UML 제대로 이해하기

이민규
㈜플라스틱소프트웨어

요약

1997년 UML이 탄생한지도 꽤 오랜 시간이 흘렀다. 이제 UML은 소프트웨어 개발에 없어서는 안될 필수적인 요소로 자리잡아가고 있다. 그런 반면에, 많은 오해와 선입견으로 그 능력을 제대로 발휘하지 못하고 있으며 이로 인하여 처음 접하는 사람들도 많은 어려움을 겪고 있다. 이 글은 UML을 처음 접하는 사람 혹은 지금 잘 사용하고 있지만 뭔가 부족하다고 생각하고 있는 사람들을 위해서 UML 그 자체를 제대로 이해할 수 있도록 하여 잠재력을 100% 이상 활용할 수 있도록 하기 위한 기초를 다지는 것이 목적이다.

목차

서론

필자는 1997년 UML(Unified Modeling Language)의 탄생 때부터 지금까지 죽 지켜보고, 공부하며, 사용하고 있다. 6년을 거쳐오면서 UML은 매우 많은 사람들의 입에 오르내리면서 사용되고 적용되며 변화하고 있다. 사실 처음부터 UML이라는 이름을 얻은 것은 아니었다. 이 분야의 대가인 Grady Booch와 James Rumbaugh는 각각 독립적인 방법론자(methodologist)였다. 이 두 사람의 방법론은 당시 산업계의 쌍두마차였고, 어떤 계기가 되었는지는 모르지만 통합된 하나의 방법론을 만들고자 했었던 것으로 보였다. 그래서 초기에는 통합 방법론(Unified Method)라는 이름으로 그들의 프로젝트는 시작되었다. 그러다가 어느 시점에서 그들은 프로세스(Process) 부분은 배제시켰고, Unified Modeling Language라는 이름으로 바뀌면서 또 다시 이분야 저명인사인 Ivar Jacobson의 이름이 함께 등장하기 시작했다. 그 이후로 OMG에 표준화 과정을 거치면서 지금까지 발전하게 된 것이다.

UML은 시작적인 표현 언어로써 쉽게 접근할 수 있을 것이라는 예측에도 불구하고, 아직도 UML은 넘기에는 어려운 산이라는 인식이 많이 팽배해 있는 듯 하다. 아직 명확하지 못한 부분도 많이 있고 또한 명백히 정의되어 있지만 UML을 사용하는 사람들에게는 잘 전달되지 못한 부분들도 많다. 필자는 UML을 오랫동안 다루어온 경험을 통해서 UML의 본질을 파악할 수 있도록 하여 좀 더 많은 사용자가 쉽게 적용하고, 그 결과 훌륭한 결실을 맺을 수 있도록 하는 것이 이 글의 목적이다.

UML에 관한 몇가지 오해

UML이 처음 등장할 때부터 많은 무성한 소문들이 퍼지기 시작했다. 어떤 이는 자신이 그리고 있는 그림이 UML인지도 모르는 사람부터, UML이 새로운 프로그래밍 언어라는 사람까지 다양했다. 그러나 그러한 오해들은 금새 풀리지만 아직도 많은 사람들이 가지고 있는 오해가 몇가지 있다. 이것이 UML을 제대로 이해하는데 걸리는 첫번째 걸림돌이라 생각이 된다.

UML은 방법론이다.

물론, UML이 탄생하게된 계기의 처음 시작은 통합된 방법론이었다. 그러나 현재까지 표준화되고 사용되고 있는 것은 방법론과는 완전하게 분리된 모델링 언어(Modeling Language)일 뿐이다. 아직 많은 사이트, 책 기타 논문이나 기사에서 “UML은 … 방법론이다.”이라는 정의가 흔히 등장한다. 초기의 몇몇 선구자들의 오해로 인하여 잘못 전달된 지식이 아직까지 영향을 미치고 있는 듯하다. 이러한 사례는 비단 UML외에도 너무나도 빈번히 일어나는 사건이다. 다시 한번 강조하지만 UML은 방법론(methodology)이 아닌 하나의 모델링 언어(modeling language)일 뿐이다.

그렇다면 여기서 흔히 나오는 질문이 바로 "방법론은 무엇이며 모델링 언어는 무엇이냐"일 것이다. 그러나 그 답변은 그리 쉽지만은 않다. "방법론"은 그 의미가 광범위하고 모호하며 철학적이기 때문이다. 비록 이 단어를 소프트웨어에 국한 시킨다고 하더라도 그것을 쉽게 정의하기란 매우 어렵다. 혹자는 "프로세스 + 표기법 = 방법론"이라는 등식을 쉽게 사용하지만 그것은 차후 또 다른 논란을 불러올 수 있기 때문에 조심스럽다. 우선 필자는 "소프트웨어를 개발하기 위한 총체적인 체계"라고 이야기하면서 여기에는 절차, 표기법, 규칙, 기법, 방식, 문화 등도 포함될 수 있다고 본다.

그리고 모델링 언어(modeling language)란 소프트웨어 모델을 표현하기 위한 언어라고 정의하는 것이 가장 무난하겠지만 가장 무성의한 답변이기도 한다. 이것을 이해하기 위해서는 소프트웨어 모델이 무엇인지를 먼저 이해해야 하고 더 근본적으로는 모델(model)을 이해해야 할 것이다. 모델에 관한 이야기는 중요하므로 뒤에서 다시 자세히 알아보도록 하고, 언어(language)라는 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. 모델링 언어는 결국 소프트웨어 개발을 위한 총체적인 체계의 한 부분일 수는 있으나 그 자체는 아니라는 점을 분명히 해둔다.

UML은 단순한 표기법이다.

또 다른 오해는 매우 심각하다. 이것은 거의 대부분의 사람들이 믿고 있기 때문이기도 하다. 이러한 오해는 오히려 UML이 시각적인 표현을 강조한다는 점에서 출발하고 있는 듯하다. 많은 교육 프로그램들이 아직, UML은 어떤 어떤 다이어그램으로 구성되어 있고, 클래스는 몇 개의 칸으로 구성된 네모로 표현되며 의존 관계는 점선 화살표로 그린다라는 식의 표기법 설명에 치중하고 있기 때문이다. 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 구문(syntax)과 의미(semantics)로 구분하기에는 복잡 오묘하지만 우선은 그런 관점에서 바라보도록 하자. UML은 단순한 표기법이라고 이해하는 사람들은 지나치게 구문(syntax)에 의존한 학습을 했기 때문이다. 만약 그러한 사람들이 모여 UML로 커뮤니케이션을 한다면 UML로 작성된 소프트웨어 모델은 귀에 걸면 귀걸이, 코에 걸면 코걸이가 되는 아주 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.

소프트웨어 모델이란?

모델(Model)

소프트웨어 모델을 이해하는 것이 UML을 꿰뚫어보는 가장 빠른 길인지도 모른다. 더 근본적으로는 모델(model)을 이해하는 것이 관건인데, 이것은 어떻게 보면 아주 쉬운 것이지만 어렵게 보면 매우 철학적 고찰이 필요한 것이다. 조금은 건방진 정의일지 모르겠지만 '모델은 자연 세계의 현상 혹은 사람의 생각 등을 일정한 관점에서 단순화 시켜 체계적으로 표현한 것'이라고 할 수 있겠다. 여기서 강조하고 싶은 단어는 "단순"과 "체계적"이라는 단어이다. 어떤 것을 너무 정확하게 표현하고 싶은 욕심에 조금만 것 하나 놓치지 않고 모조리 표현했다면 그것은 모델이 아니고, 일정한 기준 없이 중구난방식으로 표현해 도저히 이해할 수 없다면 그것 또한 모델이 아니다.

우리 실생활에서는 "모델"이란 용어가 어디에 사용될까? 우선, "모델 하우스"라는 말이 있다. 실제 건축물을 짓기전 그것과 거의 동일한 모델 하우스를 지어서 입주자들이 완성될 건축물에 대한 예측을 할 수 있도록 도와주는 것이다. 모델 하우스는 말 그대로 입주자들에게 완성될 건축물에 대한 이해가 목적이므로 실제 내장재라든지, 상하수도 시설, 냉/온방 장치, 전기 안전장치 등, 기타 그러한 목적과는 별 관계가 없는 것들은 다 생략하게 될 것이다. 그리고 "프라모델"이라는 말이 있는데, 이것은 아이들이 가지고 노는 플라스틱 모델(plastic model)을 줄인 일본식 영어이다. 이것 역시 아이들이 좋아하는 대상물 즉 인기스타, 로봇, 캐릭터, 자동차, 전쟁 무기 등을 단순한 형태만을 본따서 만든 것이다. 만약 실제 전쟁 무기 장난감에 인간에 치명적인 해를 줄 수 있는 기능이 포함되어있을 필요는 없는 것이다. 아니, 그래서는 절대로 안되겠다.

소프트웨어 모델(Software Model)

소프트웨어 모델(software model)은 실제로 개발할 소프트웨어 시스템을 단순화하여 체계적으로 정의한 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 정의해 놓은 언어라는 것이다.

소프트웨어 모델과 관점(Perspective)

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

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 모양이므로 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 바라 보는 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.

UML의 해부

흔히 UML은 몇개의 다이어그램으로 구성되어 있다고들 말한다. 그리고 각각의 다이어그램은 어떻게 생겼으며 어떤 역할을 한다라는 설명한다. 물론, 맞는 말이기는 하지만 UML을 몇 개의 다이어그램적 구성만으로 바라보는 것보다는 다음의 몇가지 관점을 함께 이해하는 것이 좋겠다.

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 정형 의미론(formal semantics)을 잘 모른다 하더라도 어떤 것이 구문이고 그 의미가 어떠하다는 직관을 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 오류이다. 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 엄격한 구문에 의거하여 개발되지 못한 점도 큰 원인 중 하나이다. 다음의 그림 1은 UML 명세에 정의되어 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 포함하고 있다. Feature 요소는 Attribute, Operation, Method와 같은 요소의 추상 요소이다. 이런 구문 규칙이 있기 때문에 Class에 Attribute와 Operation을 포함한다는 것을 UML 표기법으로 표현할 수 있는 것이다.

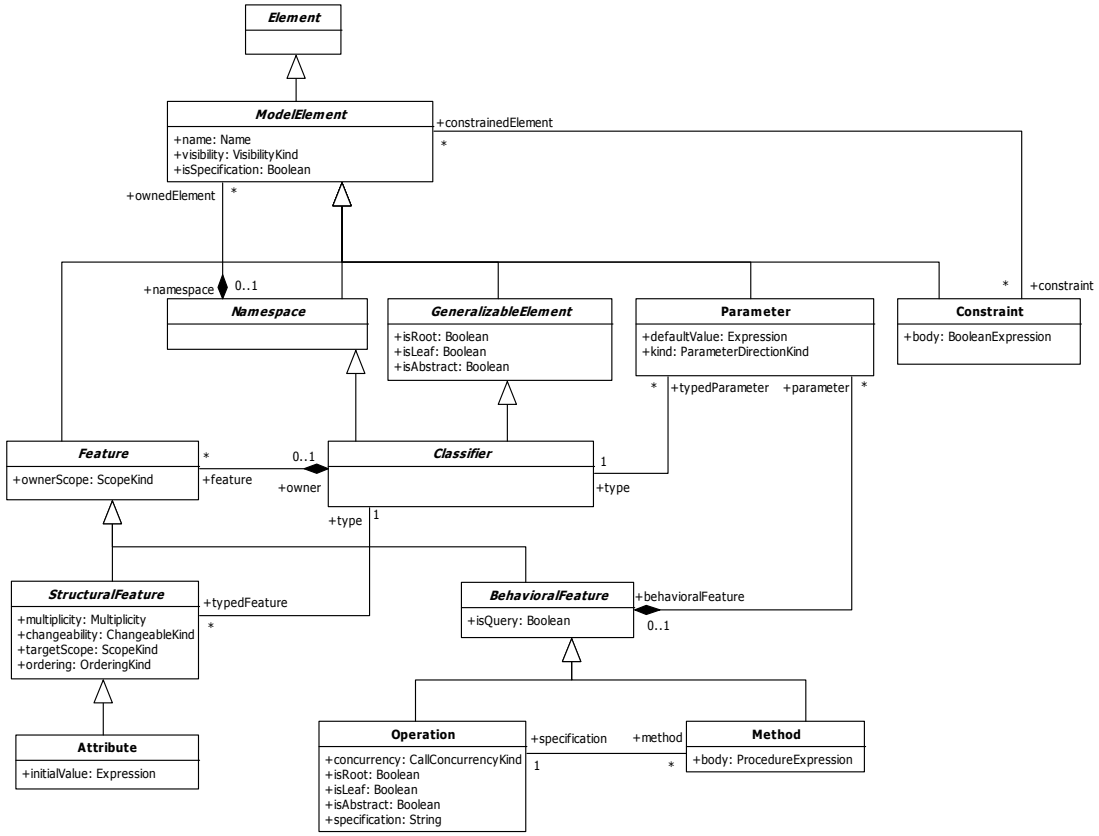

그림 1 – UML 1.4 Abstract Syntax의 일부 (Foundation::Core)

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 작성하는 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.

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 행위의 종류(interaction, state transition, activity flow, …)에 기인하여 구분된다는 것이다. 이러한 개념을 갖고 구조와 행위를 잘 구분하여 그에 적합한 다이어그램을 선택하여 작성하여야 할 것이다.

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.

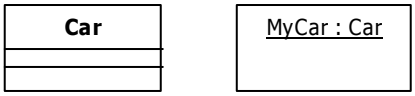

그림 3 - UML에서 클래스와 그것의 인스턴스

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 구분한다. 그리고 밑줄을 그어 클래스가 아닌 인스턴스임을 나타낸다. 클래스는 주로 클래스 다이어그램에 나타나고 그것의 인스턴스는 시퀀스 혹은 콜라보레이션 다이어그램에 나타난다.

그러나 이 두 가지(Classifier와 Instance) 이외에 역할(ClassifierRole)이라는 개념 층위가 하나 더 존재한다. 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 있다. 분명 이 예에서 '김말복'씨는 '사람'의 인스턴스이지만 여러 개의 많은 역할을 해내고 있다. UML에서는 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 개발자들의 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 방법을 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

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 프로그램도 마찬가지로 실제 의미있는 정보는 데이터베이스 내에 존재하고 그것을 표현하는 것은 표, 그래프 등 다양한 방식으로 가능하며, 이들 역시 구분되어 있다.

UML도 이와 같이 의미있는 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 보여주고 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 몇장을 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.

UML은 무엇으로 정의되었나?

램프의 요정 지니: "메타-소원은 들어줄 수 없어요."

이 장의 제목이 무슨 말인가 하면 필자가 읽었던 "괴델-에셔-바흐"라는 책에 나오는 한 귀절이다. 이 귀절을 인용한 이유는 "메타(meta)"라는 용어의 의미를 어떻게 하면 잘 전달할 수 있을까 하는 생각에서이다. 그 책의 내용에 보면 대충 이런 부분이 있다. 램프의 요정 지니가 램프의 주인에게 나타나 "주인님 3가지 소원을 말하세요."라고 말을 한다. 이때 램프의 주인은 "첫번째 소원은 나의 100가지 소원을 더 들어주는 것이다."라고 말을 하면 지니는 "메타-소원은 들어줄 수 없어요."라고 말한다. 여기서 메타-소원이라는 것은 소원에 관한 소원을 말하는 것이다. 이처럼 메타(meta)라는 용어는 어떤 개념적인 층위를 한 단계 올라선 것을 의미한다고 볼 수 있겠다. 그렇다면 꿈에 관한 꿈을 꿨다면 그것은 메타-꿈이고, 소설에 관한 소설을 썼다면 그것은 메타-소설일지도 모른다.

UML 메타모델과 MOF

메타-소원, 메타-꿈, 메타-소설이 존재한다면 모델에 관한 모델, 즉 메타-모델(meta-model)도 존재하는가? 물론 존재하고 더군다나 UML 명세 문서에서도 자주 등장하는 용어이다. 이러한 맥락에서 보면 메타-모델(metamodel)은 모델을 정의하기 위한 모델이라고 이해할 수 있다. 그렇다면 메타-모델을 정의하기 위한 모델은 메타-메타-모델(meta-meta-model)이 되고, 메타-메타-모델을 정의하기 위한 모델은 메타-메타-메타-모델이 될 것이다.

MOF(Meta-Object Facility)는 메타모델을 정의하기 위한 언어(language)와 메타데이터를 저장하는 리포지토리(repository)를 위한 프레임워크(framework)를 정의하는 OMG에서 제정한 표준이다. UML의 메타모델 뿐만 아니라 CWM(Common Warehouse Metamodel), MDA(Model Driven Architecture)에 사용되는 여러 메타모델도 MOF Model에 의거하여 정의되었다. 상세한 내용은 MOF 명세 문서를 참고하기 바란다.

사실 대부분의 사용자는 UML 메타모델을 이해하고 있는 경우는 드물고 또 반드시 알아야 할 필요는 없다. 그러나, UML 모델을 명확하게 작성한다거나, 재사용 가능한 UML 모델을 개발하여 그로부터 다양한 산출물을 자동으로 생성하고자 하는 고급 개발자라면 UML 메타모델을 이해하는 것이 필요하다.

UML은 확장성있는 언어이다.

비록 이 글에서는 소프트웨어 모델을 중심으로 다루고 있지만 사실 UML은 범용 모델링 언어이다. 즉, 다양한 분야에서 비록 소프트웨어가 아니더라도 얼마든지 UML은 사용될 수 있고 또 실제로 협소하나마 사용되고 있기도 하다. 이것은 UML의 범용성이라는 큰 장점이기도 한 반면에 특정 영역에서는 부족한 표현력 때문에 사용하기에 힘든점이 존재한다는 것은 도리어 단점이 되기도 한다. 그러나, UML은 확정성있는 언어이기 때문에 이를 확장하여 표현력을 더 높일 수도 있다. 이것을 UML의 확장 메커니즘(extension mechanism)이라고 한다.

스테레오 타입(Stereotype)과 확장 속성(Tagged value)

UML에서는 많은 모델링 요소들을 제공하고 있기는 하지만 사실 그것만으로는 다소 부족함을 느낀다. 예를 들어 클래스 모델링을 하는 경우 그것들을 좀 더 명확하게 구분하기 위해 GUI에 관계된 클래스는 별도의 표시를 해 두고 싶은 경우가 있다. 이런 경우 유용하게 사용될 수 있는게 바로 스테레오타입(stereotype)이다. 스테레오타입은 UML 요소를 좀 더 상세하게 분류할 수 있도록 도와준다. 위의 예에서와 마찬가지로 GUI에 관계된 클래스는 '<<GUI>>'라고 스테레오타입을 붙이거나, 연산 중에서 생성자에 해당하는 것에 대해서 '<<constructor>>'라고 스테레오타입을 붙일 수 있다.

확장 속성(tagged value)은 스테레오타입과 더불어 사용될 수 있는 또 다른 확장의 방법으로써 기본 UML에 제공하는 속성에 더 부가적으로 사용자가 정의한 속성을 부여하는 방법이다. 예를 들어, 속성(Attribute)을 표현했는데 이것이 직렬화(serialize)할 때 저장되지 않는 다는 정보를 표시하기 싶은 경우, UML에서는 속성(Attribute)에 그러한 용도로 사용되는 것이 없으므로 부가적인 어떤 정보를 표시해야 한다. 이런 경우 확장 속성(tagged value)를 속성(Attribute)에 부여하여 사용할 수 있다. 그러나, 대부분의 일반적인 경우 스테레오타입과 확장 속성은 함께 사용된다. 즉, 특정 스테레오타입이 연결된 경우 그에 따라 관계된 몇 가지의 확장 속성을 함께 사용할 수 있다. 예를 들어 '<<JavaAttribute>>'라는 스테레오타입을 부여한 경우 'Transient', 'IsFinal', 'IsStatic' 등과 같은 확장 속성을 함께 사용할 수 있다. 물론, 그렇게 정의되어 있는 경우에 말이다.

UML 프로파일(Profile)

UML은 매우 일반적인 언어이기 때문에, 특정한 프로그래밍 언어(Java, C++, 등)의 개념을 표현하거나 혹은 특정한 애플리케이션 도메인(금융, 항공우주, 전자상거래, ...)의 개념을 표현하기에는 부족함을 느끼게 된다. 이런 경우 앞서 설명한 스테레오타입과 확장 속성을 사용할 수 있는데, 이것을 체계적인 형태로 정의해서 하나의 패키지로 만든 것이 바로 UML 프로파일(Profile)이다. 예를 들어 UML Profile for Java라는 프로파일이 정의되었다면 Java 언어의 개념을 표현할 수 있도록 만든 것이

므로 '<<JavaClass>>', '<<JavaOperation>>', '<<JavaInterface>>' 등의 스테레오타입과 그것에 관계된 몇가지 확장 속성('IsFinal', 'IsStatic', 'Transient', 'IsSynchronized', …)이 포함되어 있을 수 있다. 이처럼 UML 프로파일은 특정한 플랫폼, 언어, 혹은 도메인 단위로 UML의 표현력을 확장 시켜줄 수 있는 매우 막강한 파워를 지닌다. 현재, OMG에서는 UML Profile for EDOC(Enterprise Distributed Object Computing), UML Profile for EAI(Enterprise Application Integration), UML Profile for CORBA 등을 표준으로 제정했거나 진행중이다.

UML Profile은 또한 MDA(Model Driven Architecture)의 핵심이기도 하다. 이제는 UML 도구를 사용할 때 사용자는 자신의 플래폼이나 도메인에 맞는 프로파일을 정의해서 사용하거나 혹은 이미 정의된 것을 가져와서 사용할 수 있게 되고, 이렇게 정확한 개념으로 표현된 모델은 코드 생성기에 의해서 소스 코드, 설정 파일 정의, 문서, IDL 정의 등 다양한 산출물을 자동으로 생성하게 될 것이다. 이것이 바로 MDA의 핵심 아이디어이기도 하다.

UML 사용에 관한 조언

높은 추상화 수준을 유지하라.

이 조언을 따르지 않는다면 UML을 사용하는 근본적인 목적을 훼손하는 일이다. 아무리 Java로 구현될 프로젝트라 할지라도 모두 Java의 코딩에 들어갈 정보들을 100% UML 모델과 일치시킨다는 것은 정말 바보 같은 일이다. 그럴바에는 차라리 UML을 쓰지 않는 편이 낫다. 그것은 아무 의미없는 중복적인 작업이기 때문이다. 어떤 객체가 다른 객체를 포함할 때, 얼마나 포함하는 가(0..1, 1..*, 0..*), 어떻게 포함하는가(aggregation or composition), 순서는 고려되는가({ordered}, {unordered}), 변경이 가능한가(changeable, addOnly, frozen)과 같은 높은 수준의 개념을 사용해야 한다.

그러나 사실 이러한 사례를 특정 도구들이 부추기고 있기도 하다. 불완전한 라운드트립-공학(Round-trip engineering)의 지원을 통해 어쩔 수 없이 그렇게 사용자들이 유도되고 있고, 오히려 더 불편하게 하고 귀찮게 하고 있다. 현재의 라운드-트립 공학은 매우 불완전한 기술이다. UML과 프로그래밍 언어 사이의 추상화 갭을 인정하지 않고 무조건 동기화시켜 버리는 일종의 쇼에 불과하다. 사용자들이 이런 기능에 갑탄하고 또 비싼 비용을 지불하고 있지만 그 최종 결과는 참담할 뿐이다. 프로젝트가 진행될수록 동기화가 이루어지지 않는 부분이 크게 늘어나고 이를 위해서 개발자들은 더 많은 시간을 할애해서 귀찮은 작업을 해야 하고, 이렇게 했음에도 불구하고 또 다시 헝클어지는 모델과 코드를 보며 얼굴을 붉혀야 하는 일이 한 두번이 아닐 것이다. 자 이제 과욕은 버리고 높은 추상화 수준을 유지하면서 체계적으로 정의된 UML 모델을 개발하기를 바란다. 이것이 나중에 더 많은 잇점을 가져다 줄 거라는 것을 믿자. 그것이 정말 궁금해서 기다리지 못하겠다면 OMG에서 MDA(Model Driven Architecture)가 무엇인지 상세히 알아보기 바란다.

최대한 명확하게 표현하라.

소프트웨어 모델은 명확하게 표현되어야 가치가 생긴다. 명확(precise)하게 표현되어야 한다는 것과 구체적(detail)으로 표현한다는 것과는 하늘과 땅이다. 이것은 앞서 말한 높은 추상화를 유지해야 한다는 것과 일맥상통 한다고 할 수 있는데, 높은 개념의 수준에서 누가봐도 오해의 소지가 없을 정도로 명확해야 한다는 것이지 실제 구현의 수준까지 세세하게 작성하라는 것은 아니다. 건축물을 짓기전에 대충 찹흙으로 빚은 형태의 모델과 명확하게 100:1로 축소하여 정교하게 만들어진 모델의 효용성에 대한 차이는 더 이상 이야기하지 않아도 잘 이해할 것이라 생각된다. 소프트웨어 모델에서 이러한 것을 달성하기 위해서, 가능하다면 다음의 노력을 기울이면 좋겠다.

1. UML의 메타모델과 의미를 OMG의 명세 문서를 통해서 전반적인 형태를 이해한다.
2. 명확한 표현을 위해서 필요하다면 UML 프로파일을 정의해서 사용하라.
3. OCL(Object Constraint Language)은 명확한 표현을 위한 매우 훌륭한 도구이므로 잘 배워서 사용할 수 있다면 더욱 좋다.

위의 3가지만 모두 잘 해낼 수 있다면 훌륭한 소프트웨어 모델러가 될 수 있겠지만 그것은 그렇게 쉽지만은 않을 것이다. 그러나, 이것을 모두 한꺼번에 해내려 하지말고 프로젝트를 수행해 나가면서 조금씩 조금씩 발전시켜 나갈 수 있다면 그리 멀지 않은 시간에 해낼 수 있으리라 본다.

기업에서는 자산(Asset)화하라

최근에 UML을 화이트보드에 그려서 서로 대화하는 용도로만 사용하라고 권장하는 이들이 있다. 물론, 매우 의미있는 일이고 효과도 있을 것이다. 그러나, 그것은 UML의 한쪽면만을 강조한 것이다. UML은 분명 커뮤니케이션의 수준이기도 하지만 기업에서는 매우 가치가 높은 자산이 되기도 한다. 두번째 면은 비교적 최근에 정립되었으며 MDA라는 개발 접근법을 통해 그것이 입증되고 있기도 하다.

현재 UML을 사용하고 있는 대부분의 기업에서는 소프트웨어 모델이 자산화되고 있지 못하고 있다. 비싼 돈을 주고 도구를 도입하여 문서화하기 위한 그림을 그리는데 사용하는 것이 고작이다. 그리고 그렇게 그려진 다이어그램(이러게 만들어진 것을 소프트웨어 모델이라고 하기에는 문제가 많다)들은 아마도 다시 열려져서 활용되는 일은 없을 것이다. 이러한 방식은 시간 낭비, 돈 낭비, 인력 낭비이다. 좀 더 치밀한 계획을 세우고 소프트웨어 모델을 자산화를 위해 노력한다면 시간, 돈, 인력의 낭비가 아니라 획기적인 생산성과 품질의 향상을 가져올 수도 있는데도 안타까운 일이다.

UML로 작성된 소프트웨어 모델이 자산화되기 위해서는 높은 추상화 수준에서 명확하게 작성되어

야만 한다. 이렇게 작성된 모델은 마치 데이터베이스에 저장된 핵심적인 정보가 다양한 그래프와 표로 얼마든지 표현될 수 있는 것처럼, 다양한 소스 코드(IDL, Java, C++, …), 문서(HTML, PDF, RTF, …), 설정 파일(manifest, ini, registry, …), 보고서(클래스당 평균 연산 개수, 총 클래스 수, 패키지별 클래스 분포, …) 등을 얼마든지 자동으로 생성해낼 수가 있다. UML 모델의 자산화는 기업의 경쟁력을 한 수준 끌어올릴 수 있을만큼 의미있는 결과를 보여줄 것이다.

끝맺음

이 글에서는 UML에 대한 많은 오해들과 UML의 본질적인 구성과 숨겨진 가능성을 소개하였고 마지막으로는 몇가지 조언을 곁들였다. 여기서는 단순히 UML의 다이어그램이 어떻게 생겼다는 등의 내용을 다루지 않는다. 많은 사람들이 그런 그다지 중요하지 않은 부분에 많이 치중해 있다는 생각이 들어, 좀 더 UML 그 자체를 이해하고 이를 통해 100% 아니 200% 이상 활용하여 큰 결과를 얻을 수 있었으면 하는 바람에서 글을 쓰게 되었다. 그러나, 역시 몇장 되지 않는 글을 통해 그것들을 모두 전달하기에는 역부족이라는 생각과 좀 더 쉽게 전달하지 못하는 나의 부족함을 느끼게 된다. 아무쪼록 이 글을 통해서 독자들이 UML을 다시 바라보게 되는 계기가 되기를 기대해본다.