드 결재 등

7.1.3 연결

- Connection의 표기

[표기 예]

<<100-T Ethernet>>

Node를 연결하는 실선으로 표기하며, 연결의 물리적 특성을 Stereo type으로 표기할 수 있다.

- Connection의 정의

두 Node 사이의 물리적인 연결을 의미한다. 두 노드 사이의 물리적인 연결 특성을 설명한다.

7.1.4 의존관계

- Dependency의 표기

[표기 예]

점선 화살표로 표현하고 필요에 따라 선 위에 설명을 붙이기도 한다.

- Dependency의 정의

객체나 컴포넌트가 다른 객체나 컴포넌트의 실행을 요청하는 경우, 즉 사물간의 실행 혹은 참조관계를 표현한다.

Class와 Class, Package와 package, Component와 Component에 주로 사용되는 관계이고, 때로는 Class-Package-Component 상호 간에도 사용되는 관계이다.

7.2 작성순서

- 노드 식별 및 정의

디플로이먼트 다이어그램을 작성할 때 시스템의 운영을 위한 HW자원을 식별하고 그 사양을 확인하는 것을 가장 먼저 수행한다. 일반적으로 프로젝트 수행 초기에 시스템 청사진(System Architecture)을 작성하는 것이 일반적인데, 이를 활용하여 HW자원을 식별한다.

- 컴포넌트 식별

디플로이먼트 다이어그램에 등장할 컴포넌트를 정한다. 컴포넌트 다이어그램이 정의되어 있을 경우, 이를 활용하면 쉽게 수행할 수 있다.

- 노드 간 구성관계 정의

디플로이먼트 다이어그램에 노드를 배치하고 노드간의 물리적 연결인 연결을 정의한다. 연결과 노드에는 Stereo type으로 하드웨어적 특성을 표현한다.

- 노드에 컴포넌트 배치

정의된 노드와 연결을 고려하여 어떤 노드에서 컴포넌트를 실행하게 될 것인가를 정의한다. 디플로이먼트 다이어그램에 SW컴포넌트들의 배치상황을 반영한다.

7.3 주의사항

- 목적을 전달할 수 있는 명확한 의미의 명칭을 부여해야 한다.

노드 명과 스테레오 타입으로 정의하는 하드웨어 특성등은 표현 방식에 기준이 없다. 하지만 시스템과 관련없는 제 3 자가 보더라도 그 의미를 이해 할 수 있게 쉽고, 명확한 용어를 사용하여 명칭을 정의하야 한다. 모호한 명칭으로 정의하면 혼란만 야기 시키는 결과를 초래한다.

- 문제 영역의 H/W에 대한 명쾌한 추상 개념을 제공하도록 작성한다.

SW 자원이 탑재되어 운영되는 보조적인 용도 뿐 아니라, 디플로이먼트 다이어그램은 시스템의 하드웨어 구성을 개념적으로 보여주는 훌륭한 도구가 된다. 이러한 용도를 살려 HW 자원의 구성에 대한 좋은 모델이 되도록 정의한다.

- Model을 만든 목적을 전달하기에 필요한 수준까지만 분해한다.

디플로이먼트 다이어그램에 모든 HW 장비가 나타날 필요는 없다. 오히려 이러한 시도는

다이어그램을 장황하고 복잡하게 만들어서 의미를 파악하기 힘들게 한다. 목적과 용도에 부합하는 요소들만 정의하면 충분하다.

8. Component Diagram

컴포넌트 다이어그램은 시스템의 구현관점에서 실행모듈(컴포넌트)을 정의하고 실행모듈간의 정적 상호작용을 정의한 모델이다. 이 다이어그램은 시스템이 어떠한 물리적 구성요소들로 -실행모듈(컴포넌트)- 구성되고 그들간의 연관성을 정의한 것이다.

컴포넌트는 매우 광범위한 의미로 사용되는 용어인데, SW 분야에서 사용되는 컴포넌트를 정의하면 다음과 같다.

Reusable application building block.

: 컴포넌트는 시스템의 재사용 가능한 구성요소입니다.

Physically replaceable or upgradeable parts of a system

: 컴포넌트는 시스템의 교체단위이자 업그레이드 단위입니다.

An independently deliverable piece of functionality providing access to its services through interfaces

: 컴포넌트는 인터페이스를 통해 그 기능이 사용되어지는, 독립적으로 인도되는 기능조각입니다.

컴포넌트 다이어그램을 작성하는 목적은 다음과 같다.

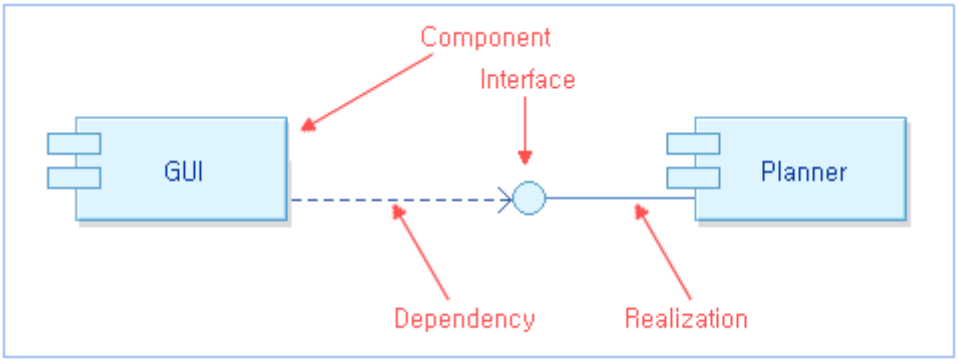
- **시스템의 실행모듈(컴포넌트)들을 정의한다.**
  컴포넌트 다이어그램은 시스템이 구축될 때, 어떤 실행모듈(컴포넌트)들로 구축될 것인지를 정의하는 용도로 사용된다. 이러한 컴포넌트는 독립적으로 배치, 교체가 가능한 단위이다. 개발 플랫폼에 따라 이러한 실행모듈의 특성은 달라진다.
- **컴포넌트간 Dependency를 정의한다.**
  컴포넌트 다이어그램은 실행모듈(컴포넌트)간의 정적인 상호작용을 정의하는 용도로 사용된다. 컴포넌트 사이의 종속관계를 표현함으로써 실행 시 상호 참조하는 연관성을 표현한다.
- **실행모듈뿐 아니라 소스코드, 데이터베이스 등의 상호작용을 모델링 한다.**
  컴포넌트 외에 소스코드나 데이터베이스 등 조각으로 나누어 정의할 수 있는 대상들에 대해, 그 대상들의 상호작용을 정의하기도 한다. 그러나 이런 용도로 사용되는 경우는 흔치 않다.

컴포넌트 다이어그램은 시스템의 설계단계의 막바지에 작성한다. 즉, 모든 클래스가 물리적으로 완전히 정의되고, 그 상호관계도 정의된 후 컴포넌트 다이어그램이 작성될 수 있다.

8.1 컴포넌트 다이어그램의 구성요소

Things 혹은 심볼 : 컴포넌트(Component), 인터페이스(Interface)

Relationship : Dependency, Realization



- 컴포넌트의 표기

컴포넌트는 탭이 달린 직사각형으로 표기하며, 컴포넌트 명은 심볼 내에 표기한다.

- 컴포넌트의 정의

컴포넌트는 독립적으로 배포되고 교체되며 재사용될 수 있는 SW조각를 의미한다. 보통의 경우 실행모듈을 말하지만, 실제 통용되는 컴포넌트라는 용어는 항상 실행모듈만을 가리키지는 않는다.

컴포넌트가 가끔은 아주 광의로 사용되어서 소스코드나 UI(User Interface), 분석, 설계 산출물들을 포함한 것을 의미하기도 한다. 컴포넌트라는 용어의 의미는 문맥에서 말하는 사람의 의도를 생각해서 받아들여야 한다.

컴포넌트의 예

※컴포넌트는 매우 다양한 크기로 정의되며. 아래 예들은 한정된 컴포넌트의 사례일 뿐이다.

결재 시스템 : 결재, 사원 등

전자 상거래 시스템 : 우편번호 검색, 신용카드 결제 등

- 인터페이스의 표기

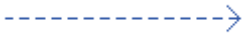
인터페이스는 두 가지 형태로 표기가 가능하다. 하나는 icon형태의 표기로 원으로 표현하는데, 이 경우 인터페이스 명은 아래쪽에 표기한다. 다른 하나는 보통의 클래스에 <<>>라는 스테레오 타입이 부가된 표기이다.

- 인터페이스의 정의

Interface는 Class의 일종이다. interface는 class나 Component의 기능을 외부에 공개할 목적으로 쓰이며, 구현은 하지 않는다. interface의 구현은 클래스나 컴포넌트에서 하게 되며,

이 클래스는 interface를 상속하여 단지 선언뿐인 interface의 구현을 담당한다. Interface는 단독으로 표시되는 경우는 거의 없으며 해당 Interface를 구현하는 Class나 Component에 붙어 다닌다.

- Dependency의 표기



점선 화살표로 표현하고 필요에 따라 선 위에 설명을 붙이기도 한다.

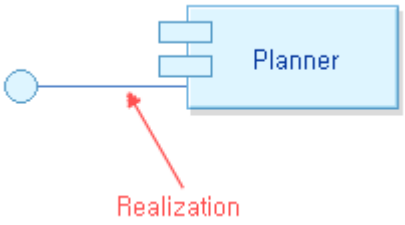
- Dependency의 정의

객체나 컴포넌트가 다른 객체나 컴포넌트의 실행을 요청하는 경우, 즉 사물 간의 실행 혹은 참조관계를 표현한다.

Class와 Class , Package와 package , Component와 Component에 주로 사용되는 관계이고, 때로는 Class-Package-Component 상호 간에도 사용되는 관계이다.

- Realization의 표기

속이 빈 삼각형의 화살표가 한쪽에 달린 점선으로 표현한다. 그러나 특별히 컴포넌트 다이어그램에서는 인터페이스와 컴포넌트간의 실선으로 표현된다.

Realization의 표기	인터페이스와 컴포넌트 간 표기
	

- Realization의 정의

정의하는 사물과 이를 구현하는 사물 간에 표현하는 관계이다. Realization은 인터페이스(정의) - 컴포넌트(구현), 유즈케이스(정의) - 컬레버레이션(구현)과 인터페이스(정의) - 클래스(구현) 사이에 허용되는 관계이다.

삼각형이 붙은 쪽이 정의하는 사물, 반대쪽이 구현하는 사물이다.

8.2 작성순서

- 컴포넌트 대상 정의

컴포넌트 다이어그램을 그리기 전에 무엇을 컴포넌트로 표현할지 클래스를 구성요소로 하는 실행모듈로 할지, 소스코드를 정의할 지 기타 무엇을 컴포넌트로 표현할 지를 정해야 한다.

- 컴포넌트 식별

컴포넌트 다이어그램에 등장할 컴포넌트를 정한다. 소스 파일일 경우 그 대상은 쉽게 식별되지만 실행모듈일 경우 간단치 않다. 여러 가지 가능한 방법으로 컴포넌트를 식별해 내는

작업을 수행 한다.

- **컴포넌트 배치 및 인터페이스 정의**

  컴포넌트 다이어그램에 컴포넌트를 배치하고 이름을 정의한다. 그리고 인터페이스를 정의할 필요가 있을 경우 인터페이스를 정의하고 컴포넌트와 realization관계로 연결한다.

- **Dependency 정의**

  컴포넌트와 컴포넌트간 의존관계를 분석하여 Dependency 관계를 정의한다.

8.3 주의사항

- **컴포넌트는 응집도는 높고 결합도는 낮은 단위로 정의되어야 한다.**

  실행모듈로서의 컴포넌트를 식별할 때, 컴포넌트는 다른 컴포넌트와 독립적이고, 기능 차별성을 갖추는 단위로 정의되어야 한다. 즉, 기능 측면에서 컴포넌트 내부는 강한 유사성을 갖는 단위들로 구성되어야 하고(높은 응집도), 다른 컴포넌트에 강하게 종속되지는 않는(낮은 결합도) 단위로 정의되어야 한다.

- **컴포넌트 크기(Granularity)의 일관성을 고려해야 한다.**

  한 시스템에서 컴포넌트의 크기에 너무 차이가 나면 바람직하지 않다. 컴포넌트의 크기는 기술구조와 시스템 특성들이 고려되어 적절한 크기로 정의해야 하며, 그 크기도 되도록 많이 차이 나지 않도록 하는 것이 좋다.

- **추상화 수준에 맞는 상세성을 일관되게 제공한다.**

  모든 모델이 마찬가지지만, 한 장의 모델에는 동일한 상세화 레벨이 유지되어야 한다. 서로 다른 추상화 레벨의 컴포넌트가 섞여 있으면 의미를 파악하기 힘들게 된다. 소스와 실행모듈을 한 장에 정의한 컴포넌트는 좋은 예가 아니다.

- **목적을 전달할 수 있는 명칭을 부여해야 한다.**

  컴포넌트,인터페이스를 비롯해 쓰이는 모든 명칭들은 명확한 표현을 사용해야 한다. 모호한 명칭으로 정의하면 혼란만 야기 시키는 결과가 된다.