UML 사용 보고서
- 소프트웨어모델링및분석 레포트

201011334 박진성
201011338 송병우
201013759 근량

Index

1. UML
    1.1. 의미
    1.2. 기능
    1.3. 구조
    1.4. Structure Diagram
    1.5. Behaviour Diagram
2. Collaboration Diagram
    2.1. Semantics
    2.2. Notation
        2.2.1. Collaboration Instance
        2.2.2. Collaboration
    2.3. Example
    2.4. Mapping
3. Activity Diagrams
    3.1. Semantics
    3.2. Notation
    3.3. Example
    3.4. Mapping
4. 활동 상태(Action State)
    4.1. Semantics
    4.2. Notation
    4.3. Presentation options
    4.4. Example
    4.5. Mapping
5. 하위 상태
    5.1. Semantics
    5.2. Notation
    5.3. Example
6. 컴포넌트
    6.1. 인터페이스
    6.2. 컴포넌트 다이어그램 작성 방법
        6.2.1. 포트 생성
        6.2.2. 제공 인터페이스(Provided Interface) 작성
        6.2.3. 요청 인터페이스(Required Interface) 작성
    6.3. 컴포넌트 모델과 UML 과의 대응관계
    6.4. 컴포넌트 사이의 연결 관계 및 구현 관계

7. Use Case Diagram
7.1. Use Case와 Actor 그리고 이들간의 관계 표현
7.2. 구성요소
7.2.1. Actor
7.2.2. Use Case
7.2.3. Actor와 Use Case의 관계
7.2.4. Use Case와 Use Case 관계
7.3. Use Case Specification
7.3.1. Actor와 Use Case 간의 상호 작용에 대한 기술 문서
7.3.2. 구성요소
7.3.3. Use Case 이름
7.3.4. Use Case 간략한 설명
7.3.5. Event 흐름
7.3.6. 특별한 요구사항
7.3.7. 사전 조건
7.3.8. 사후 조건
7.3.9. 확장점
7.3.10. System 관련 그룹
7.4. Use Case Diagram
7.4.1. include
7.4.2. exclude
7.4.3. Generalization
7.5. Use Case의 사례
8. Class Diagram
8.1. 의미
8.2. 구성
8.3. 연결
8.4. Object diagram과의 차이
8.5. Classifier
8.6. Class
8.6.1. 의미
8.6.2. 표현명
8.6.3. 용법
8.7. 사용 예
9. Deployment Diagram
9.1. 구성요소
9.1.1. Graphical node
9.1.1.1. artifact
9.1.1.2. node
9.1.1.3. deployment

9.1.2. Graphical path
9.2. 용법
9.3. 사용 예

10. Sequence Diagram

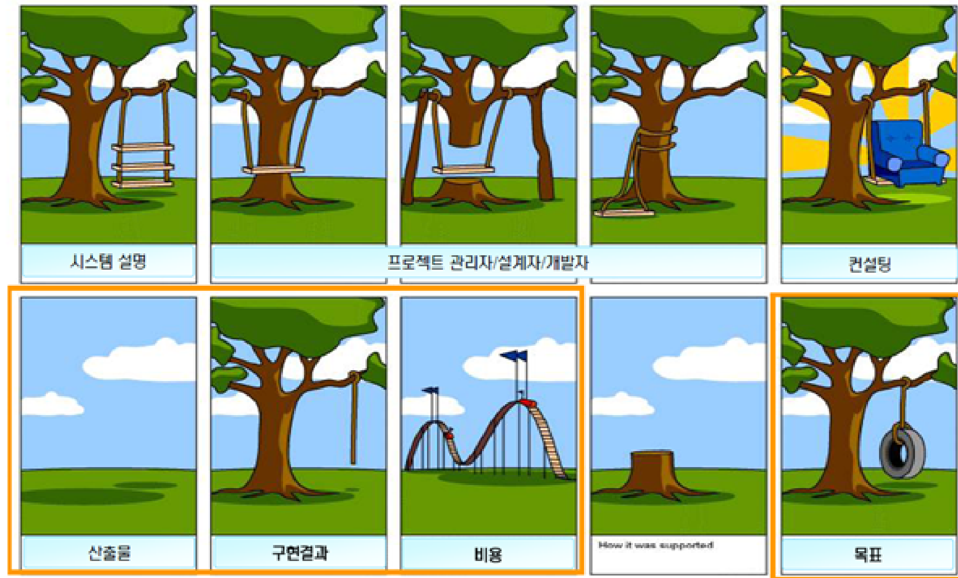
10.1. 구성요소

10.1.1. Object Lifeline

10.2. 용법

10.3. 사용 예

1. UML



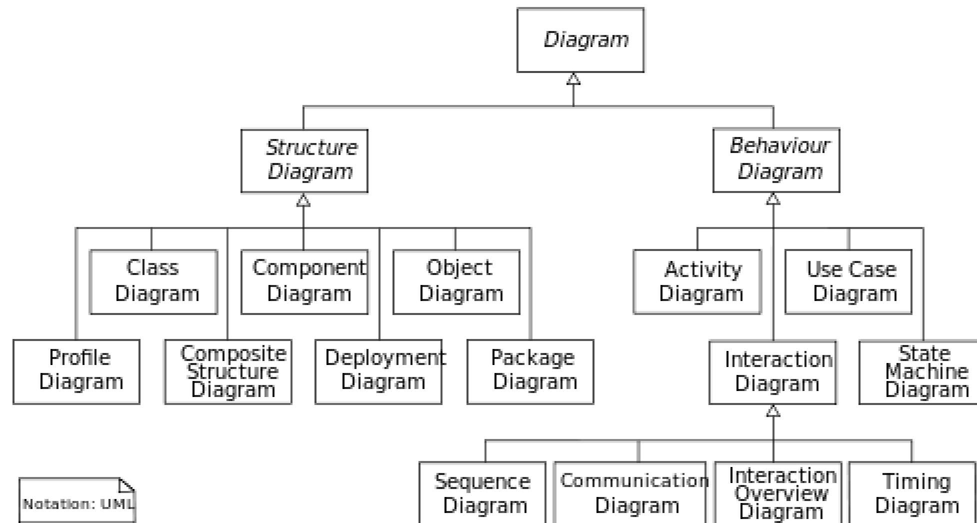
1.1. 의미

객체지향 소프트웨어를 모델링 하는 표준 그래픽 언어
1980년대 말부터 1990년대 초에 객체지향으로 모델링 하는 과정과 모델링 언어 출현
설계와 표현 방법의 급증으로 혼란을 초래
1994년 Verdix사에 합병된 Rational사에서 Ada와 C++로 개발되어 있던 환경을 Verdix 사의 코드 생성기와 런타임으로 병합.
1995년 Rational사의 세 명의 객체지향 전문가가 서로의 일을 통합하기 위해 UML을 개발했다. 1.0 버젼이 Object Management Group(OMG)에 제출되었고, 후속 개발이 추진되었다.
1997년 OMG가 UML 표준화를 추진하였고
2005년 ISO/IEC에 1.4.2 버젼이 채택되어 국제 표준으로 제정되었다.

1.2. 기능

자세한 의미(semantic) 표현
확장 메커니즘
관련 텍스트 언어 Object Constraint Language(OCL)
UML의 목적은 소프트웨어 설계 방법론이 아니라 표현을 도와주는 것이다.

1.3. 구조 (v2.2)



1.4. Structure Diagram

시스템의 설계에서 꼭 존재해야만 하는 것을 강조하여 표현한다. 구조를 표현하므로 SA에서 주로 사용된다.

1.5. Behaviour Diagram

설계된 시스템에서 어떤 일을 해야만 하는가를 강조하여 표현한다. 시스템의 기능을 표현해야 할 때에 따로 사용된다.

아래에서는 ISO 표준에 의해 전통적으로 주요하게 사용되는 다이어그램을 설명한다.

2. Collaboration Diagram

2.1. Semantics

Collaboration diagram은 인스턴스(Instance)들의 수행되어 지는 역할과 특정한 상황 속에서 요구하는 관계들의 집합이 포함된 Collaboration의 한 측면을 표현하거나 인스턴스(Instance)들과 그것 들의 관계의 집합과 함께 CollaborationInstanceSet을 표현한다. 그 다이어그램은 원하는 결과를 달성하기 위한 역할을 수행하는 인스턴스들 사이에서 특정한 메세지(Stmuli)들의 집합을 정의하는 InteractionInstanceSet을 표현하기도 한다.

하나의 Collaboration은 해당 작동(Operation)이나 특성(Classifier)의 구현(realization) 설명을 위해 사용되어 진다. Use-case와 같이, 하나의 특성(Classifier)을 설명하는 Collaboration은 대개 특성(Classifier)들과 관계(Association)들을 참조한다. 반면에, 작동(Operation)을 설명하는 Collaboration은 해당 작동(Operation)의 인자(Argument)들과 지역 변수(local variable)들을 포함하고 보통 관계(Association)들은 그 작동(Operation)이 갖고 있는 특성(Classifier)을 연결한다.

2.2. Notation

어떤 collaboration diagram은 서로 연결되어진 인스턴스(Instance), 특성 역할(Classifier Role), 관계 역할(Association Role)의 한 측면을 그래프로 보여준다. 이러한 것은 상호작용(Interation) 또는 상호작용 집합(InterationInstanceSet)에 의해서 정해진 커뮤니케이션을 포함할 수 있다.

왜냐하면 collaboration diagram은 때때로 프로시저(Procedure)들을 설계하는 데에 도움을 주기 위해 사용되어 진다. 그것 들은 흔히 링크(Links) 또는 관계 역할(Association Roles)들을 표현하는 데 화살표를 이용해서 방향을 보여준다.(링크 또는 관계 역할을 나타내는 박스들 사이에 한 방향을 가리키는 하나의 화살표.)

상호작용(Interation)의 순서는 보통 1로 시작하는 일련의 숫자와 함께 설명된다. 하나의 절차 적인 작동 흐름을 위해서, 하위 커뮤니케이션 숫자들을 중첩 호출과 함께 중첩된다. 동시에 여러 인스턴스(Instance)들의 상호작용을 하는 비절차적 배열을 위해, 모든 해당 일련의 숫자들은 같은 레벨에 있다(이 숫자들은 중첩되지 않는다).

어떠한 상호작용도 갖지 않는 Collaboration diagram은 상호작용들이 발생 할 수 있는 상황을 보여준다.

또, Single Operation 이나 모든 클래스(Class) 또는 클래스들의 그룹을 위한 상황 조차도 보여줄 수 있다.

이 표준은 아래 조건의 실시 하에 인스턴스, 링크의 생성과 파괴의 여부를 보여주는데 사용 될 수 있다.

- 인스턴스(Instance)와 링크(Link)들은 { new }로 설계 될 수 있는 것이 실행되는 동안 만들어진다.
- 인스턴스(Instance)와 링크(Link)들은 { destroyed }로 설게 될 수 있는 것이 실행되는 동안 파괴된다.
- 인스턴스(Instance)와 링크(Link)들은 { transient }로 설계 될 수 있는 실행 되고 파괴 되는 동안 만들어진다.

이러한 것들은 인스턴스(Instance)들간의 상세한 상호작용으로부터 끌어낼 수 있는 생명의 상태에 한해서 변경한다. 그것 들은 기호의 편의가 제공된다.

2.2.1. Collaboration Instance

인스턴스 레벨에서 주어지는 하나의 collaboration diagram은 한 CollaborationInstanceSet을 보여준다. CollaborationInstanceSet이란, 각기 인스턴스와 링크이 오브젝트 박스와 선으로 매핑되는 컬렉션을 뜻한다. 이러한 인스턴스들은 해당 CollaborationInstanceSet의 Collaboration의 특성 역할(Classifier Role)과 관계 역할(Association Role)들을 맞게 한다. 또한, 해당 다이어그램은 링크(Link)를 통해 전달 되어지는 자극(Stimuli)과 일치하는 선들을 연결하는 화살표를 포함한다. 그 다이어그램은 성능에 간접적으로 영향을 미치거나 접근하는 인스턴스(Instance)들이 포함하는 어떤 동작(Operation) 또는 특성(Classifier)의 구현과 관련된 인스턴스(Instances)들을 보여준다. 그 다이어그램은 인스턴스(Instance)들 간의 링크(Link)들을 프로시저의 인자들, 지역 변수들, 자기 자신의 링크들을 표현하는 것을 일시적으로 포함하여 보여준다. 개개의 속성 값들은 보통 명시적으로 보여지지 않는다. 만약 자극(Stimuli)가 속성 값들에게 보내져야만 한다면, 그 속성(Attribute)들은 관계

대신 사용할 수 있도록 설계 되어야만 한다.

2.2.2. Collaboration

특정 레벨에 주어진 어떤 collaboration diagram은 collaboration 속에 역할 들이 정의된 하나의 Collaboration을 보여준다. 함께, 이러한 역할들은 해당 Collaboration의 동작(Operation) 또는 특성(Classifier)가 붙여진 구현(realization)의 형태를 이룬다. 그 다이어그램은 Collaboration 안에 특성역할(ClassifierRole)과 관계역할(AssociationRole)들에 해당하는 클래스 박스들과 라인들의 집합을 포함한다. 이러한 경우에 해당 화살표들은 메세지(Message)들이 매핑된 라인들에 연결되어있다.

2.3. Example

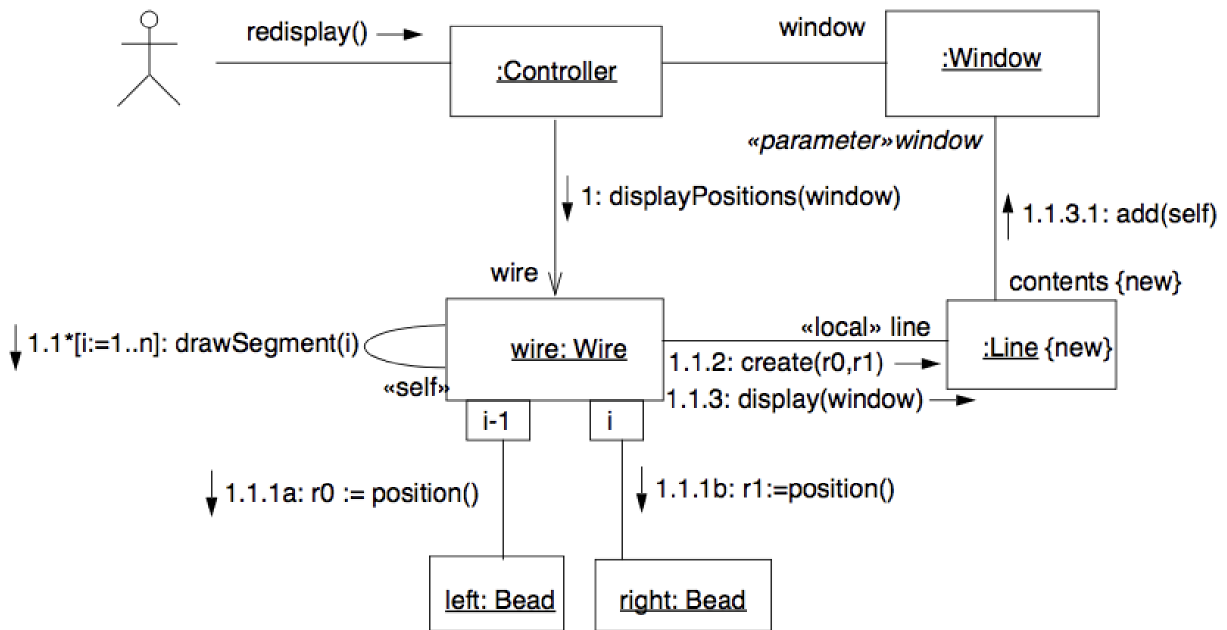


Figure 92 - Collaboration Diagram at instance level, presenting Objects, Links, and Stimuli referenced by a CollaborationInstanceSet and its InteractionInstanceSet.

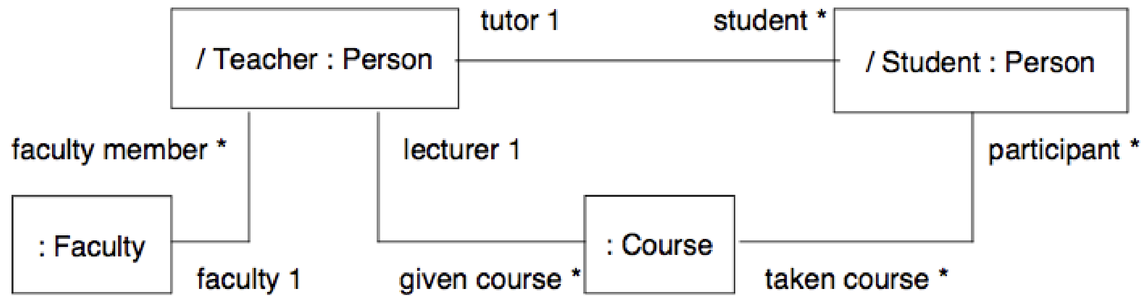


Figure 93 - Collaboration Diagram at specification level, presenting the ClassifierRoles and the AssociationRoles that belong to the Collaboration.

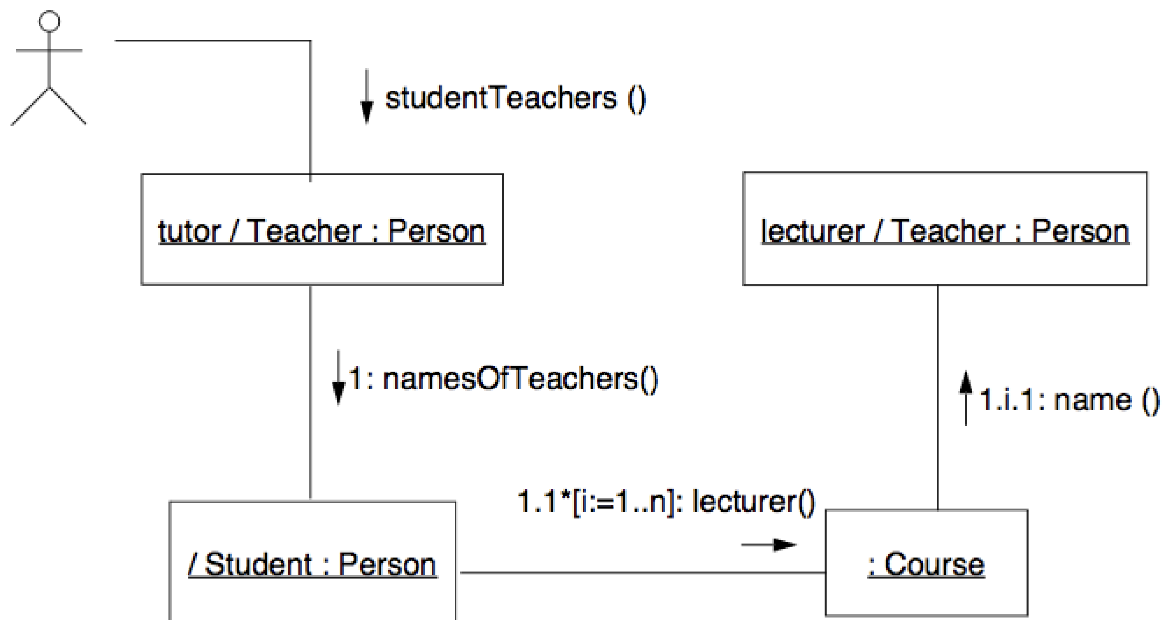


Figure 94 - Collaboration Diagram presenting a CollaborationInstanceSet in which some of the Objects play the same role. The instances conform to the Collaboration shown in Figure 93.

2.4. Mapping

Collaboration diagram은 가능한 한 상호작용(Interation), CollaborationInstanceSet, InterationInstanceSet과 함께 한 관점을 매핑한다.

3. Activity Diagrams

3.1. Semantics

Activity graph는 State machine의 변화이다. 해당 상태들은 하위 상태(Subactivity state)나 활동 상태(Action state)들의 성능을 표현하고 해당 상태의 전이(Transition)들은 하위 상태(Subactivity state)나 활동 상태(Action state)들이 완료됨에 따라 일으켜진다.

이것은 자기 자신 절차의 State machine을 표현한다.

3.2. Notation

Activity diagram은 모든(또는 적어도 대부분) 상태들이 활동상태(Action state)나 하위 상태(Subactivity state)들로 이루어져있고 원본 상태(Source state)에서 모든 전이(Transtion)들은 하위 상태(Subactivity state)나 활동 상태(Action state)의 완료에 일으켜지는 state diagram의 특별한 경우이다. 그 전체 activity diagram은 use-case, package, operation의 구현과 같이 모델을 통해서 특성(classifier)이 연결된다. 이 다이어그램의 목적은 (외부적인 이벤트와 반대되는) 내부적인 프로세싱에 의해서 주도 되는 절차에 집중하는 것 이다. 대부분(모든) 이벤트들이 발생하는 상황에서 액티비티 다이어그램의 사용방법은 내부적으로 생성되는 액션(절차적인 컨트롤 플로우)들의 완성을 표현한다. 이벤트들이 비동기적으로 발생하는 상황에서 정식 state 다이어그램을 사용한다.

3.3. Example

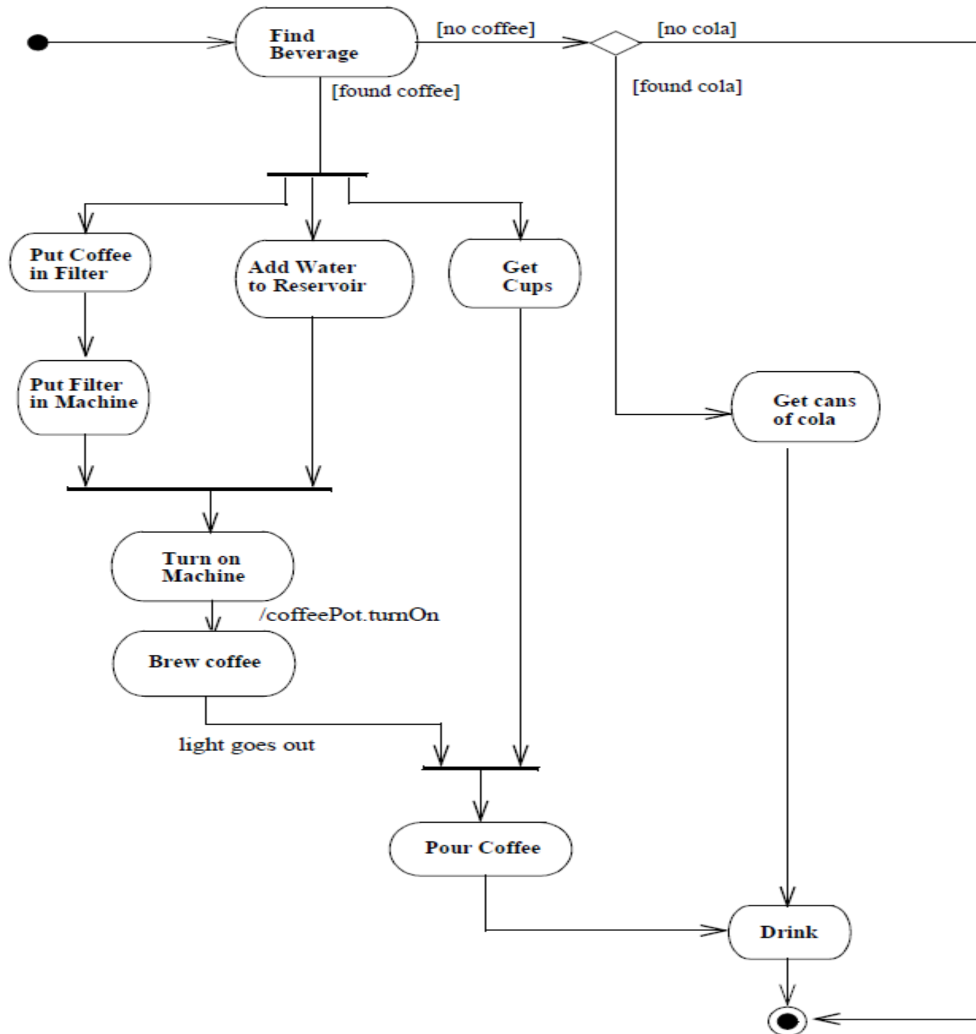


Figure 117 - Activity Diagram

3.4. Mapping

Activity 다이어그램은 Activity Graph와 연결된다.

4. 활동 상태(Action State)

4.1. Semantics

활동 상태(Action state)는 하나의 entry 활동과 적어도 한개의 암묵적인 완료 이벤트(방어 조건을 가질 수 있다면 여러 같은 전이(transition)들이 있을 수 있다)를 포함하는 outgoing 전이(transition)를 가지는 상태(state)의 줄임말 이다. 활동 상태들은 내부적인 전이(transition), 명시적인 이벤트를 기반으로 하는 outgoing 전이 또는 종료 액션(action)들을 갖지 못한다. 이러한 것에는 normal 상태들을 사용해라. 하나의 액션 상태에 의해 나가는 전이들은 어떠한 이벤트 신호도 포함하지 못한다. 같은 전이들은 상태에서의 액션에 의해 암묵적으로 일으켜진다. 이런 전이(transition)들은 보호 조건과 액션을 포함하지 않을 수 있다. 활동 상태(action state)의 정식 용도는 알고리즘(프로시져) 또는 워크플로우 프로세스의 실행에 가담하는 모델이다.

4.2. Notation

활동 상태(Action state)는 위와 아래는 스트레이트, 양쪽은 볼록한 모양으로 보여진다. 활동 식(action-expression)은 그 모양(symbol)안에 놓여진다. 활동 식은 diagram내에서 유일 할 필요가 없다.

4.3. Presentation options

활동(Action)은 인간의 언어, 수도코드 또는 프로그래밍 언어로 묘사될 수 있다. 이 것은 오브젝트 소유의 속성들과 링크들만 사용할 수 있다.
주의하라, 활동 상태(action state) 표기는 일반 상태 다이어그램에서도 사용 될 수 있다. 그러나, 그것들은 activity daigram과 함께 대부분 공통으로 사용되어 진다.

4.4. Example

Figure 118 - Action States

4.5. Mapping

활동 상태(Action state)의 표시는 entry action 상태와 매핑되어 지는 action-expression에서 ActionState와 연결된다. 이런 상태는 일반적으로 쓰인다.

5. 하위 상태(Subactivity state)

5.1. Semantics

하위 상태(Subactivity state)는 액티비티 그래프를 호출한다. 하위 상태가 들어갈 때, 일반 액티비티 그래프로써 실행 될 수 있는 “중첩된" 액티비티 그래프이다. 하위 상태는 중첩된 그래프의 마지막 상태에 도달 하거나, 하위 상태의 바깥으로 나가는 전이(transition)가 일어날 때까지 끝나지 않는다. 액티비티 그래프(activity graph)안에서 상태들은 일반적으로 어떤 이벤트들도 발생 시키지 않는 다면, 하위 상태(subactivity states)는 중첩된 그래프가 끝날 때, 일반적으로 종료되게 된다. 단일 액티비티 그래프(activity graph)는 많은 하위 상태(subactivity graph)들에 의해 호출되어 질 수 있다.

5.2. Notation

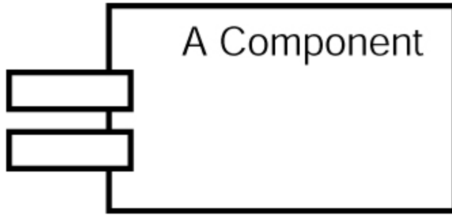
하위 상태(subactivity state)는 일반 활동 상태(action state)에서 오른쪽 아래에 중첩 액티비티 다이어그램(activity diagram)이 묘사된 아이콘이 추가된 형태로 보여진다.
이 표기는 중첩된 구조를 지원하는 어떤 UML에서도 적용 할 수 있다. 아이콘은 반드시 중첩된 구조의 형태를 암시해야 한다.

5.3. Example

Figure 119 - Subactivity States

6. 컴포넌트

컴포넌트(conponent) 스테레오 타입이 있는 클래스 타입 혹은 아이콘 형태로 표현.
컴포넌트는 인공물(시스템 사용이나 생성에 관한 정보의 조각)로 구분될 수 있는 컴퓨터 시스템 모듈러이다. 컴포넌트는 소프트웨어 시스템의 기능을 정의한다.
컴포넌트는 다른 컴포넌트가 접근할 수 있도록 인터페이스를 제공한다. 접근하고 있는 컴포넌트에서는 필수의 인터페이스를 사용한다.
컴포넌트 모델은 M = (S, V, E, I, P, N, T)로 정의한다.
S : ⊆ ℘(T) X ℘(T) : 컴포넌트 상태(state)들의 집합.
V : ⊆ N X VT : 변수(Variable)들의 집합.
E : 이벤트(event)들의 집합.
I : ⊆ ℘(E) : 인터페이스(interface)들의 집합
P : ⊆ N X N X I X PT : 포트(Port)들의 집합.
N : 컴포넌트 이름, 변수 이름, 인터페이스 이름들의 집합.
T : ⊆ S X G X P X E X ℘(A) X S : 상태전이(Transition)들의 집합.
G : Boolean 식으로 표현 가능한 수행 조건(guard condition)의 집합
A : 행위식(Expression)들의 집합.



6.1. 인터페이스

인터페이스를 나타내는 방법에는 두가지가 있다. 첫번째는 인터페이스의 정보를 가지고 있는 사각형을 텅빈 삼각형 머리를 한 점선으로 컴포넌트에 연결하는 방법이고, 두번째는 작은 원을 실선으로 컴포넌트에 연결하는 방법이다. UML2.0에서는 인터페에스가 컴포넌트에 의해 제공되며, 다른 컴포넌트에는 필수적으로 필요하다는 것을 공과 소켓 표기법으로도 표현할수도 있다. 공의 모양은 이미 알고 있듯이 작은 원 모양이고, 소켓은 다른 컴포넌트에 실선으로 연결된 작은 반원(열려있는)모양이다. 공은 제공되는 인터페이스를 의미하고, 소켓은 필수의 인터페이스를 의미한다.

6.2. 컴포넌트 다이어그램 작성 방법

6.2.1. 포트 생성

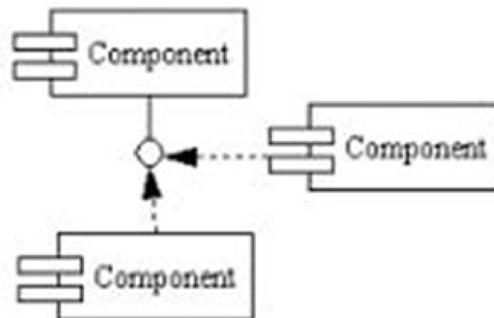
컴포넌트를 선택하고, 오른쪽 마우스 버튼으로 팝업메뉴를 디스플레이한다. Add Port 메뉴항목을 선택하고, 세부 항목으로 포트를 작성할 위치를 선택한다.

6.2.2. 제공 인터페이스(Provided Interface) 작성

컴포넌트를 선택하고, 오른쪽 마우스 버튼으로 팝업메뉴를 디스플레이한다.
Add Provided Interface 메뉴 항목을 선택한다.
제공 인터페이스는 컴포넌트의 왼쪽에 생성된다.

6.2.3. 요청 인터페이스(Required Interface) 작성

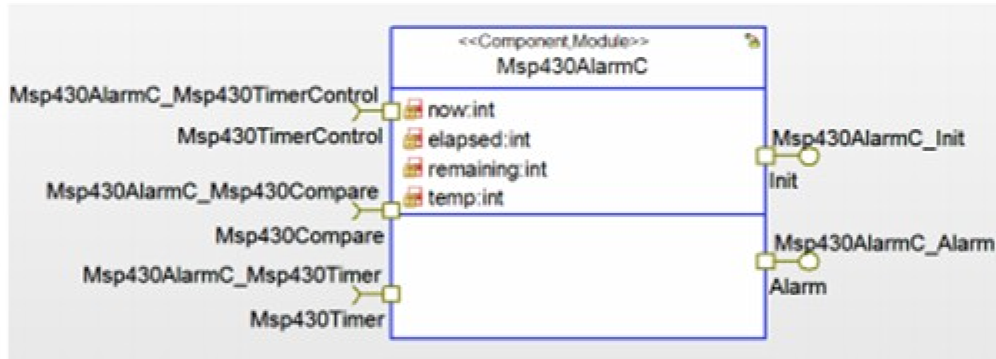
컴포넌트를 선택하고, 오른쪽 마우스 버튼으로 팝업메뉴를 디스플레이다.
Add Required Interface 메뉴 항목을 선택한다.
제공 인터페이스는 컴포넌트의 오른쪽에 생성된다.

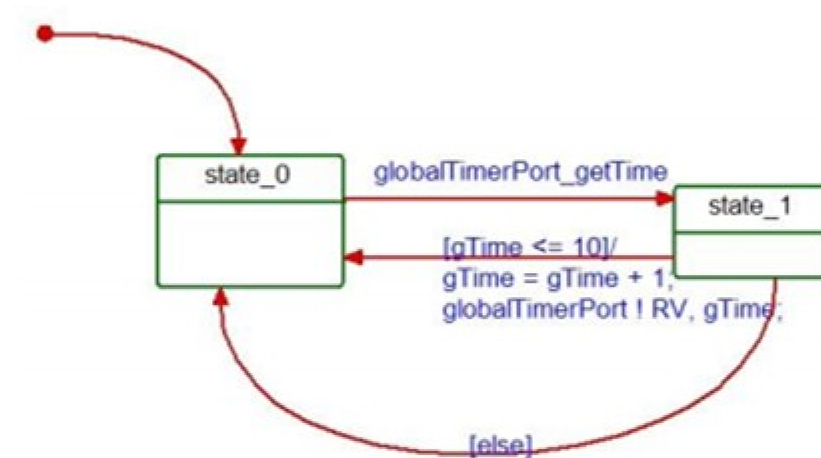


6.3. 컴포넌트 모델과 UML과의 대응 관계

컴포넌트 모델은UML에서 크게 구조 모델과 행위 모델로 나누어 지며, 구조 모델은 UML

Class Diagram, 행위 모델은 UML Statechart로 표현한다.
컴포넌트의 구조 모델을 UML모델로 표현하기 위해서 컴포넌트는 하나의 UML Class로 대응하고, 컴포넌트의 포트부분은 Class의 Port로 대응한다. 반원형 포트는 use 포트를, 원형 포트는 provide포트를 의미한다. 포트의 인터페이스는 포트 아래에 표시한다. 변수는 Class내부에 표기 되며 private와 public을 구분해서 표기한다.



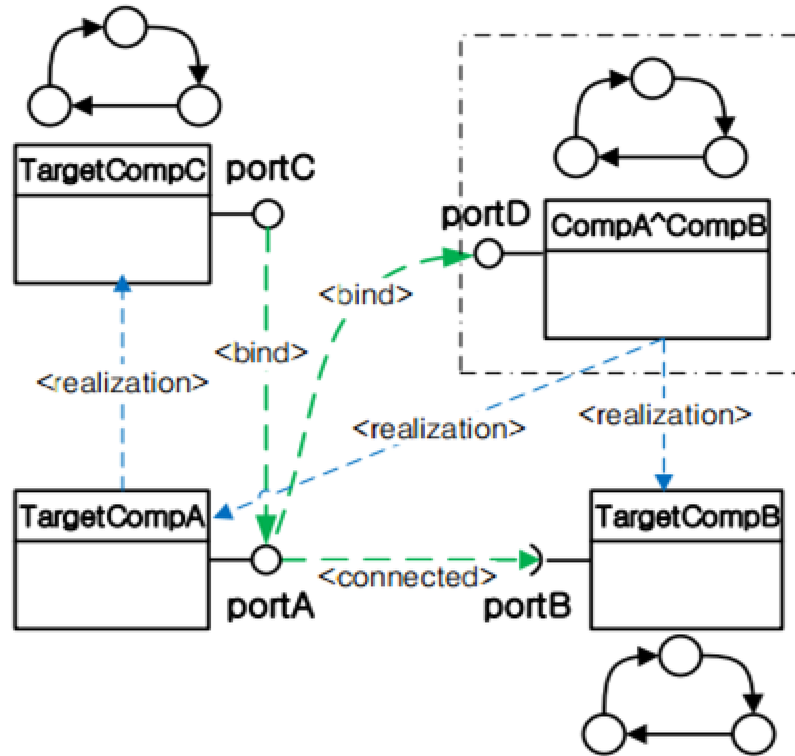


컴포넌트의 행위 모델은의 UML state diagram와 같이 표현된다. 행위모델은 초기 상태에서 부터 출발하며 “/” 를 기준으로 앞쪽에서는 선행조건이, 뒤쪽에서는 전이가 되면서 취하는 행위가 표기된다. 상태전이의 표기 방법은

$$S_i \xrightarrow{\text{[guardCondition]port_event/}\wp(\textit{Action})} S_{i+1}$$

로 트리거 이벤트에서 포트와 이벤트의 구분을 ‘_’ 기호로 표현하고 수행 조건 이 트리거 이벤트 앞에 위치하는 세부적인 내용 이외에는 3.1 장의 표기방법을 그대로 따른다. UML state diagram에서의 기본적인 표기법을 따르나 multiple region, sub-state등은 현재 지원하지 않는다.

6.4. 컴포넌트 사이의 연결 관계 및 구현 관계



컴포넌트 사이에서 시그널 전송, 함수 호출등의 서비스 요청 및 응대는 포트를 통해 이루어지며, 포트의 인터페이스에 정의 되어 있는 이벤트로만 소통할수 있다. 또한 동일한 인터페이스를 가진 포트들 만을 연결할 수 있다. 포트의 종류는 요청을 받고 결과를 외부로 제공되는 provide와 요청을 보내고 결과를 제공받는 use로 나누어지며, 컴포넌트 사이에서 포트간의 연결관계는 <connected>와 <bind>로 표현된다.
<connected>연결관계는 컴포넌트 모델 M2의 인터페이스를 제공하는 컴포넌트 모델 M1에서 M1에 인터페이스를 사용하는 컴포넌트 모델 M2 로의 방향성 그래프이다.
<connected>연결관계 집합 Cr은 $Cr \subseteq M1 \times M2$ 와 같이 표현한다.
<connected> 연결관계는 provide포트에서 use포트로 방향성을 가지고 연결되며, provide포트에서 use포트로 인터페이스에 정의된 이벤트를 제공함을 의미한다. 요청/응대한다. <connected> 연결관계는 one-to-one 연결만을 지원한다.
<bind>연결관계는 인터페이스가 같은 컴포넌트 모델 M1과 M2사이의 방향성 그래프이다. <connected>연결관계 집합 Br은 $Br \subseteq M1 \times M2$ 와 같이표현한다.
<bind> 연결관계는 provide포트에서 provide포트로 방향성을 가지고 연결되며, 연결의 방향 기준에서 목표(target) 컴포넌트 쪽에 서비스 요청이 들어왔을 때, 해당 서비스 요청을 실제로는 근원(source) 컴포넌트 쪽에서 응대함을 의미한다.
<realization>연결관계는 구현 대상인 컴포넌트 모델 M1과 구현에 필요한 컴포넌트 모델 M2간의 방향성 그래프이다. 구체화 관계 집합 Rr은 $Rr \subseteq M1 \times M2$ 와 같이 표현한다.
컴포넌트간의 <realization> 관계는 근원 컴포넌트의 구현을 위해서 목표 컴포넌트가 필요하다는 것을 나타내는 표기방법이다. 예를 들어TargetCompA는 portA에서 받은 서비스의 요청을 모두 <bind> 연결관계에 있는TargetCompC로 다시 요청하며,

TargetCompA자체의 행위모델은 정의되어 있지 않다. 따라서 TargetCompA를 실질적으로 구현하기 위해서는 TargetCompC가 반드시 필요하며 이러한 필요성을 <realization> 으로 표기한다. <realization> 관계는 추후에 본 연구에서 컴포넌트 구조를 분석하고 조합 대상 컴포넌트를 선별하는 과정
에 활용되며, cyclic한 의존 관계를 가지지 않는다.
컴포넌트의 상태(state)는 진입 상태전이(incoming transition)들의 멱집합과 퇴출 상태전이(outgoing transition)의 멱집합의 조합으로 상태는 진입 상태전이들과 퇴출 상태전이들을 기억한다. 변수(variable)는 이름과 변수 타입 VT = {public, 10private} 의 조합으로 변수의 종류는 외부에 공개되는 public과 공개 되지 않는 내부변수 private로 나누어 진다. 이벤트(event)는 시그널 혹은 함수 호출 등의 종류가 있으며, 인터페이스(interface)는 이벤트의 멱집합의 조합으로 표현된다. 포트는 근원 컴포넌트 이름, 포트 이름과 인터페이스, 포트 타입 PT = {provide, use} 의 조합으로 포트의 종류는 요청을 받고 결과를 외부로 제공되는 provide와 요청을 보내고 결과를 제공 받는 use로 나누어진다. 상태전이(Transition)은 근원 상태, 수행 조건, 이벤트를 수신하는 포트, 수신 대상 이벤트, 행위식들의 집합, 도달 상태로 표시되며 표현 양식은 아래와 같다.

$$T_i: \text{SourceState} \xrightarrow{\text{port?event[guardCondition]}/\wp(\textit{Action})} \text{TargetState}$$

상태전이는 수행 조건을 만족하며, 인터페이스와 이벤트로 표현되는 트리거 이벤트(Trigger Event)가 발생할 때 수행되며, 상태가 전이되는 시점에서 함수 호출, 시그널 전송, 변수값 전송등의 행위를 수행한다.

7. Use Case Diagram

7.1. Use Case와 Actor 그리고 이들간의 관계 표현

System 주요 기능들과 이들과 관계하는 관련 당사자들에 대한 내용을 한눈에 알아볼 수 있도록 전체적인 View를 제공

7.2. 구성요소

- Actor
- Use Case
- Actor와 Use Case의 관계

7.2.1. Actor

- Use Case와 상호작용을 하는, System 사용(System에 어떤 Action을 취한다)의 주체적인 존재
- 사람, 사물, 시스템
- Actor명은 역할로 표현
- Actor 구분

Primary Actor	Use Case 기동 Actor 관계의 화살표 방향이 Actor로부터 Use Case를 향하고 있는 모습

Supporting Actor	주로 외부 연동하는 System System에 대한 설계 진행 시점에 주로 식별

- 특정 System의 제어권 외부에 존재하며 특정 System에 요구 사항 부과
- 대상이 되는 System을 사용하는 사람이나 외부 System 항목을 나열 후, 나열된 Actor 후보가 하는 역할을 이해하기 쉬운 용어로 표현

7.2.2. Use Case

- System, Sub-System에 의해서 제공되는 상호연관성이 높은 기능성을 표현
- System과 Actor간에 발생하는 상호작용을 일련의 메시지로 표현
- (목적어 + 동사의 명사형) 표현
- Use Case 크기

Use Case 크기 크고 개수는 적게	Use Case 크기 작고 개수는 많게
유지 및 관리 대상이 적으므로 편의성 증대	Use Case 처리 Logic이 단순하고 명쾌
각 Use Case 처리의 복잡도 증가 Diagram이 간단하여 전체적인 이해의 수준이 낮아짐	유지 및 관리대상의 증가로 인해 유지/관리 부담 증가

7.2.3. Actor와 Use Case의 관계

- 모든 Use Case는 Actor로 시작
- 관계의 종류

Actor-Use Case	양방향 단방향
Actor-Actor	일반화 ▹
Use Case-Use Case	일반화 ▹ 포함 <<include>> 확장 <<exclude>>

- Actor-Use Case: Association, Communicates Association
- Actor-Actor: Generalization(상속)

7.2.4. Use Case와Use Case 관계

- Include: 최소한 2개 이상의 Use Case들이 동일한 흐름을 가진 하나의 업무기능을 공유할 때
- Exclude: 화살표의 방향은 포함관계의 방향과 반대, 확장 Use Case는

기본 Use Case에서 특정 조건이나 Actor의 선택에 따라 기동

7.3. Use Case Specification

7.3.1. Actor와 Use Case 간의 상호 작용에 대한 기술 문서

- Diagram에 표현되어 있는 Actor와 Use Case가 어떤 상호 작용을 하는지, 어떤 요청과 응답을 주고받는지, 즉 Event에 대한 기술 문서.

7.3.2. 구성요소

- Use Case 이름
- Use Case 간략한 설명
- Event 흐름(기본흐름, 대체흐름)
- 특별한 요구사항
- 사전/사후 조건
- 확장점

7.3.3. Use Case 이름

Actor의 관점에서 의미 있는 이름

7.3.4. Use Case 간략한 설명

Use Case의 역할과 목적

7.3.5. Event 흐름(기본흐름, 대체흐름)

- System 흐름에 대한 윤곽: System이 무엇을 해야 하는가.
- 개발자, 사용자, 고객, 실무자 등이 이해할 수 있는 용어

기본흐름	정상적이고 일반적인 흐름
대체흐름	기본 흐름을 진행하는 동안 발생하는 여러 가지 비정상적이거나 선택적인 행위를 설명

7.3.6. 특별한 요구사항

- Event 흐름에 표현되지 않은 비기능 요구사항
- 제약 사항이 될 수 있는 특정 Use Case에 해당되는 내용 기술
- 하나 혹은 여러 Use Case에 적용 가능

7.3.7. 사전 조건

- Use Case 시작 전에 System이 가지고 있어야 할 상태
- 기본 흐름으로부터 대체 흐름을 식별한 후 확장하는 시점에 사전조건과의 중복 여부 판단

7.3.8. 사후 조건

- Use Case가 종료된 후 System의 제약사항 기술
- 성공/실패의 경우 모두 고려

7.3.9. 확장점

특정 조건을 만나면 특정 Use Case로 확장되는 지점

7.3.10. System 관련 그룹

개발자	System 분석, 설계, 구현

Tester	System 의도 목적 달성 여부 검증
교육 담당자	사용자들에게 System 사용방법 설명
지원 담당	System 가동에 대한 책임 및 유지보수

7.4. Use Case Diagram

Use case 는 사용자 관점에서의 시스템 수행 기능이다.
Use case diagram을 그릴 때, 시스템 내의 어떤 특정 기능이 Use case 인지 아닌지의 기준은 사용자와 시스템간의 시스템 수행 기능인지를 확인하면 된다. 예를 들어, 어떠한 웹 사이트에서 회원 가입 시 입력 정보를 DB에 저장하는 기능과 같은 사용자의 눈에 보이지 않는 내부적인 수행 기능은 사용자 관점에서의 시스템 기능이 아니기 때문에 Use case 로 그리지 않는다.
관계는 Generalization, Include, Exclude 이렇게 세가지면 충분하다.
간혹 Use case diagram 을 보면 다양한 관계를 사용하는 것을 많이 볼 수 있는데, 오히려 혼란을 초래 할 수 있다. 일반적으로 Use case diagram에서는 Generalization, Include, Exclude 이렇게 세가지 관계만 사용한다.
관계를 사용하는 목적과 의미에 대해 생각해 보자.
Use case diagram 에서의 관계는 사용자에게 기능을 좀 더 쉽게 보여주고 설계에 대한 방향을 이끌어 낸다는 관점으로 접근해야 한다.

7.4.1. include

Include 를 사용하는 목적은 크게 두가지이다.
하나는 추상적이고 복잡한 하나의 Use case 를 논리적인 행동 단위의 작은 Use case 들로 분리함으로써 해당 Use case를 의미적으로 좀 더 간소화한다는 것이고,
다른 하나는 아래와 같이 두개 이상의 Use case들이 가지고 있는 공통된 행동을 추출하여 모듈화하는데에 그 목적을 두고 있다.
아래는 그 사용 예를 보여준다.

7.4.2. exclude

보조적이고 선택적인 하나의 Use case 가 다른 하나의 Use case 안에 언제 어떻게 추가되는지를 정의한다.
따라서 Extension Point 가 중요하며, 그 Extension Point에는 확장하는 Use case 가 언제, 어떻게 선택적으로 추가되는지에 대한 조건이 정의되어야 한다.

7.4.3. Generalization

클래스들 사이의 일반화와 유사하다. 자식 Use case는 부모 Use case 의 property들과 behavior 들을 상속하고 또한 behavior 들을 override 할 수 있다.

7.5. Use Case 의 사례

7.5.1. ATM 현금 출금 이벤트 흐름

<<기본 흐름>>
이 Use Case는 고객이 ATM에서 현금 인출을 하기 위해 행동 하면서 시작한다.

1. 고객이 ATM에서 현금 인출을 시작한다.
    1.1. 고객이 화면에서 메뉴를 선택하는 경우: A01
    1.2. 고객이 현금/신용카드를 넣는 경우: A02
2. 고객 인증 처리1
    2.1. System은 Card에 저장된 개인정보(은행번호, 계좌번호 등)를 읽는다.
    2.2. System이 카드 판독이 불가능한 경우: E01
    2.3. System은 고객에게 비밀번호 입력을 요구하고, 입력한 비밀번호가 맞는지 확인한다.
        Input : 고객 비밀 번호(6자리)
    2.4. 입력한 비밀번호가 다를 경우: A03
    2.5. 입력한 비밀번호가 맞는 경우: A04
3. 현금 인출 메뉴 제공
    3.1. System은 고객이 선택한 현금 인출 메뉴를 제공한다.
    3.2. System은 인출금액 입력을 요구한다.
4. 고객은 인출금액을 입력한다.
5. 인출 금액 확인
    5.1. System은 입력금액의 범위를 확인한다.(최하/최대금액)
        5.1.1. 입력금액 범위를 벗어난 경우: E02
    5.2. System은 ATM에 고객이 요청한 금액이 있는 지 확인한다.
        5.2.1. ATM에 요청 금액이 부족한 경우: E03
    5.3. System은 고객의 계좌에 요청한 금액 인출 여부를 확인한다.
        5.3.1. 은행 접속이 불가능한 경우: E04
        5.3.2. 잔액이 부족한 경우: E05
6. 현금 인출
    6.1. System은 거래 내용을 기록한다.
        Input: 거래일시, ATM 위치, 고객 계좌번호, 거래유형, 거래 금액, 거래번호 등
    6.2. System은 고객에게 현금을 지급한다.
    6.3. System은 영수증과 카드를 고객에게 제공한다.
7. 고객은 현금, 카드, 영수증을 확인한다.
8. Use Case를 종료한다.

8. Class Diagram

유사한 구조를 가지는 구조체와 구조체간의 정적인 연결을 표현하는 정적 구조체 도식이다. 또한 인터페이스, 패키지, 연결, 인스턴스, 객체와 링크까지도 포함하기도 한다. 정적 구조체 다이어그램의 짧은 표현으로 보는 것이 좋다.

8.1. 의미

정적인 상태의 시스템 구조체를 시각적으로 표현하는 모델이다. 각 클래스 다이어그램은 기반 모델의 분할을 표현하지 않는다.

8.2. 구성

정적으로 선언되는 원소들인 클래스, 인터페이스, 그 상호관계를 서로와 서로의 내용물과 연결되는 그래프로 표현한다. 기반 모델이나 여러 기반 모델의 패키지들로 이루어지는 각 패키지 형태로 구성한다.

8.3. 연결

구성요소는 하나 이상의 요소와 결합할 수 있다.
한 패키지는 하나 이상의 클래스 다이어그램으로도 표시될 수 있다. 여러 다이어그램으로 나누는 것은 표현상의 효율을 위한 것이지 모델 자체가 나누어져 있음을 의미하지 않는다.
다이어그램의 내용물들은 정적 의미 모델의 각 요소로 연결된다. 어떤 다이어그램이 패키지의 일부분이면, 그 내용물은 해당 패키지의 내부로 연결된다.

8.4. Object diagram과의 차이

Class diagram이 객체(Object)를 포함할 수는 있지만 Object diagram은 클래스(Class)를 포함하지 않는다. 이 두 가지 용어를 구분하면 사용 용도가 구별되므로 편리하다.

8.5. Classifier

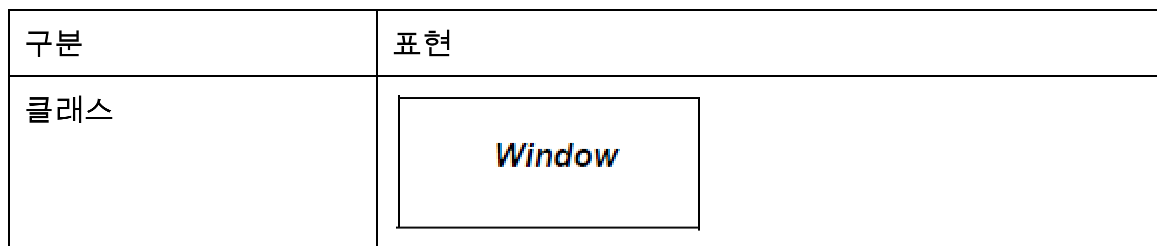
클래스, 자료형, 인터페이스의 상위 모델이다. 이 세 가지는 서로 비슷한 구문으로 표현되므로 모두 직사각형으로 표시하고 필요한 경우 형 명을 표시한다.
일반적으로 다이어그램에서 가장 많은 것이 클래스이므로 아무 형 명이 없는 직사각형으로 표시하고 나머지 자료형과 인터페이스는 앞에 형 명을 표시하는 방법을 주로 사용한다. 아래 클래스에 관한 내용은 자료형과 인터페이스에도 일정부분 적용 가능한 내용이다.

8.6. Class

8.6.1. 의미

설계하는 시스템을 구성하는 개념을 표현한다. 자료구조를 가지고 다른 요소에 특정 동작이나 연관을 갖는다. 패키지 내부의 한정 범위에서만 선언되며 그 이름은 다른 클래스 이름과 달라야 한다.

8.6.2. 표현형



구분	표현
클래스	Window

속성이 있는 클래스	**Window** size: Area visibility: Boolean *display ()* *hide ()*
속성 값이 있는 클래스	**Window** {abstract, author=Joe, status=tested} +size: Area = (100,100) #visibility: Boolean = true +default-size: Rectangle #maximum-size: Rectangle -xptr: XWindow* *+display ()* *+hide ()* *+create ()* -attachXWindow(xwin:Xwindow*)

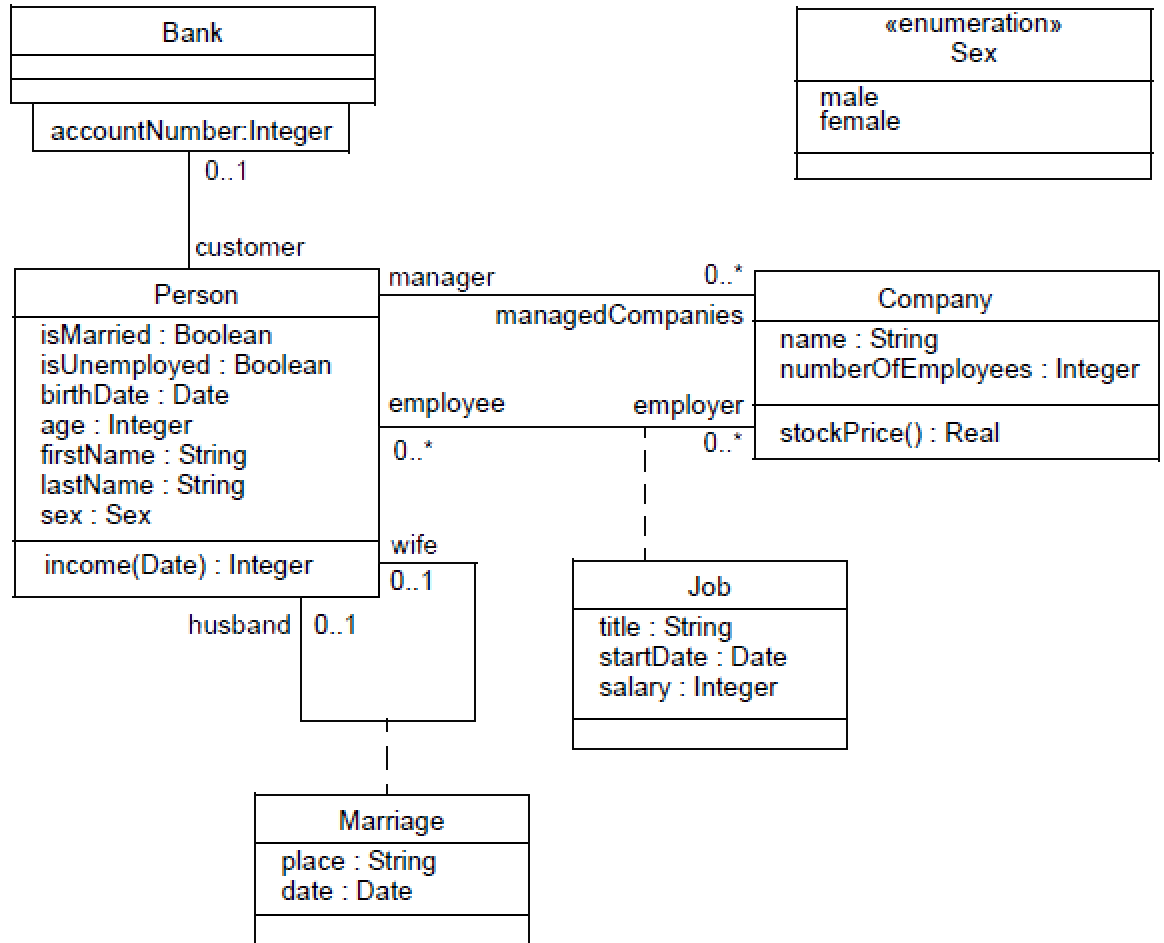
8.6.3. 용법

외곽선을 실선으로 그리며, 내부에 세 개로 나뉜 공간을 갖는다. 맨 위의 이름부분에는 클래스 이름과 { } 내부에 일반속성을 기입한다. 가운데는 속성, 아래는 작동이 들어간다.

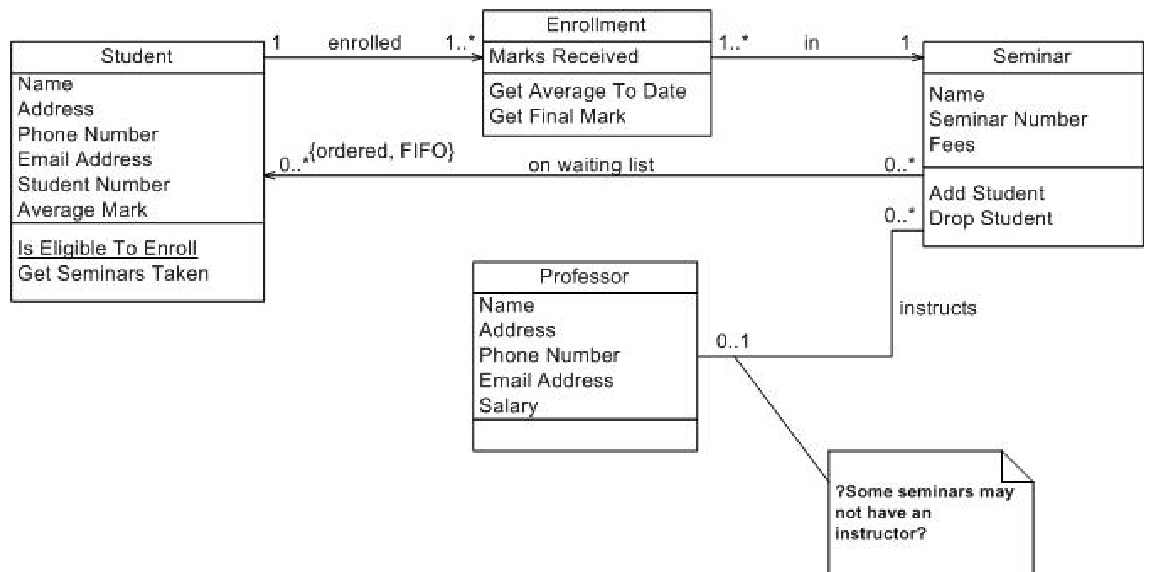
패키지 내부의 클래스는 “패키지명::클래스명” 형태로 기입한다. 패키지 내부의 패키지인 경우 “패키지명::패키지명::....::클래스명”형태로 절대적 형태를 표현할 수 있다.

8.7. 사용 예

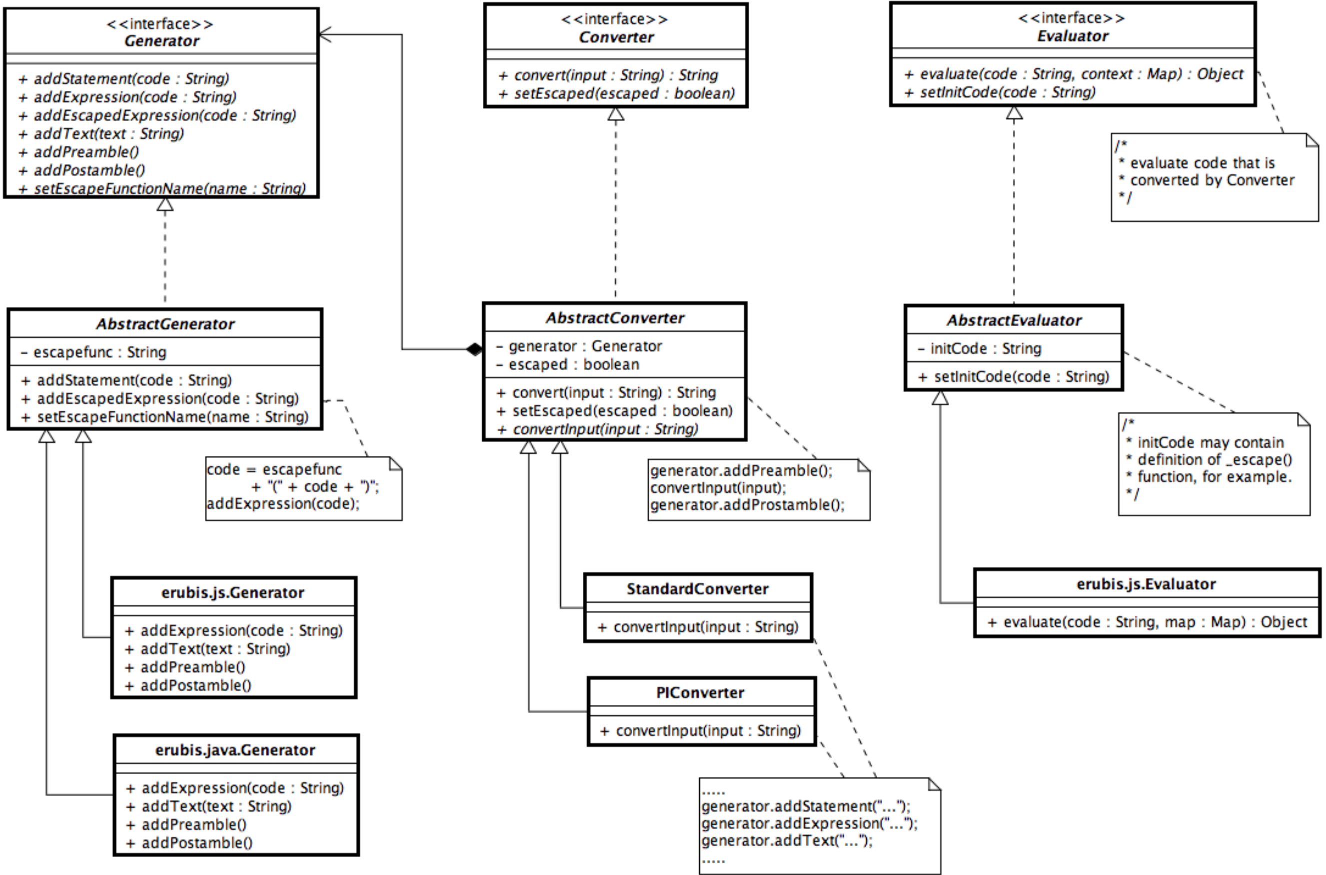
8.7.1. 클래스 사용 예



8.7.2. 학교 (v2.0)



8.7.3. Erubis-J



9. Deployment Diagram

시스템의 노드와 컴포넌트 및 그 관계를 묘사하는 정적 구조체 도식이다. 즉, 시스템의 하드웨어와 소프트웨어, 원격의 다른 시스템과 연결시켜주는 미들웨어를 보여준다. 따라서 여러 장치에서 작동하는 어플리케이션을 표현하기에 좋다.

9.1. 구성요소 (v2.1 기준)

artifact, node, deployment의 세 가지 노드와 연결선을 이용하여 작성한다.

9.1.1. graphical node : 노드

a. artifact

개발의 과정을 통해서 나온 결과물을 총칭한다. 시스템의 개발, 배포, 운영에 필요한 정보의 물리적인 부분을 명세한다.

type	artifact	속성이 있는 artifact

artifact	«artifact» ArtifactName	«artifact» ArtifactName {execution=thread, transaction =true}

b. node

artifact가 실행되는 장치 또는 실행 환경을 의미한다. 각 node간의 연결선은 통신연결을 의미한다.

type	node	artifact가 탑재된 node
node	NodeName	«executionEnvironment» NodeName artifact1 artifact2 artifact3

c. deployment

artifact를 특정 node에서 작동하기 위한 모든 활동을 의미한다.
소프트웨어 artifact가 어떻게 장치 및 실행 환경과 설치 또는 할당되어 있는지를 나타낸다.

type	deployment	속성이 있는 deployment	속성의 값이 있는 deployment
deployment	«deployment spec» Name	«deployment spec» Name execution: execKind transaction : Boolean	«deployment spec» Name execution: thread transaction : true

9.1.2. graphical path : 연결선

type	연관	의존	일반화
path			

type	deploy	선언
path	«deploy»	«manifest»

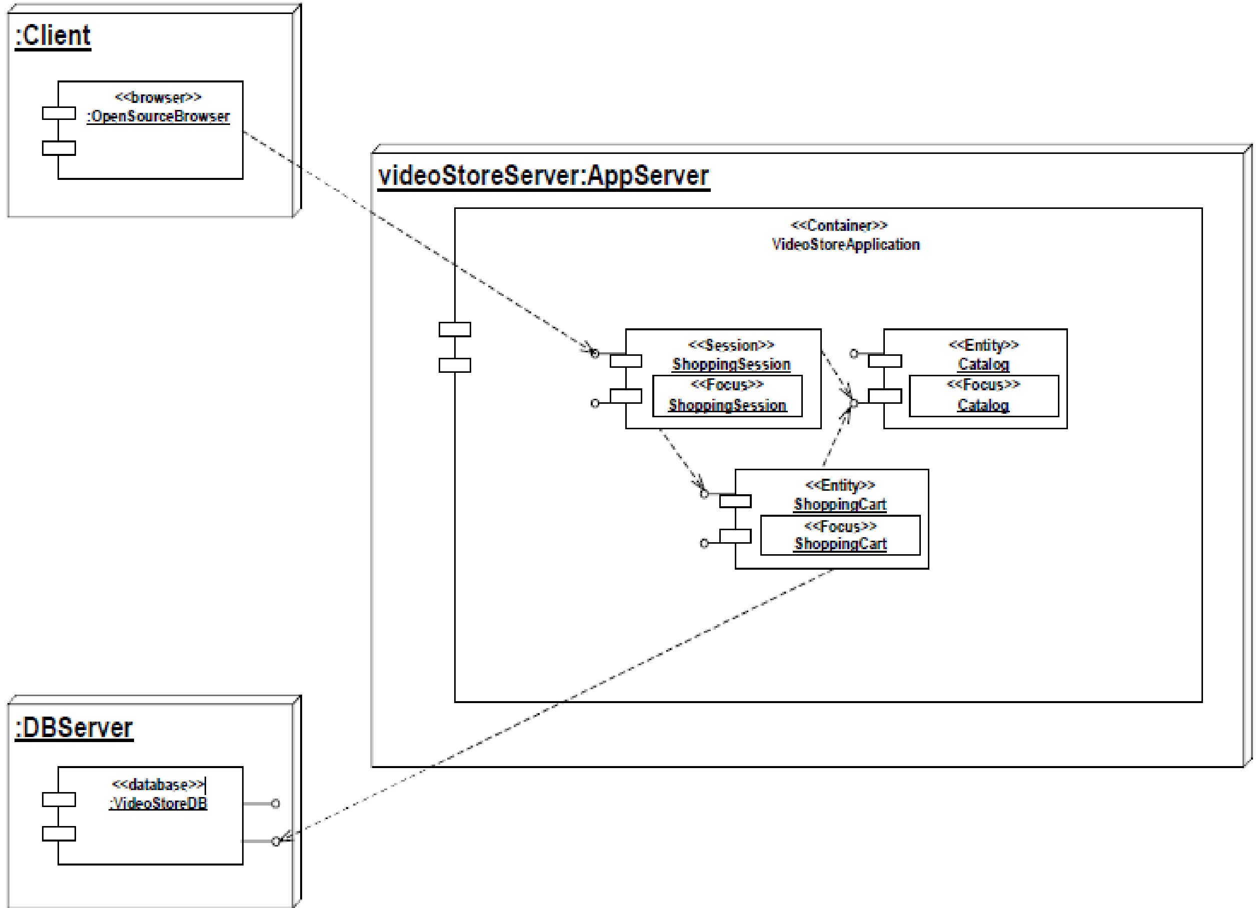
9.2. 용법

node는 입체 정육면체로 나타내고, 윗부분에 이름을 표시한다. 노드 이름에는 밑줄을

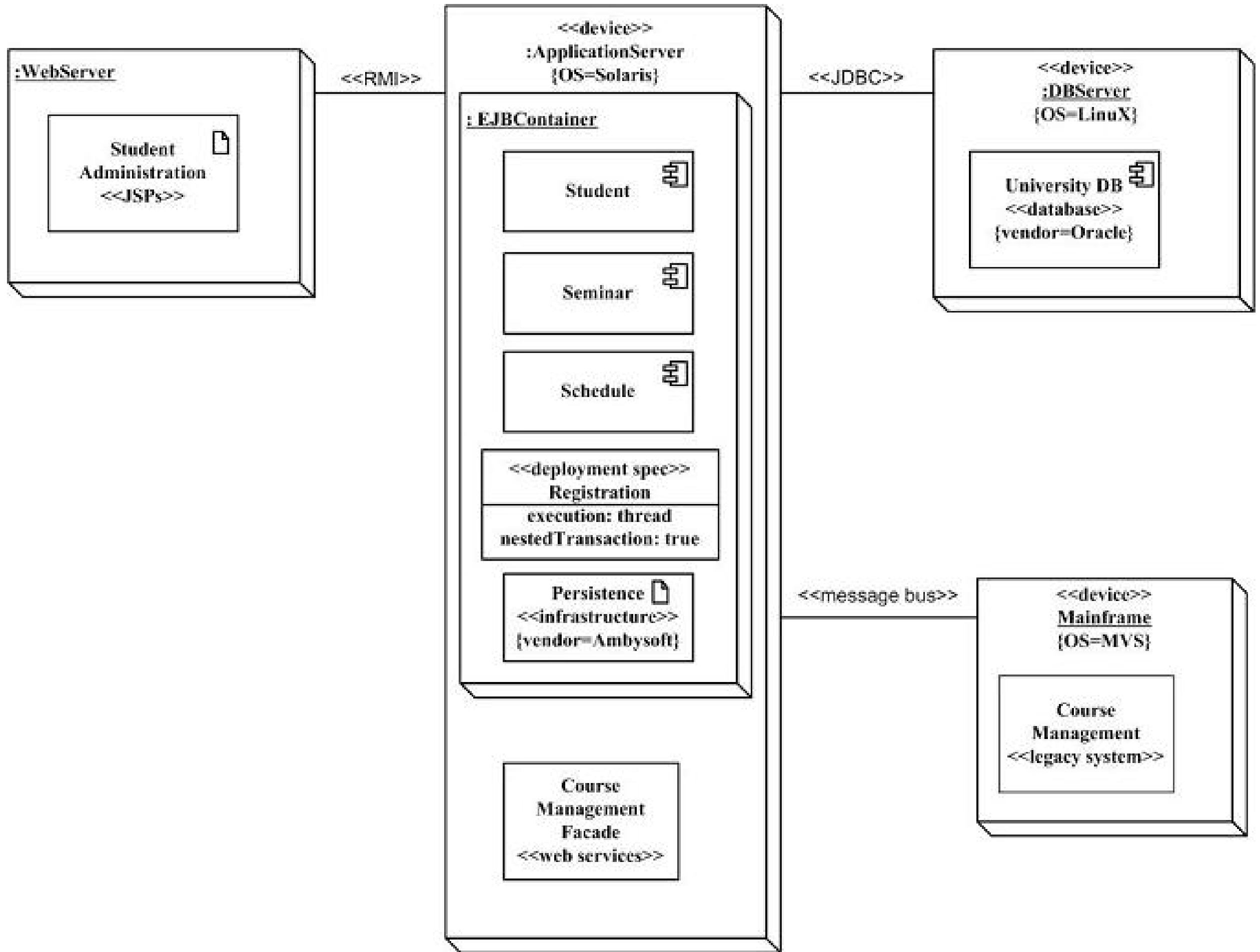
그으며, “이름:노드 타입 명” 형태로 나타낸다.
artifact에는 <<타입 명>> 을 이용해 어떤 형태에 기반하는 지를 표현할 수 있다.
node간, 혹은 artifact간에 <<become>>이 붙는 의존관계가 있을 수 있는데, 이는 전체 수행 시간 중 적은 시간동안만 해당 node 혹은 artifact에 상주한다는 것을 의미한다.

9.3. 사용 예

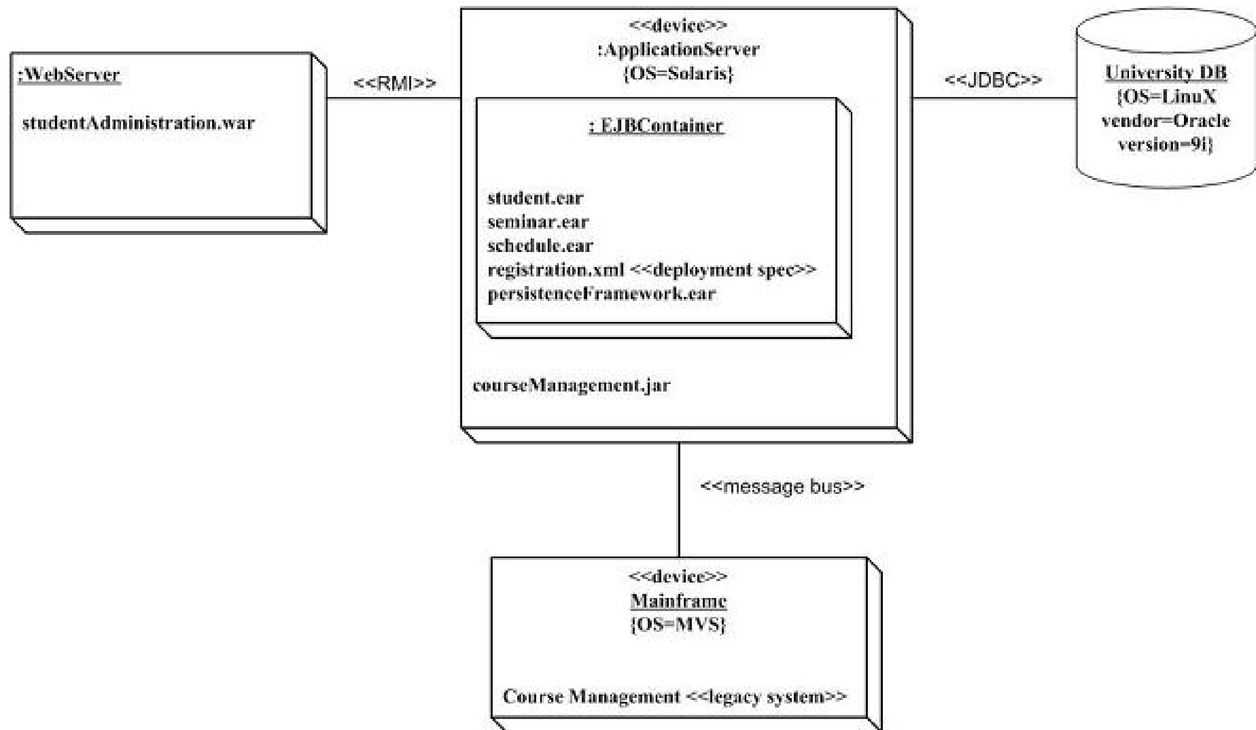
9.3.1. 온라인 비디오 대여 서비스



9.3.2. 종합 대학 정보 시스템(v2.1)



9.3.3. 위 종합 대학 정보 시스템의 보다 간결한 실용적 표현



10. Sequence Diagram

가장 일반적인 형태의 상호작용을 나타내기 위한 행위 도식이다. 여러 상호작용 요소간의 메시지 교환을 중점적으로 묘사한다. 대부분은 다이어그램의 시간 순서만이 중요하지만, 실시간 어플리케이션의 경우는 시간 축이 실제 시간의 절대값과 같은 단위이기도 하다.

10.1. 구성요소

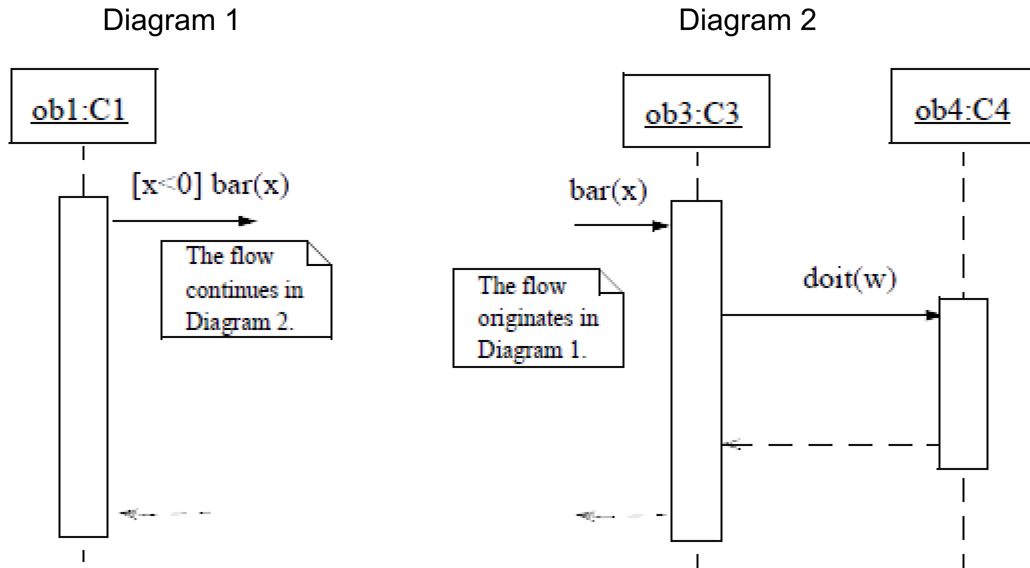
다음과 같은 구성요소이며 이는 Collaboration diagram과 같다.

10.1.1. 객체 생명선(Object Lifeline)

수직 점선으로 객체 인스턴스의 생명선을 나타낸다. 이 시간동안 인스턴스가 존재함을 의미한다. 객체 이름은 생명선의 최상단에 표시한다.

인스턴스가 실행 중간에 생성되면 활성화 화살표로 생성 주체와 연결하고, 중간에 소멸되면 커다란 “X”표시를 소멸시키는 주체 인스턴스와의 연결이나 자멸의 경우에는 해당 인스턴스로 들어오는 마지막 화살표에 한다.

상호작용 이전부터 존재하는 인스턴스는 생명선을 최초 상호작용 화살표보다 위에서 다이어그램 맨 위까지 닿도록 그리고, 마지막 상호작용 이후에도 남아있는 인스턴스는 최후 상호작용 화살표 이후로부터 다이어그램 맨 아래까지 그린다.



왼쪽의 다이어그램이 오른쪽 다이어그램으로 연결된다.

구분	표현	설명
통신	filled solid arrowhead	
절차 실행, 제어	stick arrowhead	하위 절차가 모두 완료되어야 외부 과정이 계속된다. 제어는 다른 객체의 인스턴스에 신호를 보내고 그 모든 하위 절차의 종료를 대기한다.
비동기 통신	dashed arrow with stick arrowhead	절차 제어가 없다. 바로 다음으로 넘어간다.

10.2. 용법

수직방향은 시간의 흐름을 의미하고 수평방향은 각각의 인스턴스를 의미한다.
일반적으로는 페이지의 아래 방향이 시간의 흐름과 같은 방향이다. 평행 축은 순서가 관계없다.

모든 실행 화살표가 한 방향으로 정렬되기도 하지만, 이는 항상 가능하지는 않고 정렬된 순서가 의미를 가지지는 않는다. 시간 수치나 활성화되는 활동의 설명이 변화나 활성화 화살표에 표지로 붙기도 한다.

일반적으로는 페이지의 아래 방향이 시간의 흐름과 같은 방향이고 오른쪽 방향이 서로 다른 객체이지만, 거꾸로 배치하기도 한다.
대부분은 다이어그램의 시간 순서만이 중요하지만, 실시간 어플리케이션의 경우는 시간 축이 실제 시간의 절대값과 같은 단위이기도 하다. 평행 축은 순서가 관계없다.

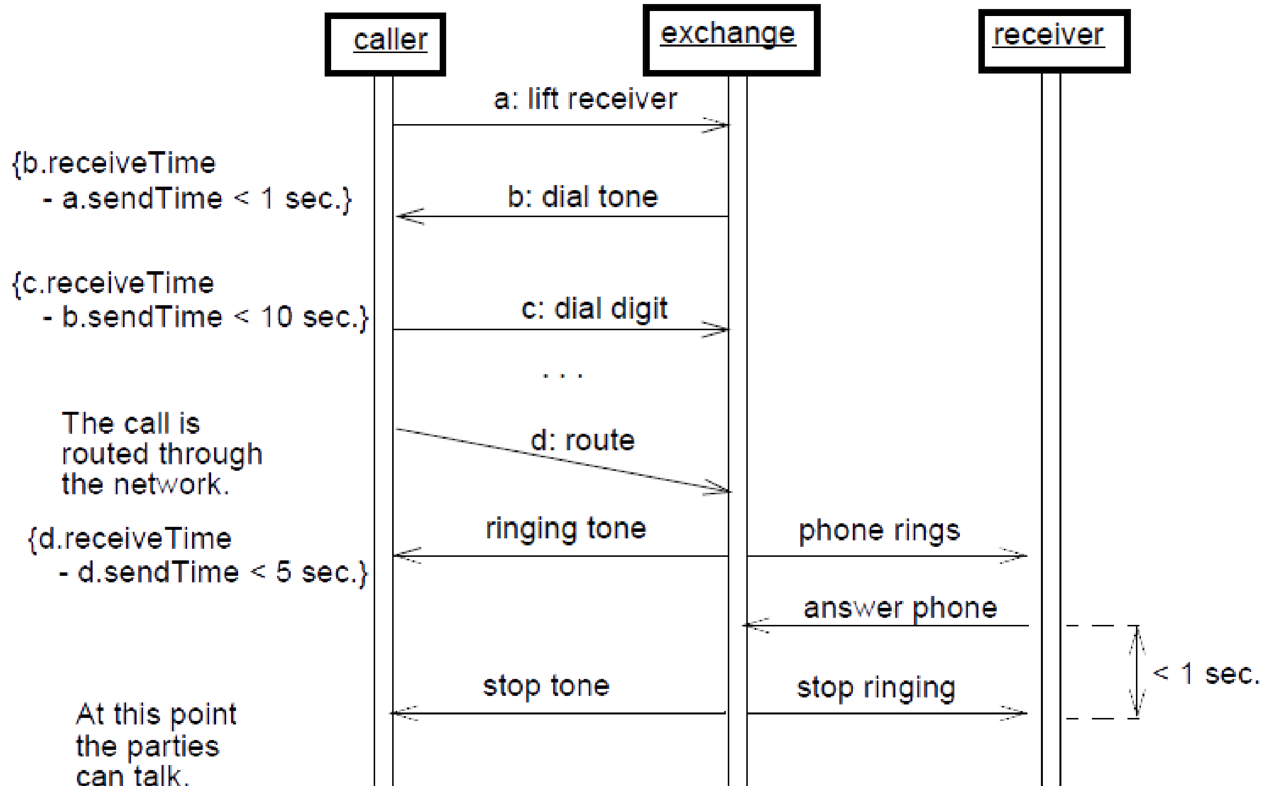
모든 실행 화살표가 한 방향으로 정렬되기도 하지만, 이는 항상 가능하지는 않고 정렬된 순서가 의미를 가지지는 않는다.

시간 값이나 활성화되는 활동의 설명이 변화나 활성화 화살표에 표지로 붙기도 한다. 시간 값은 시간 단위로 표현될 수도 이고, 활성화 화살표의 이름으로 붙기도 한다. 활성화 화살표 이름에 sendTime, recceiveTime 함수를 적용하여 실제 시간을 구해야 하는 경우도 있다. 시간 함수는 사용자가 상황에 맞게 elapsedTime, executionStartTime, queuedTime, handledTime 등으로 수정하여 사용할 수 있도록 열려있다.

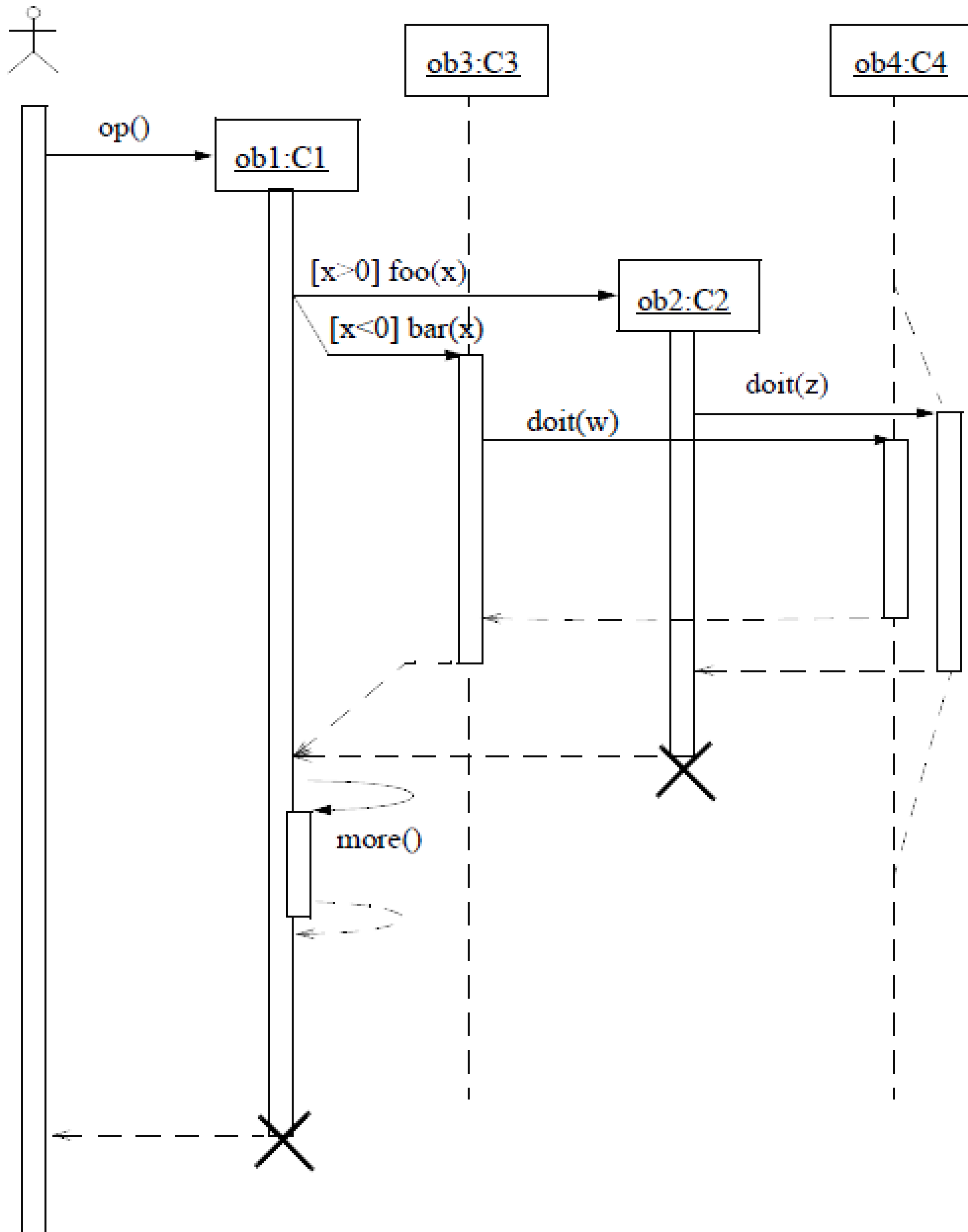
설계도에 있는 생성 표시는 연결이 활성화되는 시간 간격을 표현한다. 이 표현방법은 시각적으로 효과적이지만, 화살표가 평행인 경우 보내는 시간과 받는 시간이 구분되지 않을 수 있다. 이럴 경우 대부분은 전송 시간이 서로 같은 경우가 많기 때문에 괜찮기도 하지만, 소프트웨어 툴에서는 이렇게 혼동되지 않도록 메시지와 활성화 이름을 표시해야 한다. 툴은 시간 함수 이름을 따로 관리하여 활성화 이름과 연동되도록 하여 "b.receiveTime - a.sendTime < 1 sec"와 같은 표현의 이용이 가능해야 할 것이다. 물론 이런 표현은 의미 모델과도 연동되어야 할 것이다.

10.3. 사용 예

10.3.1. 병렬 실행형 객체(굵은 사각형)의 간단한 Sequence diagram



10.3.2. 제어, 조건, 재귀, 생성, 소멸에 대한 Sequence diagram


ob3:C3
ob4:C4
op()
ob1:C1
[x>0] foo(x)
ob2:C2
[x<0] bar(x)
doit(z)
doit(w)
more()