Introduction to UML

T4

201411258 강태준

201411265 김서우

201411321 홍유리

Index

1. UML 제대로 이해하기

2. UML 구성요소

3. Relationship

4. UML Diagram 종류

5. UML Tool

1. UML 제대로 이해하기

1) UML에 과한 몇가지 오해

ㄱ. UML은 방법론이다.

UML(통합 모델링 언어)은 통합된 방법론이지만, 현재까지 표준화되고 사용되고 있는 것은 방법론과는 완전하게 분리된 모델링 언어일 뿐이다. 방법론과 모델링 언어가 무엇인지 설명하자면, 방법론이란, 프로세스+표기법을 의미하며, 쉽게 설명하면 "소프트웨어를 개발하기 위한 총체적인 체계"라고 할 수 있다. 여기에는 절차,표기법,규칙,기법,방식,문화 등도 포함될 수 있다고 본다. 모델링 언어란 소프트웨어 모델을 표현하기 위한 언어라고 정의하는 것이 가장 무난하다. 이것을 이해하기 위해서는 소프트웨어 모델이 무엇인지를 먼저 이해해야 하고 더 근본적으로는 모델을 이해해야 한다. 모델에 관해서는 뒤에서 다시 자세히 알아보고, 언어라는 측면을 볼 때에는 대화의 수단이 된다는 것이 가장 중요할 것이다. 즉, UML은 대화의 수단이다. 사람과 사람과의 대화,지금의 나와 몇 달 후의 나와의 대화,사람과 컴퓨터와의 대화 등에서 사용될 수 있다. 모델링언어는 결국 소프트웨어 개발을 위한 총체적인 체계의 한 부분일 수는 있으나 그 자체는 아니라는 점을 분명히 해둔다.

ㄴ. UML은 단순한 표기법이다.

이것은 거의 대부분의 사람들이 믿고 있기 때문이기도 하다. 이러한 오해는 오히려 UML이 시각적인 표현을 강조한다는 점에서 출발하고 있는 듯하다. 많은 교육 프로 그램들이 아직, UML은 어떤 어떤 다이어그램으로 구성되어 있고, 클래스는 몇 개의 칸으로 구성된 네모로 표현되며 의존 관계는 점선 화살표로 그린다라는 식의 표기법 설명에 치중하고 있기 때문이 다. 어떻게 보면 당연한 현상일 것이다. UML이 시각적인 언어이고 겉으로 드러나는 것은 눈으로 잘 보이는 기하학적인 형태이므로, 초심자들에게는 우선 그 그림 모양을 우선적으로 가르치는 것이 정상적인 순서일 수도 있다.

UML은 언어라고 했다. 언어를 단순히 구문과 의미로 구분하기에는 복잡 오묘하지만 우선은 그런 관점에서 바라보도록 하자. UML은 단순한 표기법이라고 이해하는 사람들은 지나치

게 구문에 의존한 학습을 했기 때문이다. 만약 그러한 사람들이 모여 UML로 커뮤니케이션을 한다면 UML로 작성된 소프트웨어 모델은 귀에 걸면 귀걸이,코에 걸면 코걸이가 되는 아주 모호한 것으로 전락하여 UML의 정상적인 능력을 발휘하지 못하게 할 것이다. 또한 UML의 그러한 표기법은 단순한 그림이 아닌 엄연히 구분 규칙이 존재하며 그에 따른 의미도 존재한다는 것을 염두에 두어야 할 것이다.

2) 소프트웨어 모델이란?

ㄱ. 모델(Model)

소프트웨어 모델을 이해하는 것이 UML을 꿰뚫어보는 가장 빠른 길인지도 모른다. 더 근본적으로는 모델(model)을 이해하는 것이 관건인데, 여기서 강조하고 싶은 단어는 "단순"과 "체계적"이라는 단어이다. 어떤 것을 너무 정확하게 표현하고 싶은 욕심에 작은 것 하나 놓치지 않고 모조리 표현했다면 그것은 모델이 아니고, 일정한 기준없이 중구난방식으로 표현해 도저히 이해할 수 없다면 그것 또한 모델이 아니다.

ㄴ. 소프트웨어 모델(Software Model)

소프트웨어 모델(software model)은 실제로 개발할 소프트웨어 시스템을 단순화하여 체계적으로 정 의한 논리적 모델이다. 아마 소프트웨어가 눈에 보이고 손으로 만질 수 있는 것이라면 모델이라는 용어가 더 쉬웠을지도 모르지만 그렇지 못하기 때문에 이해하는데 있어서 좀 더 힘들어 보인다. 이러한 이유에서 소프트웨어 모델을 만들기 위해서는 플라스틱이나 나무, 강철 같은 물리적(physical) 재질이 아닌 수학이나 철학에서 논리적(logical) 원료를 가져올 수 밖에 없다. UML은 바로 수학이나 철학 등에서 적절한 원료를 가지고 사용자가 쉽게 소프트웨어 모델을 구축할 수 있도록 구문과 의미를 잘 정의해놓은 언어라는 것이다.

ㄷ. 소프트웨어 모델과 관점(Perspective)

하나의 소프트웨어 시스템에 대해서 단 하나의 모델이 존재해야 하는 것은 결코 아니다. 더

군다나 소프트웨어와 같이 복잡한 놈에는 더더욱 곤란한 이야기이다. 간단히 원통 모양의 물체를 표현한다 하더라도 한쪽면만 봐서는 그것이 원통 모양인지를 알 수 없다. 정면을 보면 단순히 사각형의 모양 이므로 원통인지, 사각 박스형태인지를 알 수 없다. 그리고 위측면만 보면 원 모양이므로 구(sphere) 형태인지 원통인지 알 수 없다. 이런 저런 다양한 관점에서 보아야 총체적으로 원통 모양인지를 추측할 수 있는 것이다. 소프트웨어는 바라보는 관점은 많이 있을 수 있다. 그것은 경우에 따라 가장 접합한 것을 선택하면 되겠지만 통상적으로 소프트웨어 시스템에 관여하는 이해 당사자들의 관점을 수용하는 경우가 많다. 예를 들어, 최종 사용자의 경우에는 요구 모델(requirement model)을, 시스템 분석가에게는 분석 모델(analysis model), 프로그래머에게는 구현 모델(implementation), 시스템 테스터에게는 테스트 모델 (test model) 등 다양하게 개발될 수 있다. 이렇듯 소프트웨어 모델에는 일정한 관점이 반영되게 되고, 필요에 의해 여러 가지의 관점이 필요하며 이에 따라 여러 개의 소프트웨어 모델이 개발되어야 할 수도 있다.

3) UML 해부

ㄱ. 구문(Syntax)과 의미(Semantics)

우리는 Java, C++와 같은 고급 프로그래밍 언어에서는 구문과 의미를 대체로 잘 이해한다. 비록 BNF, EBNF와 같은 정형 구문(formal syntax)이나 Operational, Denotational, Axiomatic 등과 같은 정 형 의미론(formal semantics)을 잘 모른다 하더라도 어떤 것이 구문이고 그 의미가 어떠하다는 직관 을 가지고 있다. UML도 마찬가지로 구문과 의미가 구문되어 있으며 그 차이를 이해해야 한다. UML을 단순한 그림으로 이해하기 때문에 구문에 맞지 않는 그림이 매우 많고 또한 의미에 맞지 않는 표현으로 인하여 커뮤니케이션의 수단이라는 제 역할을 재대로 수행하지 못하는 경우가 많다. 예를 들어 Package들 사이에 연관(Association) 관계를 맺는 다든지 하는 것은 명백한 구문의 오류 이다. OMG에서 제공하는 UML 명세(specification)에는 구문이 잘 정의되어 있으나, 그 내용이 다소 어려운 관계로

많은 사람들이 숙지하지 못하고 있을 뿐이다. 또한 많은 UML 도구들이 그러한 엄격 한 구문에 의거하여 개발되지 못한 점도 큰 원인 중 하나이다. 다음의 그림 1은 UML 명세에 정의 되어 있는 UML의 요약 구문의 일부를 보여준다. 구문 역시 UML 표기로 표현되어있다는 것은 참 재미있는 일이다. UML의 각 요소(element)를 보여주고 그것들 사이의 일반화(Generalization) 관계 및 연관 관계(Association) 그리고 속성(Attribute)등을 잘 보여주고 있다. 간단하게 Classifier 요소 (Class, Interface, Component 등에 대한 추상 요소)는 여러 개(없거나 하나 이상)의 Feature를 포함하 고 있다. Feature 요소는 Attribute, Operation, Method와 같은 요소의 추상 요소이다. 이런 구문 규칙 이 있기 때문에 Class에 Attribute와 Operation을 포함한다는 것을 UML 표기법으로 표현할 수 있는 것이다.

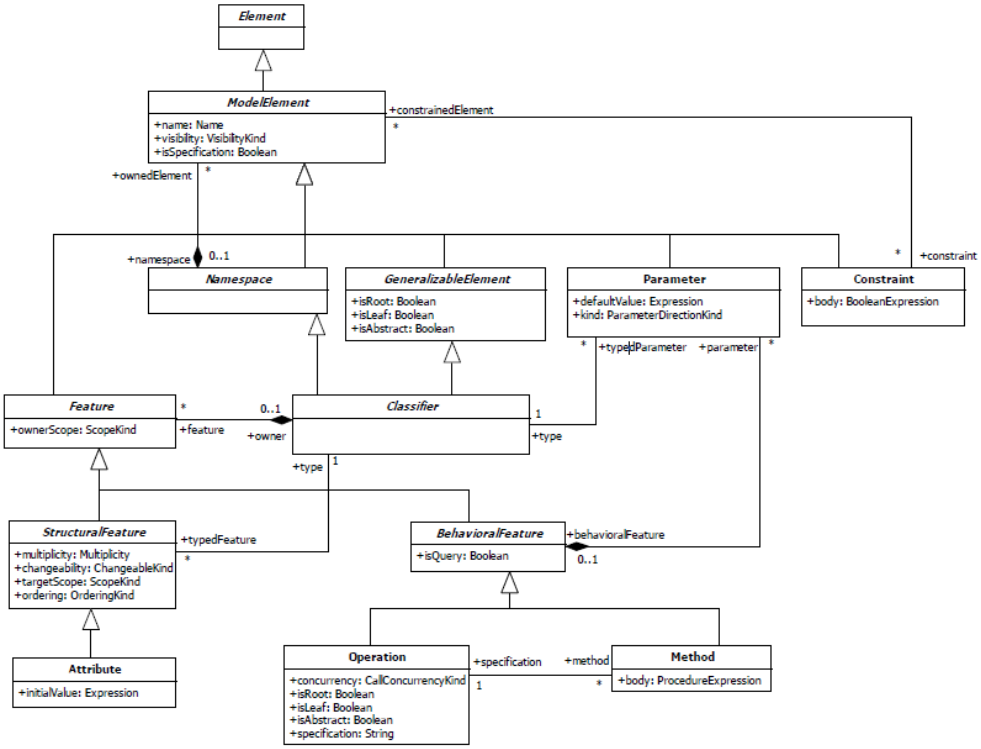


그림 1 - UML 1.4 Abstract Syntax의 일부 (Foundation::Core)

UML의 의미는 특별한 정형 의미론을 도입하지는 않는다. UML 명세에는 단지 자연어(영어)로 그 의미를 설명하고 있고 몇 가지 규칙(well-formedness rules)을 OCL(Object

Constraint Language)로 표현하고 있을 뿐이다. 그래서 UML은 모호한 언어이다. 사실 UML의 의미를 명확히 꿴다는 것은 매우 힘든 일이다. 마치, Java나 C++와 같은 언어의 의미를 모두 꿰지 못하지만 프로그램을 작성하 는 데에는 큰 무리가 없는 것과 일맥 상통한다고 할 수 있다. 여기는 UML의 의미를 다 파악해야 한다는 것이 아니라 구문과 의미가 명백히 구분되어 있다는 것을 알고 그것의 필요한 경우, UML 명세를 통해서 하나 하나 숙지해 나가자는 것이다.

ㄴ. 구조(Structure)와 행위(Behavior)

우리가 표현하고자 하는 소프트웨어 모델은 대체로 구조(structure)와 행위(behavior)로 나눌 수 있다. "클래스, 컴포넌트 등이 어떻게 소프트웨어를 구성하는가?"는 구조를 의미하는 것이고 "어떤 클래스의 인스턴스가 메시지를 받았을 때 어떻게 동작하는가?"는 행위를 의미하는 것이다. UML에는 여러 개의 다이어그램이 존재하는데 각 다이어그램을 구조적인 것과 행위적인 것으로 분류하면 그림 2와 같다.

구조적(Structural)	행위적(Behaviroal)
Class Diagram	Use Case Diagram
Component Diagram	Sequence Diagram
Deployment Diagram	Collaboration Diagram
	Statechart Diagram
	Activity Diagram

그림 2 - UML 다이어그램의 구조적/행위적 분류

여기서 재미있는 것은 구조적인 것에 해당하는 다이어그램들은 주로 소프트웨어를 구성하는 요소의 알갱이 크기(granularity)에 기인하여 구분된다는 것과 행위적인 것에 해당하는 다이어그램들은 행 위의 종류(interaction, state transition, activity flow, ...)에 기인하여 구분된다는 것이다. 이러한 개념을 갖고 구조와 행위를 잘 구분하여 그에 적합한 다이어그램

을 선택하여 작성하여야 할 것이다.

ㄷ. UML의 삼분법(Trichotomy)

UML에는 크게 3개의 개념적 층위(layer)가 존재한다. 그것은 바로 클래스(Classifier, '분류자'로 용어를 사용할 수도 있으나 '클래스'라는 용어에 대부분 다 친숙하고 또 그렇게 이해하더라도 무방하므로 '클래스'라는 용어를 사용한다.), 역할(ClassifierRole), 그리고 인스턴스(Instance)이다.

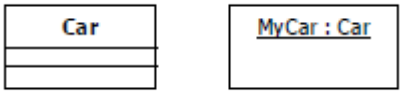


그림 3 – UML에서 클래스와 그것의 인스턴스

우리는 대부분 클래스(Classifier)와 인스턴스(Instance)는 잘 구분한다. UML에서도 사람들은 이 두 가지는 잘 구분해서 쓰는 듯 하다. 그림 3에서와 같이 'Car'라는 클래스(Class)는 사각형에 굵은 글씨체로 이름을 표현하고, 그것의 인스턴스는 자신의 이름에 이어 ':'으로 그것의 클래스 이름을 구분 한다. 그리고 밑줄을 그어 클래스가 아닌 인스턴스임을 나타낸다. 클래스는 주로 클래스 다이어그램 에 나타나고 그것의 인스턴스는 시퀀스 혹은 콜라보레이션 다이어그램에 나타난다. 그러나 이 두 가지(Classifier와 Instance) 이외에 역할(ClassifierRole)이라는 개념 층위가 하나 더 존 재한다. 간단한 예를 들어보자. '사람'이라는 클래스의 인스턴스 '김말복'씨는 회사에서 '팀장'을 맡고 있다. 집에가면 아이 둘 딸린 '가장'이고 자신이 취미로 몸담고 있는 밴드에서는 '드러머'를 맡고 있 다. 분명 이 예에서 '김말복'씨는 '사람'의 인스턴스이지만 여러 개의 많은 역할을 해내고 있다. UML 에서는 이러한 역할이라는 개념을 도입해서 사용할 수 있도록 허용하고 있다. 이로 인하여 최근에 나온 몇 가지 UML 도구에서는 Sequence Role Diagram, Collaboration Role Diagram 이라는 약간 다른 개념의 다이어그램 형태를 제공하고 있다(PLASTIC 2003, Rational XDE 등).

UML에서 역할(ClassifierRole)이 등장하게 된 배경은, 디자인 패턴(Design Pattern)이 많은 개발자들 의 큰 반향을 일으키고 따라서 이것을 UML에 적절히 표현할 수 있는 개념적 베이스를 제공하기 위한 것으로 보여진다. 패턴은 대체로 객체들의 적절한 협동을 통해 특정한 문제를 해결하는 방법 을 정리한 것인데, UML에서 패턴은 하나의 협동(Collaboration)으로 표현된다.

그림 4 – UML에서 협동(Collaboration)의 표현

패턴은 어떤 목적(문제의 해결)을 이루기 위해 객체들이 어떻게 협동하는가에 초점이 맞추어져 있기 때문에 패턴에서 표현되는 각각은 클래스도 아니며 인스턴스도 아니다. 간단히 Visitor 패턴의 경우 방문을 하는 Visitor가 있고 방문을 당하는 Node가 있다. Visitor와 Node는 특정 클래스의 정의를 표현하고자 하는 것이 아니고 더군다나 인스턴스를 지칭하는 것도 아니다. 다만, 객체가 수행해야 할 역할(role)을 의미하는 것이다. 실제 프로그래밍에서도 Visitor와 Node를 직접 클래스로 정의하는 경우는 드물 것이다. 자신에 개발하는 애플리케이션의 어떤 클래스가 Visitor의 역할을 또 다른 클래스가 Node의 역할을 해야 하는 것이다. 즉, 패턴이라는 것은 역할이라는 개념적 층위에서 정의되어야 하는 것이다.

패턴은 UML에서 하나의 협동(Collaboration)으로 정의되고, 역할(ClassifierRole), 역할-연관 (AssociationRole)과 같은 별도의 요소들로 모델링된다. 이것들은 그림 5에서 보듯이 Class와 Association 그리고 Object와 Link에 각각 대응된다. 연산에 대한 호출도 ClassifierRole 수준에서는 Message라는 요소에 대응하고 Instance 수준에서는 Stimulus로 대응된다. 이러한 UML의 개념적 층위를 이해하고 모델링을 한다면 더 명확한 소프트웨어 모델을 작성하는데 도움이 될 것이다.

Classifier Level	ClassifierRole Level	Instance Level
Class	ClassifierRole	Object
Association	AssociationRole	Link
	Message	Stimulus

그림 5-UML에서의 개념 층위와 그에 대응하는 몇 가지 요소

ㄹ. 모델(Model)과 다이어그램(Diagram)

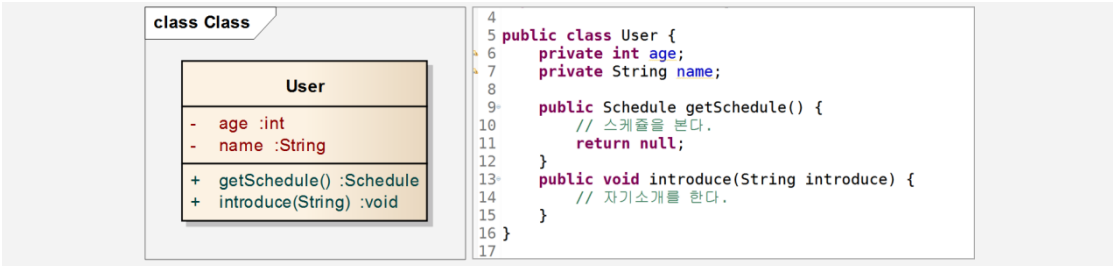
초기의 웹(World-Wide-Web)은 HTML만으로 구성되었으나 최근에는 XML이 많이 사용되고 있다. HTML은 실제 정보와 그것을 표현하는 스타일이 뒤섞여 있어 불편함이 이만 저만이 아니었으나, XML이 등장하면서 실제 정보는 XML로 정의하고 그것을 표현하는 스타일은 CSS나 XSLT와 같은 것을 사용하고 있다. 즉 모델(model)과 뷰(view)를 구분한 것이다. 데이터베이스를 활용한 프로그램 도 마찬가지로 실제 의미 있는 정보는 데이터베이스 내에 존재하고 그것을 표현하는 것은 표, 그래프 등 다양한 방식으로 가능하며, 이들 역시 구분되어 있다. UML도 이와 같이 의미 있는 정보와 그것을 표현하는 것이 구분되어 있다는 것을 이해하는 사람이 그리 많이 있어보지는 않는다. 실제 소프트웨어 모델이라는 것은 논리적인 것으로써 눈에 보이는 것이 아니다. 다만 UML의 다양한 다이어그램을 통해 그것의 일부 단면을 시각적인 그림으로 보여 주고 있을 뿐이다. 클래스(Class)라는 요소는 Name, IsAbstract, IsLeaf, IsRoot, IsActive 등의 속성을 가지고 있고 이에 대한 값들이 실제로 의미가 있는 것이고, 이것을 다이어그램에서는 네모 상자에, 이탤릭체(IsAbstract), 굵은 테두리선(IsActive)으로 나타내는 것 뿐이다. 비록 하나의 클래스(Class) 요소라 할지라도 여러 장의 다이어그램에 걸쳐 나타날 수도 있는 것이다. UML 모델링 도구를 사용할 때 항상 유념해야 하는 부분은 다이어그램 몇 장을 끄적거려 나타내는 것이 아니라 하나의 체계적인

소프트웨어 모델을 작성한다는 생각을 가지고 그것을 모두 시각적으로 표현해내기 위해, 다양한 관점에서 여러 개의 다이어그램들을 그려낸다라는 생각으로 작업에 임해야 한다.

2. UML 구성요소

1) Class (클래스)

클래스는 보통 3 개의 compartment(구획)으로 나누어 클래스의 이름, 속성, 기능을 표기한다. 속성과 기능은 옵션으로 생략이 가능하지만 이름은 필수로 명시해야 한다.

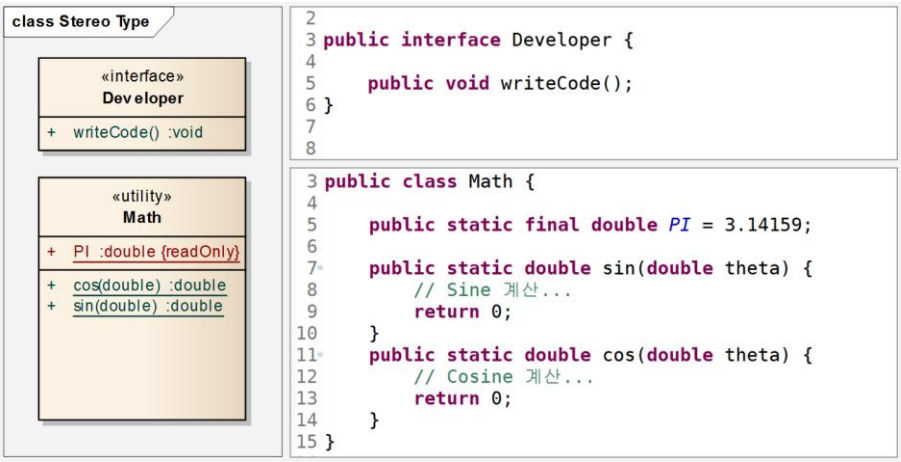


1. [그림 3] 클래스

클래스의 세부사항은 필드와 메소드의 Access modifier (접근제한자), Field name (메소드명), 데이터타입, parameter(매개변수), 리턴 타입 등을 나타낼 수 있다. 클래스의 세부사항들을 상세하게 적는 것이 유용할 때도 있지만, UML 다이어그램은 필드나 메소드를 모두 선언하는 곳이 아니기 때문에 다이어그램을 그리는 목적에 필요한 것만 사용하는 것이 좋다. 추가로 보통 3 개의 구획(compartment)을 사용 하지만 다른 미리 정의되거나 사용자 정의 된 모델 속성(비즈니스 룰, 책임, 처리 이벤트, 발생된 예외 등)을 나타내기 위한 추가 구획도 사용할 수 있다고 한다.

2) Stereo Type (스테레오 타입)

스테레오 타입이란 UML 에서 제공하는 기본 요소 외에 추가적인 확장요소를 나타내는 것으로 쌍 꺾쇠와 비슷하게 생긴 길러멧(guillemet, « ») 사이에 쓴다. 이 길러멧이란 기호는 쌍 꺾쇠와는 좀 다른 것으로 폰트 크기보다 작다. 종이나 화이트보드에 그릴 때는 상관없지만 공식적인 문서라면 이 기호를 구분해서 사용하는 것이 좋을 것 같다.

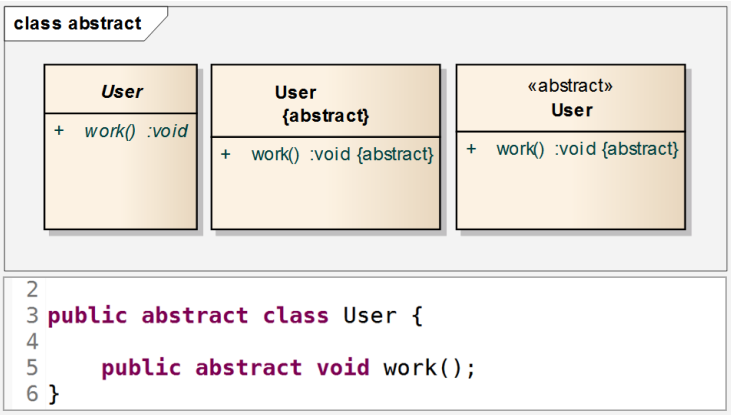


[그림 4] 스테레오 타입

위의 다이어그램은 인터페이스와 유틸리티 클래스를 표현하고 있으며 필드, 메소드 밑의 밑줄은 static(정적)필드 또는 메소드를, readOnly 는 final 키워드를 사용하는 상수를 의미한다. 스테레오 타입으로 많이 사용되는 것은 «interface», «utility», «abstract», «enumeration» 등이 있다.

3) Abstract Class/Method (추상 클래스 / 메소드)

추상클래스란 1 개 이상의 메소드가 구현체가 없고 명세만 존재하는 클래스를 말한다.

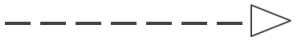
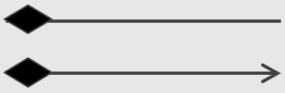


[그림 5] 추상클래스

추상 클래스의 이름과 메소드는 italic 체나, {abstract} 프로퍼티를 사용하여 표기한다. UML 툴에서는 italic 체로 표기하는 것이 많지만 종이나 칠판에 그릴 때는 힘들게 italic 체로 기울여서 적는 것 보다는 {abstract} 프로퍼티로 표기하는 것이 쉽고 명확할 것 같다.
또한 공식적인 것은 아니지만 스테레오타입을 사용하여 추상 클래스를 표기하기도 한다.

3. Relationship

클래스 다이어그램의 주 목적은 클래스간의 관계를 한눈에 쉽게 보고 의존 관계를 파악하는 것에 있다. 그렇기 때문에 클래스 다이어그램에서 가장 중요한 것이 클래스간의 관계이다.

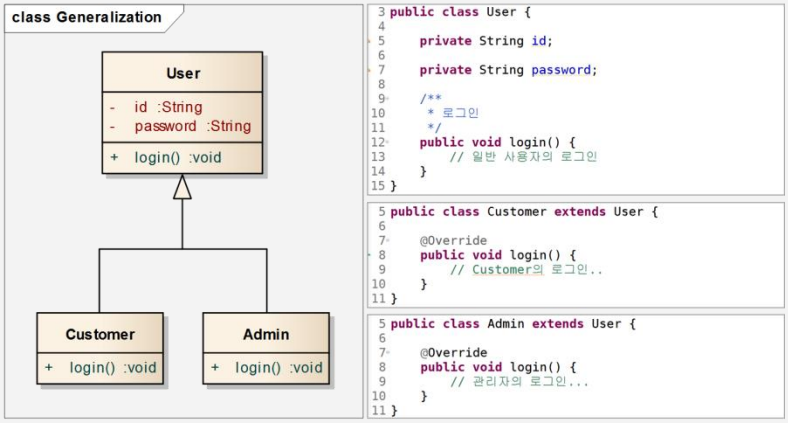
관계	UML 표기
Generalization (일반화)	
Realization (실체화)	
Dependency (의존)	
Association (연관)	
Directed Association (직접연관)	
Aggregation (집합, 집합연관)	
Composition (합성, 복합연관)	

[그림 6] 클래스 관계 종류

위의 그림은 클래스간의 관계들의 몇 가지 종류와 표기법을 나타낸 것이다.

1) Generalization (일반화)

Generalization 은 슈퍼(부모)클래스와 서브(자식)클래스간의 Inheritance(상속) 관계를 나타낸다. 여기서 Generalization 이란 서브 클래스가 주체가 되어 서브 클래스를 슈퍼 클래스로 Generalize 하는 것을 말하고 반대의 개념은 슈퍼 클래스를 서브 클래스로 Specialize(구체화) 하는 것이다. 상속은 슈퍼 클래스의 필드 및 메소드를 사용하며 구체화 하여 필드 및 메소드를 추가 하거나 필요에 따라 메소드를 overriding(오버라이딩) 하여 재정의 한다. 또는 슈퍼 클래스가 추상 클래스인 경우에는 인터페이스의 메소드 구현과 같이 추상 메소드를 반드시 오버라이딩 하여 구현하여야 한다.

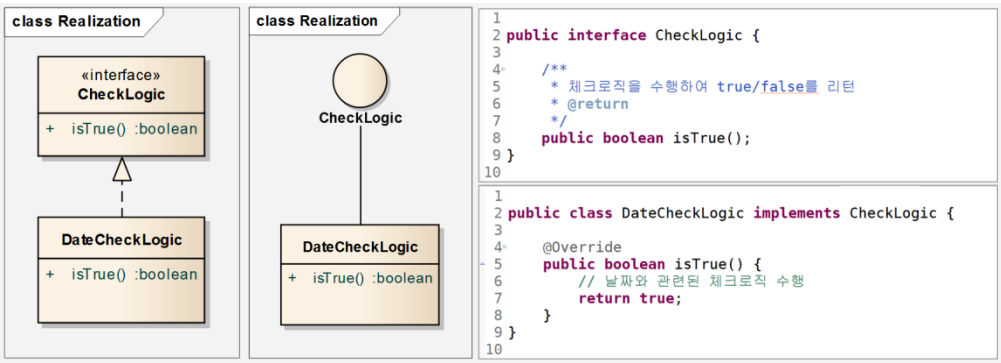


[그림 7] Generalization

위와 같이 표기법은 클래스 사이에 실선을 연결하고 슈퍼 클래스 쪽에 비어 있는 삼각형으로 나타내고 자바에서는 extends 키워드를 사용하여 상속을 구현한다.

2) Realization (실체화)

Realization 은 interface 의 spec(명세, 정의)만 있는 메소드를 오버라이딩 하여 실제 기능으로 구현 하는 것을 말한다.



[그림 8] Realization

Realization 을 나타내는 표기법은 2 가지가 있다.

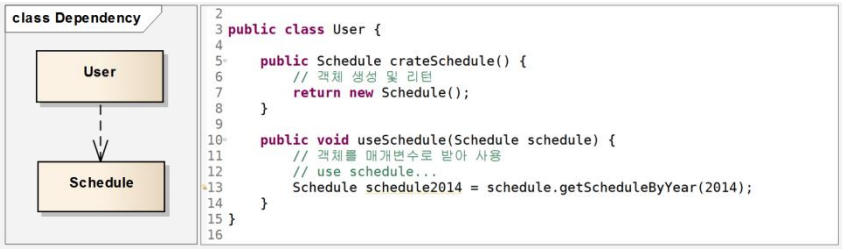
첫 번째는 인터페이스를 클래스처럼 표기하고 스테레오 타입 «interface»를 추가한다. 그리고 인터페이스와 클래스 사이의 Realize 관계는 점선과 인터페이스 쪽의 비어있는 삼각형으로 연결한다.

두 번째는 인터페이스를 원으로 표기하고 인터페이스의 이름을 명시합니다. 그리고 인터페이스와 클래스 사이의 관계는 실선으로 연결한다.

자바에서는 위와 같이 implements 키워드를 사용하여 인터페이스를 구현한다.

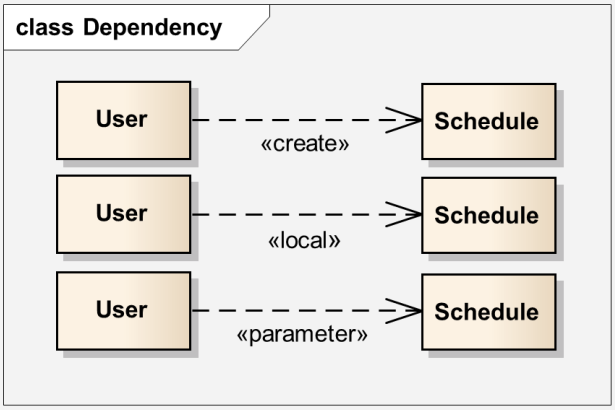
3) Dependency (의존)

Dependency 는 클래스 다이어그램에서 일반적으로 제일 많이 사용되는 관계로서, 어떤 클래스가 다른 클래스를 참조하는 것을 말한다.



[그림 9] Dependency

위의 그림은 자바에서 참조하는 형태에 대해 코드를 보여주고 있다. 참조의 형태는 메소드 내에서 대상 클래스의 객체 생성, 객체 사용, 메소드 호출, 객체 리턴, 매개변수로 해당 객체를 받는 것 등을 말하며 해당 객체의 참조를 계속 유지하지는 않는다.

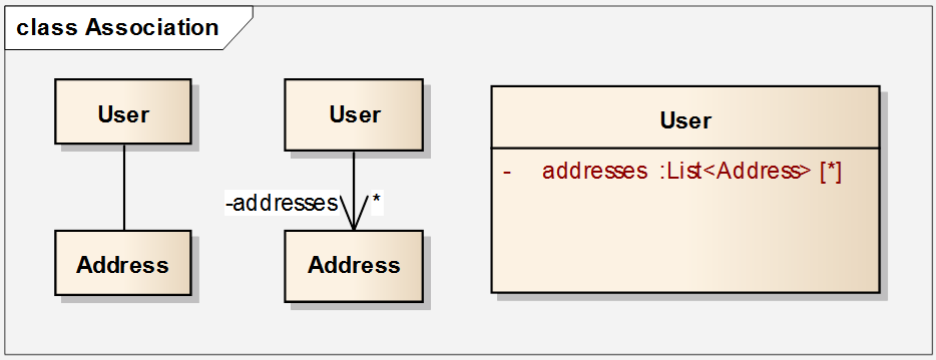


[그림 10] Dependency Stereo Type

추가로 위와 같이 스테레오 타입으로 어떠한 목적의 Dependency 인지 의미를 명확히 명시할 수도 있는데 Dependency 의 목적 또는 형태가 중요할 경우 사용할 수도 있을 것 같다.

4) Association (연관), Directed Association(방향성 있는 연관)

클래스 다이어그램에서의 Association 은 보통 다른 객체의 참조를 가지는 필드를 의미합니다. 아래 클래스 다이어그램은 두 가지 형태의 Association 을 나타내고 있다.




```
1 import java.util.List;
2
3
4 public class User {
5     private List<Address> addresses;
6 }
7
```

[그림 11] Assocication

첫 번째 다이어그램은 일반적인 Association 으로 단지 실선 하나로 클래스를 연결하여 표기하고, 두 번째 다이어그램은 Directed Association 으로 클래스를 실선으로 연결 후 실선 끝에 화살표를 추가한다. Association 과 Directed Association 의 차이는 화살표가 의미하는 navigability(방향성)인데 이것에 따라 참조 하는 쪽과 참조 당하는 쪽을 구분한다.

두 번째 다이어그램은 User 에서 Address 쪽으로 화살표가 있으므로 User 가 Address 를 참조하는 것을 의미한다. Navigability 가 없는 Association 은 명시되지 않은 것으로 User 가 Address 를 참조할 수도, Address 가 User 를 참조할 수도, 또는 둘 다일 수도 있는 것을 의미한다. 화살표 옆에 있는 –addresses 는 roleName(역할명)을 나타내고 Address 가 User 클래스에서 참조될 때 어떤 역할을 가지고 있는지를 의미한다.

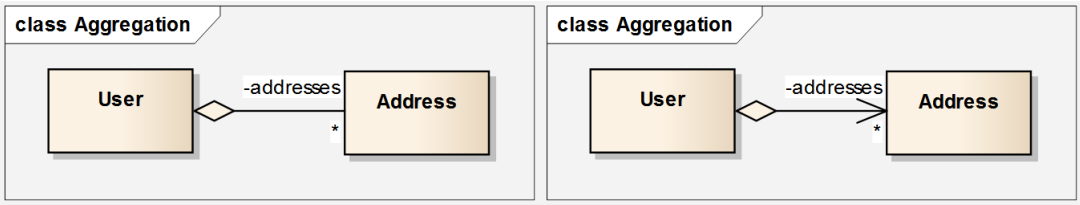
*는 Multiplicity(개수)을 나타내는데 대상 클래스의 가질 수 있는 인스턴스 개수 범위를 의미한다. 0...1 과 같이 점으로 구분하여 앞에 값은 최소값, 뒤에 값은 최대값을 의미하는데 *은 0...*과 같은 의미로 객체가 없을 수도 있고 또는 수가 정해지지 않은 여러 개일 수도 있다는 것을 의미한다.

세 번째의 다이어그램은 두 번째의 다이어그램과 비슷한 의미를 가지고 있지만 다른 형태인 속성 표기법으로 나타낸 것이다. 여기서 보는 바와 같이 roleName 은 보통

클래스의 필드명이 된다. 속성 표기법이 두 번째 클래스 다이어그램과 조금 다른 점은 여러 개의 객체에 대한 Container(컨테이너)가 List 라는 것까지 알려주고 있다.

5) Aggregation (Shared Aggregation, 집합)

Aggregation 은 Shared Aggregation 이라고도 하며 Composition(Composite Aggregation)과 함께 Association 관계를 조금 더 특수하게 나타낸 것으로 whole(전체)와 part(부분)의 관계를 나타낸다. Association 은 집합이라는 의미를 내포하고 있지 않지만 Aggregation 은 집합이라는 의미를 가지고 있다.



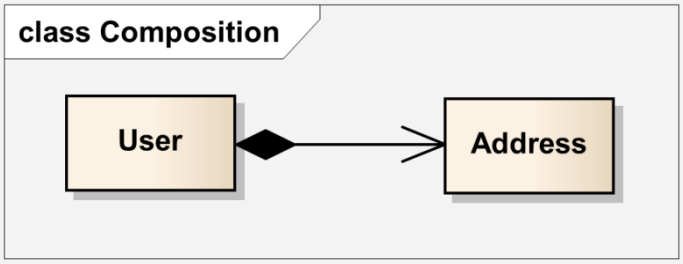

```
1 import java.util.List;
2
3
4 public class User {
5     private List<Address> addresses;
6 }
7
```

[그림 16] Aggregation

표기법은 위와 같이 whole 과 part 를 실선으로 연결 후 whole 쪽에 비어있는 다이아몬드를 표기한다. Part 쪽에는 화살표를 명시하여도 되고 명시하지 않아도 된다. Aggregation 의 다이아몬드가 이미 navigability 의 방향을 표현하고 있기 때문이다. 그런데 코드를 보면, 위에서 보았던 Association 의 코드와 똑같습니다. Association 과 Aggregation 은 집합이라는 개념적인 차이는 있지만 코드에서는 이 차이를 구분하기 힘들다.

6) Composition (Composite Aggregation, 합성)

Composition(또는 Composite Aggregation)도 Aggregation 과 비슷하게 whole(전체)와 part(부분)의 집합 관계를 나타내지만 개념적으로 Aggregation 보다 더 강한 집합을 의미한다.



[그림 19] Composition

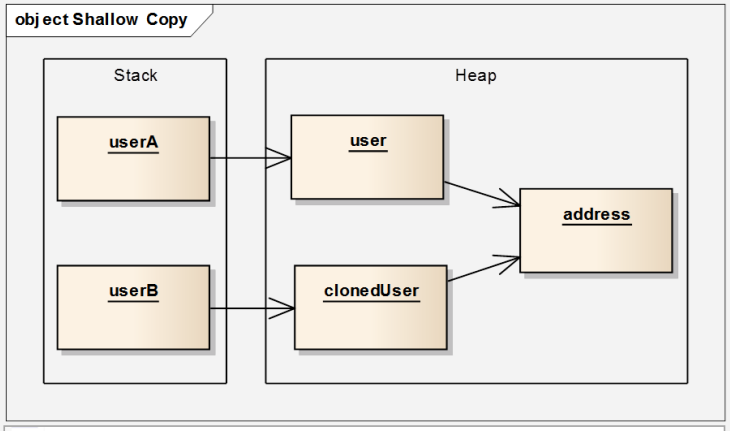
Composition 의 표기법 또한 위와 같이 Aggregation 과 비슷하지만 다이아몬드의 내부가 채워져 있다는 점만 다르다. 그럼 Composition 의 개념과 코드에서는 Aggregation 과 어떤 차이가 있는지 보겠다. Composition 은 Aggregation 보다 강한 집합이라고 했다. 여기서 강한 집합이란 part 가 whole 에 종속적이어서 part 가 whole 의 소유이다. 반면 Aggregation 은 part 가 whole 에 대해 독립적이어서 whole 이 part 를 빌려 쓰는 것과 비슷하다. 이러한 의미 때문에 Aggregation 과는 다르게 명확하게 나타나는 점이 있다.

- 첫 번째, part 를 가지는 whole 인스턴스가 part 인스턴스의 전체 수명을 책임진다.
- 두 번째, part 에 해당하는 인스턴스는 공유 될 수 없다.

첫 번째의 whole 인스턴스가 part 인스턴스의 전체 수명을 책임진다는 의미는 다음과 같다.

- whole 인스턴스가 part 인스턴스를 생성
- whole 인스턴스가 소멸되면 part 인스턴스도 함께 소멸
- whole 인스턴스가 복사되면 part 인스턴스도 함께 복사

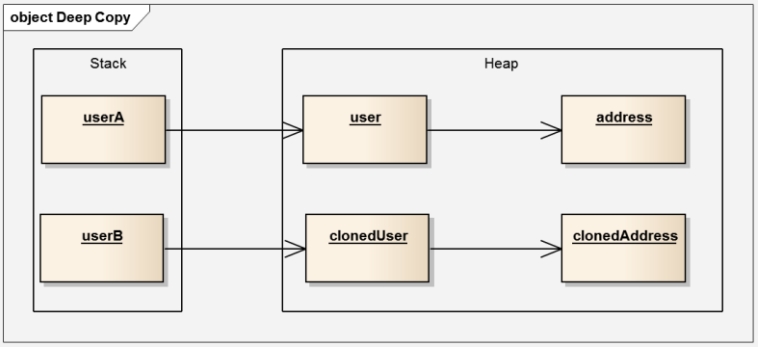
두 번째의 part 에 해당하는 인스턴스는 공유 될 수 없다는 의미를 먼저 보겠다.




```
5
6         User userA = new User();
7         User userB = (User) userA.clone();
8         userB.getAddress().setZipCode("111-111");
9
```

[그림 20] Shallow Copy

위의 예제는 클래스 다이어그램이 아닌 객체 다이어그램으로 [그림 19]를 객체로 표현한 것이다. 참조변수 userA 가 참조하고 있는 user 객체를 clone 하여 clonedUser 객체를 만들고 참조변수 userB 에서 참조하고 있다. user 객체는 제대로 복사가 됐지만 user 객체 안에서 참조하고 있는 address 는 clonedUser 객체도 똑같이 참조하고 있다. 이것을 shallow copy(얕은 복사)라고 하며 이 경우에 part 에 해당하는 address 객체가 공유 된 것이다. 또한 첫 번째 조건의 의미 중 whole 인스턴스가 복사되면 part 인스턴스도 복사되어야 한다는 조건도 충족시키지 못한다.



[그림 21] Deep Copy Object Diagram

위의 다이어그램에서는 user 객체가 복사되어 clonedUser 객체가 생성될 때 user 객체가 참조하여 가지고 있는 address 객체 또한 같이 복사 되어 clonedUser 객체는 새로운 clonedAddress 객체를 참조하여 가지고 있다.

```java
public class User implements Cloneable {

    private Address address;

    public Address getAddress() throws CloneNotSupportedException {
        return address.clone();
    }

    @Override
    protected User clone() throws CloneNotSupportedException {
        //
        User user = new User();
        user.address = this.address.clone();
        return user;
    }
}
```

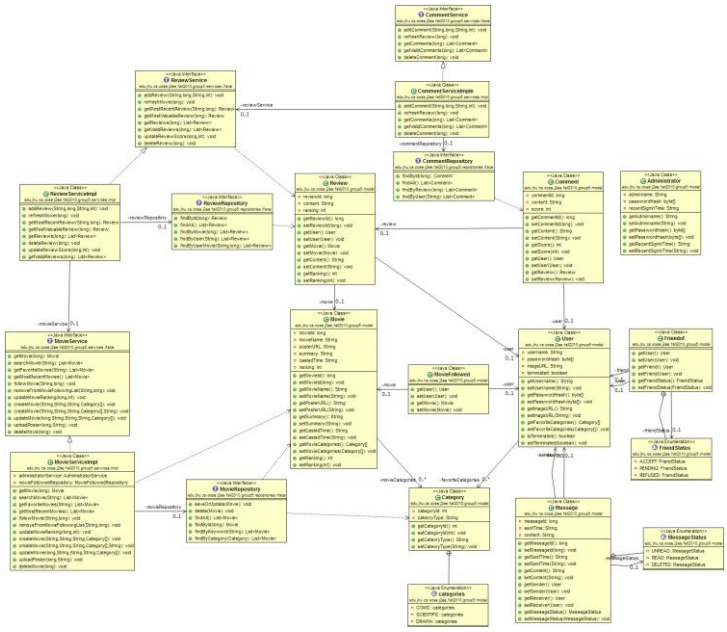
```java
public class Address implements Cloneable {

    private String zipCode;

    public String getZipCode() {
        return zipCode;
    }
    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
    @Override
    protected Address clone() throws CloneNotSupportedException {
        Address address = new Address();
        address.setZipCode(zipCode);
        return address;
    }
}
```

[그림 22] Deep Copy Code

Aggregation 관계와 Composition 관계를 UML 툴에서 그린 후 Code Generation을 하면 똑같은 코드가 생성된다. 하지만 Composition에서는 개발자가 구현해야 할 부분이 몇 가지 있다. 위에서 본 part에 해당하는 인스턴스가 공유되지 않게 하기 위한 Deep Copy 구현과 part를 가지는 whole 인스턴스가 part 인스턴스의 수명 전체를 책임져야 한다는 것에 따라 whole 클래스의 생성자 또는 기타 메소드 내에서 part 인스턴스를 생성해야 하고 외부에서 part 객체를 생성하지 못하도록 whole 클래스에는 part 인스턴스에 대한 setter가 있으면 안된다. 다른 언어에서는 생명주기와 관련된 다른 추가적인 것도 있겠지만 자바에서의 객체 소멸은 Garbage Collector가 수행하므로 part 인스턴스의 소멸은 신경쓰지 않아도 된다.

4. UML Diagram 종류

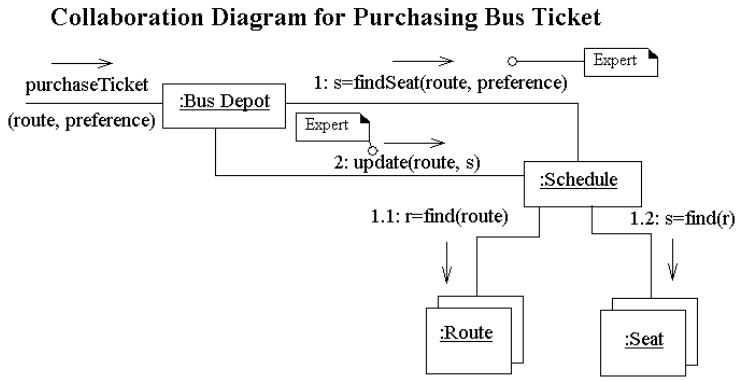
1) Class Diagram



Class Diagram의 예

Class Diagram의 경우 시스템 내부에 존재하는 Class들을 선별하여 나타내고 각 Class들의 속성(Attribute)과 행위(Behavior)를 기입한다. 또한 정적인 관점에서 Class 구조를 표현한다. 여기서 Class들 사이에 관계(Relationship)를 가질 수 있다. 예를 들어 연관관계(Association)은 Class와 Class가 어떠한 연관을 가지고 있음을 나타내고 여기서 세부적으로 복합연관(Composition)과 집합 연관관계 (Aggregation) 등으로 나뉘어 질 수 있다. 이외에 상속관계(Generalization), 의존관계(Dependency)가 나타날 수 있다. Class Diagram은 데이터 모델링을 수행하는 용도가 포함되며, Class Diagram을 그리고자 할 때 항상 추상화 단계를 고려하여서 그리도록 하여야 할 것이다. 추상화의 단계가 높은 경우 대충의 속성과 행위를 기입하고 대충의 관계를 기입하여도 충분할 것이다. 이 단계에서 너무 상세한 내용을 찾고 기입하다 보면 Class Diagram 내부에서 구현의 단계에서 이루어져야 할 일이 이루어지는 오류를 범하게 된다. 이러한 오류는 실제 구현 단계에서 커다란 위험의 요소가 될 수 있다.

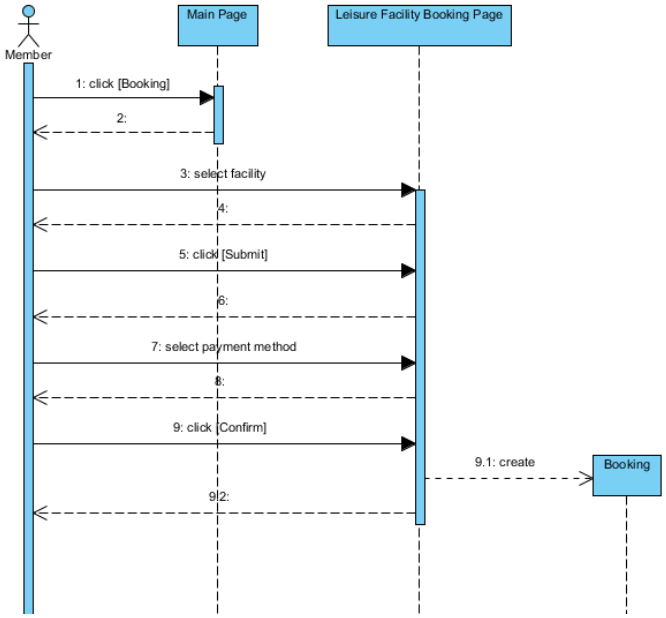
2) Collaboration Diagram



Collaboration Diagram의 예

Collaboration Diagram 또한 Sequence Diagram과 함께 메시지의 흐름을 나타낸다. 하지만 Collaboration Diagram은 인스턴스와 인스턴스들 사이의 관계, 즉 인스턴스들이 어떻게 협동하는지 또한 표기하게 된다. 하나의 협동-인스턴스집합(CollaborationInstanceSet)에 포함된 인스턴스(Instance)들의 협동 모델을 직접적으로 표현한다. 협동 역할 Diagram(Collaboration Role Diagram)은 역할(ClassifierRole) 중심의 관점을 반영한 반면, 협동 Diagram(Collaboration Diagram)은 인스턴스(Instance) 중심의 관점을 반영한 Diagram 이다.

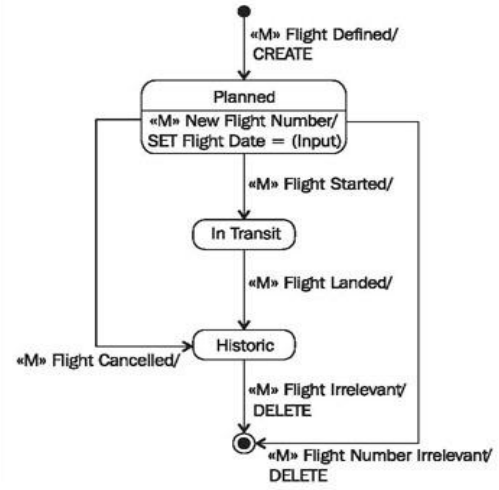
3) Sequence Diagram



Sequence Diagram의 예

Sequence Diagram은 Collaboration Diagram과 함께 시스템의 동적인 면을 나타내는 대표적인 Diagram이다. 시스템 실행 시 생성되고 소멸되는 인스턴스를 표기하고 인스턴스들 사이에 주고 받는 메시지를 나타내게 된다. Collaboration Diagram 또한 메시지의 흐름을 나타내지만 Sequence Diagram 만의 특징이라면 가로축을 시간 축으로 하여 시간의 흐름을 나타내어 메시지의 순서에 중점을 두고 있다. Sequence Role Diagram은 역할 중심의 관점을 반영한 반면, Sequence Diagram은 인스턴스 중심의 관점을 반영한 것이다.

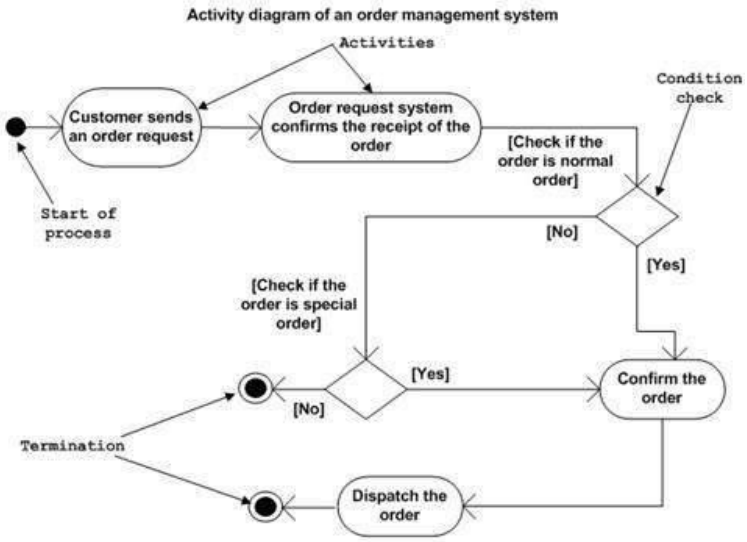
4) Statechart Diagram



Statechart Diagram의 예

Statechart Diagram은 특정 개체의 동적인 행위를 상태(State)와 그것들간의 전이(Transition)를 통해 묘사한다. 일반적으로 클래스의 인스턴스에 대한 행위를 묘사하는데 사용되지만 그 밖의 요소들에 대해서도 얼마든지 사용될 수 있다. 시스템 실행 시 인스턴스의 상태는 메시지를 주고 받음으로써 혹은 어떠한 이벤트를 받음으로써 많은 변화가 있을 수 있다. 실제 시스템에서 실행 시에 많은 인스턴스가 생성되고 소멸된다. 이렇게 무수한 인스턴스의 상태 전부를 모두 Diagram으로 나타내는 것은 불가능하다. 결국 Statechart Diagram은 특별히 관심을 가져야 할 인스턴스에 관하여 그리고 특정 조건에 만족하는 기간 동안의 상태를 표시하여야 한다.

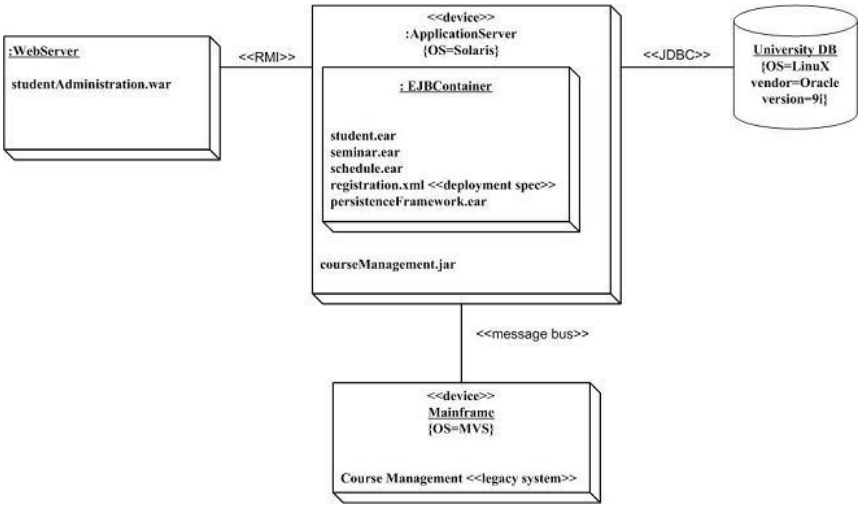
5) Activity Diagram



Activity Diagram의 예

Activity Diagram은 Statechart Diagram의 특별한 형태로써, 시스템 내부에 존재하는 여러 가지 행위들 그리고 각 행위의 분기, 분기되는 조건 등을 모두 묘사한다. 이러한 Activity Diagram에서는 순차적인 제어의 흐름뿐 아니라, 병렬적으로 수행되는 활동과 분기가 이루는 대안들에 대해서도 표현해줍니다. 그리고 일반적으로 작업흐름(Workflow)을 표현하기 위해 많이 사용되며, 클래스, 패키지 혹은 연산 등의 개체에 대해 주로 사용된다. 또한 특정 오퍼레이션(Operation) 내에서의 Work Flow 표현을 위해 Activity Diagram을 사용한다.

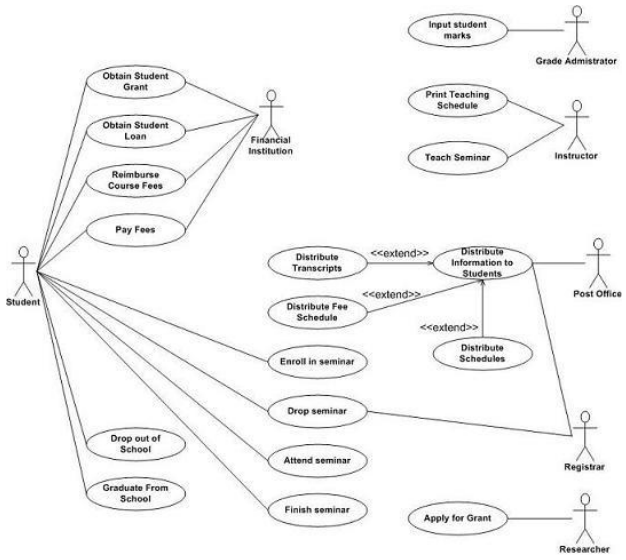
6) Deployment Diagram



Deployment Diagram의 예

Deployment Diagram은 물리적인 컴퓨터 및 장비 등의 하드웨어 요소들과 그것에 들이 배치되는 소프트웨어 컴포넌트, 프로세스 및 객체들의 형상을 묘사하는 Diagram이다.

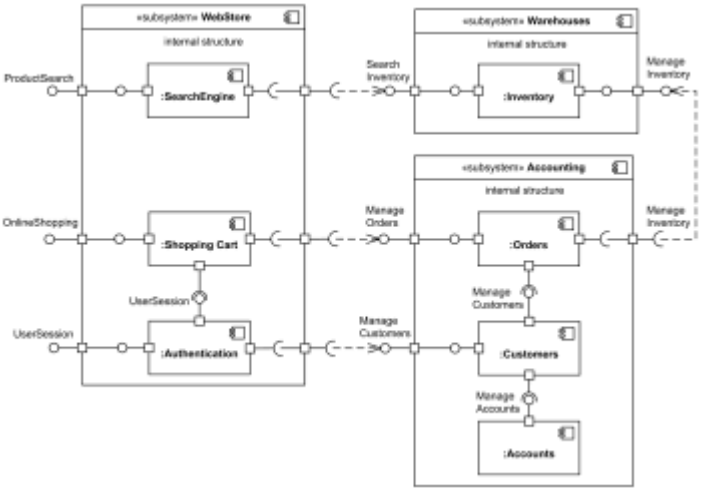
7) Usecase Diagram



Usecase Diagram의 예

Usecase Diagram은 Usecase를 그려놓은 Diagram이다. 여기서 Usecase란 말 그대로 컴퓨터 시스템과 사용자가 상호작용을 하는 하나의 경우이다. 예를 들어 보험처리 프로그램의 경우에 "고객이 보험증권에 sign한다." , "보험 판매원이 판매통계량을 종합한다." 등이 use case가 된다. 이러한 Usecase Diagram은 시스템 구축의 초기에 이 시스템이 어떠한 일을 하는지에 대한 부분을 사용자 입장에서 이해할 수 있을 정도로 기술을 하여야 한다. 이러한 Usecase Diagram은 사용자와의 대화수단으로 그리고 앞으로 구축해 나갈 때의 밑바탕이 된다.

8) Component Diagram



Component Diagram의 예

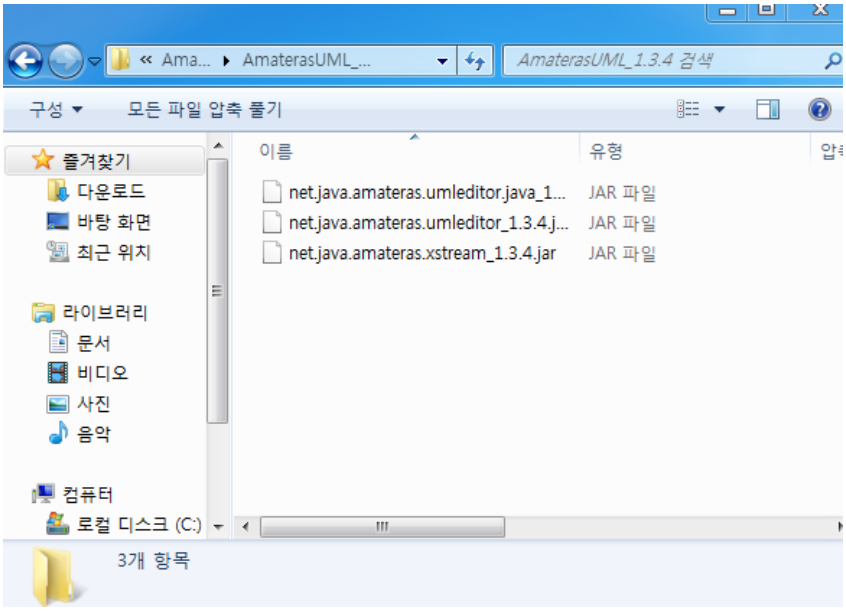
Component Diagram은 소프트웨어 컴포넌트 사이의 의존관계를 나타내는 Diagram이다. 소프트웨어 컴포넌트를 구성하는 요소들과 그것들을 구현하는 요소들도 모두 표현할 수 있다.

5. UML Tool

1) Amateras UML

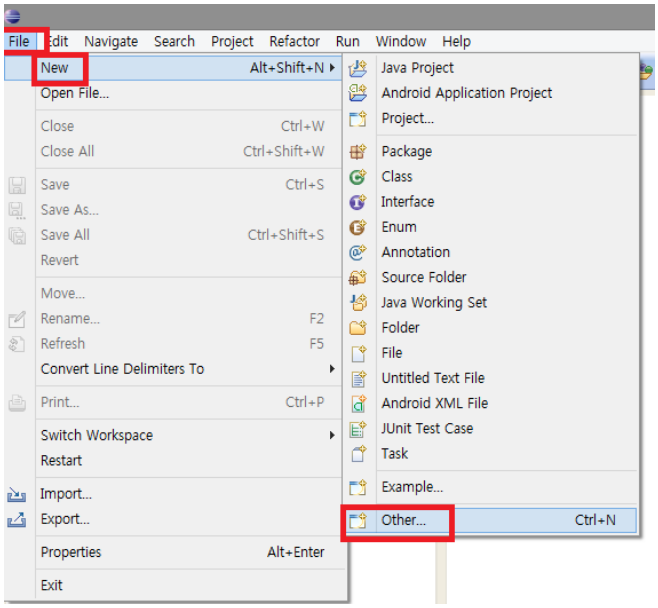
Amateras UML은 JAVA 개발 환경인 Eclipse에서 쉽게 사용할 수 있는 UML 도구로 사용하기가 편하여 많이 사용 되는 UML 도구 이다.

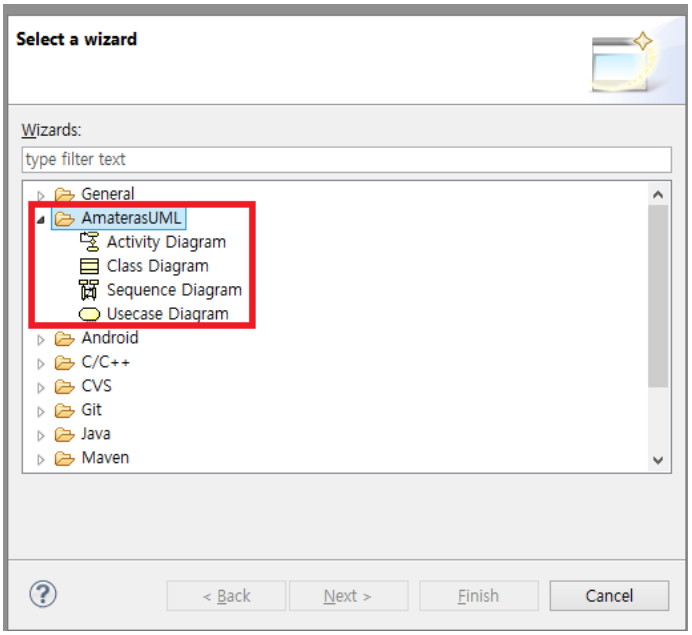
ㄱ. http://sourceforge.jp/projects/amateras/releases/ 에 접속하여 Amateras UML의 zip 파일을 다운 받은 뒤에 압축을 푼다.



ㄴ. 압축을 풀면 위와 같은 파일이 생성되는데 위의 파일을 Eclipse의 plugin 폴더에 넣어주면 Amateras UML을 Eclipse 에서 사용할 수 있다.

ㄷ. Eclipse에서 File > New > Other 을 누른 뒤 AmaterasUML이 있는지 확인한다.



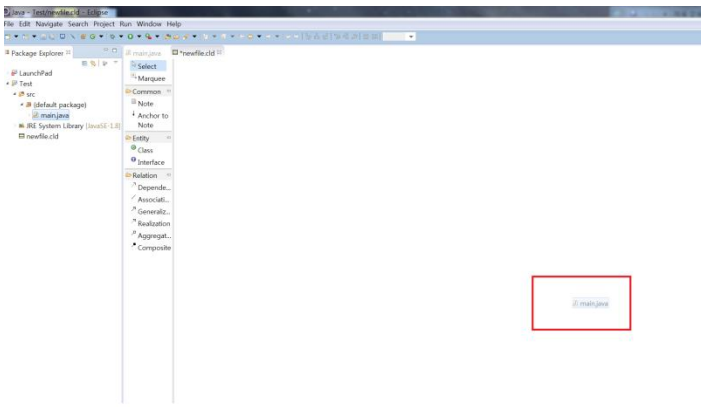


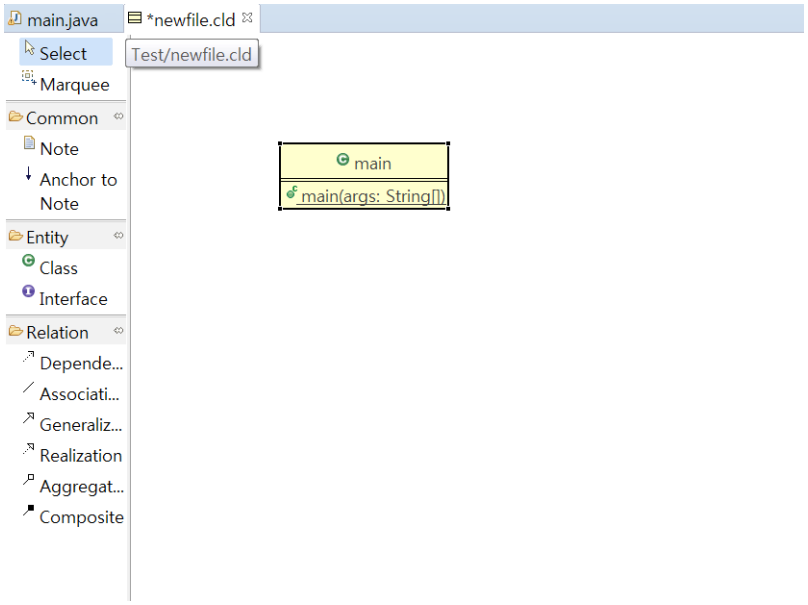
AmaterasUML은 Activity Diagram, Class Diagram, Sequence Diagram, Usecase Diagram 총 4개의 UML을 제공한다.

AmaterasUML이 널리 사용되는 이유는 편리함이 가장 크다.

☞ Import

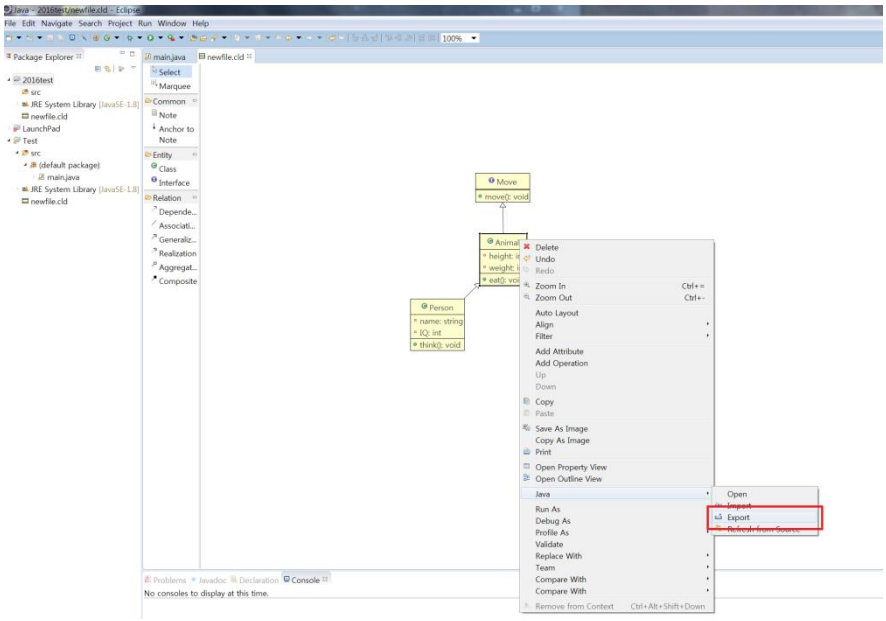
JAVA 파일이 있다면 그 파일을 드래그 하여 넣으면 자동으로 다이어그램을 생성하여 준다.

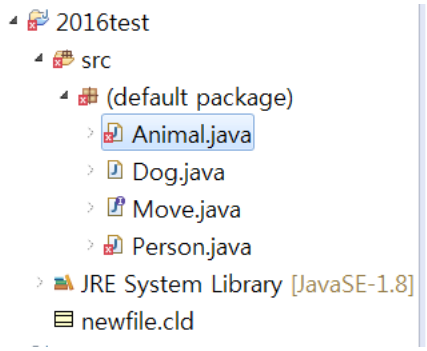




☞ Export

또한 미리 그린 다이어그램을 JAVA 파일로 변환해주기 까지 한다.





2) Star UML

StarUML™은 UML(Unified Modeling Language)을 지원하는 소프트웨어 모델링 플랫폼이다. UML 버전 1.4 에 기반을 두고 있으며, UML 버전 2.0 의 표기법을 적극적으로 지원하고 있다. 총 11 가지의 다양한 종류의 다이어그램을 제공할 뿐만 아니라 UML 프로파일 개념과 템플릿 기반의 문서 및 코드 생성을 지원하여 MDA(Model Driven Architecture) 접근방법을 적극적으로 지원해준다. 또한 사용자의 환경에 대한 맞춤 능력이 우수하고 기능에 대한 확장성이 매우 뛰어난 것이 장점이다.